

Программирование для Microsoft® WINDOWS на C#

Charles Petzold

Programming Microsoft® WINDOWS with C#

Microsoft® Press

Чарльз Петцольд

Программирование для Microsoft® WINDOWS на C#

Том 1

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 1./ Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2002. – 576 с.: ил.

ISBN 5-7502-0210-0

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 1 состоит из 11 глав, 3 приложений и снабжен компакт-диском, содержащим примеры программ,

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, MicrosoftPress, 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 и 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 5-7502-0210-0

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 Вырезание и буксировка

Оглавление I тома

Введение XIII

Эволюция Windows-программирования XIII
Требования к читателю XV
Требования к системе XVI
Структура книги XVI
Прилагаемый компакт-диск XVII
Техническая поддержка XVII
Особые благодарности XVIII

Глава 1 Работаем с консолью 1

Версия на C# 2
Анатомия программы 4
Пространства имен C# 6
Консольный ввод-вывод 7
Типы данных C# 9
Выражения и операторы 14
Условия и циклы 16
Переходим к объектам 20
Программирование в тональности до-диез 24
Статические методы 26
Обработка исключений 27
Выброс исключений 29
Получение и присваивание значений свойств 31
Конструкторы 34
Экземпляры и наследование 38
Общая картина 41
Соглашения об именовании 43
Выходим за рамки консоли 44

Глава 2 Hello, Windows Forms 45

Информационное окно 47
Формы 52
Отображение формы 54
Это приложение, и мы хотим его запускать 56
Вариации на ту же тему 58
Свойства форм 59
Ввод, управляемый событиями 61
Обработка события Paint 62
Показ текста 65
Шрифт 66
Кисть 67
Координаты точек 67
Событье Paint — особое событье! . 70

Несколько форм, несколько обработчиков ... 70
Наследование форм ... 73
Метод OnPaint ... 75
Как быть с Main? ... 76
События и методы «On...» ... 77

Глава 3 Основные структуры 82

Классы и структуры ... 82
Двумерные координаты ... 84
Массивы точек ... 86
Структура Size ... 86
Версии с плавающей точкой ... 88
Прямоугольник — это точка и размер ... 90
Свойства и методы Rectangle ... 90
Форма подходящего размера ... 93
Форма и ее клиентская область ... 94
Преобразования точек ... 97
Структура цвета ... 99
141 название для разных цветов ... 99
Перья и кисти ... 100
Системные цвета ... 102
Известные цвета ... 104
Какие цвета использовать? ... 105
Выполнение заливки для перерисовки ... 106
Центрирование «Hello World» ... 108
Измерение строки ... 112
Текст в прямоугольнике ... 114

Глава 4 Упражнение по выводу текста 117

Информация о системе ... 117
Разделение текстовых строк ... 118
Значения свойств ... 119
Разбивка на колонки ... 121
Вокруг одни объекты ... 123
Отображение информации о системе ... 128
Windows Forms и полосы прокрутки ... 130
Прокрутка панели ... 130
Наследование класса ScrollableControl ... 134
Прокрутка при отсутствии элементов управления ... 135
Реальные числа ... 138
Отслеживание изменений ... 139
Оптимизация программы ... 142
Задел на будущее ... 144

Глава 5 Линии, кривые и заливка областей 150

Как получить объект Graphics ... 151
Вкратце о перьях ... 152
Прямые линии ... 153

Кое-что о печати ... 154
Свойства и состояние ... 160
Сглаживание ... 161
Сочлененные линии ... 164
Кривые и параметрические уравнения ... 168
Вездесущий прямоугольник ... 171
Многоугольники ... 173
Эллипсы; простой метод ... 174
Дуги и секторы ... 175
Заливка прямоугольников, эллипсов и секторов ... 182
Ошибка смещения на 1 пиксел ... 183
Многоугольники и режим заливки ... 184

Глава 6 Работа с клавиатурой 188

Игнорируемая клавиатура ... 188
У кого фокус? ... 189
Клавиши и символы ... 190
Нажатые и отпущенные клавиши ... 191
Перечисление Keys ... 193
Проверка клавиш-модификаторов ... 200
Не теряйте связи с реальностью ... 201
Клавиатурный интерфейс для программы SysInfo ... 202
Метод KeyPress для символов ... 204
Управляющие символы ... 204
Просмотр клавиш ... 205
Вызов API Win32 ... 210
Обработка информации при вводе с иноязычных клавиатур ... 212
Фокус ввода ... 215
Пропавшая каретка ... 216
Отображение набираемых символов на экране ... 220
Проблема с текстом, выводимым справа налево ... 224

Глава 7 Страницы и преобразования 226

Достижение независимости от устройства при помощи текста ... 226
А сколько это в твердой валюте? ... 227
Сколько точек в одном дюйме? ... 230
А что с принтером? ... 231
Ручное преобразование координат ... 232
Единицы измерений и масштабирование страницы ... 234
Толщина пера ... 238
Преобразования страницы ... 241
Сохранение состояния объекта Graphics ... 242
Метрические размеры ... 243
Произвольные координаты ... 247
Ограничения преобразования страницы ... 251
Знакомство с глобальным преобразованием ... 250
Глобальное преобразование ... 255
Линейные преобразования ... 256

Немного о матрицах 258
Класс Matrix 259
Сдвиг и подобные ему преобразования 261
Комбинирование преобразований 264

Глава 8 Приручение мыши 266

Темная сторона мыши 267
Игнорируемая мышь 268
Несколько быстрых определений 268
Сведения о мыши 269
Колесико мыши 270
Четыре основных события мыши 271
Работа с колесиком 273
Движение мыши 277
Отслеживание курсора и захват мыши 279
Слежка с приключениями 282
Делаем код универсальным при помощи интерфейсов 289
Одиночные и двойные щелчки 293
Свойства, связанные с мышью 294
События при перемещении курсора мыши 294
Курсор мыши 296
Упражнение на определение позиции курсора 303
Добавление клавиатурного интерфейса 306
Как заставить работать дочерние объекты 308
Определение позиции курсора в тексте 313
«Пишем» мышью 314

Глава 9 Текст и шрифты 320

Шрифты в Windows 320
Основные шрифты 321
Высота шрифта и межстрочный интервал 324
Шрифты по умолчанию 325
Начертания шрифта 326
Создание шрифта по названию 328
Единицы измерения размера шрифта 332
Преобразование единиц измерения 337
Свойства и методы класса Font 338
Новые шрифты из FontFamily 344
Метрики дизайна 346
Массивы гарнитур 350
Коллекции шрифтов 356
Вариации метода DrawString 357
Сглаживание текста 358
Определение размеров строки 360
Параметры StringFormat 361
Преобразование знаков и текста 363
Горизонтальное и вертикальное выравнивание 365

Вывод «горячих» клавиш ... 370
Отсечение и подгонка текста ... 371
Позиции табуляции ... 378

Глава 10 Таймер и время 384

Класс Timer ... 385
Структура DateTime ... 388
Местное и всемирное время ... 390
Число отсчетов ... 393
Мировые системы летосчисления ... 395
Удобная интерпретация ... 397
Простые локализованные часы ... 402
Часы в стиле ретро ... 406
Аналоговые часы ... 411
Пятнашки ... 417

Глава 11 Изображения и битовые карты 424

Поддержка битовых карт ... 426
Форматы файлов битовых карт ... 427
Загрузка и вывод изображений ... 431
Сведения об изображении ... 435
Представление изображений ... 439
Вписывание изображения в прямоугольник ... 441
Вращение и наклон ... 446
Вывод части изображения ... 448
Рисование на изображении ... 452
Подробнее о классе Image ... 457
Класс Bitmap ... 460
Программа «Hello World» с битовой картой ... 462
Изображение-тень ... 463
Двоичные ресурсы ... 466
Анимация ... 471
Список изображений ... 477
Класс PictureBox ... 480

Приложение А Файлы и потоки 484

Важнейший класс файлового ввода-вывода ... 484
Свойства и методы FileStream ... 486
Проблема, связанная с FileStream ... 490
Другие потоковые классы ... 490
Чтение и запись текста ... 491
Ввод-вывод двоичных файлов ... 499
Класс Environment ... 502
Разбор имен файлов и каталогов ... 504
Параллельные классы ... 505
Работа с каталогами ... 506
Получение информации о файлах и работа с ними ... 512

Приложение Б Математические функции 515
Числовые типы данных 515
Целочисленное переполнение 516
Десятичный тип данных 518
Бесконечность с плавающей точкой и NaN 520
Класс Math 522
Остаток с плавающей точкой 524
Степени и логарифмы 525
Тригонометрические функции 526
Приложение В Работа со строками 528
Тип char 530
Конструкторы и свойства строк 531
Копирование строк 533
Преобразования строк 535
Конкатенация строк 535
Сравнение строк 537
Поиск в строках 540
Отсечение и заполнение 541
Манипуляции со строками 542
Форматирование строк 543
Сортировка и поиск в массивах 543
Класс StringBuilder 546

Введение

В этой книге рассказывается о том, как писать программы для Microsoft Windows. Способов написания таких программ много, но здесь для этого используются новый объектно-ориентированный язык C# (произносится «Си-шарп») и современная библиотека классов *Windows Forms*, которая входит в состав Microsoft .NET Framework - платформы, представленной публике летом 2000 г. и реализованной спустя примерно полтора года после этого события.

Microsoft .NET Framework — это обширный набор классов, предоставляющий программистам многое из того, что необходимо для создания приложений для Интернета, Web и Windows. Периодика освещала .NET в основном как средство программирования для Web, а в этой книге .NET обсуждается с *другой* стороны. Windows Forms используют для написания как традиционных автономных приложений для Windows (сейчас их иногда называют *клиентскими*), так и интерфейсной части распределенных приложений.

Библиотека Windows Forms предоставляет практически все, что нужно для создания полноценных приложений для Windows. Правда, есть один существенный пробел: в этой библиотеке начисто отсутствует поддержка мультимедиа. В Windows Forms нет даже функции, которая заставила бы пищать динамик компьютера! Я, было, порывался написать собственные мультимедийные классы, но благоразумно (надеюсь) решил повременить с этим, ожидая появления в следующем выпуске Windows Forms гибкой, мощной и простой в использовании поддержки мультимедиа.

Классы, определенные в .NET Framework, нейтральны по отношению к языку. Microsoft выпустила новые версии C++ и Visual Basic, а также новый язык программирования — C#. Другие производители также приспосабливают создаваемые ими языки программирования для работы с классами .NET. Компиляторы новых языков преобразуют исходный текст в .exe-файл, содержащий код на промежуточном языке. В период выполнения этот код компилируется в машинные коды, подходящие для исполняющего программу микропроцессора. Таким образом, .NET потенциально независима от аппаратной платформы.

Я выбрал для этой книги именно C#, так как C# и .NET в прямом смысле слова созданы друг для друга. Поскольку .NET Framework нейтральна по отношению к языку, по этой книге можно учиться писать приложения Windows Forms и на других языках .NET.

Эволюция Windows-программирования

Первая версия Windows выпущена Microsoft в 1985 г. С тех пор Windows постоянно улучшалась и обновлялась, но самые грандиозные изменения относятся к 199.3 (Windows NT) и 1995 (Windows 95) годам, когда Windows перешла с 16-разрядной архитектуры на 32-разрядную,

В первое время после выхода Windows был только один способ создания приложений для Windows — вызов функций интерфейса прикладного программирования (API) Windows из программ на языке C. Хотя Microsoft Pascal также позволял вызывать функции API Windows, этим языком пользовались редко.

За эти годы к программированию для Windows было приспособлено множество других языков, включая Visual Basic и C++. Как C++, так и C# — это объектно-ориентированные языки, поддерживающие большинство типов, операторов, выражений и конструкций C. Поэтому C++ и C# (а также Java) иногда называют языками программирования *на основе C* или языками *семейства C*.

С выходом .NET число подходов, предлагаемых Microsoft для написания Windows-приложений на языках семейства C, увеличилось до трех:

Эволюция подходов (ориентированных на продукцию Microsoft) к программированию для Windows на языках семейства C

Год выхода	Язык	Интерфейс
1985	C	Интерфейс прикладного программирования (API) Windows
1992	C++	Библиотека Microsoft Foundation Class (MFC)
2001	C* или C++	Windows Forms (часть .NET Framework)

Я не собираюсь учить вас тому, какой интерфейс или язык выбирать для создания приложений для Windows, — вы должны это сделать самостоятельно в зависимости от поставленной перед вами задачи и доступных ресурсов.

Если вам нужны дополнительные источники по Windows API, то многие нашли полезной в этом плане мою книгу *Programming Windows* (5-е изд., Microsoft Press, 1998).

Мне никогда не нравилась MFC. Еще до выхода этой библиотеки у меня сложилось отрицательное впечатление об ее конструкции, более того, я считаю, что MFC едва ли можно назвать объектно-ориентированной библиотекой. Но это мое личное мнение, другие программисты с успехом используют MFC, и это до сих пор один из самых популярных подходов к программированию для Windows. Хорошее пособие для изучения MFC — книга Джефа Прозиса (Jeff Prosise) *Programming Windows with MFC* (2-е изд., Microsoft Press, 1999). Для более опытных программистов также могу порекомендовать книгу Джеффри Рихтера (Jeffrey Richter) *Programming Applications for Microsoft Windows* (Microsoft Press, 1999)[1].

С моей точки зрения, библиотека Windows Forms организована намного лучше, чем MFC, и в моем представлении она намного ближе к идеальному объектно-ориентированному интерфейсу для Windows. За 14 месяцев, отданных этой книге, использование Windows Forms стало моим любимым подходом к программированию для Windows.

В терминах программирования как MFC, так и Windows Forms работают, вызывая функции Windows API. С точки зрения архитектуры, можно сказать, что эти интерфейсы расположены наверху Windows API. Это интерфейсы более высокого уровня, предназначенные для облегчения программирования для Windows. Вообще MFC и Windows Forms позволяют решать поставленные задачи с меньшим числом операторов, чем при использовании функций API.

Очевидно, что MFC и Windows Forms не только повышают производительность программиста, но и, подобно любому интерфейсу более высокого уровня, обла-

[1] *Рихтер Дж.* Windows для профессионалов. М.: Русская Редакция, 2000. — *Прим. перев.*

дают меньшей гибкостью по сравнению с интерфейсом более низкого уровня. Windows API позволяет делать много такого, что невозможно при использовании классов Windows Forms.

К счастью, приложив немного дополнительных усилий, можно вызвать из программы Windows Forms функцию API Windows. Здесь я применял это средство лишь иногда — сталкиваясь со слишком большими пробелами в функциональности .NET. Обычно, следуя принципу своей философии, я стараюсь не нарушать изоляцию внутренних механизмов Windows, предлагаемую Windows Forms.

Требования к читателю

Для плодотворной работы с этой книгой надо иметь возможность компилировать и исполнять программы на C#. Для компиляции программ нужен компилятор C#, а для их исполнения — исполняющая среда .NET (или CLR), которая представляет собой набор динамически подключаемых библиотек

Все это есть в Microsoft Visual C#, современной интегрированной среде разработки. Вместо Visual C# можно приобрести более мощную (и, естественно, более дорогую) среду Microsoft Visual Studio .NET, которая в дополнение к C# позволяет программировать на C++ и Visual Basic.

Если вы сторонник варианта «дешево и сердито», можно скачать бесплатный пакет для разработки программ .NET (.NET Framework SDK), В него входят компилятор C# и исполняющая среда .NET. Для этого откройте страницу *http://msdn.microsofl.com/downloads,* выберите слева ссылку Software Development Kits и найдите в списке .NET Framework (помните, что в любой момент содержимое и адрес этого и других Web-узлов, упомянутых в этой книге, может измениться, а в порой узлы могут вовсе исчезнуть).

Эта книга написана, исходя из предположения, что вы умеете программировать хотя бы на C, Знание C++ или Java полезно, но не обязательно. Поскольку C# — новый язык, в первой главе дается краткое введение в C# и разъясняются необходимые понятия объектно-ориентированного программирования. При изложении материала я часто буду обсуждать различные понятия C# по мере знакомства с ними,

Однако эта книга — не исчерпывающее пособие по C#. Для повышения уровня знаний и мастерства владения языком обращайтесь к другим источникам. Несомненно, их станет еще больше по мере роста популярности языка. Книга Тома Арчера (Tom Archer) «Inside C#»[2] дает сведения не только о программировании на C#, но и о более глубоких процессах, а «Microsoft Visual C# Step by Step» (Microsoft Press, 2001) Джона Шарпа (John Sharp) и Джона Джэггера (Jon Jagger) больше похожа на учебник.

Я иногда буду ссылаться на функции API Windows. Как я уже говорил, в случае затруднений обращайтесь за дополнительным разъяснением к моей книге «Programming Windows».

Требования к системе

Как сказано в предыдущем разделе, для эффективной работы с этой книгой читатель должен иметь возможность компилировать и исполнять программы на O*. к системе же предъявляются такие требования;

- *Арчер Т.* «Основы C#», М.: «Русская Редакция», 2001. — *Прим.перев.*

- Microsoft .NET Framework SDK (минимум), Microsoft Visual C# *или* Microsoft Visual Studio .NET (рекомендуется);
- Microsoft Windows NT 4.0, Windows 2000 или Windows XP.

Чтобы программы, написанные на C#, можно было исполнять на других компьютерах, на них должна быть установлена исполняющая среда .NET (также называемая *свободно распространяемым пакетом* .NET Framework). Этот пакет поставляется вместе с .NET Framework SDK, Visual C# и Visual Studio .NET и может быть установлен в вышеупомянутые версии ОС Windows, а также в Windows 98 и Windows Millennium Edition (Me).

Чтобы установить файлы примеров с компакт-диска, прилагаемого к этой книге, на жесткий диск, потребуется около 2,1 Мб свободного места (полностью компилированные примеры занимают более 20 Мб),

Структура книги

В первом выпуске Windows 1.0 весь API умещался в трех динамически подключаемых библиотеках — KERNEL, USER и GDI. Хотя с тех пор число и объем DLL Windows сильно увеличилось, все же полезно делить функции (или классы) Windows на три категории. Во-первых, это функции ядра. Они реализованы во внутренней части ОС и, как правило, отвечают за многозадачность, управление памятью и операции файлового ввода-вывода. Термином *user* здесь обозначен пользовательский интерфейс. К нему относятся функции для создания окон, работы с меню и диалоговыми окнами, а также элементами управления, такими как кнопки и полосы прокрутки. GDI (Graphics Device Interface) — это интерфейс графических устройств, часть Windows, ответственная за вывод графической информации (включая текст) на экран и принтер.

Книга начинается вводными главами. Темы глав с 5 (где рассказывается о рисовании прямых и кривых) по 24 (о буфере обмена Windows) чередуются: главы с нечетными номерами посвящены программированию графики, а с четными — пользовательскому интерфейсу,

Обычно в подобных книгах уделяют мало внимания таким далеким от Windows темам, как файловый ввод-вывод, вычисления с плавающей точкой и манипулирование строками. Однако новизна .NET Framework и C# диктует необходимость включения руководства по соответствующим классам .NET. Некоторые разделы таковыми и являются, а именно три приложения, посвященные обработке файлов, математическим вычислениям и работе со строками. К этим приложениям можно обращаться в любое время после прочтения 1 главы.

Я попытался упорядочить главы и материалы каждой главы так, чтобы построить изложение каждой следующей темы на основе предыдущей и обойтись минимумом «опережающих ссылок». Эта книга написана так, чтобы ее можно было читать последовательно с начала до конца, почти как полную версию < Противостояния»[3] или «Закат и падение Римской Империи»[4].

Хорошо, когда такую книгу можно не только читать подряд, но и использовать как справочник. С этой целью большинство важных методов, свойств и перечис-

[3] Роман С. Кинга . — *Прим. ред.*

[4] **Фундаментальный** труд историка Е- Гиббона. — *Прим. ред.*

лений, используемых при программировании в Windows Forms, вынесены в таблицы в тех главах, где они обсуждаются. Но даже в такой большой книге невозможно охватить *все* о Windows Forms, поэтому она не может заменить официальную документацию по классам .NET.

Написание программ Windows Forms требует определенных усилий, поэтому вы найдете здесь множество примеров кода в виде законченных программ, фрагменты которых можно совершенно свободно копировать и вставлять в собственные разработки (здесь они приводятся именно *для этого*). Но не распространяйте код или программы в исходном виде, поскольку для этой цели служит книга.

У компилятора C# есть потрясающая функция, позволяющая записывать комментарии в виде тэгов XML. Однако я не использовал ее, поскольку приведенные здесь программы, как правило, содержат мало комментариев, а основное описание кода приводится в тексте,

Возможно, вы знаете, что Visual C# позволяет интерактивно конструировать внешний вид приложений. Вы можете располагать разные элементы управления (кнопки, полосы прокрутки и т. п.) на поверхности окна программы, а Visual C# генерирует соответствующий код. Подобные методики полезны для быстрой разработки диалоговых окон и приложений, богатых элементами управления, однако здесь я проигнорировал эту возможность Visual C#.

В этой книге мы не дадим Visual C# генерировать код за нас, а будем учиться писать его самостоятельно.

Прилагаемый компакт-диск

На компакт-диске содержатся все программы-примеры. Можно загружать решения (.sln-файлы) или проекты (.csproj-файлы) в Visual C# и компилировать их.

Честно говоря, я никогда особо не пользовался компакт-дисками, прилагаемыми к книгам. При изучении нового языка программирования я предпочитаю набивать исходный текст самостоятельно, даже если это чужая программа. Мне кажется, что так можно быстрее выучить язык, но это моя привычка.

Если вы потеряете или повредите компакт-диск, не шлите мне писем с просьбой заменить его: по условиям договора с издательством я не могу этого сделать. Microsoft Press является единственным распространителем этой книги и прилагаемого к ней диска. Для замены компакт-диска, получения дополнительных сведений или технической поддержки свяжитесь с Microsoft Press (см. контактную информацию ниже в разделе «Техническая поддержка»).

Техническая поддержка

Я приложил все усилия, чтобы обеспечить точность сведений, изложенных в книге и записанных на компакт-диске. Поправки к этой книге предоставляются Microsoft Press через World Wide Web по адресу:

`http://www.microsoft.com/mspress/support/`

Чтобы напрямую подключиться к Базе знаний Microsoft Press и найти нужную информацию, откройте страницу:

`http://www.microsoft.com/mspress/support/search.asp`

Пожалуйста, присылайте комментарии, вопросы и предложения, касающиеся этой книги или прилагаемого к ней диска, в Microsoft Press:
по обычной почте

Microsoft Press
Attn: *Programming Microsoft Windows with C#* Editor
One Microsoft Way
Redmond, MA 98052-6399

или по электронной почте

MSPINPUT@MICROSOFT.COM

Пожалуйста, обратите внимание, что по этим адресам не предоставляется техническая поддержка. Информацию о технической поддержке C#, Visual Studio или .NET Framework вы найдете на Web-узеле Microsoft Product Support по адресу:

http://support.microsoft.com

Особые благодарности

Писателю обычно приходится работать в одиночку, но, к счастью, вокруг всегда находятся люди, облегчающие его труд.

Я хочу поблагодарить моего агента Клодетт Мур (Claudette Moore) из Литературного агентства Мур за то, что она постоянно поддерживала этот проект «на плаву» и взяла на себя труд разобраться со всеми запутанными юридическими вопросами,

Как всегда, сплошным удовольствием была работа с сотрудниками Microsoft Press, которые в очередной раз спасли меня от конфуза. Если бы не редактор проекта, Сэлли Стикни (Sally Stickney) и технический редактор Джин Росс (Jean Ross), примеры программ в этой книге были бы путаными и полны ошибок. Хотя порой мне кажется, что мои редакторы обладают сверхчеловеческими способностями, это, увы, не так. Любые ошибки или невнятные фразы остались исключительно по моей вине.

Да, хочу еще выразить признательность Иоганну Брамсу за музыкальный аккомпанемент к моей работе и Энтони Троллопу, чьи произведения позволили мне отвлекаться от работы во время вечернего чтения.

Мои друзья, собирающиеся у меня по воскресеньям, вторникам и четвергам помогали мне и оказывали поддержку, порой незаметную со стороны, но всегда бесценную, и будут делать это дальше.

Больше всех я хотел бы поблагодарить мою невесту Дэйдру, обеспечившую инфраструктуру совсем иного рода (.NET тут ни при чем), позволяющую мне жить, работать и любить.

Чарльз Петцольд
Ноябрь 2001, Нью-Йорк.

ГЛАВА

1



Работаем с консолью

Учебное пособие Брайена Кернигана и Денниса Ритчи «Язык программирования Си», лаконичное и, наверное, поэтому очень любимое программистами, начинается с примера программы, получившей известность как программа «hello-world»[1]:

```
ffinclude <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Хотя такие программы вряд ли могут задействовать всю мощь современного компьютера, они очень полезны на практике, так как позволяют начинающему программисту убедиться, что компилятор и все относящиеся к нему файлы установлены правильно. Кроме того, программы «hello-world* помогают оценить затраты усилий при работе на данном языке: в одном такая программа может состоять из одной строки, в другом — иметь устрашающие размеры. И, наконец, программы «hello-world» очень полезны авторам книг по программированию, так как показывают, на чем сосредоточиться в начале обучения.

Как знают все С-программисты, точка входа в программу на С — функция с именем *main,* функция *printf* показывает форматированный текст, а файл stdio.h — файл заголовков, содержащий описание *printf* и других стандартных библиотечных функций С. Угловые, простые и фигурные скобки используются, чтобы заключать в них информацию или группировать операторы языка.

---

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language,* 2nd ed. (Englewood Cliffs, NJ: Prentice Hall, 1988). В первой редакции книги эта программа была такой же, но без оператора *#include.*
Имеется перевод на русский: Б. Керниган, Д. Ритчи, *Язык программирования Си* (М.: Финансы и статистика, 1992). — *Прим. перев.*

Традиционные программы «hello-world» создаются для работы в среде, поддерживающей изящный старомодный тип компьютерного интерфейса — *командную строку,* или *консоль.* Этот «чисто текстовый» тип интерфейса восходит к устройству, называемому *телетайпом,* которое в свою очередь, основано на старинном устройстве работы с текстами — *пишущей машинке.* Когда пользователь вводил данные на клавиатуре телетайпа, устройство печатало символы на рулоне бумаги и отправляло их на удаленный компьютер. Удаленный компьютер в ответ отправлял свои символы, которые телетайп принимал и печатал на бумаге. В этой модели ввода/вывода отсутствовала концепция позиционирования текста на странице. Поэтому функция *printf* показывает текст там, где в данный момент находится печатающая головка телетайпа или курсор командной строки видеотерминала.

Интерфейс командной строки существует и в Microsoft Windows в форме окна приложения, называемого приглашением MS-DOS или командной строкой. Хотя с появлением графических интерфейсов интерфейс командной строки считается устаревшим, программы, работающие в режиме командной строки, часто оказываются проще тех. что работают в графической среде. Поэтому с них хорошо начинать изучение новых языков программирования.

Версия на C#

В этой книге я расскажу о C# (произносится «си-шарп», как тональность «Лунной сонаты» Бетховена[2]). C#, разработанный Андерсом Хейлзбергом (Anders Hejlsberg) из Microsoft, — современный объектно-ориентированный язык программирования, содержащий элементы C, C++. Java, Pascal и даже BASIC. В этой главе дается поверхностное и, конечно же. неполное описание языка.

Файлы исходного кода C# имеют расширение .cs («с sharp»). Моя первая версия программы «hello-world» на C# содержится в файле ConsoleHelloWorld.cs.

ConsoleHelloWorld.cs

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

class ConsoleHelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
    }
}
```

У вас есть пара вариантов компиляции программы в зависимости от того, сколько денег вы готовы потратить и насколько вы требовательны к удобству среды программирования.

[2] C# в музыкальной нотации — до-диез; англофоны произносят это как «си-шарп». — *Прим. перев.*

Дешевле всего скачать .NET Framework Software Development Kit (SDK) с *http://msdn.microsofl.com*. При установке SDK устанавливаются и DLL (dynamic-link libraries, динамически подключаемые библиотеки), содержащие исполняющую среду .NET. Здесь же и техническая документация по .NET. Кроме того, вы получите компилятор C#, вызываемый в командной строке, который можно использовать для компиляции приведенных в этой книге программ,

Для написания программ на C# годится любой текстовый редактор, начиная с Microsoft Notepad. Компилятор C# называется csc.exe. ConsoleHelloWorId.cs компилируется такой командной строкой:

```
csc consolehelloworld.cs
```

И все! Этап компоновки отсутствует. (Как вы увидите в главе 2, компиляция программ Windows Forms потребует указания кое-каких дополнительных параметров.) Компилятор создаст файл ConsoleHelIoWorld.exe, который можно запускать из командной строки.

Кроме того, можно создавать, компилировать и выполнять программы в Visual C# .NET, новейшей интегрированной среде разработки Microsoft. Visual C# .NET необходима профессиональным разработчикам на C#. Она крайне полезна для определенных типов программ Windows Forms, которые рассматривают окно программы как форму, содержащую управляющие элементы, например, кнопки, поля ввода текста, полосы прокрутки. Но применять Visual C# .NET не обязательно. Я считаю, что одно из самых больших удовольствий программирования для Windows на C# в том, что библиотека Windows Forms не требует использования отдельных файлов, Практически все содержится в файле исходного кода на C#, а все, что нужно для его заполнения, — пораскинуть мозгами и поработать руками.

Далее я опишу действия, которые выполняются для создания программ этой книги в Visual C# .NET. Каждый пример программы из этой книги — это *проект*, под который отводится свой каталог на диске. В Visual C# .NET проекты обычно группируются в *решения*. Например, я создавал решение для каждой из глав этой книги. Решение также является каталогом. Проекты же — подкаталогами каталога решения.

Для создания решения выберите в меню File | New | Blank Solution. В диалоговом окне New Project выберите его местоположение на диске и введите имя решения. Именно так я и создавал решения для каждой из глав этой книги.

После загрузки решения в Visual C# .NET можно создать проекты для этого решения. Выберите File | Add Project | New Project. (Можно также щелкнуть правой кнопкой имя решения в Solution Explorer и выбрать в контекстном меню Add | New Project.) В диалоговом окне Add New Project выберите тип проекта на вкладке Visual C# Projects. Можно выбрать один из нескольких шаблонов. Если вы не хотите, чтобы Visual C# .NET генерировала для вас код (лично я предпочитаю писать собственный), выберите шаблон Empty Project (пустой проект). Именно так я и создавал проекты для этой книги.

При работе с проектом можно выбрать Project | Add New Item для создания нового файла исходного кода на C#. (И здесь тоже можно щелкнуть правой кнопкой имя проекта в Solution Explorer и выбрать этот пункт в контекстном меню.) В диалоговом окне Add New Item в списке Categories выберите Local Project Items,

В разделе Templates выберите Code File. При использовании этого шаблона Visual C# .NET не будет генерировать код для вас.

При любом способе создания и компиляции ConsoleHelloWorld — в командной строке или в Visual C# .NET — файл .exe будет небольшим — 3 или 4 Кб в зависимости от того, помещает ли в него компилятор отладочную информацию. Исполняемый файл состоит из операторов MSIL (Microsoft Intermediate Language). MSIL представлен на рассмотрение ECMA (European Computer Manufacturer's Association. Европейская ассоциация производителей компьютеров) в качестве стандарта под названием CIL (Common Intermediate Language). При запуске программы среда CLR (Common Language Runtime) компилирует промежуточный язык в «родной» машинный код компьютера и подключает соответствующие DLL .NET. Вероятно, сейчас вы используете компьютер архитектуры Intel, поэтому исполняющая среда будет генерировать 32-битный машинный код Intel x86.

Посмотреть на MSIL можно, запустив дизассемблер Intermediate Language Disassembler (ildasm.exe):

```
ildasm consolehelloworld.exe
```

Чтобы получить документацию по набору инструкций MSIL, скачайте файл, описанный сокращением «CIL», с *http://msdn.microsoft.com/net/ecma*. Могут оказаться полезными и другие файлы на этой Web-странице. Можно даже писать код прямо на MSIL и собирать его, используя ассемблер Intermediate Language Assembler (ilasm.exe).

Так как написанные на C# программы компилируются в промежуточный язык, а не напрямую в машинный код, исполняемые файлы независимы от платформы. Возможно, в будущем исполняющую среду .NET перенесут на компьютеры отличной от Intel архитектуры. Если это произойдет, исполняемые файлы, которые вы создаете сегодня, будут на них запускаться. (Позвольте мне добавить «теоретически», чтобы не показаться безнадежно наивным.)

Кроме того, используя .NET Framework и программируя на C#, вы создаете *управляемый код* (managed code) — код, который может исследоваться и анализироваться другими программами для определения его возможностей. Наличие управляемого кода необходимо для обмена двоичными исполняемыми файлами через Интернет.

Анатомия программы

Давайте еще раз взглянем на программу ConsoleHelloWorld.

Как и в C++ и Java (а также во многих реализациях C), пара символов «/» является началом однострочного комментария. Все, что справа от них, не учитывается при компиляции программы. C* поддерживает и многострочные комментарии, заключаемые в комбинации символов «/*» и «*/». Интересно, что в C* такие комментарии могут содержать операторы XML (Extensible Markup Language), которые в дальнейшем можно отформатировать и задействовать для генерации документации по коду. Это очень удобно, и я советую вам изучить все, что с ней связано, но я вынужден не использовать ее в примерах программ этой книги.

ConsoleHelloWorld.cs

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

class ConsoleHelloWorld
{
     public static void Main()
     {
          System.Console.WriteLine("Hello, world!");
     }
}
```

Точка входа в программу «hello-world» на C# — функция *Main* внутри первой пары фигурных скобок. Подобно C, C++ и Java, C# чувствителен к регистру. Но имя *Main* точки входа в программу на C# пишется с заглавной буквы, тогда как в этих трех языках программирования *main* пишется строчными буквами. Пустые скобки показывают, что *Main* не имеет параметров, а ключевое слово *void* — что она не возвращает значения. Можно также указать, чтобы *Mam* принимала массив строк символов в качестве входного параметра и возвращала целое значение. Ключевые слова *public* и *static* я рассмотрю немного позже. Ключевое слово *public* не является здесь строго необходимым; программа будет компилироваться и запускаться и без него.

Main располагается внутри определения *класса*. Класс — основной структурный и организационный элемент объектно-ориентированных языков программирования, подобных C#. Проще всего сказать, что класс — совокупность связанных между собой кода и данных. Я назвал этот класс *ConsoleHelloWorld*. В этой книге я буду, как правило, но не всегда определять по одному классу в исходном файле. У файла будет такое же имя, как и у класса, но с расширением .cs. Это соглашение об именах не является необходимым в C#; эта концепция появилась в Java, и мне она нравится. Итак, файл, содержащий класс *ConsoleHelloWorld* называется ConsoleHellcWorlcl.cs.

System.Console.WriteLine напоминает вызов функции, и это, действительно, так. Данная функция принимает один параметр — текстовую строку — и показывает ее на консоли, в окне командной строки, на вашем допотопном телетайпе или где-то еще. Если вы скомпилируете и запустите программу, она покажет строку:

```
Hello, world!
```

и завершит работу.

Длинное имя функции *System.Console.WriteLine* состоит из таких частей:

- *System* — пространство имен;
- *Console* — класс, определенный в этом пространстве имен;
- *WriteLine* — метод, определенный в этом классе; метод — то же самое, что и функция, процедура или подпрограмма.

Пространства имен C#

Пространство имен (namespace) — концепция, позаимствованная из C++ и позволяющая обеспечить уникальность всех имен, используемых в конкретной программе или проекте. Иногда программисту при работе над крупным проектом не хватает удобочитаемых глобальных имен или нужны библиотеки классов сторонних разработчиков, в которых конфликтуют имена. Допустим, вы работаете над крупным проектом на C# и приобрели (в виде DLL) пару библиотек классов фирм Bovary Enterprises и Karenina Software. Обе содержат класс *SuperString*, который совершенно по-разному реализован в каждой из DLL, но полезен и в той, и в другой версии. К счастью, такое дублирование — не проблема, так как обе компании следовали принципам C#: использовали пространства имен. Bovary поместила код своего класса *SuperString* в описание имени класса, подобное такому:

```
namespace BovaryEnterprises.VeryUsefulLibrary
{
    class SuperString
    {
    }
}
```

Нечто подобное проделала и Karenina:

```
namespace KareninaSoftware.HandyDandyLibrary
{
    class SuperString
    {
    }
}
```

В обоих случаях сначала идет имя компании, затем название продукта. В ваших программах, использующих эти библиотеки, можно обращаться к конкретному классу *SuperString* через полное имя:

```
BovaryEnterprises.VeryUsefulLibrary.SuperString
```

или:

```
KareninaSoftware.HandyDandyLibrary.SuperString
```

Конечно же, придется много стучать по клавишам, но такое решение определенно работает.

Использование пространств имен было бы довольно вредным явлением, если б не существовал способ уменьшить объем ввода с клавиатуры. Для этого предназначено ключевое слово *using*. Пространство имен указывается один раз в операторе *using*, затем это пространство имен можно не вводить при обращении к его классам. Вот альтернативный вариант программы «hello-world» на C#:

ConsoleHelloWithUsing.cs

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

см. след. стр.

```
class ConsoleHelloWithUsing
{
    public static void Main()
    {
        Console.WriteLine("Hello, world!");
    }
}
```

Если в проекте два различных класса *SuperString,* из затруднения выводит использование псевдонимов с ключевым словом *using:*

```
using Emma = Bovary.VeryUsefulLibrary;
using Anna = Karenina.HandyDandyLibrary;
```

Теперь к этим двум классам можно обращаться как:

```
Emma.SuperString
```

и:

```
Anna.SuperString
```

Подробнее об использовании *using* см. в руководстве по языку C#.

Более 90 пространств имен, определенных в .NET Framework, начинаются со слова *System* и 5 — со слова *Microsoft.* Наиболее важными для этой книги пространствами имен являются само *System, System.Drawing,* содержащее различные классы для работы с графикой, и *System.Windows.Forms.*

Пространства имен даже позволяют давать имена, уже задействованные в .NET Framework, собственным классам. Сама .NET Framework также повторно использует некоторые имена классов. Так, она содержит три класса *Timer,* находящиеся в пространствах имен *System.Timers, System.Threading* и *System.Windows.Forms.*

Что происходит с классами, определенными без использования пространств имен, такими как *ConsoleHelloWorld* и *ConsoleHelloWithUsing* в наших примерах? Эти имена классов входят в *глобальное пространство имен* (global namespace). Это не проблема для небольших самодостаточных программ, подобных этим. Однако, если я определю в этой книге класс, который может оказаться полезным в программе кого-либо другого, то помещу его в пространство имен *Petzold.ProgrammingWindowsWithCSharp.*

Консольный ввод-вывод

Пространства имен также играют важную роль в документировании .NET. Например, документацию по классу *Console* нужно искать в документации пространства имен *System.* Вы увидите, что *WriteLine* — не единственный метод вывода класса *Console.* Метод *Write* очень похож на *WriteLine* тем, что также осуществляет вывод на консоль. Отличие в том, что *WriteLine* завершает свой вывод возвратом каретки.

Имеется 18 описаний метода *Write* и 19 — метода *WriteLine,* каждое со своими параметрами. Такое использование нескольких версий одного и того же метода известно как *перегрузка* (overload). Компилятор обычно может определить, какой из перегруженных методов нужно вызвать программе, по количеству и типам передаваемых методу параметров.

Следующая программа демонстрирует три разных способа вывода одной и той же информации.

ConsoleAclder.cs

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

class ConsoleAdder
{
    public static void Main()
    {
        int a * 1509; '
        int b = ?44;
        int c " a + b;

        Console.Write("The sum of ");
        Console.Write(a);
        Console.Write(" and ");
        Console.Write(b);
        Console.Write(" equals ");
        Console.WriteLine(c);

        Console.WriteLine("The sum of " + a + " and " + to + " equals ° * c);

        Console.WriteLine("The sura of {0} and {1} equals {2}", a, b, c);
    }
}
```

Программа выведет:

```
The sum of 1509 and 744 equals 2253
The sum of 1509 and 744 equals 2253
The sum of 1509 and 744 equals 2253
```

C-программистов обнадежит тот факт, что C# поддерживает знакомый тип *int* и не использует оператор присваивания :=, как Algol или Pascal.

В первом способе программа для показа строки использует вызываемые по отдельности методы *Write* и *WriteLine*, каждый из которых принимает один параметр. *Write* и *WriteLine* могут принимать параметры любых типов и преобразовывать их в строку для показа.

Во втором способе применяется техника, непривычная для программистов на C, зато хорошо знакомая программистам на BASIC, — конкатенация строк при помощи знака «+». C# преобразовывает переменные в строки, объединяет все строки в единое целое и передает результат в функцию *WriteLine*. В третьем способе используется строка форматирования с тремя полями подстановки — {0}, {1} и {2} — для трех других параметров. Поля подстановки могут содержать дополнительную информацию для форматирования. Скажем, {0:C} представляет число как денежную сумму: показывается знак доллара (или другой символ валюты в зависимое-

ти от региональных параметров ОС), запятая, два десятичных знака, если число отрицательное, то строка заключается в скобки. Поле подстановки {0:X8} показывает число в шестнадцатеричном виде, при необходимости дополняя его нулями до 8 знаков. Вот некоторые примеры спецификаций формата в применении к целому числу 12 345:

Различные спецификации формата в применении к целому числу 12345

Тип форматирования	Код формата	Результат
Currency (денежные суммы)	C	Я 2345.00
	C1	$12,345.0
	C7	$12,345.0000000
Decimal (десятичный)	D	12345
	D1	12345
	D7	0012345
Exponential (экспоненциальный)	E	1.234500E+004
	E1	1.2E+004
	E7	1.2345000E+004
Fixed point (с фиксированной точкой)	F	12345.00
	F1	12345.0
	F7	12345.0000000
General (общий)	G	12345
	G1	1E4
	G7	12345
Number (числовой)	N	12,345.00
	N1	12,345.0
	N7	12,345.0000000
Percent (процент)	P	1,234,500.00
	P1	1,234,500.0
	P7	1,234,500.0000000
Hexadecimal (шестнадцатеричный)	X	3039
	X1	3039
	X7	0003039

Даже если вы не собираетесь использовать вывод на консоль при программировании для .NET, вполне вероятно, что эти спецификации формата пригодятся вам при работе с методом *String.Format*.Точно так же, как *Console.Write* и *Console.WriteLine* являются в .NET аналогами функции C *printf*, метод *Stringformat* в .NET аналогичен функции *sprintf*.

Типы данных C#

Я определил несколько целых чисел с ключевым словом *int* и использовал строки, заключенные в кавычки, так что вы уже знакомы с двумя типами данных, поддерживаемыми C#. На самом деле в C# поддерживаются восемь целочисленных типов данных:

Целочисленные типы данных C#

Количество бит	Со знаком	Без знака
8	*sbyte*	*byte*
16	*short*	*usbort*
32	*in!*	*uint*
64	*long*	*ulong*

Кроме того, в C# поддерживаются два типа данных с плавающей точкой, *float* и *double,* соответствующие стандарту ANSI/IEEE 754-1985 — *IEEEStandardfor Binary Floating-Point Arithmetic* (стандарт IEEE двоичной арифметики с плавающей точкой). Количество бит, используемое порядком и мантиссой типов *float* и *double,* таково:

Количество бит, используемое типами данных C# с плавающей точкой

Тип C#	Порядок	Мантисса	Всего бит
float	8	24	32
double	11	53	64

Кроме того, в C# поддерживается тип *decimal,* использующий 128 бит для хранения данных. В состав числа этого типа входят 96-битная мантисса и десятичный масштабирующий множитель от 0 до 28. Тип данных *decimal* обеспечивает точность около 28 десятичных знаков. Он удобен для хранения и выполнения вычислений над числами с фиксированным количеством десятичных знаков, например, денежных сумм или процентных ставок. Я подробней рассмотрю *decimal,* работу с числами и математические вычисления в C# в приложении Б.

Если вы в тексте программы на C# введете число 3.14, компилятор будет считать, что оно имеет тип *double.* Чтобы указать, что его тип *float* или *decimal,* используйте для типа *float* суффикс *f,* а для *decimal* — суффикс *m.*

Вот небольшая программа, показывающая минимальное и максимальное значения каждого из 11 числовых типов данных.

MinAndMax.cs

```
//---------------------------------------
// MinAndMax.cs © 2001 by Charles Petzold
//---------------------------------------
using System;
class MinAndMax
{
    public static void Main()
    {
        Console.WriteLine("sbyte:   {0} to {1}", sbyte.MinValue,
                                                 sbyte.MaxValue);
        Console.WriteLine("byte:    {0} to {1}", byte.MinValue,
                                                 byte.MaxValue);
        Console.WriteLine("short:   {0} to {1}", short.MinValue,
                                                 short.MaxValue);
```

см. след. стр.

```
            Console.WriteLine("ushort:  {0} to {1}",  ushort.MinValue,
                                                      ushort.MaxValue);
            Console.WriteLine("int:     {0} to {1}",  int.MinValue,
                                                      int.MaxValue);
            Console.WriteLine("uint:    {0} to {1}",  uint.MinValue,
                                                      uint.MaxValue);
            Console.WriteLine("long:    {0} to {1}",  long.MinValue,
                                                      long.MaxValue);
            Console.WriteLine("ulong:   {0} to {1}",  ulong.MinValue,
                                                      ulong.MaxValue);
            Console.WriteLine("float:   {0} to {1}",  float.MinValue,
                                                      float.MaxValue);
            Console.WriteLine("double:  {0} to {1}",  double.MinValue,
                                                      double.MaxValue);
            Console.WriteLine("decimal: {0} to {1}",  decimal.MinValue,
                                                      decimal.MaxValue);
        }
    }
```

Как вы заметили, я поставил после каждого типа данных точку и слова *MinValue* или *MaxValue*. Эти два идентификатора являются полями структуры. К концу главы все, что делает эта программа, станет для вас понятным, а пока что простс взглянем на результаты ее выполнения:

```
sbyte:    -128 to 127
byte:     0 to 255
short:    -32768 to 32767
ushort:   0 to 65535
int:      -2147483648 to 2147483647
uint:     0 to 4294967295
long:   -9223372036854775808 to 9223372036854775807
ulong:    0 to 18446744073709551615
float:    -3.402823E+38 to 3.402823E+38
double:   -1.79769313486232E+308 to 1.79769313486232E+308
decimal: -79228162514264337593543950335 to 79228162514264337593543950335
```

В C# поддерживается также тип *bool*, который может принимать только одно из двух значений: *true* или *false*, являющихся ключевыми словами C#. Результаты операций сравнения (==, !=, <, >, <= и >=) — значения типа *bool*. Можно также описать данные типа *bool* явно. Приведение типа *bool к* целым типам допускается *(true* преобразуется в 1, а *false* — в 0), но оно должно быть явным

Тип данных *char* служит для хранения одного символа, а *string* — для хранения строк из нескольких символов. Тип данных *char* отличается от целых типов, и его не следует путать с типами *sbyte* или *byte*. К тому же переменные типа *char* занимают 16 бит (но это не значит, что его можно путать с *short* или *ushort)*.

Тип *char* — 16-битный, так как C# использует кодировку Unicode[3] вместо ASCII, Вместо 7-битного представления символов в базовом варианте ASCII и 8 бит на

[3] Дополнительную информацию см. на Web-странице *http://www.unicode.org* или в The Unicode Consortium, The Unicode Standard Version 3.0 (Reading, Mass.: Addison-Wesley, 2000),

символ в расширенных наборах символов, ставших общепринятыми на компьютерах, в Unicode используется целых 16 бит для кодировки одного символа. Это позволяет отобразить все буквы, условные обозначения и другие символы всех письменностей мира, которые могут использоваться при работе с компьютером. Unicode является расширением кодировки ASCII. Его первые 128 символов совпадают с символами ASCII.

Типы данных не обязательно определять в начале метода. Как и в *C++,* можно определить типы данных в том месте метода, где они потребовались.

Переменную можно определить и сразу же инициализировать:

```
string str = "Hello, World!";
```

Как только переменной типа *string* присвоено значение, ее отдельные символы нельзя изменить, однако можно присвоить ей новое значение. Строки не заканчиваются нулем, а количество символов в переменной типа *string* можно определить так:

```
str.Length
```

Length является *свойством* типа данных *string;* концепцию свойств я опишу дальше в этой главе. В приложении В содержится подробная информация по работе со строками в C#.

Чтобы описать переменную-массив, поставьте после типа данных пустые квадратные скобки:

```
float[]arr;
```

Тип данных переменной *arr* — массив чисел с плавающей точкой, но на самом деле *arr* — это указатель. В языке C# массив является *ссылочным* типом (reference type). Это относится и к строке. Другие типы, о которых я рассказывал ранее, являются *размерными* (value types),

Значение переменной *arr* при первоначальном определении — *null.* Для выделения памяти массиву нужно воспользоваться оператором *new* и указать количество элементов в массиве:

```
arr=newfloat[3];
```

Можно также применить сочетание двух приведенных выше операторов:

```
float[]arr=newfloat[3];
```

Кроме того, при описании массива можно инициализировать его элементы:

```
float[] arr = new float[3] { 3.14f, 2.17f, 100 };
```

Количество инициализирующих значений должно совпадать с объявленным размером массива. При инициализации массива можно опустить размер:

```
float[] arr = new float[] { 3.14f, 2.17f, 100 };
```

и даже — оператор *new.*

```
float[] arr = { 3.14f, 2.17f, 100 };
```

В дальнейшем в программе можно присвоить переменной *arr* массив типа *float* другого размера:

```
arr = new float[5];
```

При этом выделяется память для хранения пяти значений типа *float,* каждое из этих значений первоначально равно 0.

Вы можете спросить: «А что случилось с первым блоком памяти, выделенным для трех значений типа *float?*» В C# нет оператора *delete.* Поскольку на исходный блок памяти больше не ссылается ни одна из переменных программы, он становится объектом *сборки мусора* (garbage collection). В какой-то момент Commor. Language Runtime освободит память, изначально выделенную массиву.

Как и в случае строк, количество элементов массива можно определить, используя выражение:

```
arr.Length;
```

Кроме того, C# позволяет создавать многомерные и *невыровненные* (jagged) массивы, являющиеся массивами массивов.

Если не нужно взаимодействие с кодом, написанным на другом языке, указатели в программах на C# требуются редко. По умолчанию параметры передаются методам *по значению.* Это означает, что метод может как угодно изменять параметр, а значение параметра в вызывающем методе не изменится. Чтобы изменить такое поведение параметров, можно использовать ключевые слова *ref* (reference — ссылка) или *out,* Например, определить метод, изменяющий передаваемую в качестве параметра переменную, можно так:

```
void AddFive(ref int i)
{
    i += 5;
)
```

А метод, присваивающий переменной значение, выглядит так:

```
void SetToFive(out int i)
{
    i = 5;
}
```

В первом примере переменной *i* необходимо присвоить значение перед вызовом *AddFive,* тогда метод *AddFive* сможет изменить ее значение. Во втором — *i* при вызове метода может не иметь никакого значения,

Важную роль в C# и .NET Framework играют перечисления. Многие константы .NET Framework определены как перечисления. Приведем пример из пространства имен *SystemIO:*

```
public enum FileAccess
{
    Read = 1,
    Write,
    ReadWrite
}
```

Перечисления всегда являются целыми типами данных, по умолчанию они имеют тип *int.* Если не указать значение явно (в данном примере для *Read* оно указано), первому элементу перечисления присваивается нулевое значение. Следующим элементам присваиваются значения в порядке возрастания,

FileAccess можно использовать при работе с несколькими классами файлового ввода-вывода. (Файловый ввод-вывод подробно рассмотрен в приложении А). Необходимо указывать имя перечисления, затем, через точку имя элемента;

```
file.Open(FileMode.CreateNew, FileAccess.ReadWrite)
```

FileMode — это еще одно перечисление класса *System.IO.* Если эти два перечисления в методе *Open* поменять местами, компилятор сообщит об ошибке. Перечисления помогают программисту избегать ошибок, связанных с константами.

Выражения и операторы

Один из самых важных справочников для С-программистов — таблица, в которой приведен порядок выполнения всех операций языка С. (Когда-то можно было купить эту таблицу, отпечатанную на футболке вверх ногами — для удобства чтения.) Аналогичная таблица C# немного отличается от нее в первых двух строках: в них присутствуют некоторые дополнительные операторы и отсутствует оператор «,» (запятая),

Я хотел бы обсудить некоторые тонкости использования двух видов операторов И и ИЛИ, так как они могут вызывать путаницу; по крайней мере, так было у меня при первом знакомстве с ними.

Заметьте, что операторы &, ^ и | называются *логическими* И, исключающим ИЛИ и ИЛИ; в С они назывались *побитовыми.* В C# логические И, исключающее ИЛИ и ИЛИ определены для целых типов данных и для типа *bool.* Для целых типов они выполняются как побитовые, точно так же, как в С. Например, результатом вычисления выражения:

```
0x03 | 0x05
```

будет 0x07. Для переменных или выражений типа *bool* результат выполнения логических операторов также имеет тип *bool.* Результатом логического оператора И является *true,* когда оба операнда — *true.* Результатом логического исключающего ИЛИ является *true,* когда один операнд — *true,* а другой — *false.* Результат логического ИЛИ — *true,* когда хотя бы один из операндов — *true,*

Порядок вычисления в C#

Тип оператора	Оператор	Ассоциативность
Простой	C) D /0 . *x++ >'++ new typeof sizeofchecked unchecked*	Слева направо
Унарный	+ ? ! ~ *++x ??x(type)*	Слева направо
Мультипликативный	•/ %	Слева направо
Аддитивный	+ '	Слева направо
Сдвиг	<< >>	Слева направо
Отношение	< > <= >= *is as*	Слева направо
Равенство	== !=	Слева направо
Логическое И	,\:	Слева направо
Логическое исключающее ИЛИ	^	Слева направо
Логическое ИЛИ	|	Слева направо
Условное И	*&£*	Слева направо

Порядок вычисления в C# *(продолжение)*

Тип оператора	Оператор	Ассоциативность
Условное ИЛИ	||	Слева направо
Условие	?:	Справа налево
Присваивание	= += -= *= /= %= <<= >>= &= ^= |=	Справа налево

Операторы && и || называются в языке C *логическими*. В C# они называются *условными* И и ИЛИ и определены только для типа данных *bool*.

C-программисты привыкли использовать && и || в конструкциях вида:

```
if (a != 0 && b >= 5)
```

Кроме того, C-программисты знают, что если результат вычисления первого выражения — *false* (т. е. если *a* равно 0), то второе выражение не вычисляется. Это важно знать, так как второе выражение может выполнять присваивание или вызывать функцию. Аналогично при использовании оператора || второе выражение не вычисляется, если результат первого выражения — *true*.

В C# операторы && и || используются точно так же, как в C. Они называются условными И и ИЛИ, так как второй операнд в них вычисляется только при необходимости.

В C# можно использовать операторы & и | так же, как и && и ||, например:

```
if (a != 0 & b >= 5)
```

При таком использовании операторов & и | в C# вычисляются оба выражения независимо от результата первого выражения.

Второй вариант оператора *if* допустим и в C, и он будет работать так же, как в C#. Однако большинство C-программистов, вероятно, написало бы такой оператор только по ошибке. Этот оператор просто выглядит для меня *неправильным* и заставляет звенеть звоночек в моей голове, так как я приучил себя рассматривать & как побитовое И, а && как логическое И. Но в C результатом отношений или логических выражений является значение типа *int*: 1, если выражение истинно, 0 — в противном случае. Поэтому побитовый оператор И также будет работать правильно.

Программист на C может сделать исходный оператор, использующий &&, немного короче, записав его:

```
if (a && b >= 5)
```

Это прекрасно работает в C, потому что C рассматривает любые ненулевые выражения как истинные. Однако в C# такая конструкция не допускается, так как оператор && определен только для типа *bool*.

Но при использовании побитового оператора И в сокращенной форме выражения программист на C столкнется с большими проблемами:

```
if (a & b >= 5)
```

Если *b* равно 7, выражение справа получит значение 1. Если *a* равно 1, 3 или любому другому нечетному числу, то результатом побитовой операций будет *true*. Если *a* — П. 2 или любое четное число, то результатом побитового И будет 0, а резуль-

татом всего выражения — *false*.Скорее всего такие результаты совершенно не то, на что рассчитывал программист. Именно поэтому С-программисты столь болезненно реагируют на побитовые операторы И и ИЛИ в логических выражениях. В C# такие операторы запрещены, так как целые числа и значения типа *bool* не могут совместно использоваться в логических операторах И, исключающее ИЛИ и ИЛИ.

В плане приведения типов C# гораздо строже, чем С. Если нужно преобразовать один тип данных в другой, а ограничения C# не позволяют этого, может пригодиться класс *Convert.* Он определен в пространстве имен *System* и содержит многие методы, среди которых, возможно, есть те, что вам подойдут. Если требуется интерфейс с существующим кодом, можно использовать класс *Marshal,* определенный в пространстве имен *System.RuntimeInteropServices.*Он содержит метод *Copy,* позволяющий передавать данные между массивами C# и областями памяти, на которые ссылаются указатели.

Условия и циклы

В C# поддерживаются многие условные операторы, операторы цикла и управления выполнением программы, применяемые в С. В этом разделе я рассмотрю операторы, содержащие ключевые слова *if, else, do, while, switch, case, defaultfor, foreach, in, break, continue* и *goto.*

Конструкции *if u else* выглядят точно так же, как в С:

```
if (a == 5)
{
    ⋮
};
else if (a < 5)
{
    ⋮
}
else
{
    ⋮
}
```

Однако в C# выражения в скобках должны иметь тип *bool.* Такое ограничение позволяет избежать общеизвестного подводного камня языка С — ошибочного использования в проверяемом выражении оператора присваивания вместо оператора сравнения:

```
if (a = 5)
```

Такой оператор вызовет в C# ошибку компиляции, и скажите за это спасибо C#.

Но, конечно, ни один компилятор не обеспечивает полной защиты от рассеянности программиста. В одной из своих первых программ на C# я определил переменную *trigger* типа *bool,* но вместо оператора

```
if (trigger)
```

я, чтобы сделать программу чуть понятнее, хотел ввести:

```
if (trigger == true)
```

Но, к сожалению, я ввел:

```
if (trigger = true)
```

Это совершенно корректный, с точки зрения C#, оператор, но, очевидно, он делает не то, что я хотел.

Кроме того, C# поддерживает операторы *do* и *while*. Можно проверять условие перед выполнением блока:

```
while (a < 5)
{
    ⋮
}
```

или после:

```
do
{
    ⋮
}
while (a < 5);
```

Здесь выражение также должно иметь тип *bool*. Во втором случае блок выполнится хотя бы раз независимо от значения *a*.

В конструкции *switch/case* языка C# есть ограничение, которого нет в C, В C допускается оператор:

```
switch (a)
{
case 3:
    b = 7;
            // В C# запрещено прохождение через case.
case 4:
    c = 3;
    break;

default:
    b = 2;
    c = 4;
    break;
}
```

Если *a* равно 3, выполнится первый оператор, затем программа «перешагнет» через *case*, в котором *a* равно 4. Может, вы именно так и задумали, но возможно, что вы просто забыли поставить оператор *break*. Чтобы избежать ситуаций такого рода, компилятор C# сообщает об ошибке. C# позволяет проходить через *case*, только если *case* не содержит операторов. А вот такая конструкция допустима в C#:

```
switch (a)
{
case 3:
case 4:
    b = 7;
    c = 3;
```

```
        break;

default:
        b = 2;
        c = 4;
        break;
}
```

Если требуется нечто более сложное, можно использовать оператор *goto,* описанный в этом разделе далее,

Одна из приятных возможностей C#: в операторе *switch* можно использовать строковую переменную, сравниваемую в операторах *case* со строковыми константами:

```
switch (strCity)
{
case "Boston":
        :
        break;

case "New York":
        :
        break;

case "San Francisco":
        :
        break;

default:
        :
        break;
}
```

Конечно, такой фрагмент кода вызовет ужас у заботящихся о производительности программистов на C или C++. Все эти сравнения строк просто *не могут* быть особо эффективными. На самом деле, благодаря технологии *интернирования строк* (string interning), заключающейся в поддержке таблицы всех уникальных строк, используемых программой, он работает гораздо быстрее, чем может показаться.

Цикл *for* выглядит в C# точно так же, как в C и C++:

```
for (i = 0; i < 100; i += 3)
{
        :
}
```

Как и в C++, в C# широко используется определение параметра цикла прямо в операторе *for*:

```
for (float f = 0; f < 10.05f; f += 0.1f)
{
        :
}
```

Удобным дополнением к этому типу цикла является оператор *foreach*, заимствованный C# из Visual Basic. Предположим, *arr* — массив значений типа *float* Если требуется показать все его элементы в одной строке, разделив их проблемами, обычно делают так:

```
for (int i = 0; i < arr.Length; i++)
    Console.Write("{0} ", arr[i]);
```

Оператор *foreach,* в котором используется ключевое слово *in,* упрощает *эту* операцию:

```
foreach (float f in arr)
    Console.Write("{0} ", f);
```

Для параметра цикла *foreach* (в данном случае переменной *f)* надо в операторе *foreach* указать тип данных; в следующем за *foreach* блоке операторов параметр цикла доступен только для чтения. Следовательно, *foreach* не годится для инициализации элементов массива:

```
int[] arr = new int[100];

foreach (int i in arr)          // Так нельзя!
    i=55;
```

Интересно, что оператор *foreach* может работать не только с массивами, но и с любыми классами, реализующими интерфейс *IEnumerable*, определенный в пространстве имен *System.Collections.* Более ста классов .NET Framework реализуют интерфейс *lEnumerable.* (Здесь я лишь коснусь интерфейсов, а в главе 8 рассмотрю их подробно.)

Оператор *break* обычно используется в конструкциях *switch/case*, а также для выхода из циклов *while, do, for* и *foreach*. Оператор *continue* служит для перехода к концу блока, выполняемого в цикле *while, do, for* или *foreach* и выполнении следующей итерации цикла (если эта итерация должна выполняться согласно условик: цикла).

И, наконец, оператор *goto*:

```
goto MyLabel;
    :
MyLabel:
```

полезен для выхода из блоков с большой вложенностью и написания запутанного кода. Кроме того, в C# поддерживается оператор *goto* в конструкции *switch/case* для перехода к другой ветви:

```
switch (a)
!
case 1:
    b = 2;
    goto case 3;

case 2:
    c = 7;
    goto default;
```

```
case 3:
    c = 5;
    break;

default:
    b = 2;
    break;
}
```

Если вместо *break* используется *goto*, то *break* в конце ветви не нужен. Эта возможность компенсирует запрет проходить через *case*.

Переходим к объектам

В большинстве традиционных процедурных языков, таких как Pascal, Fortran, BASIC, PL/I, C и COBOL, мир делится на код и данные. В основном на них пишут код для «перемалывания» данных.

На всем протяжении истории программирования программисты стремились упорядочить код и данные, особенно в больших программах. Например, взаимосвязанные функции группировались в одном исходном файле. В нем могли находиться и переменные, используемые только этими функциями и нигде более. И, конечно, общепринятым в традиционных языках средством консолидации взаимосвязанных *данных* являются структуры.

Допустим, вы разрабатываете приложение, которое будет работать с датами, в частности, вычислять день года — порядковый номер дня в году. Скажем, для 2 февраля значение дня года равно 33, а для 31 декабря — 366, если год високосный, и 365, если нет. Наверно, вы сочтете разумным описать дату как единую сущность. Например, в C данные, относящиеся к дате, можно описать в структуре с тремя полями:

```
struct Date
{
    int year;
    int month;
    int day;
};
```

Затем можно определить переменную типа *Date:*

```
struct Date today;
```

Можно обращаться к отдельным полям, поставив после имени переменной структурного типа точку и указав имя поля:

```
today.year  = 2001;
today.month = 8;
today.day   = 29;
```

Можно, наоборот, работать с данными структуры как с группой, указывая имя переменной (в данном случае *today).* Например, в C можно одним приемом определить структурную переменную и инициализировать ее:

```
struct Date birthdate = < 1953, 2, 2 } ;
```

Для написания функции вычисления дня года будет полезна небольшая функция, определяющая, каким является год — обычным или високосным:

```
int IsLeapYear(int year)
{
    return (year K 4 == 0) && ((year X 100 != 0) || (year % 400 == 0»;
}
```

Ее использует функция вычисления дня года *DayOfYear*:

```
int DayOfYear(struct Date date)
i
    static int HonthDays[12] = {  0,  31,  59,  90, 120, 151,
                                181, 212, 243, 273, 304, 334 };

    return MonthDays[date.month - 1] + date.day +
                    ((date.month > 2) && IsLeapYear(date.year));
!
```

Обратите внимание, что функция обращается к полям входной структуры, указывая через точку имя поля.

Вот как выглядит полностью работоспособная версия структуры *Date* и относящихся к ней функций на языке C:

Г CDate.c

```
//-------------------------------------
// CDate.c © 2001 by Charles Petzold
//-------------------------------------
#include <stdio.h>

struct Date
{
   .  int year;
     int month;
     int day;
ft
int IsLeapYear(int year)
{
     return (year % 4 == €} && ((year % 100 != 0) || (year % 400 == 0));
I
int DayOfYear(struct Bate date)
{
     static int MonthDays[12] = {  0,  31,  59,  90, 120. 151,
                                 181, 212, 243, 273, 304, 334 };

     return MonthDays[date.month - 1] + date.day +
                     ((date.month > 2} && IsLeapYear(date.year));
}
intffialn(void)
{
     struct Date mydate;
```

см. след. стр.

```
    'mydate.month = 8;
     mydate.day   = 29;
     mydate.year  * 2001;

     printf("Day of year = %i\n", DayOfYear(mydate));

     return 0;
>
```

Я поместил функцию *main* в конец программы, чтобы не пришлось использовать упреждающее объявление.

Программа на С имеет такой вид, потому что структуры языка С могут содержать только данные. Код и данные существуют отдельно друг от друга. Однако функции *IsLeapYear* и *DayOfMonth* тесно связаны со структурой *Date,* поскольку они работают с полями структуры *Date.* Поэтому имеет смысл объединить эти функции с самой структурой *Date.* Перемещение функций внутрь структуры превращает программу на С в программу на C++. Вот эта программа на C++:

CppDateStruct.cpp

```
//-----------------------------------------------------
// CppDateStruct.cpp © 2001 by Charles Petzold
//-----------------------------------------------------
#include <stdio.h>

struct Date
{
     int year;
     int month;
     int day;

     int IsLeapYear()
     {
          return (year % 4 == 0) && ((year X 100 E= 0} if (year X 400 == 0));
     }
     int DayOfYear()
     {
          static int MonthDays[12] = {   0,   31,   59,   90,  120,  151,
                                        181,  212,  243,  273,  304,  334 };

          return MonthDays[month - 1} + day + ((month > 2) && IsLeapYear());
     }
};
int  main(void)
{
     Sate  mydate;

     fflydat&.month=8;
     mydate.day   - 29;
     mydate.year  * 2001;
```

см. след, стр.

```
    printf("Day of year = %i\n", mydate.DayOfYear());

    return 0;
}
```

Объем кода уменьшился. *IsLeapYear* и *DayOfJYear* теперь объявлены без параметров. Они могут напрямую обращаться к полям структуры, так как являются частью той же структуры. Теперь эти функции имеют право называться *методами*.

Кроме того, в описании переменной, *mydate* функции *main* исчезло ключевое слово *struct*. Теперь тип *Date* выглядит, как обычный тип данных, а *mydate* — как переменная этого типа. На жаргоне объектно-ориентированного программирования *mydate* называется *объектом* типа *Date* или *экземпляром Date*. Иногда говорят (в основном любители громких слов), что *создается экземпляр* (instantiated) объекта *Date*.

И, наконец, самое важное. Метод *DayOfJYear* вызывается аналогично обращению к полям данных структуры. После имени объекта через точку указывается имя метода. Более тонкое изменение заключается в смещении акцента. Раньше мы обращались к функции *DayOfYear*, чтобы она обработала данные, представленные структурой *Date*. Теперь мы обращаемся к структуре *Date*, содержащей реальную календарную дату, чтобы она вычислила свой день года — *DayOfJYear*.

Сейчас мы с вами занимаемся объектно-ориентированным программированием, по крайней мере одним из его аспектов. Мы объединяем код и данные в единое целое,

Однако в самых передовых объектно-ориентированных языках элемент программы, содержащий код и данные, называется не *структурой (struct),* а *классом* (ключевое слово *class)*. Для перехода от *struct* к *class* в C++ требуется добавить всего одну строку кода, ключевое слово *public* в начале описания новоиспеченного класса *Date*.

CppDateClass.cpp

```
//-----------------------------------------------------
// CppDateClass.cpp © 2061 by Charles Petzold
//-----------------------------------------------------
#include <stdio.h>

class Date
{
public:
    int year;
    int month;
    int day;

    int IsLeapYear()
    {
        return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
    }
    int DayOfYear()
```

см. след. стр.

```
    {
        static int MonthDays[12] * {  0,  31,  59,  90, 120, 151,
                                    181, 212, 243, 273, 304, 334 };

        return MonthDays[month - 1] + day + ((month > 2) && IsLeapYear());
    }
};
int main(void)
I
    Bate mydate;

    mydate.month = 8;
    mydate.day   = 2S;
    mydate.year  = 2001;

    printf("Day of year = %i\n", mydate.DayOfYear());

    return S;
}
```

В C++ и C# классы очень похожи на структуры. И в том, и в другом языке *class* — это *не в точности* то же самое, что и *struct,* но отличия *class* от *struct* в этих двух языках разные. Я рассмотрю отличия *class* и *struct* в C# в этой главе и подробнее — в главе 3. Б C++ все поля и методы *struct* по умолчанию открытые (public), т. е. к ним можно обращаться из-за пределов структуры. Например, чтобы я мог обратиться к полям и методам *Date* из функции *main,* они должны быть открытыми. В классах языка C++ все поля и методы по умолчанию закрытые (private), и, чтобы сделать их открытыми, нужно использовать ключевое слово *public.*

Этот пример я выполнял в C++, а не в C#, так как C++ разрабатывался как язык, совместимый с C и позволяющий плавно перейти из мира C в мир C++. Теперь пора выполнить этот пример на C#.

Программирование в тональности до-диез

На самом деле версия этой программы на C# не так уж сильно отличается от версии на C++.

CsDateClass.cs

```
//-----------------------------------------
// CsDateClass.cs © 2081 by Charles Petzold
//-----------------------------------------
using System;

class CsDateClass
{
    public static void Main()
    {
```

см. след. стр.

```
            Date mydate * flaw Date();

            mydate.month = 8;
            mydate.day   = 29;
            mydate.year  = 2001;

            Console.WriteLine("Day of year * {0}", mydate.DayOfYear());
        }
    }
    class Date
    {
        public int year;
        public int month;
        public int day;

        public static bool IsLeapYear(int year)
        {
            return (year X 4 == 0) SA ((year X 106 != 0) || (year % 400 == 0»;
        }

        public int DayOfYear()
        {
            int[] MonthDays = new int[] {  0,  31,  S9,  SO,  120,  151,
                                          181,  212,  243,  273,  304,  334 };

            return MonthDays[month - 11 + day +
                             (month > 2 && IsLeapYear(year) ? 1 : 0);
        }
    }
```

Одно из внесенных мною изменений в том, что метод *Main,* содержащийся теперь в собственном классе, я поместил в начало программы, а класс *Date* — в конец программы. Я был вправе это сделать, так как в C# не требуется упреждающее объявление.

В программе на C++ я определял объект *Date* так:

```
Date mydate;
```

В C* нужна конструкция:

```
Date mydate = new Date();
```

Как и при определении массива, ключевое слово *new* служит для выделения памяти новому объекту типа *Date.* (О скобках после *Date я* расскажу ниже.)

Еще одно изменение в том, что в программе на C# ключевое слово *public* требуется перед каждым полем и методом, к которому обращаются вне класса. Ключевое слово *public* называется *модификатором доступа* (access modifier), так как показывает, как можно обращаться к полям и методам. Альтернативами *public* являются *private* и *protected,* которые я рассмотрю дальше в этой главе.

Заметьте, что метод *IsLeapYear* возвращает значение типа *bool.* В методе *DayOfYear я* использовал условный оператор (?:), чтобы получить значение 1, прибавляемое ко дню високосного года. Я мог бы также привести выражение *bool к* типу *int.*

Давайте немного поговорим на жаргоне. *Date* — это *класс* (class). Класс *Date* содержит пять *членов* (member). Три члена — *year, month* и *day* — содержат данные и называются *полями* (fields). Остальные два члена содержат код и называются *методами* (methods). Переменная *mydate* является *объектом* (object) типа *Date*. Кроме того, ее называют *экземпляром* (instance) класса *Date*.

Статические методы

Переписывая программу с C++ на C#, я внес в нее еще одно изменение: добавил модификатор *static* в определение *IsLeapYear* и добавил в этот метод параметр *year*. Это изменение не необходимо: удаление у метода *IsLeapYear* ключевого слова *static* и параметра не повлияет на работу программы.

Но модификатор *static* имеет настолько большое значение в C# и ,NET, что я не хотел откладывать его рассмотрение ни на секунду.

На всем протяжении этой главы я отображал текст на консоли, вызывая метод *WriteLine* класса *Console*:

```
Console.WriteLine(...);
```

Однако при вызове метода *DayOfYear* указывается не класс (в данном случае — *Date),* а *mydate* — объект типа *Date*:

```
mydate.DayOfYear();
```

Уловили разницу? В первом случае указан класс *Console*; во втором случае — объект *mydate*.

Разница как раз и заключается в модификаторе *static*. Метод *WriteLine* в классе *Console* определен как *static*:

```
public static void WriteLine(string value)
```

Статические методы относятся к самому классу, а не к его объектам. Для вызова метода, определенного как *static*, необходимо указать перед ним имя класса. Для вызова метода, не определенного как *static,* нужно указать перед ним имя объекта — экземпляра класса, в котором определен этот метод.

Это различие относится и к членам класса, содержащим данные. Любые поля, определенные как *static,* имеют одно и то же значение для всех экземпляров класса. Вне определения класса к членам, содержащим данные, необходимо обращаться, указывая имя класса, а не имя объекта этого класса. Поля *MinValue* и *MaxValue,* которые я ранее использовал в программе MinAndMax, — статические.

К чему приводит определение *IsLeapYear* как *static*? Во-первых, нельзя вызывать *IsLeapYear*, указывая перед ним экземпляр класса *Date:*

```
mydate.IsLeapYear(1997)     // Не будет работать!
```

Необходимо вызывать *IsLeapYear*, указывая имя класса:

```
Date.IsLeapYear(1997)
```

Внутри определения класса, например в методе *DayOfYear*, перед *IsLeapYear* вообще не нужно ничего ставить. Еще одно следствие: *IsLeapYear* должен иметь параметр — проверяемый год. Преимущество определения *IsLeapYear* как *static* в

том, что для его использования не надо создавать экземпляр *Date*. Аналогично для статических методов, определенных в классе *Console*, не нужно создавать экземпляр класса *Console*. На самом деле вы не сможете создать экземпляр *Console*, но даже если бы это удалось, то все равно было бы бесполезно, так как в классе *Console* нет нестатических методов.

Статический метод не может обращаться к нестатическим методам или полям своего класса. Дело в том, что нестатические поля различны для разных экземпляров класса, а нестатические методы возвращают различные значения для разных экземпляров класса. Когда вы ищете что-нибудь в справочнике по .NET Framework, всегда обращайте внимание, определено ли оно как *static* или нет. Это крайне важно. Я также постараюсь быть в этой книге очень аккуратным и обращать внимание, когда что-либо определено как *static*.

Поля также могут определяться как *static*, и тогда они совместно используются всеми экземплярами класса. Статические поля хорошо подходят для массивов, которые требуется инициализировать постоянными значениями, как в случае массива *MonthDays* в программе CsDateClass. Как видно из этой программы, массив повторно инициализируется при каждом вызове метода *DayOfYear*.

Обработка исключений

Разные ОС, графические среды, библиотеки и функции по-разному сообщают об ошибках. Одни возвращают логические значения, другие — коды ошибок, третьи — значение NULL, четвертые подают звуковой сигнал, пятые приводят к аварийному сбою системы.

В C# и .NET Framework для сообщения об ошибках предпринята попытка унифицировать использование технологии *структурной обработки исключений* (structured exception handling).

Чтобы изучить ее, начнем с того, что в программе CsDateClass присвоим полю *month* объекта *Date* значение 13:

```
mydate.month = 13;
```

Перекомпилируйте и запустите программу. Если откроется диалоговое окно выбора отладчика, щелкните No. Вы получите в командной строке сообщение:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the
bounds of the array.
   at Date.DayOfYear()
   at CsDateClass.Main()
```

При компиляции с установкой режима отладки информация будет конкретнее: вы увидите номера строк исходного кода. Но в любом случае выполнение программы завершится преждевременно.

Заметьте, что сообщение правильно: индекс массива *MonthDays*, действительно, вне границ массива. В языке C проблемы такого рода приводили к другим видам ошибок, например, к переполнению стека. C# выполняет проверку корректности индекса перед доступом к элементу массива по его индексу. Программа реагирует на неправильный индекс, инициируя простой процесс, который называется *выбросом* (throwing) или *генерацией* исключения.

В C# есть очень полезная возможность — сделать так, чтобы программы сами определяли возникновение исключений и обрабатывали их, Когда программа проверяет возникновение исключения, говорят, что она *перехватывает* (catch) исключение. Для перехвата исключения операторы, которые могут вызвать исключение, помещаются в блок *try,* а операторы, выполняемые при обработке исключения, — в блок *catch.* Например, чтобы учитывать возможность указания неправильной даты, в программу CsDateClass можно поместить такой код:

```
try
{
     Console.WriteLine("Day of year = {0}", mydate.DayOfYear());
}
catch (Exception exc)
{
     Console.WriteLine(exc);
}
```

Класс *Exception* определен в пространстве имен *System,* а *exc* — объект типа *Exception,* определенный в программе. Этот объект получает информацию об исключении. В данном примере я просто передал *exc* методу *Console.WriteLine* в качестве параметра. При задании неправильного месяца этот оператор покажет точно такой же блок текста с описанием ошибки, как тот, что я показывал ранее. Но теперь программа не завершается преждевременно, и можно обрабатывать ошибку более изящно.

Даже одна строка кода может вызвать исключения разных типов. Поэтому можно определить несколько разных блоков *catch:*

```
Try
{
     ⋮
}
catch (NullReferenceException exc)
{
     ⋮
}
catch (ArgumentOutOfRangeException exc)
{
     ⋮
}
catch (Exception exc)
{
     ⋮
}
```

Обратите внимание, что исключение самого общего типа находится в конце.

Можно также добавить блок *finally:*

```
Try
{
     ⋮
}
```

```
catch (Exception exc)
{
    ⋮
}
finally
{
    ⋮
}
```

Независимо от того, возникнет исключение или нет, код *блока finally* все равно выполнится: после кода блока *catch* (если исключение возникнет) или после кода блока *try* (если нет). В *блок finally* можно поместить, например, код освобождения ресурсов.

Вы можете спросить: «А зачем блок *finally?* Разве нельзя просто поместить код освобождения ресурсов после блока *catch?*» Можно. Но вы можете закончить блок *try* или *catch* оператором *goto.* В этом случае код блока *finally* все равно выполнится перед выполнением оператора *goto.*

Можно также пропустить блок *catch*:

```
try
{
    ⋮
}
finally
{
    ⋮
}
```

В этом случае вы увидите окно с информацией об отладчике и текстовым описанием исключения (тем же, что мы показывали при помощи *Console.WriteLine*), затем выполнится код блока *finally,* и программа завершится нормально.

Выброс исключений

В программе работы с датами CsDateClass меня беспокоит то, что мы не добрались до истинной причины проблем. Метод *DayOfYear* выбрасывает исключение из-за того, что индекс массива *MonthDays* находится вне границ. Но настоящая проблема возникает в программе раньше и связана с оператором, который я предложил добавить в программу:

```
mydate.month = 13;
```

Сразу после его выполнения мы имеем дело с объектом *Date,* содержащим неправильную дату. В этом и заключается настоящая проблема. Просто получилось так, что *DayOJYear* оказался первым методом, который отрицательно реагирует на эту проблему. Но допустим, что вы введете в программу другой оператор:

```
mydate.day = 47;
```

Метод *DayOJYear* выполнится без сообщений об ошибках и посчитает результат, несмотря на некорректность даты.

Есть ли способ встроить в класс защиту от присваивания полям некорректных значений программой, использующей этот класс? Проще всего описать поля как *private,* а не *public*.

```
private int year;
private int month;
private int day;
```

Модификатор *private* делает эти три поля доступными только методам, находящимся внутри определения класса *Date*. На самом деле в C# атрибут *private* используется по умолчанию, так что для внесения этих изменений достаточно просто удалить атрибут *public*.

```
int year;
int month;
int day;
```

Конечно, такое изменение порождает уже другую проблему. Как программа, использующая класс *Date,* должна устанавливать значения года, месяца и дня?

Одно из решений, которые могут прийти в голову, — определить методы класса *Date*, предназначенные специально для присваивания значений этих трех полей и для получения ранее присвоенных значений полей. Вот, например, два простых метода для присваивания и получения значения закрытого поля *month:*

```
public void SetMonth (int month)
{
    this.month = month;
}
public int GetMonth ()
{
    return month;
}
```

Заметьте: оба метода определены как *public*. Кроме того, я дал параметру метода *SetMonth* имя, совпадающее с именем поля! При этом перед именем поля нужно указывать слово *this* и точку. Внутри класса ключевое слово *this* означает экземпляр класса, вызывающего метод. В статических методах ключевое слово *this* не допускается.

Вот версия *SetMonth,* проверяющая правильность значения месяца:

```
public void SetMonth (int month)
{
    if (month >= 1 && month <= 12)
        this.month = month;
    else
        throw new ArgumentOutOfRangeException("Month");
}
```

В этом примере показан синтаксис выброса исключения. Я выбрал исключение *ArgumentOutOfRangeException*, так как оно наиболее адекватно идентифицирует проблему. Ключевое слово *new* используется для создания нового объекта типа *ArgumentOutOfRangeException*. Этот объект — то, что блок *catch* получает в ка-

честве параметра. Параметром *ArgumentOutOfRangeException* является текстовая строка, идентифицирующая вызвавший проблему параметр. При выводе сообщения об исключении эта строка показывается вместе с другой информацией об ошибке.

В C# имеется достойная альтернатива методам *Get* и *Set*. Когда вы собираетесь писать методы, начинающиеся со слов *Get* и *Set,* более того, когда вы собираетесь писать любой метод, возвращающий информацию об объекте и не требующий параметров, вам следует подумать о такой возможности C#, как *свойство* (property),

Получение и присваивание значений свойств

Как вы уже знаете, в классах C# могут присутствовать члены, содержащие данные, называемые *полями,* и члены, содержащие код, — *методы.* Кроме того, в классах C# может быть еще один вид членов, содержащих код. Эти члены называются *свойствами* и играют крайне важную роль в .NET Framework.

Свойства как бы размывают различия между кодом и данными. Для использующих класс программ свойства выглядят как поля с данными и часто именно так и рассматриваются. Однако внутри класса свойства, конечно же, являются кодом. Во многих программах открытое свойство обеспечивает другим классам доступ к закрытому полю класса. Свойство имеет преимущество перед полем, так как в него можно поместить проверку допустимости значения.

Некоторые программисты на C# (в том числе и я) дают закрытым полям имена, начинающиеся со строчных букв, а открытым свойствам — имена, начинающиеся с заглавных букв. Ниже приведено простейшее определение свойства *Month* обеспечивающего доступ к закрытому полю *month:*

```
public int Month
{
     set
     {
          month = value;
     }
     get
     {
          return month;
     }
}
```

Программа, использующая класс с таким свойством, обращается к свойству точно так же, как если бы это было поле:

```
mydate.Month = 7;
```

ИЛИ:

```
Console.WriteLine(mydate.Month);
```

ИЛИ:

```
mydate.Month += 2;
```

В последнем примере значение *Month* увеличивается на 2. Смотрите, насколько синтаксис этого оператора прозрачнее, чем синтаксис аналогичного оператора, использующего методы *SetMonth* и *GetMonth*, с которыми мы работали ранее:

```
mydate.SetMonth(mydate.GetMonth() + 2);        // Хорошо, что мы от этого
                                               // избавились!
```

Рассмотрим определение этого свойства подробно. Ключевое *слово public* означает, что это свойство доступно вне класса. Тип данных *int* — что свойство является 32-битным целым. Само свойство имеет имя *Month*.

Внутри тела свойства содержатся два *аксессора* (accessors) — *set* и *get*. Не обязательно использовать оба. У многих свойств есть только открытый *get*-аксессор, а *set* или вообще не определен, или определен *как private*. Такие свойства называются *неизменяемыми* (read-only). Можно также создать свойство с *set*-аксессором и без *get*-аксессора, но это требуется гораздо реже.

Внутри определения *set*-аксессора специальное слово *value* служит для указания значения, присваиваемого свойству оператором:

```
mydate.Month = 7;
```

Аксессор *get* всегда должен содержать оператор *return*, чтобы вернуть значение программе, использующей свойство.

Следующая программа использует свойства *Year*, *Month* и *Day* и выполняет в *set*-аксессорах проверку допустимости значений.

CsDateProperties.cs

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

class CsDateProperties
{
     public static void Main()
     {
          Bats mydate = new Date();

          try
          {
               mydate.Month = 8;
               mydate.Day   = 29;
               mydate.Year  = 2001;

               Console.WriteLine("Day of year = {0}", mydate.DayOfYear);
          }
          catch (Exception exc)
          {
               Console.WriteLine(exc);
          }
     }
```

см. след. стр.

```
}
class Date
{
                                                        // Поля
    int year; :
    int month;
    int day;
    static int[] MonthDays = new int[] {   Q,  31,  59,  90, 120, 151,
                                         181, 212, 243, 273, 304, 334 };

                                                        // Свойства
    public int Year
    {
        set
        {
            if (value < 1600)
                throw new ArgumentOutOfRangeException("Year");
            else
                year = value;
        }
        get
        {
            return _ear;
        }
    }
    public int Month
    I
        set
        {
            if (value < 1 I! value > 12)
                throw new ArgumentOutOfRangeException("Month");
            else
                month = value;
        }
        get
        {
            return month;
        }
    }
    public int Bay
    {
        set
        {
            if (value < 1 || value > 31)
                throw new ArgumentOutOfRangeException("Day");
            else
                day * value;
        }
        get
```

см. след. стр.

```
            {
                return day;
            }
        }
        public int DayOfYear
        {
            get
            {
                return MonthDays[month - 1] + day +
                                (month > 2 && IsLeapYear(year) ? 1 : 0);
            }
        }
                                                        - // Метод
        public static bool IsLeapYear(int year)
        {
            return (year % 4 == 0) && ((year % 100 != 0) || (year K 400 == 0));
        }
}
```

Я оставил код в блоках *try* и *catch*, так что вы можете поэкспериментировать с некорректными датами. Кстати, для свойства *Year* я установил минимальное значение 1600, так как для более ранних дат метод *IsLeapYear* не имеет особого смысла, Осталась нерешенной одна проблема: метод не проверяет согласованность полей между собой. Например, можно задать дату 31 февраля. Такая проверка согласованности наложила бы ограничения на порядок присвоения свойств, так что я обойдусь без нее.

Кроме того, я преобразовал *DayOfYear* из метода в неизменяемое свойство, потому что значение дня года больше похоже на свойство даты, чем на метод. Иногда бывает сложно определить, что использовать — метод или свойство? Единственное очевидное правило: есть параметр — используй метод,

Конструкторы

Снова вернемся к версии программы на С. Я уже упоминал, что при определении структурной переменной можно инициализировать поля структуры:

```
struct Date birthdate = { 1953, 2, 2 } ;
```

Однако я не рассматривал, можно ли применять этот способ в разных версиях С, Структуры или классы языка C++ можно инициализировать таким образом, но тогда инициализация будет зависеть от количества полей в *struct или class* и порядка, в котором расположены эти поля, так что это не очень хорошая идея. В C# такая инициализация не допускается. Но было бы очень хорошо, если бы в C# было *нечто,* позволяющее решать подобные задачи.

Еще один момент. В предыдущей версии программы на C# мы реализовали проверку допусгимости значений во всех *set*-аксессорах свойств класса. Однако ситуация, когда класс содержит некорректную дату, по-прежнему возможна — это ситуация, когда объект только что создан:

```
Date mydate = new Date();
```

Можно решить эти две проблемы при помощи средства, называемого *конструктором* (constructor). Конструктор — метод класса, выполняемый при создании объекта этого класса. Как видите, после слова *new* в конструкции:

```
Date mydate = new Date();
```

идет нечто, похожее на вызов метода без параметров. И это так! Это вызов конструктора класса *Date* по умолчанию. У каждого класса есть конструктор по умолчанию, существующий независимо от того, определен он явно или нет. Но если явно определить конструктор класса *Date* по умолчанию, то можно добиться, что объект *Date* всегда будет содержать допустимую дату.

Можно также определить конструкторы с одним или несколькими параметрами. Например, в классе *Date* можно определить конструктор с тремя параметрами, который будет инициализировать объект *Date* конкретной датой. Этот конструктор позволит создавать объект *Date* таким образом:

```
Date birthdate = new Date(1953, 2, 2);
```

Внутри класса конструктор выглядит как метод за исключением того, что имеет то же имя, что и класс, в котором определен, и того, что в конструкторе не определяется тип возвращаемого значения. Если указать в конструкторе тип возвращаемого значения или определить другой метод, не указав тип возвращаемого значения, компилятор сообщит об ошибке. Это правильно; ведь если при определении конструктора имя класса указано неверно, вы об этом узнаете при компиляции.

Вот простой пример конструктора, параметры которого определяют дату:

```
public Date(int year, int month, int day)
{
    this.year  = year;
    this.month = month;
    this.day   = day;
}
```

Но в нем не используется реализованная в свойствах проверка ошибок. Правильнее присвоить значения свойствам, а не полям:

```
public Date(int year, int month, int day)
{
    Year  = year;
    Month = month;
    Day   = day;
}
```

Но можно пойти еще дальше. Можно выполнить проверку совместимости трех параметров конструктора.

А как быть с конструктором по умолчанию? Принято определять конструкторы классов по умолчанию, присваивающие объектам нулевые значения или значения, более-менее эквивалентные нулевым. Для класса *Date* таким значением, вероятно, будет 1 января 1600 года, так как это самая ранняя допустимая дата. Вот новая версия программы:

CsDateConstructors.cs

```
//-----------------------------------------------
// CsDateConstructors.cs в 2001 by Charles Petzold
//-----------------------------------------------
using System;

class CsDateConstructors
{
     public static void Main()
     {
          try
          {
               Bate mydate = new Date(2001, 8, 29);

               Console.WriteLine("Day of year - " + mydate.DayOfYear);
          }
          catch (Exception exc)
          {
               Console.WriteLine(exc);
          }
     }
}
class Date
{
                                                          // Поля
     int year; .
     int month;
     int day;
     static int[] MonthDays = new int[] {   0,   31,   59,   90,  120,  151,
                                          181,  212,  243,  273,  304,  334 };

                                                          // Конструкторы
     public Date()
     {
          Year = 1600;
          Month * 1;
          Day = 1;
     }
     public Date(int year, int month, int day)
     {
          if ( (month == 2 &&  IsLeapYear(year) && day > 29) ||
               (month == 2 && !IsLeapYear(year) && day > 28) i [
              ((month == 4  | | month == 6 ||
                            month == 8  ||  month == 11) && day > 30»
          {
               throw new ArgumentOutOfRangeException("Day");
          >
          else
          {
```

см. след. стр.

```
            Year  = year;
            Month = month;
            Day   = day;
        }
    }
                                                        // Свойства
    public int Year
    {
        set
        {
            if (value < 1600)
                throw new ArgumentOutOfRangeException("Year");
            else
                year = value;
        }
        get
        {
            return year;
        }
    }
    pyblic int Month
    {
        set
        {
            if (value < 1 || value > 12)
                throw new ArgumentOutOfRangeException("Month");
            else
                month = value;
        }
        get
        {
            return month;
        }
    }
    public int Day
    {
        set
        {
            if (value < 1 || value > 31)
                throw new ArgumentOutOfRangeException("Day");
            else
                day - value;
        }
        get
        {
            return day;
        }
    }
    public int DayOfYear
```

См. ***след. стр.***

```
    {
        get
        {
            return MonthDays[month - 1} + day +
                                (month > 2 && IsLeapYear(year) ? 1 : 0);
        }
    1
                                                        // Метод
    public static bool IsLeapYear(int year)
    {
        return (year % 4 == 0) && ((year X 100 != 0) || (year it 400 == 0));
    }
}
```

Экземпляры и наследование

Вполне возможно, что по прошествии некоторого времени пользования классом вы подумаете: «Этот класс очень хорош, но лучше, если б...» Если у вас есть исходный код класса, можете просто открыть его в редакторе, добавить новый метод, перекомпилировать и начать использовать. Но у вас может и не быть исходного кода. Возможно, что вам доступна только скомпилированная версия класса в виде DLL.

Может случиться и так: вы хотите, чтобы некоторые функции класса выполнялись иначе. Но класс уже используется в существующем виде другими приложениями и вполне их устраивает. Изменения требуются только для одного нового приложения, и вы предпочли бы не «засорять» код исходной версии.

В объектно-ориентированных языках, подобных C#, реализована возможность *наследования* (inheritance). Можно определить новый класс, основанный на существующем классе. Говорят, что вы *наследуете* (inherit) класс от существующего класса или создаете *подкласс* (subclass) существующего класса. В новом классе должны содержаться только новые данные и код. Все классы C# и .NET Framework являются наследниками класса *Object* или классов-наследников *Object.* Говорят также, что все классы в конечном счете являются *производными* от *Object.*

Давайте создадим новый класс *DatePlus,* унаследованный от *Date, DatePlus* будет обладать новым свойством *DaysSince1600.* Благодаря наличию этого свойства, можно сделать, чтобы *DatePlus* вычислял разницу в днях между двумя датами.

Вот программа, в которой определен класс *DatePlus,*

CsDateInheritance.cs

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

class CsDateInheritance
{
    public static void Main()
```

см. след. стр.

```
    {
        DatePlus birth = new DatePlus(1953, 2, 2);
        DatePlus today = new DatePlus(2001, 8, 29);

        Console.WriteLine("Birthday = {0}", birth);
        Console.WriteLine("Today - " + today);
        Console.WriteLine("Days since birthday = {0}", today - birth);
    }
}
class OatePlus: Date
{
    public DatePlus() {}
    public DatePlus(int year, int month, int day): base(year, month, day) O

    public int DaysSince1600
    {
        get
        {
            return 365 - (Year - 1606) +
                   (Year - 1597) / 4 -
                   (Year - 1601) / 100 *
                   (Year - 1601) / 400 + DayOfYear;
        V
    }
    public override string ToString()
    {
        string[] str = { "Jan", "Feb", "Har", "Apr", "May", "Jun",
                         "Jul", "Aug", "Sep", "Oct", "Nov", "Bee" };

        return String.Format("{0} {1} {2}", Day, str[Month - 1], Year);
    }
    public static int operator -(DatePlus date1, DatePlus date2)
    Л
        return date1.DaysSince1600 - date2.DaysSince1600;
    }
}
```

Эту программу необходимо компилировать вместе с файлом CsDateConstructors.cs — самой последней реализацией класса *Date*. Так как имеется два класса, содержащих метод *Main,* нужно указать компилятору, какой из классов, содержащих метод *Main,* использовать для входа в программу.

При компиляции в командной строке надо ввести:

```
csc CsDateConstructors.cs CsDateInheritance.cs /main:CsDateInheritance
```

При этом нужно быть внимательным к регистру букв. Имена файлов можно вводить в любом регистре, но параметр */main* обращается к классу, и регистр символов этого параметра должен соответствовать имени класса, определенному в файле. В случае Visual C# .NET нужно добавить CsDateConstructors.cs в проект CsDateInheritance. Для этого выберите в меню Project | Add Existing Item. При выборе файла

CsDateConstructors.cs в диалоговом окне Add Existing Item щелкните стрелку рядом с кнопкой Open и выберите Link File. Этот параметр позволяет избежать создания копии файла CsDateConstructors.cs, а также проблем, возникающих в случае, когда версия одной из копий этого файла изменилась, а другую при этом забыли обновить.

Обратите внимание на первую строку определения класса *DatePlus*:

```
class DatePlus: Date
```

Это означает, что *DatePlus* наследуется от *Date*. Классу *DatePlus* не нужно выполнять в конструкторах специальных действий. Поэтому для него определен конструктор по умолчанию с пустым телом:

```
public DatePlus() {}
```

При создании экземпляра класса вызываются конструкторы по умолчанию всех объектов, от которых унаследован класс, начиная с конструктора по умолчанию класса *Object* и кончая конструктором по умолчанию класса объекта, который вы создаете.

Это правило неприменимо к конструкторам не по умолчанию. Конструктор с тремя параметрами не выполняет никаких специальных действий в *DatePlus,* но нужно определить его в классе и явно вызвать конструктор *базового* (base) класса, от которого наследуется *DatePlus,* — класса *Date*. Определение конструктора не по умолчанию имеет такой синтаксис:

```
public DatePlus(int year, int month, int day): base(year, month, day) {}
```

Так как конструктор не делает в классе *DatePlus* ничего специфического, его тело пусто.

В классе *DatePlus,* кроме свойства *DaysSince1600,* реализованы две изящные возможности. Во-первых, в *DatePlus* определен оператор вычитания (-) для объектов этого класса, Это называется *перегрузкой* (overloading) *оператора*. Обычно оператор минус определяется только для чисел, но здесь он хорошо подходит и для работы с датами. Тело этого перегруженного оператора довольно простое: в нем просто вычитаются *DaysSince1600* двух дат.

Например, если определить два объекта *DatePlus:*

```
DatePlus birth = new DatePlus(1953, 2, 2);
DatePlus today = new DatePlus(2001, 8, 29);
```

то можно посчитать разницу в днях между этими датами при помощи выражения

```
today - birth
```

Заметьте: я не реализовал в этом классе перегруженный оператор сложения. Не имеет смысла складывать друг с другом две даты. Однако, я мог бы реализовать сложение даты и числа, результатом которого является новая дата. Но тогда потребовалось бы писать код, чтобы преобразовать новое значение *DaysSince1600* в дату. Зато довольно легко реализовать операторы сравнения (<, >, <= и >=),

Во-вторых, я уже упоминал, что все объекты в конечном счете наследуются от *Object.* В классе *Object* реализован метод *ToString,* предназначенный для преобразования объекта в читабельную текстовую строку. Мы уже использовали *ToString.*

При конкатенации числовой переменной с текстовой строкой автоматически вызывается метод *ToString* этой переменной. При передаче объекта в метод *Console.WriteLine* также вызывается метод *ToString* объекта.

Но по умолчанию метод *ToString* класса *Object* возвращает имя класса, например строку «DatePlus». Ничего страшного: ведь любой класс, происходящий от *Object* (т. е. любой класс C#) может *переопределить* (override) метод *ToString класса Object* своим собственным. В классе *DatePlus* реализован собственный метод *ToString*, использующий статический метод *String.Format* для преобразования даты в текстовую строку. Стало возможным вызвать метод *Console.WriteLine*, указав в качестве параметра объект *DatePlus*, и показать форматированную дату. Вывод программы CsDateInheritance выглядит так

```
Birthday = 2 Feb 1953
Today = 29 Aug 2001
Days since birthday = 17740
```

Теперь мы готовы более подробно рассмотреть модификаторы доступа. Если поле, свойство или метод определены как *private*, они видимы и доступны только внутри класса, Если поле, свойство или метод определены *как public*, они видимы и доступны в других классах. Если поле, свойство или метод определены как *protected*, они видимы и доступны только внутри класса и любого класса, унаследованного от этого класса.

Метод *ToString* класса *Object* определен с модификатором *virtual*. Метод, определенный как *virtual*, может переопределяться наследниками класса. При переопределении метода используется модификатор *override*, показывающий, что метод заменяется собственной версией, реализованной в этом классе. Модификатор *override* необходим, чтобы нельзя было ошибиться, случайно переопределив виртуальный метод, когда этого не планировалось.

Кроме того, в классах можно переопределять методы, не объявленные как *virtual*. В этом случае новый метод должен содержать модификатор *new*.

Кроме *ToString*, класс *Object* содержит несколько других методов, в том числе *GetType*. *GetType* возвращает объект типа *Type*. *Type* — это класс, определенный в пространстве имен *System*, позволяющий получить информацию об объекте, в частности, о его методах, свойствах и полях. Оператор C# *typeof* также возвращает объект типа *Type*. Отличие между ними в том, что *GetType* применяется к объекту, а *typeof* — *к* классу. В методе *Main* программы CsDataInheritance результатом выражения:

```
today.GetType() == typeof(DatePlus)
```

будет *true*.

Общая картина

Документация библиотек классов в .NET Framework упорядочена по пространствам имен. Каждое пространство имен логически объединяет классы и другие элементы и реализуется в конкретной DLL.

В каждом из пространств имен можно встретить элементы пяти видов. Только эти пять видов элементов определяются в C# на внешнем уровне:

- *class* (класс), с которым вы уже познакомились;
- *struct* (структура), во многом похожая на класс;
- *interface* (интерфейс) — аналогичен *class* и *struct*, но в нем определяются только сами методы, а их тела не определяются (пример интерфейса см. в главе 8);
- *enumeration* (перечисление) — список констант с заранее определенными целыми значениями;
- *delegate* (делегат) — прототип вызова метода.

Классы и структуры выглядят в C# очень похожими. Однако *class* является *ссылочным типом*. Это значит, что объект на самом деле служит указателем на выделенный ему блок памяти, А *struct* является *размерным типом*, более похожим на обычную числовую переменную. Подробнее я рассмотрю различия между ними в главе 3. О делегатах я расскажу в главе 2: они обычно используются вместе с *событиями* (event),

Одни классы .NET Framework содержат статические методы и свойства, к которым обращаются, указывая имя класса и имя метода или свойства. Чтобы использовать другие классы, нужно создавать их экземпляры в приложениях Windows Forms. Третьи классы .NET Framework наследуются вашими приложениями.

В состав классов и структур могут входить такие члены:

- *поля* — объекты определенных типов;
- *конструкторы,* выполняемые при создании объекта;
- *свойства* — блоки кода с аксессорами *set* и *get;*
- *методы* — функции, принимающие аргументы и возвращающие значения;
- *операторы,* реализующие стандартные операции, такие как + и –, определяемые для объекта или *приведения типов* (cast);
- *индексаторы* (indexer), позволяющие обращаться к объекту как к массиву;
- *события,* о которых я расскажу в главе 2;
- другие встроенные классы, структуры, интерфейсы, перечисления и делегаты.

Ранее я рассмотрел числовые и строковые типы, поддерживаемые языком C#. Все основные типы C# реализованы как классы или структуры в пространстве имен *System*. Например, тип данных *int*, является псевдонимом структуры *Int16*. Вместо определения переменной типа *int:*

```
int a = 55;
```

можно использовать:

```
System.Int16 a = 55;
```

Эти два оператора функционально идентичны. По этой же причине в одних случаях можно видеть строки C#, определенные так:

```
string str = "Hello, world!";
```

а в других — при определении арок тип данных *String* пишется с прописной буквы:

```
String str = "Hello, world!";
```

Использование и прописных, и строчных букв в этих операторах не означает, что C# иногда бывает нечувствительным к регистру. *String* с заглавной буквы —

это класс *String* пространства имен *System*. Если вы не указали оператор *using для* пространства имен *System* и хотите использовать *String* вместо *string,* придется ввести:

```
System.String str = "Hello, world!";
```

Вот типы C#, соответствующие классам и структурам пространства имен *System*

Псевдонимы типов данных C#

Со знаком		Без знака	
Тип .NET	Псевдоним C#	Тип .NET	Псевдоним C#
System.Object	*object*	*System.Enum*	*enum*
System.String	*string*	*System.Char*	*char*
System.SByte	*sbyte*	*System.Byte*	*byte*
System.Int16	*short*	*System.UInt16*	*ushort*
System.Int32	*int*	*System.UInt32*	*uint*
System.Int64	*long*	*System.UInt64*	*ulong*
System.Single	*float*	*System.Double*	*double*
System.Decimal	decimal	System.Boolean	*bool*

Поскольку базовые типы являются классами и структурами, у них могут быть поля, методы и свойства. Именно поэтому свойство *Length* можно использовать для получения количества символов в объекте *string,* а числовые типы данных имеют поля *MinValue* и *MaxValue*. Процедуры и методы, применяемые для поддержки массивов, реализованы в классе *System.Array*.

Соглашения об именовании

Дальше я буду использовать соглашения об именовании, основанные на принципах .NET Framework и на системе, называемой венгерской нотацией в честь легендарного программиста Microsoft Чарльза Симони (Charles Simonyi).

Для определяемых мной имен классов, свойств и событий я буду применять *стиль языка Pascal* (Pascal casing): используются заглавные и строчные буквы, слове начинается с заглавной буквы, внутри слова также могут присутствовать заглавные буквы.

Для определяемых мной полей, переменных и объектов я буду применять *стиль верблюда* (camel casing): первая буква — строчная, но имя может содержать заглавные буквы (горбы).

Для переменных стандартных типов я буду использовать префикс, набранный строчными буквами и показывающий тип переменной. А префиксы такие;

Тип данных	Префикс
bool	*b*
byte	*by*
short	s
Int	*i, x, y, cx, cy*
long	*l*
float	*f*

(*продолжение*)

Типданных	Префикс
char	*ch*
string	*str*
object	*obj*

Префиксы *x* и *y* означают, что переменная содержит координату точки, а *cx* и *cy* — что переменная содержит ширину или высоту (*c* — это сокращение от *count*.)

В именах объектов различных классов, я буду использовать префикс — имя класса, записанное строчными буквами, возможно, сокращенное. Например, объект типа *Point* может называться *ptOrigin*. Иногда в программах будет создаваться только один экземпляр данного класса, Тогда объект будет иметь то же имя, что и класс, но записанное строчными буквами. Например, объект типа *Form* будет называться *form*, а объект типа *PaintEventArgs*— *pea*.

У переменных-массивов перед остальными положенными им префиксами будет стоять префикс *a*.

Выходим за рамки консоли

Осенью 1985 г. Microsoft выпустила первую версию Windows. Кроме того, тогда же она выпустила Windows Software Development Kit (SDK), который показал программистам, как писать приложения для Windows на языке C.

Первая программа «hello-world» в Windows 1.0 SDK оказалась немного скандальной. HELLO.C имела длину примерно 150 строк, также имелся файл ресурсов HELLO.RC длиной не менее 20 строк. Надо заметить, что программа создавала меню и показывала диалоговое окно, но даже без этих удобств она содержала 70 строк кода. У ветеранов программирования на C первое знакомство с программой «hello-world» для Windows вызывало смех или ужас.

В известном смысле вся история новых языков программирования и библиотек классов для Windows связана с борьбой за то, чтобы сделать программу «hello-world» для Windows небольшой, простой и элегантной.

Давайте посмотрим, оправдают ли Windows Forms эти ожидания.

ГЛАВА

2



Hello, Windows Forms

Программы из предыдущей главы, конечно же, не были Windows-программами. Они не создавали свои собственные окна, не выводили графики и ничего не знали про мышь. Весь ввод и вывод пользователя осуществлялся через класс *Console*. Пришла пора двигаться вперед. В дальнейшем в книге мы не забудем класс *Console* полностью, но его роль будет сведена к использованию в рутинных операциях вроде ведения журналов или несложной отладки.

Но в чем же разница между консольным и Windows-приложением? Интересно, что эта разница не столь отчетлива, как кажется. Одно и то же приложение может содержать элементы и консольного, и Windows-приложения. Приложение может запускаться как консольное, затем становиться Windows-приложением, а потом снова превращаться в консольное. Windows-приложения также могут без проблем использовать вывод на консоль. Консольное приложение может показывать информационное окно Windows для сообщения об ошибке, а затем продолжать вывод на консоль, после того как пользователь закроет информационное окно.

Для компилятора C# разница между консольным приложением и Windows-приложением определяется ключом компилятора *target* (его можно сократить до *t*). Консольное приложение создается с помощью ключа:

```
/target:exe
```

Он же действует по умолчанию, если ключ *target* не указан. Для создания исполняемого файла Windows служит:

```
/target:winexe
```

Кроме того, ключ *target* может иметь значение *library* или *module*. В Microsoft Visual Studio .NET используется диалоговое окно Property Pages. В разделе General Common Properties установите для параметра Output Type значение Console Application или Windows Application.

Этот ключ компилятора не делает ничего особо сложного. Он лишь устанавливает в исполняемом файле флажок, показывающий, как загружать и запускать программу. Если исполняемый файл помечен как консольное приложение и запускается из Windows, ОС Windows создает окно командной строки, в котором программа запускается и осуществляет вывод на консоль. Если консольное приложение запускается из окна командной строки, то приглашение MS-DOS не появляется снова до тех пор, пока программа не завершится. Если исполняемый файл помечен как приложение Windows, окно командной строки не создается, и программа ничего не выводит на консоль. При запуске такой программы из окна командной строки, приглашение MS-DOS появляется снова сразу же после запуска программы. Таким образом, если скомпилировать приложение Windows Forms как консольное, ничего плохого не случится!

Следует иметь в виду, что окно командной строки ведет себя по-разному в зависимости от того, как запускается программа — в режиме окончательной версии (release mode) или в режиме отладки (debug mode), В первом случае по завершении работы программы вы увидите стандартное сообщение «Press any key to continue». Благодаря этому, вы сможете посмотреть информацию, выведенную на консоль, и после ее просмотра закрыть окно. Во втором — после запуска программы из Windows окно консоли исчезает без предупреждения сразу по завершении работы программы. Приходится смотреть вывод на консоль до завершения работы программы.

Во всех файлах проектов Visual Studio .NET, относящихся к программам этой книги, указано, что программы являются консольными приложениями. Поэтому при их выполнении сначала появляется окно командной строки. Консоль весьма полезна: если нужно посмотреть, что происходит внутри программы, можно вставить операторы *Console.Write* или *Console.WriteLine* *в любом месте любой программы* этой книги. (Но, как я уже говорил, если вы запустите программу в режиме отладки и завершите программу, посмотреть вывод этих операторов вы не сможете. Кроме того, в этом случае не помещайте операторы *Write* или *WriteLine* после закрытия окна программы.) Почти все загадки, связанные с работой программ, поможет разгадать пара операторов *ConsoleWriteLine.* Кроме того, в пространстве имен *System.Diagnostics* есть класс *Debug,* который можно использовать для этих целей вместо класса *Console.*

Конечно, я бы не стал поставлять Windows-программу, скомпилированную как консольное приложение, пользователям, не являющимся разработчиками. Их могло бы смутить появление окна командной строки, если только они не имеют опыта работы с UNIX или UNIX-подобными средами. Но это всего лишь ключ компилятора, который можно изменить в любое время,

Настоящее отличие между консольным приложением и Windows-приложением заключается в способе получения программой входных данных от пользователя, В консольное приложение информация вводится с клавиатуры при помощи методов *Console.Read* или *Console.ReadLine*; в приложение Windows Forms информация вводится с: клавиатуры (и других устройств) при помощи *событий* (events), которым в этой книге уделено большое внимание.

Я создавал проекты этой главы в Visual Studio .NET почти так же, как проекты главы 1. Я указывал, что это проект C#, а также Empty Project (пустой проект). При

Ниже показан другой возможный вариант вызова *MessageBox.Show* вашей первой программой, работающей с Windows Forms:

```
MessageBox.Show("Hello, world!", "MessageBoxHelloWorld");
```

Если не указывать второй параметр, текста в заголовке не будет.

Для указания кнопок информационного окна можно выбрать одно из значений перечисления:

Перечисление *MessageBoxButtons*

Член перечисления	Значение
OK	0
OKCancel	1
AbortRetryIgnore	2
YesNoCancel	3
YesNo	4
RetryCancel	5

Например, кнопки OK и Cancel (Отмена) покажет такой вызов:

```
MessageBox.Show("Hello, world!", "MessageBoxHelloWorld",
                MessageBoxButtons.OKCancel);
```

При использовании одной из версий *MessageBox.Show* без этого параметра показывается только кнопка OK. Кнопки *AbortRetryIgnore* восходят к печально известному сообщению MS-DOS, которое выводилось при попытке обратиться к устройству (обычно дисководу), которое почему-либо не отвечало. В графических средах использовать эти кнопки, пожалуй, не стоит, если только вы не стремитесь выглядеть до смешного старомодным.

Кроме того, можно указать одно из значений перечисления *MessageBoxIcon* чтобы вывести в информационном окне значок:

Перечисление *MessageBoxIcon*

Член перечисления	Значение
None	0x00
Hand	0x10
Stop	0x10
Error	0x10
Question	0x20
Exclamation	0x30
Warning	0x30
Asterisk	0x40
Information	0x40

Как можно видеть по значениям, в информационных окнах на самом деле используются только четыре уникальных значка. Вот пример показа окна со значком:

```
MessageBox.Show("Hello, world!", "MessageBoxHelloWorld",
                MessageBoxButtons.OKCancel, MessageBoxIcon.Exclamation);
```

Если задано значение *MessageBoxButtons*, соответствующее двум или трем кнопкам, можно задействовать перечисление *MessageBoxDefaultButton*, определяющее кнопку по умолчанию:

Перечисление *MessageBoxDefaultButton*

Член	Значение
Button1	0x000
Button2	0x100
Button3	0x200

Например, вызов:

```
MessageBox.Show("Hello, world! ", "MessageBoxHelloWorld",
                MessageBoxButtons.OKCancel, MessageBoxIcon.Exclamation,
                MessageBoxDefaultButton.Button2);
```

делает вторую кнопку — Cancel (Отмена) — кнопкой по умолчанию. После появления информационного окна именно она является выделенной и реагирует на ввод с клавиатуры, например, на нажатие пробела.

Еще одно перечисление, используемое методом *Show* класса *MessageBox*, - *MessageBoxOptions* — определяет параметры окна:

Перечисление *MessageBoxOptions*

Член	Значение
DefaultDesktopOnly	0x020000
RightAlign	0x080000
RtlReading	0x100000
ServiceNotification	0x200000

Однако эти параметры применяются редко.

Если в информационном окне более одной кнопки, наверное, вам нужно будет знать, какую кнопку щелкнул пользователь для его закрытия. Об этом сообщает возвращаемое *MessageBox.Show* значение, принимающее одно из следующих значений перечисления:

Перечисление *DialogResult*

Член	Значение
None	0
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

Вот пример использования возвращаемого *MessageBox.Show* значения:

```
DialogResult dr = MessageBox.Show("Do you want to create a new file?",
                                  "WonderWord",
                                  MessageBoxButtons.YesNoCancel,
                                  MessageBoxIcon.Question);
if (dr == DialogResult.Yes)
{
     // Обработка "Да" (Yes)
}
else if (dr == DialogResult.No)
{
     // Обработка "Нет" (No)
}
else
{
     // Обработка "Отмена" (Cancel)
}
```

Можно использовать и конструкцию *switch/case,* скажем, так:

```
switch (MessageBox.Show("Do you want to create a new file?",
                        "WonderWord",
                        MessageBoxButtons.YesNoCancel,
                        MessageBoxIcon.Question)
{
case DialogResult.Yes:
     // Обработка "Да" (Yes)
     break;

case DialogResult.No:
     // Обработка "Нет" (No)
     break;

case DialogResult.Cancel:
     // Обработка "Отмена" (Cancel)
     break;
}
```

Иногда информационные окна бывают полезны, если нужно быстро что-то посмотреть. Допустим, вы хотите показать имя каталога Windows с псевдонимом My Documents (Мои документы). Эту информацию можно получить с помощью класса *Environment* пространства имен *System,* используя статический метод *GetFolderPath* с одним параметром — членом перечисления *Environment.SpecialFolder.* Эти два имени отделяются точкой, так как *SpecialFolder*— перечисление, определенное в классе *Environment,*

MyDocumentsFolder.cs

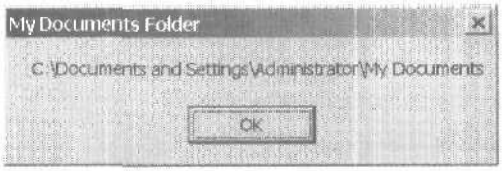
```
//-----------------------------------------
// MyDocumentsFolder.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
```

см. след. стр

```
using System.Windows.Forms;

class MyDocumentsFolder
{
     public static void Main()
     {
          MessageBox.Show(
               Environment.GetFolderPath(Environment.SpecialFolder.Personal),
               "My Documents Folder");
     }
}
```

На моем компьютере эта программа выводит такое информационное окно:



Формы

Конечно, для создания Windows-программ одних информационных окон недостаточно. Чтобы начать разработку полноценного приложения для Windows, понадобится то, что в программировании для Windows традиционно называется *окном,* а в .NET — *формой.* Обычно в программах Windows Forms формы создаются для главного окна приложения. Кроме того, формы служат для построения диалоговых окон приложения.

Форма, используемая в качестве главного окна приложения, обычно состоит из *заголовка* (caption bar, title bar) с именем приложения, *меню* (menu bar), расположенного под заголовком, и внутренней области — *клиентской области* (client ;trea). Форма окружается рамкой, позволяющей изменять ее размеры, или тонкой рамкой, не позволяющей изменять размеры. До главы 14 мы будем создавать формы, не используя меню.

В дальнейшем я собираюсь рассмотреть несколько нестандартных и нетрадиционных подходов к созданию форм и покажу их на экране, перед тем как применить общепринятый испытанный метод. Надеюсь, при таком подходе вы будете лучше понимать происходящее.

Для начала рассмотрим программу, которую я считаю самой короткой программой, создающей формы. Она называется NewForm.cs.

NewForm.cs

```
//---------------------------------------
// KewForra.cs © 2001 by Charles Petzold
//---------------------------------------
class NewForm
{
```

см. след. стр.

```
    public static void Main()
    {
        new System.Windows.Forms.Form();
    }
}
```

Эту программу можно сделать еще короче, лишь использовав более короткое имя класса, опустив комментарии и «лишние» пробелы и удалив модификатор доступа *public,* не являющийся строго обязательным.

Form — это класс пространства имен *System.WindowsForms*В программе NewForm применяется оператор *new* для создания нового экземпляра класса *Form.* Как вы уже знаете, можно сделать программу чуть-чуть длиннее, поместив директиву *using* в начале программы:

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

тогда единственный оператор модуля *Main* будет иметь вид;

```
new Form();
```

Можно также определить объект типа *Form*:

```
Form formOfMine;
```

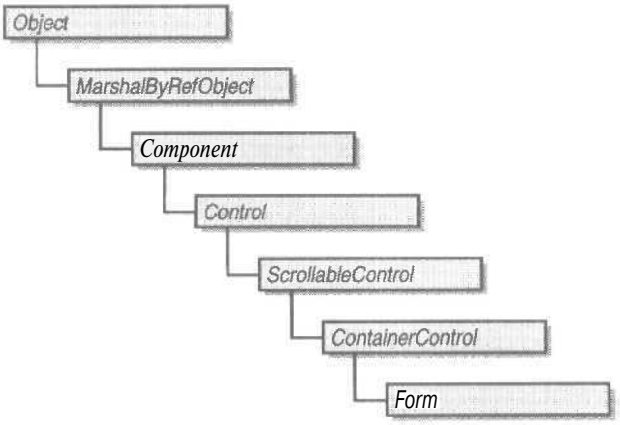
и потом присвоить этому объекту результат оператора *new*:

```
formOfMine = new Form();
```

И, наконец, можно выполнить эти два действия одной строкой кода:

```
Form formOfMine = new Form();
```

Класс *Form* происходит от *ContainerControl,* но на самом деле у него длинная родословная, начинающаяся с класса *Object,* от которого происходят все классы .NET Framework:



Объекты интерфейса пользователя (кнопки, полосы прокрутки, поля редактирования и т. п.) обозначаются термином *элемент управления* (control). Основная часть поддержки этих объектов, в частности, ввод с клавиатуры и мыши и визуальное отображение, реализована в классе *Control.* В классе *ScrollableControl* до-

бавлена поддержка автоматической прокрутки, которую мы рассмотрим в главе 4. Класс *ContainerControl* позволяет элементам управления, подобно диалоговым окнам, выступать в качестве *родителей* (parent) других элементов управления, т. е. другие элементы управления появляются на элементе-контейнере,

Программа NewForm действительно создает форму, но возникает небольшая проблема: конструктор класса *Form* завершает работу, не показывая создаваемую форму. Форма создается, но не делается видимой. При завершении программы форма уничтожается.

Отображение формы

Следующая версия этой программы — ShowForm — исправляет этот недостаток,

ShowForm.cs

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

class ShowForm
{
     public static void Main()
     {
          Form form = new Form();

          form.Show();
     }
}
```

В этой версии есть оператор *using*, сокращающий количество ввода с клавиатуры. В противном случае в обоих местах, где встречается *Form* с заглавной буквы, перед *Form* пришлось бы поставить *System Windows forms. Слово form* со строчной буквы означает экземпляр класса *Form*, создаваемый программой. Можно использовать любое другое имя. (Но при программировании на языках, нечувствительных к регистру, например Visual Basic, использовать *form* нельзя, так как компилятор не сможет отличить это имя от имени класса *Form;* для экземпляров *Form* придется выбирать другие имена.)

Show — один из двух методов, унаследованных *Form* от *Control* и влияющих на видимость формы (или элемента управления):

Методы *Control* (выборочно)

Метод	Описание
void Show()	Делает элемент управления видимым.
void Hide()	Делает элемент управления невидимым.

Вместо:

```
form.Show();
```

можно написать;

```
form.Visible = true;
```

Show является методом. *Visible* выглядит как поле, но на самом деле это свойство:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
bool	*Visible*	Чтение/запись

ShowForm позволяет увидеть форму, но для этого приходится внимательно следить за ней! Сразу после показа форма исчезает. Если ваш компьютер намного быстрее моего, вы вообще можете не увидеть форму.

Такое поведение дает один из возможных ответов на поставленный мной вопрос о различиях между консольным приложением и Windows-приложением: Когда программа командной строки завершает работу, ее вывод остается на консоли. Когда завершает работу Windows-приложение, оно заметает свои следы, уничтожая окно и всю отображенную информацию.

Можно ли замедлить программу, чтобы лучше ее рассмотреть? Знакомы ли вы с концепцией *sleep?* Если вы тщательно изучите пространство имен *System.Threading*, то обнаружите класс *Thread* и статический метод этого класса, приостанавливающий выполнение программы (точнее, потока программы) на заданное время, указанное в миллисекундах,

Следующая программа дважды вызывает метод *Sleep* (с параметрами, указывающими задержку 2.5 секунды) и позволяет лучше рассмотреть форму.

ShowFormAndSleep.cs

```
//-------------------------------------------------
// ShowFormAndSleep.cs © 2001 by Charles Petzold
//  ------------  '                    -     ' -----
using System.Threading;
using System.Windows.Forms;

class ShowFormAndSleep '
{
     public static void Main()
     {
          Form form = new Form();

          form.Show();

          Thread.Sleep(2500);

          form.Text = "My First Form";

          Thread.Sleep(2500);
     }
}
```

Кроме того, в этой версии присваивается значение свойству *Text:*

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
String	*Text*	Чтение/запись

Text — очень важное свойство. Для кнопок свойство *Text* содержит текст, показываемый на кнопке; для полей редактирования — текст в поле. Для форм *Text* — это текст, появляющийся в заголовке формы. При запуске программы вы в течение 2,5 секунд видите форму с пустым заголовком, затем появляется заголовок, и еще через 2,5 секунды форма исчезает.

Мы, безусловно, добились прогресса, но боюсь, что метод *Sleep* — не самый подходящий способ фиксации формы на экране.

Это приложение, и мы хотим его запускать

Волшебный метод, который нам нужен, называется *Kun* и входит в состав класса *Application* пространства имен *System.Windows.forms.* Как в случае классов *Console* и *MessageBox,* экземпляр класса *Application* создать нельзя — все члены класса определены как *static.* Следующая программа создает форму, присваивает значения ее свойствам *Text* и *Visible,* а затем вызывает *Application.Run.*

RunFormBadly.cs

```
//-----------------------------------------
// RunF o rmBadly.cs © 2001 by Charles Petzold
//-----------------------------------------
y§ing System.Windows. Forms;

class RunFormBadly
{
     public static void Main()
     {
          Form form = new Form();

          form.Text = "Mot & flood Idea... ";
          form.Visible = true;

          Application.Run();
     }
}
```

Пожалуй, это уже успех! Форма, отображаемая этой программой показана на следующей странице.

Можно захватить мышью заголовок и перемещать форму по экрану, захватить рамку формы и изменить ее размер, щелкнуть кнопки свертывания или развертывания, вызвать системное меню, называемое в Windows Forms *меню управления* (control box), щелкнув значок в левом верхнем углу окна, или щелкнуть кнопку закрытия в правом верхнему углу, чтобы закрыть окно.



Но есть туг и один очень серьезный недостаток, который вы, верно, уже заметили: при закрытии формы не происходит возврата из метода *ApplicationRun*, и программа остается запущенной, хотя ее форма невидима. Это еще больше бросается в глаза, если программа скомпилирована как консольное приложение: после закрытия окна программы в окне командной строки не появляется знакомое сообщение «Press any key to continue». Для завершения программы можно нажать Ctrl+C. Если программа не компилировалась как консольное приложение, можно вызвать Task Manager (Диспетчер задач), щелкнуть вкладку Processes (Процессы), найти приложение RunFormBadly и завершить его вручную. (Вот и еще одна веская причина компилировать программу как консольное приложение: можно завершить программу, нажав Ctrl+C.)

Вот более правильный способ вызова метода *ApplicationRun*. Ему в качестве параметра передается объект типа *Form,*

RunFormBetter.cs

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

class RunFormBetter
{
     public static void Main()
     {
          Form for» * new Form();

          form.Text - "Hy Very Own Form";

          Application.Run(form);
     }
}
```

Заметьте: в этой версии нет ни вызова *Show,* ни присваивания значения свойству *Visible*. Форма автоматически делается видимой методом *ApplicationRun*. Кроме того, при закрытии формы, передаваемой методу в качестве параметра, *Applica-*

tionRun возвращает управление методу *Main,* и программа нормально завершает работу.

Программистам, имеющим опыт работы с Win32 API, скорее всего известно, что *ApplicationRun* вводит программу в цикл обработки сообщений и что для формы, передаваемой в качестве параметра, имеется код, отправляющий в цикл сообщение выхода при закрытии формы. Так что *ApplicationRun* — это то, что делает программу настоящим Windows-приложением.

Вариации на ту же тему

Попробуем создать две формы, чтобы лучше понимать, как они создаются.

TwoForms.cs

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

class TwoForms
{
     public static void Main()
     {
          Form form1 = new Form();
          Form form2 = new Form();

          form1.Text = "Form passed to Run()";
          form2.Text = "Second form";
          form2.Show();

          Application.Run(form1);

          MessageBox.Show("Application.Run() has returned " +
                          "control back to Main. Bye, bye!",
                          "TwoForms");
     }
}
```

Здесь создаются две формы, *form1* и *form2,* им присваиваются разные заголовки, чтобы можно было отличить их друг от друга. Для *form2* вызывается метод *Show, form1* передается как параметр методу *ApplicationRun.* Когда *ApplicationRun* возвращает управление *Main,* открывается информационное окно.

Запустите TwoForms пару раз, чтобы посмотреть, что происходит. Если сначала закрыть *form2,* то с *form1* ничего не произойдет. Единственный способ заставить *ApplicationRun* вернуть управление и увидеть информационное окно — это закрыть и *form1.* Но если вы сначала закроете *form1,* то с экрана исчезнут обе формы, *ApplicationRun* вернет управление *Main,* и откроется информационное окно.

Так что *ApplicationRun* делает еще одну вещь: когда вы закрываете форму, передаваемую как параметр *ApplicationRun,* метод закрывает все остальные формы,

созданные программой. Если вы не передадите объект *Form* методу *ApplicationRun* (как было в RunFormBadly), то программа должна явно вызвать метод *ApplicationExit*, чтобы заставить *ApplicationRun* вернуть управление. Но откуда программе вызывать *ApplicationExit*, если она находится где-то внутри вызова *ApplicationRun*? Вскоре вы узнаете, как программы могут устанавливать обработчики событий, возвращающие управление программе и дающие потенциальную возможность вызвать *ApplicationExit*.

Свойства форм

Как и во многих других классах, в классе *Form* определяется ряд собственных свойств, и, кроме того, *Form* наследует свойства от своих предков, в частности, от *Control*. О двух таких свойствах я уже говорил — это *Text* и *Visible*. В следующей программе присваиваются значения некоторым свойствам, чтобы продемонстрировать гибкость, с которой можно создавать и показывать формы.

FormPropertles.cs

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

class FormProperties
{
    public static void Main()
    {
        Form form = new Form();

        form.Text            = "Form Properties";
        form.BackColor       = Color.BlanchedAlmond;
        form.Width           *= 2;
        form.Height          /= 2;

        form.FormBorderStyle = FormBorderStyle.FixedSingle;
        form.MaximizeBox     = false;
        form.Cursor          = Cursors.Hand;
        form.StartPosition   = FormStartPosition.CenterScreen;

        Application.Run(form);
    }
}
```

Свойство *BackColor* определяет цвет фона формы. В следующей главе вы узнаете, что *Color* — структура, определенная в пространстве имен *System.Drawing* (обратите внимание на оператор *using*), которая содержит 141 свойство. На самом деле эти свойства являются именами цветов, перечисленных в файле Colors.pdf на компакт-диске, прилагаемом к этой книге.

Свойства *Width* и *Height* определяют начальные размеры формы. Два оператора, изменяющие эти свойства, выполняют операции *get* и *set*, удваивая ширину окна и вдвое уменьшая высоту окна относительно значений по умолчанию.

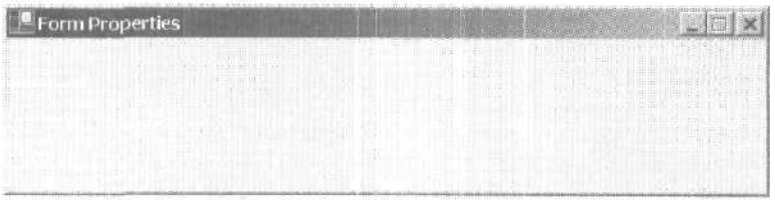
FormBorderStyle — перечисление, определяющее не только вид и функциональность рамки формы, но и другие аспекты работы с формой. Возможные значения таковы:

Перечисление *FormBorderStyle*

Член	Значение	Описание
None	0	Рамка и заголовок отсутствуют.
FixedSingle	1	То же самое, что и *FixedDialog*.
Fixed3D	2	Имеет «объемный» вид.
FixedDialog	3	Рекомендуется использовать в диалоговых окнах.
Sizable	4	Используется по умолчанию.
FixedToolWindow	5	Маленький заголовок, меню управления отсутствует.
SizableToolWindow	6	То же, что и FixedToolWindow, но с рамкой, позволяющей изменять размер.

При использовании стиля *FormBorderStyle.Sizable* форма показывается с заголовком; в заголовке слева находится кнопка меню управления, затем идет текст заголовка, затем идут кнопки свертывания/развертывания и справа — кнопка закрытия. В стилях *FixedToolWindow* и *SizableToolWindow* (инструментальное окно) заголовок имеет меньшие размеры, а кнопок управления, свертывания/развертывания нет,

Стиль *FormBorderStyle FixedSingle,* который я использовал в приведенной выше программе, не позволяет пользователю изменять размер формы. Кроме того, я присвоил свойству *MaximizeBox* значение *false*, поэтому кнопка развертывания, как видно ниже, отключена:



Свойство *Cursor* определяет вид курсора мыши, когда курсор находится в клиентской области формы. Свойство *StartPosition* определяет, где первоначально показывается форма. Значение *CenterScreen* перечисления *FormStartPosition* указывает, что форма появится в центре экрана, а не в месте, автоматически определяемом Windows, как это было бы по умолчанию.

Анализируя программу FormProperties, можно поразмышлять над тем, как структурируются программы Windows Forms. Чтобы заставить форму взаимодействовать с пользователем, нужно вызвать *Application.Run*, но, пока форма не закроется, *Application.Run* не вернет управление.

Словом, пока что не видно, куда поместить свой код!

Ввод, управляемый событиями

Многие консольные программы вообще не взаимодействуют с пользователем. Типичное консольное приложение получает всю нужную информацию из командной строки, выполняет свои функции и завершает работу. Если в консольной программе требуется взаимодействие с пользователем, она получает входные данные с клавиатуры. В .NET Framework консольные программы получают данные с клавиатуры, вызывая методы *Read* или *ReadLine* класса *Console*. После того, как программа перестает получать данные с клавиатуры, программа продолжает работу.

Программы, предназначенные для работы в графических средах, используют другую модель ввода. Одна из причин этого состоит в существовании разных типов устройств ввода. Информация вводится не только с клавиатуры, но и с помощью мыши. Кроме того, программы могут создавать элементы управления, например кнопки, меню, полосы прокрутки, взаимодействующие с пользователем от имени главной программы,

Я полагаю, что теоретически программная среда, поддерживающая разные устройства ввода, могла бы взаимодействовать со всеми этими устройствами, используя технологию *последовательного опроса* (serial polling). При последовательном опросе программа проверяет наличие ввода с клавиатуры, при его отсутствии проверяет мышь, если с помощью мыши ничего не вводили, проверяет ввод через меню, а меню в свою очередь проверяет клавиатуру и мышь и т. д. До эры Windows программы, работающие в символьном режиме и использующие мышь, выполняли последовательный опрос.

Однако гораздо лучшей моделью ввода с различных устройств является модель ввода, *управляемая событиями* (event-driven). В Windows Forms каждый тип ввода связан с определенным методом класса. Когда происходит определенное событие (скажем, нажимается клавиша, перемещается мышь или выбирается пункт меню программы), то как бы из-за пределов программы вызывается соответствующий метод.

Сначала такая модель ввода кажется хаотичной. Так как пользователь стучит по клавишам, двигает мышь, нажимает кнопки, использует полосы прокрутки и выбирает пункты меню, постоянно вызываются методы программы со всевозможных направлений. Но на практике все оказывается более упорядоченным, так как все вызовы методов выполняются в одном потоке. Только после того как один метод завершит обработку своего события, вызывается другой метод другого события,

После того как программа для Windows Forms выполнила инициализацию формы, *всё,* что она делает, *каждый,* даже самый небольшой, фрагмент ее кода - это ответ на событие. Большую часть времени программа бездействует, находясь где-то внутри вызова *Application.Run* и ожидая события. Часто имеет смысл воспринимать программы для Windows Forms как *конечные автоматы* (state machine), состояние которых целиком определяется изменениями, вносимыми событиями.

События играют настолько важную роль, что их поддержка органично встроена в .NET Framework и (Ж События наряду с конструкторами, полями, методами и свойствами являются членами классов. Когда программа определяет метод, выполняющий обработку события, этот метод называется *обработчиком события* (event handler). Параметры обработчика должны соответствовать функции-прототипу — *делегату* (delegate). Мы в скором времени рассмотрим, как это все работает.

Как вы узнаете в главе 6, имеется три разных типа событий клавиатуры. Один сообщает о нажатии клавиши, другой — об отпускании клавиши, третий — о том, что при нажатии определенной клавиши или комбинации клавиш сгенерирован код символа,

В главе 8 я вас познакомлю с 7 типами событий мыши, сообщающих о ее перемещении и об одинарных или двойных щелчках ее кнопок.

В главе 10 вы узнаете о событии таймера. Оно периодически оповещает форму об истечении определенного периода времени. Программы, показывающие часы, используют событие таймера, чтобы каждую секунду обновлять время.

В главе 12, когда мы начнем создавать элементы управления (кнопки, текстовые поля, списки) и помещать их на формы, вы узнаете, что эти элементы управления передают информацию форме через события. Именно события сообщают о нажатии кнопки или изменении текста в поле.

В главе 14 вы узнаете, что меню также передают информацию форме через события. Есть события, сообщающие о том, что будет показано выпадающее меню, о выборе пункта меню и о щелчке пункта меню,

Но одно из самых странных событий — даже не верится, что оно может быть событием, — является в то же время и одним из самых важных. Это *Paint.* Оно сообщает программе о необходимости отобразить информацию в окне.

Ничто так отчетливо не раскрывает огромную разницу между программами командной строки и графическими, как событие *Paint.* Программа командной строки отображает информацию, когда сочтет нужным. Программа Windows Forms также может отображать информацию, когда захочет, но делает это несколько иначе. *Paint* информирует программу о том, что вся клиентская область или ее часть *недействительна* (invalid) и требует перерисовки.

Как клиентская область становится недействительной? Когда форма только что создана, вся клиентская область недействительна, так как программа еще ничего в ней не показывала. В программе происходит первое событие *Paint,* сообщающее о необходимости показать что-нибудь в клиентской области.

Когда вы перемещаете окна по экрану, так что они перекрывают друг друга, Windows не запоминает вид клиентской области, закрытой другим окном. Потом, когда клиентская область снова открывается, программа должна восстановить ее вид. Поэтому в ней и происходит событие *Paint.* Когда окно программы после свертывания возвращается в нормальное состояние, в ней также происходит событие *Paint.*

Windows-программа должна быть способна перерисовать свою клиентскую область в любой момент времени. Она должна запоминать всю нужную для этого информацию или иметь возможность быстро обратиться к этой информации. Может показаться, что структуризация программ для правильной обработки события *Paint* накладывает слишком жесткие ограничения, но в противном случае работа программы замедлится.

Обработка события *Paint*

Правильнее всего рассматривать события на примерах, На практике при обработке события *Paint* вам в первую очередь потребуется знакомство с *PaintEventHandler*— делегатом, определенным в пространстве имен *System.Windows.Forms*одним оператором, который имеет такой вид (в синтаксисе C#):

```
public delegate void PaintEventHandler(object objSender, PaintEventArgs pea);
```

Если вам кажется, что это прототип функции, вы недалеки от истины. Вторым параметром является класс *PaintEventArgs,* также определенный в пространстве имен *System.WindowsForms*— вскоре мы его рассмотрим.

Для обработки события *Paint* одной из программ, приведенных ранее в этой главе, нужно определить в классе статический метод с такими же параметрами и таким же типом возвращаемого значения, что и у делегата *PaintEventHandler*:

```
static void MyPaintHandler(object objSender, PaintEventArgs pea)
{

}
```

Затем этот обработчик события подключается к событию *Paint* класса *Form* посредством специальной синтаксической конструкции:

```
form.Paint += new PaintEventHandler(MyPaintHandler);
```

Paint — это событие, определенное в классе *Control* и, благодаря наследованию, входящее в класс *Form.* Единственными двумя операторами, применимыми к *Paint,* являются операторы присваивания += и -=. Оператор += устанавливает обработчик события, подключая метод к событию. Его синтаксис:

```
object.event += new delegate(method)
```

При отключении метода от события используется такой же синтаксис, но с оператором -= :

```
object.event -= new delegate(method)
```

Однако отключение метода от события требуется редко. Обычно обработчик события устанавливается и никогда не отключается от события.

Два параметра обработчика события *Paint* — объект, который я назвал *objSender.* и класс *PaintEventArgs,* для которого я использовал сокращенное название *pea.* Первый параметр содержит ссылку на объект, к которому применяется событие *Paint;* в нашем случае это объект *form.* Данный объект называется отправителем (sender), так как событие исходит от этого объекта,

Класс *PaintEventArgs* определен в пространстве имен *System.Windowsforms.* Он содержит два свойства, *Graphics* и *ClipRectangle,* доступные только для чтения:

Свойства *PaintEventArgs*

Тип	Свойство	Доступ	Описание
Graphics	*Graphics*	Чтение	Крайне важный объект графического вывода
Rectangle	*ClipRectangle*	Чтение	Недействительный прямоугольник

Свойство *Graphics* содержит экземпляр класса *Graphics,* определенного в пространстве имен *System.Drawing. Graphics* — важнейший класс библиотеки Windows Forms, по важности не уступающий *Form.* Этот класс служит для рисования графики и вывода текста на форме. Б пространстве имен *SystemDrawing* реализована система графического программирования — GDI+. Это усовершенствованная версия интерфейса GDI (Graphics Device Interface. интерфейс графического устройства) системы Windows. О свойстве *ClipRectangle* см. главу 4.

В подавляющем большинстве программ этой книги вы будете встречать оператор:

```
Graphics grfx = pea.Graphics;
```

в первой строке обработчика события *Paint.* Свой объект типа *Graphics* вы можете называть как угодно. Кто-то из программистов пишет *graphics* строчными буквами, но это название настолько длинное, что некоторые просто используют букву *g*! Я выбрал компромиссный подход.

Прежде чем углубиться в этот новый материал, рассмотрим программу, содержащую обработчик события *Paint.*

PaintEvent.cs

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

class PaintEvent
{
     public static void Main()
     {
          Form form   - new Form();
          form.Text   - "Paint Event";
          form.Paint += new PaintEventHandler(MyPaintHandler);

          Application,flun(forBi);
     >
     static void MyPaintHandler(object objSender, PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;

          grfx.Clear(Color.Chocolate);
     }
}
```

После создания формы в методе *Main,* метод *MyPaintHandler* подключается к событию *Paint* формы. В обработчике события программа получает в составе класса *PaintEventArgs* объект *Graphics* и вызывает метод *Clear* этого объекта, *Clear* — это простой метод (возможно, самый простой из методов рисования), определенный в классе *Graphics:*

Методы *Graphics* (выборочно)

Метод	Описание
void Clear(Color clr)	Закрашивает всю клиентскую область некоторым цветом,

Параметр метода — объект типа *Color,* который я подробней рассмотрю в главе 3- Как я упоминал при рассмотрении программы FormProperties, показанной

выше, самый простой путь получить цвет — указать одно из 141 имени цветов, являющихся статическими свойствами структуры *Color*.

Чтобы получить представление о частоте, с которой программа получает события *Paint,* попробуйте вставить в *MyPaintHandler* оператор:

```
Console.WriteLine("Paint Event");
```

Кроме того, в следующей главе будет показана пара программ, визуально демонстрирующих частоту возникновения событий *Paint*.

С этого момента все программы для Windows Forms этой книги будут содержать в начале по меньшей мере три оператора *using*.

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

Вообще говоря, эти операторы — минимум, необходимый любому нетривиальному приложения Windows Forms.

Можно заметить связь между этими тремя операторами *using* и тремя DLL, ссылки на которые указывались при компиляции программы. Для программистов на C или C++ естественно думать, что операторы *using* идентичны операторам *#include*. Но это не так! Операторы *using* ближе к операторам *With* языка Visual Basic. Они существуют только для того, чтобы не требовалось вводить полные имена классов. Все, что в программах на C и C++ обычно делают файлы заголовков (например, объявление типов, функций и классов), здесь обеспечивают DLL, указываемые как ссылки (references), и те же DLL связываются с запускаемой программой для реализации этих классов.

Показ текста

Класс *Graphics* содержит много методов для рисования графических фигур, например, линий, кривых, прямоугольников, эллипсов и растровых изображений. Метод этого класса, показывающий текст в форме, называется *DrawString* (не спутайте этот метод со шнурком, который не дает сползти вашим штанам[1]).

DrawString имеет шесть перегруженных версий, но первые три параметра у них всегда одинаковы. Самая простая версия *DrawString* определяется так:

```
void DrawString(string str, Font font, Brush brush, float x, float y)
```

Вы могли предположить, что параметрами *DrawString* являются строка, которую требуется показать, и координаты места, в котором она появится. Предположить, что к параметрам метода относятся также шрифт, используемый для показа текста, и нечто, называемое *Brush* (служит для указания цвета текста), вы вряд ли могли, но все же это так. Наличие этих двух параметров является составной частью того, что имеется в виду, когда говорят о GDI+ как о графической системе программирования *без информации о состоянии* (stateless). Все, что нужно системе для показа различных графических фигур, содержится прямо в вызовах методов.

[1] Drawstring — шнурок, затягивающий сумку, мешок. — *Прим.перев.*

Обратной стороной такого подхода является то, что вызов *DrawString* содержит слишком много информации. Вы можете поймать себя на том, что сокращаете второй и третий параметры до одной буквы или ищете другие пути, чтобы сделать вызов метода короче.

Первый параметр *DrawString* — текстовая строка, которую требуется отобразить, например:

```
grfx.DrawString("Hello, world!", ...);
```

Остальные параметры давайте рассмотрим подробнее.

Шрифт

Второй параметр *DrawString* — шрифт, используемый для рисования текста. Это объект типа *Font*. Класс *Font* определен в пространстве имен *System.Drawing*. О классе *Font* я многое расскажу в главе 9. Достаточно сказать, что для программ Windows Forms доступно большое количество шрифтов, размеры которых можно масштабировать. Пока мы используем шрифт по умолчанию. Очень удобно, что каждый класс, унаследованный от *Control*, наследует свойство *Font*, в котором хранится шрифт элемента управления по умолчанию,

Свойства *Control* (выборочно)

Тип	Свойство	Доступ	Описание
Font	*Font*	Чтение/запись	Шрифт элемента управления по умолчанию

Сначала вас может смутить, что как свойство, так и класс имеют имя *Font*, но гарантирую, что через несколько месяцев это перестанет вас беспокоить.

При установке обработчика события *Paint* для формы объект, для которого выполняется событие, получают приведением типа первого параметра к типу этого объекта:

```
Form form = (Form)objSender;
```

Это приведение типа работает, так как *objSender* действительно является объектом типа *Form*. Если *objSender* не объект типа *Form* (или класс, унаследованный от *Form*), этот оператор сгенерировал бы исключение. Так что в обработчике события вы можете обращаться к шрифту по умолчанию, используя *form.Font*. Вызов *DrawString* будет выглядеть примерно так:

```
grfx.DrawString(str, form.Font, ...);
```

Если *DrawString* вызывается несколько раз, можно сначала определить объект типа *Font* и присвоить ему шрифт по умолчанию:

```
Font font = form.Font;
```

В этом операторе слово «font» используется многократно! Первое слово *Font* — это класс, определенный в пространстве имен *System.Drawing*; *font* строчными буквами — объект этого класса. Последний *Font* — это свойство класса *Form*. Вызов *DrawString* принимает вид:

```
grfx.DrawString(str, font, ...);
```

Для краткости можно именовать объект типа *Font* просто буквой *f*

Кисть

Третий параметр *DrawString* указывает «цвет» символов шрифта. Слово «цвет» я взял в кавычки, так как параметр на самом деле имеет тип *Brush* (кисть), а кисть может гораздо больше, чем просто закрашивать поверхность одним цветом. В кистях могут использоваться градиентная заливка, или шаблоны-орнаменты, или растровые изображения. Кисти так разнообразны и обладают такими мощными возможностями, что я посвятил им почти целую главу, Но так как это будет глава 17, а вы сейчас читаете вторую, нам пока хватит очень простых кистей.

Проще всего добиться многообразия цветов — использовать класс *Brushes*. Заметьте: *Brushes* стоит во множественном числе в отличие от *Brush*, которое также является именем класса, Класс *Brushes* содержит 141 статическое неизменяемое свойство с такими же именами цветов, что и в классе *Color,* перечисленных в файле Colors.pdf на компакт-диске, прилагаемом к этой книге. Свойства класса *Brushes* — это объекты типа *Brush*. Так как эти свойства статические, к ним обращаются по имени класса и имени свойства:

```
grfx.DrawString(str, font, Brushes.PapayaWhip, ...);
```

Наверное, вы сейчас думаете: «Конечно, очень приятно, что можно выводить текст различными цветами, используя градиенты и шаблоны, но будем реалистами: вероятно, 97,5 % текста, который мне нужно показывать, имеет обычный черный цвет. За небольшим исключением я буду просто передавать *Brushes.Black* как третий параметр *DrawString*». Можно также определить объект типа *Brush:*

```
Brush brush = Brushes.Black;
```

и передавать этот объект в 97,5 % вызовов *DrawString:*

```
grfx.DrawString(str, font, brush, ...);
```

Конечно, можно даже использовать имя *b* для краткости.

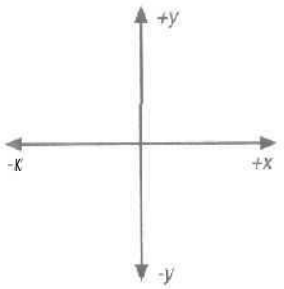
Но боюсь, что такое использование *BrushesBlack* может оказаться ошибочным, Вы делаете неявное предположение, что цвет фона формы не будет черным, как кисть. Разве не может так быть? Очень даже может. В этом случае текст окажется невидимым.

И все же я пока разрешаю вам использовать *Brushes.Black* в вызовах *DrawString,* но только если вы будете присваивать свойству *BackColor* формы значение *Color.White* или любой другой цвет фона, на котором виден черный текст. Более правильные подходы я рассмотрю в главе 3.

Координаты точек

И, наконец, последние два параметра *DrawString* указывают горизонтальную *(x)* и вертикальную *(y)* координаты точки, где появится левый верхний угол строки текста.

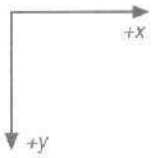
Если вы получили математическое образование или школьная математика травмировала вас настолько, что шрамы навсегда остались в вашем мозге, вы можете себе представить такую двумерную систему координат:



Она известна как *декартова система координат* и названа по имени французского математика и философа Рене Декарта (1596—1650), который заслужил эту честь изобретением аналитической геометрии и перед которым специалисты в области компьютерной графики в неоплатном долгу.[2] Начало координат — точка (0, 0) — является центром. Значения координаты *x* увеличиваются слева направо, значения *y* увеличиваются снизу вверх,

Однако это не совсем та система координат, которая используется в большинстве графических сред. Координатная система, в которой *y* увеличивается снизу вверх, не соответствует способу письма большинства западных языков. Кроме того, на заре компьютерной графики программисты выполняли запись напрямую в память видеоадаптера. Буферы видеопамяти начинались с верха экрана, так как луч развертки перемещался сверху вниз. Это в свою очередь вызвано тем, что луч развертки телевизоров также перемещался сверху вниз — такое решение было принято где-то в 60-х.

В среде Windows Forms, как и в большинстве графических сред[3], координатная система по умолчанию имеет начало в левом верхнем углу и выглядит так:



Я написал «координатная система *по умолчанию*», так как от нее можно перейти к какой-нибудь другой. Это удовольствие нас ожидает в главе 7.

Когда вы рисуете на форме при помощи объекта *Graphics,* получаемого из класса *PaintEventArgs,*являющегося в свою очередь параметром обработчика события *Paint,* все координаты отсчитываются от левого верхнего утла клиентской области формы.

---

[2] Факсимиле и перевод на английский труда Декарта по аналитической геометрии (1637 г.) см. в книге *The Geometry ojRene Descartes* (New York: Dover, 1954).

[3] Исключение составляет OS/2 Presentation Manager, в котором координата увеличивалась снизу вверх. Это отлично подходило для программирования графики, но в других случаях не всегда было удобно. Программистам приходилось использовать координаты, растущие снизу вверх, например, при указании положения элементов управления в диалоговых окнах. Это приводило к тому, что диалоговые окна разрабатывались снизу вверх. Подробнее см.: Charles Petzold, *Programming the OS/2 Presentation Manager* (Redmond, WA: Microsoft Press, 1989) или Charles Petzold, *OS/2 Presentation Manager Programming* (Emeryville, CA: Ziff-Davis Press, 1994).

Единицами измерения являются пикселы. Значения *x* увеличиваются слева направо, значения *y* — сверху вниз.

Повторю: координаты отсчитываются от левого верхнего угла *клиентской области*. Это область внутри формы, не занятая заголовком, рамкой и меню (если они есть у формы). При использовании объекта *Graphics* из класса *PaintEventArgs* вы не можете рисовать *за пределами* клиентской области. Значит, можно не беспокоиться, что вы нарисуете что-нибудь не там, где предполагалось.

Координаты точки, передаваемые методу *DrawString*, указывают положение левого верхнего угла первого символа строки. При указании координат (0, 0) строка текста показывается в левом верхнем углу клиентской области.

Теперь применим полученные знания в программе *PaintHello*.

PaintHello.cs

```
//------------------------------------------------------------------------
// PaintHello.cs © 2001 by Charles Petzold
//------------------------------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class PaintHello
{
    public static void Main()
    {
        Form form      = new Form();
        form.Text      = "Paint Hello";
        form.BackColor = Color.White;
        form.Paint     += new PaintEventHandler(MyPaintHandler);

        Application.Run(form);
    }
    static void MyPaintHandler(object objSender, PaintEventArgs pea)
    {
        Form      fora = (Form)objSender;
        Graphics grfx = pea.Graphics;

        grfx.DrawString("Hello, world!", form.Font, Brushes.Black, 0, 0);
    }
}
```

Итак, мы получили первую и, как можно видеть, не такую уж простую программу, показывающую текст в форме. Текст выводится в левом верхнем углу клиентской области:



Событье *Paint* — особое событье!

Будьте осторожны при написании методов-обработчиков события *Paint*. Эти методы вызываются часто и иногда довольно неожиданно. Лучше всего, если обработчик *Paint* может перерисовать клиентскую область быстро, без задержек.

Ранее в этой главе я предлагал использовать для простой отладки информационные окна. Но ни в коем случае не помещайте вызов *MessageBox.Show* в обработчик события *Paint*! Информационное окно перекроет часть клиентской области, в результате чего снова произойдет событие *Paint*. И снова, снова... По этой же причине *никогда* не помещайте вызовы *Console.Read* или *Console.ReadLine* в *любые* обработчики событий. Однако вызовы *Console.Write* или *Console.WriteLine* вполне безопасны.

Кроме того, не делайте ничего, что может накапливаться. В одной из своих первых программ для Windows Forms я написал *обработчик Paint,* который обращался к свойству *Font,* создавал новый шрифт вдвое большего размера и присваивал свойству *Font* этот новый шрифт. Каждый раз при возникновении события *Paint* шрифт становился вдвое больше, чем раньше. (*«Дорогая, я увеличил шрифт!»*, последняя серия.)

Может показаться, что выполнение всей прорисовки в обработчике события *Paint* накладывает некоторые ограничения. Бывает. Поэтому в Windows Forms есть несколько методов, повышающих гибкость рисования.

Во-первых, вы можете получить объект *Graphics* вне обработчика события *Paint,* вызвав метод *CreateGrapbics,* имеющийся у класса *Control* и унаследованный *Form*. Во-вторых, можно инициировать событие *Paint* из других событий, используя метод *Invalidate* класса *Control*. Я покажу, как это делается, когда буду рассматривать события клавиатуры, мыши и таймера в главах 6, 8 и 10.

Несколько форм, несколько обработчиков

Чтобы лучше понять, как работает обработчик события *Paint,* рассмотрим несколько вариаций на эту тему. В следующей программе используется один и тот же обработчик события *Paint* для двух создаваемых форм.

PaintTwoForms.cs

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

class PaintTwoForms
{
     static Form form1, form2;

     public static void Main()'
     {
          form1 я new Form();
          form2 = new Form();

          form1.Text      = "First Form";
          form1.BackColor = Color.White;
          f orm1 .Paint += new PaintEventHandler(MyPaintHandler);

          form2.Text      = "Second Form";
          form2.BackColor = Color.White;
          form2.Paint   += new PaintEventHandler(MyPaintHandler);
          form2.Show();

          Application.Run(form1);
     }
     static void MyPaintHandler(object objSender, PaintEventArgs pea)
     {
          Form     form * (Form)objSender;
          Graphics grfx - pea.Graphics;
         , string   str;

          if(form == form1)
               str  = "Hello from the first form";
          else
               str  = "Hello from the second form";

          grfx.DrawString(str, form.Font, Brushes.Black, 0, 0);
     }
}
```

Обратите внимание, что объекты *Form* являются полями и поэтому доступны как *Main,* так и обработчику события *Paint.* Каждый вызов обработчика событии *Paint* относится к одной из двух форм, создаваемых программой. Обработчик события может определить, к какой форме относится событие, сравнив параметр *objSender* (приведенный к типу *Form) с* двумя объектами *Form,* являющимися полями. Если закрыть глаза на небольшую проблему с регистром символов, можно заменить всю эту конструкцию *if/else* одним оператором:

```
str = "Hello from the " + form.Text;
```

А теперь поступим наоборот: создадим одну форму и подключим к ней два обработчика события *Paint*.

TwoPaintHandlers.cs

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

class TwoPaintHandlers
{
     public static void Main()
     {
          Form form      = new Form();
          form.Text      = "Two Paint Handlers";
          form.BackColor * Color.White;
          fora, Paint    +» new PaintEventHandler(PaintHandler1);
          form.Paint     += new PaintEventHandler(PaintHandler2);

          Application.Run(form);
     >
     static void PaintHandler1(object objSender, PaintEventArgs pea)
     {
          Form     form * (Form)objSender;
          Graphics grfx = pea.Graphics;

          grfx.DrawString("First Paint Event Handler", form.Font,
                          Brushes.Black, 0, 0);
     }
     static void PaintHandler2(object objSender, Paint Event A rgs pea)
     {
          Form     form - (Form)objSender;
          Graphics grfx = pea.Graphics;

          grfx.DrawString("Second Paint Event Handler", form.Font,
                          Brushes.Black, 0, 100);
     }
}
```

Эта программа подчеркивает один из интересных аспектов подключения обработчиков событий, Если имеется более одного обработчика события, все обработчики вызываются по порядку. Заметьте, что методу *DrawString* передаются координаты (0, 0) в первом обработчике и (0, 100) — во втором. Я предположил, что высота шрифта по умолчанию не превышает 100 пикселов. Мне кажется, что можно на это рассчитывать.



Наследование форм

Итак, вы уже научились создавать формы, присваивать значения их свойствам (например, тексту заголовка или цвету фона), подключать обработчики событий. Если вы можете подключить обработчик события *Paint,* значит, можете подключить обработчики событий клавиатуры, мыши, меню и т. д.

Но боюсь, придется сказать горькую правду: обычно все делается не так.

Чтобы полностью использовать мощь класса *Form,* вы должны не просто создать форму. Вы должны ее *сотворить. Control* родил *ScrollableControl, ScrollableControl* родил *ContainerControl,* а *ContainerControl* родил *Form,* и теперь *Form* может породить воистину поразительные формы — вы только должны указать, какими они будут,

Такие формы создаются определением класса, который является наследником *Form.* Рассмотрим такую программу.

InheritTheForm.cs

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

class InheritTheForm: Form
{
     public static void Main()
     {
          InheritTheForm form = new InheritTheForm();
          form.Text = "Inherit the Form";
          form.BackColor = Color.White;

          Application.Run(form);
     }
}
```

Обратите внимание на оператор *class:*

```
class InheritTheForm: Form
```

Та часть, что идет за именем класса *Form,* означает, что *InheritTheForm* является наследником *Form* и наследует все методы и свойства *Form.*

Как и раньше, этот класс содержит статический метод *Main,* являющийся точкой входа в программу. Однако в *Main* создается новый экземпляр *InheritTheForm,* а не *Form.* Так как *InheritTheForm* — наследник *Form,* у него также имеются свойства *Text* и *BackColor,* которым в программе присваиваются значения. Объект типа *Form* можно передавать в качестве параметра методу *Application.Run,* значит, и любой объект типа, унаследованного от *Form,* можно передавать как параметр *Application.Run.*

Программа InheritTheForm создает форму, выполняет инициализацию, которая в данном случае заключается в присваивании значения свойству *Text,* затем вызывает *Application.Run,* передавая созданную форму как параметр. Более традиционный подход заключается в переносе инициализации формы в конструктор класса.

InheritWithConstructor.cs

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

class InheritWithConstructor: Form
{
     public static void Main()
     {
          Application.Run(new InheritWithConstructor());
     }
     public InheritWithConstructor()
     {
          Text = "Inherit with Constructor";
          BackColor = Color.White;
     }
}
```

Напоминаю, что для конструктора не указывается тип возвращаемого значения и что у конструктора по умолчанию пустой список параметров.

Родословная *Form* начинается с *Object* и содержит еще пять классов, Когда в *Main* создается объект *InheritWithConstructor,* сначала вызывается конструктор по умолчанию класса *Object,* затем — конструктор по умолчанию класса *MarshalByRefObject* и т. д. до конструктора по умолчанию класса *Form,* и, наконец, последним вызывается конструктор по умолчанию класса *InheritWithConstructor.*

Заметьте, что я не указывал перед свойствами *Text* и *BackColor* имя объекта, который в предыдущих программах этой главы назывался *form.* Перед этими свойствами ничего не нужно ставить, так как это свойства класса *InheritWithConstructor,* Они являются свойствами *InheritWithConstructor,* так как этот класс происходит от классов *Control и Form,* в которых изначально определены эти свойства.

Если бы я захотел что-нибудь поставить перед этими свойствами, то это было бы ключевое слово *this*:

```
this.Text = "Inherit with Constructor";
this.BackColor = Color.White;
```

Ключевое слово *this* означает текущий объект.

Метод *OnPaint*

Какие преимущества дает наследование от класса *Form* в сравнении с созданием экземпляра этого класса? Хотя большинство методов и свойств *Form* определены как *public,* некоторые важные свойства определены как *protected.* Эти защищенные методы и свойства доступны только наследникам *Form.* Одно из таких защищенных свойств — *ResizeRedraw*, которое я рассмотрю в главе 3.

Один из защищенных методов, наследуемых *Form* от *Control.* — *OnPaint.* Однако он вам нужен не для того, чтобы его *вызывать*, но чтобы *переопределить* — тогда вам не потребуется подключать обработчик события *Paint.* У метода *OnPaint* один параметр — объект типа *PaintEventArgs.* Можно из этого параметра получить объект *Graphics* точно так же, как и в обработчике события *Paint.*

Вот окончательный вариант программы Windows Forms «hello-world».

HelloWorld.cs

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

class HelloWorld: Form
{
     public static void Main()
     {
          Application.Run(new HelloWorld());
     }
     public HelloWorld()
     {
          Text = "Hello World";
          BackColor = Color.White;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;

          grfx.DrawString("Hello, Windows Forms!", Font,
                          Brushes.Black, 0, 0);
     }
}
```

Это официальный, сертифицированный, проверенный программистами и одобренный мамой способ создания формы на языке C# при помощи библиотеки классов Windows Forms. Вот почему это первая программа книги, названная просто HelloWorld. (Правда, в следующей главе я покажу лучший способ указания цвета фона и цвета текста.) Обратите внимание, что в *OnPaint* я ничего не ставил перед *Font*. Методу *OnPaint* не нужен параметр *objSender*, так как форма, для которой вызывается *OnPaint*, всегда определяется как *this*.

Вот как выглядит окно программы:



Но всегда найдется умник, который сидит на галерке и тянет руку, нахально спрашивая: «А можно отцентрировать текст в окне?»

Да. можно, и в следующей главе я покажу три различных способа, как это сделать.

Как быть с *Main?*

При рассмотрении программ вроде HelloWorld может вызвать удивление метод *Main*. *Main* — метод класса *HelloWorld*, в то же время *Main* создает экземпляр класса *HelloWorld*. Это может показаться странным: программа уподобляется Мюнхгаузену, который вытянул себя из болота за волосы. Как метод *Main* вообще может выполняться, если еще не создан экземпляр класса *HelloWorld*?

Отвечаю: метод *Main* определен как *static*. Существование статических методов не зависит от того, созданы ли экземпляры каких-либо объектов класса. По идее ОС загружает программу в память и начинает выполнение программы с вызова:

```
HelloWorld.Main();
```

Если *Main* не определен как *static,* ОС не сможет его вызвать. Если вы удалите слово *static* из определения *Main*, компилятор сообщит об отсутствии у программы точки входа.

Может, вам покажется удобнее поместить *Main* в свой класс, как в программах на C# главы 1. В этом нет ничего неправильного, и кое-кто предпочитает этот подход. Его демонстрирует программа SeparateMain.cs, по функциональности не отличающаяся от HelloWorld.

Согласен: она выглядит чище с точки зрения архитектуры, и я немного поэкспериментировал, структурируя примеры программ моей книги таким образом, Но у меня нет желания добавлять в каждую программу три строки кода, и я не вижу

смысла заводить два класса там, где можно обойтись одним. Кроме того, я отказался от такого подхода, когда начал замечать, что статические методы и свойства классов .NET возвращают экземпляры классов, к которым они относятся. Например, *Chocolate* — статическое свойство класса *Color,* тем не менее, оно возвращает экземпляр *Color.*

SeparateMain.cs

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

class SeparateMain
{
     public static void Main()
     {
          Application.Run(new AnotherHelloWorld());
     }
}
class AnotherHelloWorld: Form
{
     public AnotherHelloWorld()
     {
          Text = "Another Hello World";
          BackColor = Color.White;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;

          grfx.DrawString("Hello, Windows Forms!", Font,
                          Brushes.Black, 0, 0);
     }
}
```

События и методы «On...»

Как вы уже видели, при создании экземпляра *Control* или любого класса, происходящего от *Control,* например *Form,* можно установить обработчик события *Paint.* Для этого сначала нужно определить статический метод с такими же типами параметров и возвращаемого значения, что и у делегата *PaintEventHandler:*

```
static void MyPaintHandler(object objSender, PaintEventArgs pea)
{
     // Код, выполняющий прорисовку
}
```

Затем нужно установить обработчик этого события для конкретного объекта, например для *form*, при помощи кода:

```
form.Paint += new PaintEventHandler(MyPaintHandler);
```

Однако в классе, *происходящем* от *Control*, не нужно (хотя и можно) устанавливать обработчик события *Paint*. Можно просто переопределить метод *OnPaint*:

```
protected override void OnPaint(PaintEventArgs pea)
{
    // Код, выполняющий прорисовку
}
```

Вы увидите, что все события, определенные в Windows Forms, похожи. Для каждого события имеется соответствующий защищенный метод, Имя метода состоит из слова On, за которым идет имя события. Для каждого из событий, с которыми мы будем сталкиваться, я буду приводить таблицу такого вида:

События *Control* (выборочно)

Событие	Метод	Делегат	Параметр
Paint	*OnPaint*	*PaintEventHandler*	*PaintEventArgs*

Таблица содержит имя события, соответствующий метод, делегат, используемый при установке обработчика события, и параметры обработчика события и метода.

Бы можете предположить (я тоже так думал), что метод *OnPaint* по существу является предустановленным обработчиком события *Paint*. Но это не так. Метод *OnPaint* класса *Control* отвечает за вызов всех установленных обработчиков события *Paint*.

Давайте затронем эту концепцию. Показанный ранее класс *HelloWorld* является наследником *Form*, а созданный нами класс *InheritHelloWorld* будет наследовать классу *HelloWorld*.

InheritHelloWorld.cs

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

class InheritHelloWorld: HelloWorld
{
    public new static void Main()
    {
        Application.Run(new InheritHelloWorld());
    }
    public InheritHelloWorld()
    {
        Text = "Inherit " + Text;
```

см. след. стр.

```
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        grfx.DrawString("Hello from InheritHelloWorld!",
                        Font, Brushes.Black, 0, 100);
    }
}
```

Сначала решим «хозяйственные» вопросы. При создании проекта InheritHelloWorld в Visual Studio .NET я создал новый файл C# InheritHelloWorld.cs как обычно, но, кроме того, потребовалось включить в проект HelloWorld.cs. Для этого я выбрал пункт меню Add Existing Item и затем — Link File в раскрывающемся меню рядом с кнопкой Open. Поэтому не потребовалось создавать копию файла HelloWorld.cs.

Заметьте: метод *Main* содержит ключевое слово *new,* означающее, что он замещает методы *Main,* определенные в классах-предках, например в *HelloWorld.* Кроме того, вы должны сообщить Visual Studio .NET, какой из методов *Main* является точкой входа в программу. Это делается в диалоговом окне Properties. В General | Common Properties, укажите Startup Object InheritHelloWorld.

При запуске компилятора C# из командной строки укажите в командной строке оба исходных файла, а также ключ компилятора:

```
/Main: InheritHelloWorld
```

чтобы определить, метод *Main* какого класса является точкой входа в программу.

Как я уже говорил, при создании нового объекта, относящегося к унаследованному классу, при помощи конструктора по умолчанию выполняются все конструкторы по умолчанию всех классов-предков, начиная с *Object.* На предпоследнем этапе этого процесса вызывается конструктор *HelloWorld,* присваивающий свойству *Text* формы значение «Hello World». И, наконец, последним выполняется конструктор *InheritHelloWorld,* присваивающий значение свойству *Text:*

```
Text = "Inherit " + Text;
```

То, что в заголовке этой программы написано «Inherit Hello World», подтверждает, что конструкторы выполняются именно в такой последовательности.

Метод *OnPaint* класса *InheritHelloWorld* переопределяет метод *OnPaint* класса *HelloWorld.* При запуске *InheritHelloWorld* показывается текст «Hello from InheritHelloWorld!». Я специально расположил этот текст в точке с координатами (0, 100), чтобы было видно, что метод *HelloWorld.OnPaint* при этом не выполняется. Метод *OnPaint* класса *HelloWorld* переопределен.

Теперь взглянем на программу, делающую немного другую вещь. В ней не определяется класс, унаследованный от *HelloWorld,* а создается экземпляр класса *HelloWorld.*

InstantiateHelloWorld.cs

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

class InstantiateHelloWorld
{
     public static void Main()
     {
          Form form  - new HelloWorld( ) ;
          form.Text   = "Instantiate" + form.Text;
          form.Paint += new PaintEventHandler(MyPaintHandler);

          Application.Run(form);
     }
     static void MyPaintHandler(object objSender, PaintEventArgs pea)
     {
          Form     form - (Form)objSender;
          Graphics grfx = pea.Graphics;

          grfx.DrawString("Hello from InstantiateHelloWorld! ",
                          form.Font, Brushes.Black, 0, 100);
     }
}
```

Во-первых, обратите внимание, что класс *InstantiateHelloWorld* не наследует *HelloWorld*, или *Form,* или любому другому классу (конечно же, кроме *Object*):

```
class InstantiateHelloWorld
```

Вместо этого создается новый экземпляр класса *HelloWorld*, который помещается в переменную *form,* как в предыдущих программах этой главы, создававших экземпляры класса *Form;*

```
Form form = new HelloWorld();
```

Программа может помещать объект *HelloWorld* в переменную типа *Form,* потому что *HelloWorld* является наследником *Form.* При создании объекта *HelloWorld* вызывается конструктор *HelloWorld*, присваивающий свойству *Text* формы значение «Hello World». Следующий оператор добавляет в начало свойства *Text* слово «Instantiate». Затем программа устанавливает для формы обработчик события *Paint.*

Но в клиентской области InstantiateHelloWorld появится *не* «Hello from InstantiateHelloWorld!», а текст «Hello, Windows Forms!», отображаемый методом *OnPaint* класса *HelloWorld.* Что произошло?

Метод *OnPaint* класса *Control* отвечает за вызов всех установленных обработчиков события *Paint.* Так как класс *HelloWorld* перекрывает *OnPaint,* эта работа не выполняется. Поэтому в документации .NET рекомендуется: если вы переопреде-

ляете защищенный метод с префиксом *On,* следует вызвать метод *On* базового класса так:

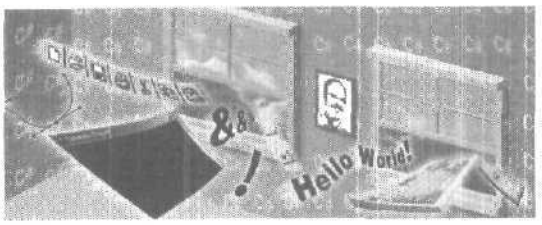
```
base.OnPaint(pea)
```

Попробуйте поместить этот оператор в начале метода *OnPaint* программы HelloWorld и заново собрать InstantiateHelloWorld. Программа будет работать так, как вы, вероятно, и хотели: показывать свою текстовую строку («Hello from InstantiateHelloWorld!») и, кроме того, показывать строку «Hello, Windows Forms!».

В усовершенствованной версии события будут происходить в таком порядке.

- Когда клиентская область становится недействительной, вызывается метод *OnPaint.* Это метод класса *HelloWorld,* переопределяющий метод *OnPaint,* заданный в классах-предках.
- Метод *OnPaint* из HelloWorld вызывает метод *OnPaint* своего базового класса. (Не забывайте, что я говорю об усовершенствованной версии HelloWorld, в которой имеется вызов *base.OnPaint.*) Была бы логичной реализация метода *OnPaint* в *Form,* но, вероятно, *Form* не переопределяет метод *OnPaint,* и на самом деле вызывается метод *OnPaint* класса *Control.*
- Метод *OnPaint* класса *Control* вызывает все установленные обработчики события *Paint.* В данном случае единственным обработчиком является метод *MyPaintHandler* из InstantiateHelloWorld, который выводит текст в точке с координатами (0, 100).
- После вызова всех установленных обработчиков события *Paint* метод *OnPaint* класса *Control* возвращает управление методу *OnPaint* класса HelloWorld.
- Метод *OnPaint* из HelloWorld выводит текст в точке с координатами (О, О).

Документация по Windows Forms рекомендует вызывать метод *On* базового класса, если переопределяется метод *On.* Однако обычно это требуется, только когда вы определяете класс, экземпляры которого собираетесь создавать, и этот класс устанавливает обработчики событий для переопределяемых вами методов *On.* Так бывает, но не очень часто. Но иногда все-таки приходится вызывать методы базового класса в методах, переопределяющих методы *On.* Как вы узнаете в главе 3, одним из таких случаев является использование *OnResize.*



ГЛАВА

3

Основные структуры

Компьютеры первоначально были предназначены для выполнения числовых вычислений и перемалывания чисел — это по-прежнему то, что компьютеры делают лучше всего. Практически все языки программирования имеют механизмы для хранения чисел в переменных, выполнения арифметических операций, организации циклов по диапазону чисел, сравнения чисел и отображения чисел в читабельном виде,

Для многих языков следующим шагом после чисел стала работа с текстом в виде строк символов. Эти строки предназначены главным образом для того, чтобы позволять компьютерным программам взаимодействовать с пользователями. Конечно, внутри компьютера символы представляются, как и все остальное, числами.

Основная задача объектно-ориентированного программирования — сделать так, чтобы типы данных, выходящие за рамки стандартных чисел и строк символов, легко определялись и с ними было легко работать. Мы уже видели несколько примеров этой легкости, в частности класс *Form,* который трудно представить в виде числа или строки символов.

Довольно часто в программировании для графических сред используют четыре типа данных;

- точки с двумерными координатами;
- двумерные размеры — ширина и высота;
- прямоугольники;
- цвета.

Эти четыре типа данных находятся в центре внимания данной главы.

Классы и структуры

Эти четыре типа данных — на самом деле их семь, так как три из них реализованы и в целочисленной форме, и в форме с плавающей точкой, — определены в пространстве имен *System.Drawing.* Интересно, что эти семь типов данных *не* реа-

лизованы как классы. Они являются структурами, а не классами, и это семь из восьми структур определенных в *System.Drawing*.

Структурный тип данных (определяемый ключевым словом *struct)* пришел а C# из C и C++. (В Java нет структурного типа данных.) В C++ классы и структуры очень схожи. В структурах C++ все методы и поля по умолчанию открыты, а в классах C++ — по умолчанию закрыты. Конечно, можно использовать ключевые слова *public*, *private* или *protected* для изменения видимости любого метода или поля в структурах и классах C++,

Классы и структуры в C# также очень схожи, но у них другие отличия, чем в C++. В C# все методы и поля, а также свойства и события по умолчанию закрыты и в классах, и в структурах. Однако классы — это *ссылочные типы* (reference type), а структуры — это *размерные типы* (value type). В чем же различие? Давайте рассмотрим выражение:

```
new Form()
```

Этот оператор выделяет блок памяти в области универсальной памяти, известной как *куча* (heap). Этот блок памяти должен быть достаточно большим, чтобы вместить экземпляр объекта *Form,* значит, он должен быть достаточно большим, чтобы вместить все поля экземпляра *Form* (т. е. нестатические поля). Если вы посмотрите документацию по .NET, то занимаемое объектом пространство покажется небольшим, но имейте в виду, что вы не видите *закрытые* (private) поля, а этих полей, несомненно, много.

Значение, возвращаемое выражением *new*, по существу является ссылкой на блок памяти, расположенный в куче. Эта ссылка на память и сохраняется в переменной *form* оператором вида:

```
Form form = new Form();
```

То есть *ссылочный тип* означает: объект является указателем (ссылкой) на блок памяти,

Допустим, имеются такие операторы:

```
Form form2 = form;
form2.Text = "Form 2 Text";
```

Вставим их в программу PaintEvent из главы 2, прямо перед вызовом *Application.Run*. Что произошло? У отображаемой нами формы, на которую ссылается *form,* появился заголовок «Form 2 Text». Почему это случилось? Потому что *form* является ссылкой. Оператор:

```
Form form2 = form;
```

просто копирует этот указатель в *form2*. Данный оператор *не* создает новый экземпляр класса *Form*. Переменные *form* и *form2* идентичны, т. е. ссылаются на один и тот же блок памяти и поэтому обращаются к одному и тому же объекту.

Понятно, что в повсеместном присваивании указателей нет нужды. Рассмотрим такую последовательность операторов:

```
tnt a = 5;
tnt b = a;
в = 10;
```

Вы не хотели бы, чтобы это означало, что *a* и *b* являются одинаковыми указателями, которые всегда ссылаются на одно и то же число, и что *b* равно 10! Это было бы абсурдом. Поэтому числа в C# являются *размерными типами*. Переменная любого числового размерного типа *не* является указателем на область памяти, в которой хранится число, а содержит само число.

В документации пространства имен *System* вы обнаружите, что большинство базовых типов — *Boolean, Byte, Char, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32, UInt64* — определены как структуры, а не как классы. Структуры унаследованы от *ValueType*, который в свою очередь унаследован от *Object*. Вы можете думать о размерных типах, как о «легковесных объектах», и действительно, использовать *struct* надо только для типов, которые имеют небольшие размеры и могут часто создаваться и уничтожаться.

Двумерные координаты

Координатная точка — тип данных, довольно распространенный в графических средах. Он невелик, поэтому для него больше подходит использование структуры, а не класса. В .NET Framework это структура *Point*. В двумерной координатной системе (такой как поверхность видеодисплея или лист бумаги принтера) точка обозначается обычно парой чисел (x, y), где x — горизонтальная координата, а y ~ вертикальная координата. В главе 2 я рассказал о системе координат, определенной в Windows Forms, но тип данных *Point* не обязательно относится к какой-нибудь конкретной координатной системе. Можно использовать *Point* в любой двумерной системе координат.

Структура *Point* имеет два доступных для чтения и записи свойства, *X* и *Y*, которые определены, как 32-битные целые числа. *X* и *Y* могут быть отрицательными. Хотя *Point* — структура, а не класс, вы не сможете просто определить переменную типа *Point* и присвоить значения этим двум свойствам:

```
Point pt;
pt.X = 23;          // Компилятор сообщит об ошибке!
pt.Y = 47;
```

Вы получите сообщение компилятора «Use of unassigned local variable» (использование неопределенной локальной переменной). По-прежнему нужно использовать *new* для создания экземпляра структуры или класса. При объявлении:

```
Point pt = new Point();
```

свойствам *X* и *Y* присваивается начальное значение 0. Затем им можно присвоить конкретные значения:

```
pt.X = 23;
pt.Y = 47;
```

Или использовать такое объявление для инициализации свойств:

```
Point pt = new Point(34, 55);
```

Или, в редких случаях побитовой упаковки в .NET Framework, можно указать две координаты как 16-битные значения, объединенные в 32-битное целое, например:

```
Point pt = new Point(0x01000010);
```

В результате этого объявления свойство *X* получит значение 16 (0x0010), а свойство *Y* — 256 (0x0100). Я не советую рассматривать точки как 32-битные целые; такое объявление в основном полезно людям, вынужденным продолжать использовать функции Win32 API, иногда работающие с упакованными координатами.

Единственный случай, когда можно обойтись без *new,* — использование метода, свойства или поля, возвращающего *Point.* На самом деле структура *Point* заключает один такой элемент в себе. Это статическое поле *Empty:*

```
Point pt = Point.Empty;
```

Заметьте, что *Point c* заглавной буквы в правой части оператора означает саму структуру *Point,* а не экземпляр структуры *Point.* Указать тип *Point* необходимо, так как поле *Empty* — статическое. В результате выполнения этого оператора *X* и *Y* получат нулевые значения. Кроме того, в *Point* имеется неизменяемое свойство *IsEmpty,* возвращающее *true,* если оба свойства *X и* У равны 0.

Вот полный список свойств *Point:*

Свойства *Point*

Тип	Свойства	Доступ
int	*X*	Чтение/запись
int	*Y*	Чтение/запись
bool	*IsEmpty*	Чтение

Point наследует метод *GetType* от *Object,* переопределяет методы *GetHashCode, ToString, Equals* из *Object,* унаследованные от *Object* через *ValueType,* и содержит свой метод *Offset.* Вот полный список открытых экземплярных (т. е. нестатических) методов структуры *Point:*

Экземплярные методы *Point*

```
Type GetType()
int GetHashCode()
string ToString()
bool Equals(Point point)
void Offset(int dx, int dy)
```

О трех статических методах структуры *Point* я расскажу попозже.

Метод *ToString* преобразовывает объект *Point* в строку символов, доступную для чтения. Например, после выполнения операторов:

```
Point pt = new Point(5, 201);
string str = pt.ToString();
```

переменной *str* присваивается текстовая строка *{X?5,Y?201}.* Метод *ToString* вызывается *Console.Write, Console.WriteLine* и *String.Format* для преобразования объектов в строки.

Метод *Equals* проверяет, равны ли две точки друг другу, как, например, в операторе:

```
if (pt1.Equals(pt2))
```

Равенство (equality) означает, что свойства *X* и *Y* двух объектов *Point,* равны друг другу. Однако в C# удобнее применять такой оператор равенства:

```
if (pt1 == pt2>
```

Метод *Equals* предназначен для языков, не поддерживающих операторы равенства. Кроме того, со структурами *Point* можно использовать оператор неравенства:

```
if (pt1 != pt2)
```

Метод *Offset:*

```
pt.0ffset(21, -12);
```

в основном является тем же самым, что и прибавление двух смещений к свойствам:

```
pt.X += 21;
pt.Y += -12;
```

Массивы точек

Массивы структур *Point* широко используются в программировании для графических сред. Например, массив структур *Point* может задавать сложную кривую или расположение кнопок калькулятора. Создать массив, скажем, из 23 точек, позволяет оператор:

```
Point[] apt = new Point[23];
```

В C# массивы индексируются, начиная с нулевого элемента. Поэтому в данном случае допускаются элементы массива с *apt[0]* по *apt[22].* При создании массива структур, каждый из элементов массива инициализируется точкой (0, 0).

При создании массива можно инициализировать его элементы, но потребуется немного больше стучать по клавишам, чем при инициализации массива структур В C:

```
Point[] apt = new Point[3] { new Point(25, 50),
                             new Point(43, 32),
                             new Point(27, 8) };
```

Количество инициализаторов должно точно совпадать с размерностью массива. На самом деле размерность массива указывать не обязательно:

```
Point[] apt = new PoInt[] { new Point(25, 50),
                            new Point(43, 32),
                            new Point(27, 8) };
```

Можно даже опустить первое выражение *new.*

```
Point[] apt = { new Point(25, 50),
                new Point(43,32),
                new Point(27, 8) };
```

Структура Size

Структура *Size* очень похожа на *Point,* но вместо свойств *X* и *Y* в ней есть свойства *Width* и *Height:*

Свойства *Size*

Тип	Свойства	Доступ
int	*Width*	Чтение/запись
int	*Height*	Чтение/запись
bool	*IsEmpty*	Чтение

Можно создать новую структуру *Size* так же, как структуру *Point*:

```
Size size = new Size(15, 20);
```

Свойства *Width* и *Height* структуры *Size* могут иметь отрицательные значения. (Причины этого будет можно увидеть дальше в этой главе, когда мы будем изучать прямоугольники.)

Структуры *Point* и *Size* так похожи, что могут создаваться друг из друга. Вот полный список конструкторов *Point:*

Конструкторы *Point*

```
Point()
Point(int xyPacked)
Point(int x, int y)
Point(Size size)
```

А вот полный список конструкторов *Size*:

Конструкторы *Size*

```
Size()
Size(int width, int height)
Size(Point point)
```

Кроме того, можно приводить объект *Point к* объекту *Size* и наоборот:

```
pt = (Point) size;
size = (Size) pt;
```

В структуре *Size* перегружаются операторы сложения и вычитания, поэтому можно складывать два размера, например;

```
size3 = size1 + size2;
```

или вычитать два размера:

```
size2 = size3 - size1;
```

Здесь происходит именно то, что можно предположить: свойства *Width* и *Height* просуммируются или вычтутся по отдельности. Кроме того, можно применять совмещенные операторы присваивания:

```
size2 += size1;
size3 -= size2;
```

В структуре *Point* также перегружаются операторы сложения и вычитания, но я не говорил об этих операторах раньше, так как они тоже используют структуру

Size. Так как сложение двух точек не имеет смысла, единственные объекты, которые можно складывать с объектами *Point* или вычитать из них, — это объекты *Size*:

```
pt2 = pt1 + size;
pt3 = pt2 - size;
pt += size;
pt -= size;
```

Версии с плавающей точкой

В C# поддерживается два типа данных с плавающей точкой: *float* и *double*. Тип данных *double* используется в Windows Forms и GDI+ редко, а *float,* наоборот, часто. Можно удивиться, зачем нужны координаты с плавающей точкой, когда рисование ведется в пикселах, но в главе 7 вы увидите, что можно использовать координатные системы, основанные не на пикселах, а на других единицах.

Структура *PointF* очень похожа на структуру *Point* — только свойства *X* и *Y* принимают значения типа *float,* а не *int*. Аналогично структура *SizeF* очень похожа на структуру *Size* за исключением того, что *Width* и *Height* имеют тип *float*. Структуры *PointF* и *SizeF* поддерживают операторы сложения, вычитания, равенства и неравенства точно так же, как *Point* и *Size*.

Вот ошибка, которую я допускаю примерно раз в неделю:

```
PointF ptf = new PointF();
ptf.X = 2.5;                // Ошибка!
ptf.Y = 3E-2;               // Ошибка!
```

Компилятор предположит, что значения имеют тип *double,* а не *float,* и поэтому сообщит: «Literal of type double cannot be implicitly converted to type 'float'» (литерал типа *double* не может неявно преобразовываться в тип *float)*. Одно из решений — привести значения к типу *float;*

```
ptf.X = (float)2.5;
ptf.Y = (float)3E-2;
```

Более легкий метод — поставить за литералом символ *f*, означающий *float*.

```
ptf.X = 2.5f;
ptf.Y = 3E-2f;
```

Как и у *Point*, у структуры *PointF* есть конструктор, позволяющий инициализировать ее в момент создания:

```
PointF ptf = new PointF(2.5f, 3E-2f);
```

Структура *PointF* не содержит метода *Offset*.

Целые значения *могут* неявно приводиться к *типу float,* поэтому можно присваивать целые значения полям с плавающей точкой:

```
PointF ptf = new PointF(127, 42);
```

Можно привести структуру *Point к* структуре *PointF:*

```
ptf = (PointF)pt;
```

Однако нельзя привести структуру *PointF* к *Point:*

```
pt = (Point)ptf;      // Ошибка!
```

Вместо преобразования *PointF* в *Point* можно использовать один из статических методов, предусмотренных для этой цели в структуре *Point:*

Статические методы *Point*

```
Point Round(PointF ptf)
Point Truncate(PointF ptf)
Point Ceiling(PointF ptf)
```

Например:

```
pt = Point.Round(ptf);
```

Перед методом *Round* должно стоять имя структуры, так как это статический метод.

Метод *Round* округляет свойства *X* и *Y* до ближайшего целого, если дробная часть равна 0,5, выполняется округление до ближайшего большего целого. Метод *Truncate* убирает дробную часть, округляя ее до 0. Например, координаты 0,9 и -0,9 станут равными 0. Метод *Ceiling* округляет величину до следующего большего целого значения, т. е. 0,9 станет 1, а -0,9 — 0.

Аналогично можно неявно преобразовать *Size* в *SizeF,* но для преобразования *SizeF* в *Size* понадобятся следующие методы:

Статические методы *Size*

```
Size Round(SizeF sizef)
Size Truncate(SizeF sizef)
Size CeilingCSizeF sizef)
```

Кроме того, структура *SizeF* содержит два следующих экземплярных метода. Это единственные экземплярные методы, которые не наследуются и не переопределяются *SizeF:*

Экземплярные методы *SizeF*

```
PointF ToPointF()
Size ToSize()
```

Метод *ToSize* аналогичен методу *Truncate.*

Как ни странно, хотя можно приводить структуры *Point* и *Size,* приводить *Point* к *PointF, Size* к *SizeF* и *SizeF* к *PointF,* нельзя привести *PointF* к *SizeF.* Однако *PointF* имеет конструктор, принимающий параметр типа *SizeF.* Ниже для сравнения представлен полный список конструкторов четырех структур:

Сравнение конструкторов

Point	PointF	Size	SizeF
0	*0*	*0*	*0*
(x, y)	*fay)*	*(cx, cy)*	*(cx, cy)*
(size)		*(point)*	*(pointf)*
(xyPacked)			*(sizef)*

Прямоугольник — это точка и размер

Структура *Rectangle* определяет прямоугольник, как комбинацию структур *Point* и *Size*. Идея в том, что *Point* определяет расположение левого верхнего угла прямоугольника, а *Size* — ширину и высоту этого прямоугольника. Это не значит, что его ширина и высота должны быть неотрицательны. Сама структура *Rectangle* не накладывает таких ограничений. Однако в главах 4 и 5 вы увидите, что структура *Rectangle* используется в некоторых функциях рисования, и этим функциям нужны неотрицательные ширина и высота. У структуры *Rectangle* два конструктора:

Конструкторы *Rectangle*

```
Rectangle(Point pt, Size size)
Rectangle(int x, int y, int width, int height)
```

Опытные Windows-программисты, осторожнее используйте этот последний конструктор и в тех случаях, когда прямоугольник определяется четырьмя числами: вторые два числа — не координаты нижнего правого угла прямоугольника!

Вот конструкторы структуры *RectangleF*:

Конструкторы *RectangleF*

```
RectangleF(PointF ptf, SizeF sizef)
RectangleF(float x, float y, float width, float height)
```

Легко заметить, что конструкторы для структур *Rectangle* и *RectangleF* одинаковы во всем, кроме типов данных,

Фактически в целом реализации структур *Rectangle* и *RectangleF* до того похожи, что можно поклясться, что при их создании использовался какой-то шаблон. Помимо типов данных, единственное различие в том, что в структуре *RectangleF* определяется преобразование из *Rectangle* в *RectangleF,* в то время как в структуре *Rectangle* определены три статических (и уже знакомых вам) метода, позволяющих выполнять преобразование из *RectangleF* в *Rectangle*:

Статические методы *Rectangle* (выборочно)

```
Rectangle Round(RectangleF rectf)
Rectangle Truncate(RectangleF rectf)
Rectangle Ceiling(RectangleF rectf)
```

С этого момента я буду говорить только о структуре *Rectangle,* но все сказанное мной будет применимо и к *RectangleF.* Типы данных, связанные с *RectangleF,* — это, конечно, *float, PointF* и *SizeF,* а не *int, Point* и *Size.*

Свойства и методы Rectangle

В структуре *Rectangle* определяется совокупность свойств, которые дают любую желаемую информацию:

Свойства *Rectangle*

Тип	Свойства	Доступ
Point	*Location*	Чтение/запись
Size	*Size*	Чтение/запись
int	*X*	Чтение/запись
int	*Y*	Чтение/запись
int	*Width*	Чтение/запись
int	*Height*	Чтение/запись
int	*Left*	Чтение
int	*Top*	Чтение
int	*Right*	Чтение
int	*Bottom*	Чтение
bool	*IsEmpty*	Чтение

Свойство *Left* возвращает то же значение, что и *X,* а *Top* — то же значение, что и *Y.* Свойство *Right* возвращает значение суммы *X* и *Width,* а *Bottom* — значение суммы *Y* и *Height,* даже если *Width* и *Height* отрицательны. Иначе говоря, *Left* может быть больше, чем *Right,* а *Bottom* — больше, чем *Top.*

Единственными операторами, перегруженными в *Rectangle,* являются операторы равенства (==) и неравенства (!=). Кроме того, в *Rectangle* реализован метод *Equals.* Хотя сложение и вычитание не допускаются в структурах *Rectangle,* есть методы, позволяющие по-разному манипулировать структурами *Rectangle* или создавать новые структуры *Rectangle* из существующих.

Для Windows-программистов, привыкших думать в терминологии «левый, верхний — правый, нижний» есть статический метод, создающий *Rectangle* из двух координат:

Методы *Rectangle* (выборочно)

```
static Rectangle FromLTRB(int xLeft, int yTop, int xRight, int yBottom)
```

Так как это статический метод, перед его именем должно стоять имя структуры. Метод возвращает новый объект *Rectangle,* Вызов:

```
rect = Rectangle.FromLTRB(x1, y1, x2, y2);
```

равносилен:

```
rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);
```

Методы *Offset* и *Inflate* манипулируют структурой *Rectangle* и компенсируют отсутствие в ней операторов сложения и вычитания:

Методы *Rectangle* (выборочно)

```
void Offset(int x, int y)
void Offset(Point)
void Inflate(int x, int y)
void Inflate(Size size)
static Rectangle Inflate(Rectangle rect, int x, int y)
```

Метод *Offset* сдвигает прямоугольник в другое положение. Вызов метода:

```
rect.Offset(23, -46);
```

равносилен:

```
rect.X += 23;
rect.Y += -46;
```

или:

```
rect.Location += new Size(23, -46);
```

Перегруженная версия метода *Offset* принимает параметр *Point* (наверное, было бы правильнее использовать тип *Size):*

```
rect.Offset(pt);
```

Это вызов равносилен:

```
rect.X += pt.X;
rect.Y += pt.Y;
```

или:

```
rect.Location += (Size)pt;
```

Метод *Inflate* воздействует на расположение и размер *Rectangle:*

```
rect.Inflate(x, y);
```

идентичен:

```
rect.X -= x;
rect.Y -= y;
rect.Width  += 2 * x;
rect.Height += 2 * y;
```

Таким образом, прямоугольник становится большим (или меньшим, если параметр отрицателен) во всех направлениях. Центр прямоугольника остается там же. Перегруженная версия *Inflate* использует объект *Size* для указания параметров растяжения по двум осям. Статическая версия создает новый растянутый объект *Rectangle* из имеющегося объекта *Rectangle.*

Следующие методы выполняют объединение и пересечение пары объектов *Rectangle:*

Методы *Rectangle* (выборочно)

```
static Rectangle Union(Rectangle rect1, Rectangle rect2)
static Rectangle Intersect(Rectangle rectl, Rectangle rect2)
void Intersect(Rectangle rect)
```

У метода *Intersect* имеется перегруженная версия, не являющаяся статической. Она используется так:

```
rect1.Intersect(rect2);
```

Это выражение равносильно:

```
rectl = Rectangle.Intersect(rect1, rect2);
```

Остальные уникальные методы *Rectangle* возвращают логические значения:

Методы *Rectangle* (выборочно)

```
bool Contains(Point)
bool Contains(int x, int y)
bool Contains(Rectangle rect)
bool IntersectsWith(Rectangle rect)
```

И, наконец: и в *Rectangle*, и в *RectangleF* очень удобно переопределяется *ToString*. Этот метод возвращает такую строку:

```
{X=12,Y=5,Width=30,Height=10}
```

Форма подходящего размера

Какой размер у вашей формы? Это не личный вопрос! Когда программа создает форму, форма имеет определенный размер и занимает определенное место на экране. Размер и расположение формы не фиксированы, однако если у формы есть рамка изменяемого размера, пользователь может перетаскивать ее, изменяя размеры формы. Если у формы есть заголовок, пользователь может перетаскивать ее в новое место на экране. Для программы было бы полезно знать размер своей формы. В точном знании, *где* форма расположена на экране, пользы мало, но знание это не совершенно неуместно.

У класса *Form* есть не менее 13 свойств — почти все они унаследованы от класса *Control,* — которые отражают эту информацию. Эти свойства, кроме двух, можно изменять, и они позволяют программе изменять размер и расположение формы, Мы видели пример таких изменений в программе FormProperties главы 2.

Вот девять свойств (все они унаследованы классом *Form* от *Control),* которые показывают размер формы и ее расположение на экране:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ	Комментарии
Point	*Location*	Чтение/запись	Относительно экрана
Size	*Size*	Чтение/запись	Размер всей формы
Rectangle	*Bounds*	Чтение/запись	Равно *Rectangle(Location, Size)*
int	*Width*	Чтение/запись	Равно *Size.Width*
int	*Height*	Чтение/запись	Равно *Size.Height*
int	*Left*	Чтение/запись	Равно *Location.X*
int	*Top*	Чтение/запись	Равно *Location.Y*
int	*Right*	Чтение	Равно *Location.X* + *Size.Width*
int	*Bottom*	Чтение	Равно *Location.Y* + *Size.Height*

Вообще список свойств, с которыми обычно приходится иметь дело, можно сократить до четырех чисел: координаты *x* и *y* левого верхнего угла формы относительно левого верхнего угла монитора, высота и ширина формы. Я полагаю, что единственная причина того, что *Right* и *Bottom* доступны только для чтения, состоит в неопределенности того, что должно произойти при присваивании им значений. Что тут прикажете делать: передвигать форму или изменять ее размер?

Хотя вы можете присвоить свойствам *Width* и *Height* какие угодно значения, система накладывает определенные ограничения. Ограничение снизу состоит в том, что размеры формы должны быть достаточны для отображения заголовка, содержащего определенный текст. Ограничение сверху: размер формы не может быть больше, чем при развертывании на весь экран.

Однако не делайте так:

```
Size.Width *= 2;
```

Это присваивание значения свойству свойства. По причинам, не доступным пониманию людей, не разрабатывающих компиляторы, это не допускается.

В классе *Form* определены два дополнительных свойства, относящихся к размеру и расположению:

Свойства *Form* (выборочно)

Тип	Свойства	Доступ
Point	*DesktopLocation*	Чтение/запись
Rectangle	*DesktopBounds*	Чтение/запись

Эти свойства похожи на свойства *Location и Bounds,* но учитывают панель задач (taskbar) Windows. Рабочий стол (desktop) — это часть экрана, не занятая панелью задач. Если панель задач расположена слева, то *DesktopLocation.X* будет меньше, чем *Location.X;* если она сверху, то *DesktopLocation.Y* будет меньше, чем *Location.Y. DesktopBounds* определяются по *DesktopLocation* и стандартному свойству *Size,* на которое не влияет расположение панели задач.

Форма и ее клиентская область

Размеры, о которых я говорил, относятся к форме в целом — вместе с рамкой и заголовком. Как правило, приложение в первую очередь интересует размер клиентской области формы. *Клиентская область* — внутренняя область формы, на которой приложение может рисовать все, что сочтет нужным, при возникновении события *Paint* и размещать элементы управления. В клиентскую область не входят заголовок и рамка формы. Если и форме под заголовком располагается меню (я расскажу о добавлении меню в главе 14), из клиентской области также исключается область, занятая меню. Если форма отображает полосы прокрутки справа и снизу окна (как это сделать, я покажу в главе 4), то из клиентской области также исключаются эти полосы прокрутки.

В классе *Form* только два свойства (изначально реализованные в *Control*), относящихся к размеру клиентской области:

Свойства *Client* (выборочно)

Тип	Свойства	Доступ
Size	*ClientSize*	Чтение/запись
Rectangle	*ClientRectangle*	Чтение

Свойство *ClientSize* показывает высоту и ширину клиентской области в пикселах. Свойство *ClientRectangle* не дает дополнительной информации по сравнению с *CUentSize,* потому что свойства *X* и *Y ClientRectangle* всегда равны 0! *ClientRectangle*

иногда полезно в методах, которым требуются параметры *Rectangle*. Две последние программы этой главы иллюстрируют, как использовать *Rectangle* в этих целях.

Еще раз: избегайте присваивать значения свойствам свойств. Такое присваивание не будет работать:

```
ClientSize.Width += 100;          // Не будет работать!
```

Вместо этого делайте так:

```
ClientSize = new Size(ClientSize.Width + 100, ClientSize.Height);
```

Или так:

```
ClientSize += new Size(100, 0);
```

Следующая программа показывает все 13 свойств, отвечающих за размер и местоположение клиентской области.

FormSize.cs

```
//-- _____ ------------
// FormSize.cs 9 2001 by Charles Petzold
//--------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class FormSize: Form
{
     public static void Main()
     {
          Application.Run(new FormSize());
     }
     public FormSize()
     {
          Text = "Form Size";
          BackColor = Color.White;
     }    protected override void OnMove(EventArgs ea)

     {
          Invalidate();
     }
     protected override void OnResize(EventArgs ea)
     {
          Invalidate();
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx - pea.Graphics;
          string   str  = "location: "       + Location      + "\n"   +
                          "Size: "           t Size          + "\n"   +
                          "Bounds: "          + Bounds       + "\n"   +
                          "Width: "          + Width         *  "\n"  *
```

см. след. стр.

```
                "Height: "          + Height          + "\n"    +
                "Left: "            + Left            + "\n"    +
                "Top; "             + Top             + "\n"    +
                "Right: "           +  flight         * "\n"    +
                "Bottom: "          + Bottom          + "\n\n" +
                "DesktopLocation: " + DesktopLocation + "\n"    +
                "DesktopBounds: "   + DesktopBounds   + "\n\n"
                "ClientSize: "      + CHentSize       + "\n"    +
                "ClientRectangle: " + ClientRectangle;

            grfx. DrawString(str, Font, Brushes.Black, 0, 0);
        }
    }
```

Эта несложная программа знакомит с парой вещей, с которыми мы раньше не сталкивались. Во-первых, переопределяются методы *OnMove* и *OnResize*. Префиксы *On* показывают, что данные методы связаны с событиями:

События *Control* (выборочно)

Событие	Метод	Делегат	Параметр
Move	*OnMove*	*EventHandler*	*EventArgs*
Resize	*OnResize*	*EventHandler*	*EventArgs*

Эти методы вызываются, когда форма передвигается или изменяет размеры. В настоящем программировании метод *OnMove* почти никогда не переопределяется. Программы обычно не заботятся, где им расположиться на экране. Однако переопределение метода *OnResize* встречается довольно часто. Мы увидим массу примеров использования *OnResize* в следующих главах.

В ответ на оба этих события я вызываю простейшую из шести перегруженных версий метода *Invalidate*:

Методы *Invalidate* класса *Control*

```
void Invalidate();
void Invalidate(Rectangle rectInvalid);
void Invalidate(bool bIncludeChildren);
void Invalidate(Rectangle rectInvalid, bool bIncludeChildren);
void Invalidate(Region rgnInvalid);
void Invalidate(Region rgnInvalid, bool bIncludeChildren);
```

Этот метод делает недействительной всю клиентскую область, или ее прямоугольное, или непрямоугольное подмножество; при этом дочерние элементы управления, размещенные в ней, в зависимости от параметра могут делаться или не делаться недействительными. Вызов *Invalidate* информирует Windows, что клиентская область больше не является действительной. В итоге форма получит вызов *OnPaint*, и вы сможете перерисовать клиентскую область. Вызов *Invalidate* — стандартный способ принудительной перерисовки окна программы.

Вызов *OnPaint* не произойдет сразу. Сначала должно выполниться текущее событие, обрабатываемой формой (в данном случае *Resize* или *Move*), и даже по-

том, если имеются другие необработанные события (например, ввод с клавиатуры или мыши), вызов *OnPaint* должен дожидаться своей очереди. Если нужно сразу обновить клиентскую область, после вызова *Invalidate* надо вызвать метод *Update* класса *Control:*

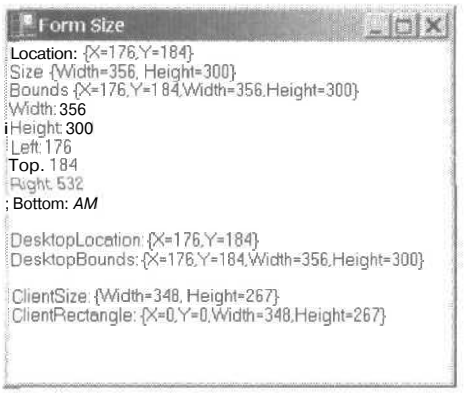
Методы *Control* (выборочно)

```
void Update();
```

Это приведет к немедленному вызову *OnPaint*; после того как *OnPaint* вернет управление, вернет управление и *Update.*

Программа FormSize также показывает объединение строк с объектами *Point*, *Size* и *Rectangle.* Когда вы помещаете свойство или другой объект в строку, как в этой программе, вызывается метод объекта *ToString.*

Кроме того, я задействовал преимущество метода *DrawString*, состоящее в том, что метод распознает символы перевода строки ASCII и правильно устанавливает промежутки между строками текста. И все же вывод немного неряшливый:



В следующей главе мы узнаем, как красиво расположить текст по столбцам.

У объекта *Form* есть четырнадцатое свойство, унаследованное от *Control* и связанное с размером клиентской области:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
Rectangle	*DisplayRectangle*	Чтение

По умолчанию это свойство такое же, как и *ClientRectangle.* Оно не изменяется, пока вы не начнете размещать элементы управления на клиентской области,

Преобразования точек

Как было показано в паре программ главы 2, при рисовании графики на форме используется координатная система, связанная с левым верхним утлом клиентской области. Эти координаты называются *координатами клиентской области* (client area coordinates), Раньше, говоря о расположения формы относительно экрана, я неявно вводил другую координатную систему. Она связана с левым верхним уг-

лом экрана, и такие координаты называются *экранными координатами* (screen coordinates). *Координаты рабочего стола* (desktop coordinates) идентичны координатам экрана, но только если панель задач Windows не расположена на верхней или на левой границе экрана. И, наконец, *координаты формы* (form coordinates) связаны с левым верхним углом формы, который обычно является углом рамки формы.

Свойство *Location* относится к точке в экранных координатах, эквивалентной точке (0, 0) в координатах формы. Таким образом, это свойство позволяет приложению преобразовывать координаты точки из одной координатной системы в другую в виде формул:

$$x_{экр} = x_{формы} + Location.X$$
$$y_{экр} = y_{формы} + Location.Y$$

Аналогично свойство формы *DesktopLocation* позволяет программе преобразовывать друг в друга координаты рабочего стола и координаты формы:

$$x_{раб} = x_{формы} + DesktopLocation.X$$
$$y_{раб} = y_{формы} + DesktopLocation.Y$$

Путем легкой алгебраической манипуляции можно получить формулы преобразования между координатами рабочего стола и экранными координатами:

$$x_{раб} = x_{экр} + DesktopLocation.X - Location.X$$
$$y_{раб} = y_{экр} + DesktopLocation.Y - Location.Y$$

У Form нет аналогичных свойств, позволяющих приложению выполнять преобразования между координатами клиентской области и любой другой из трех других координатных систем. Это можно сделать так: получить с помощью свойства *CaptionHeight* класса *SystemInformation* высоту стандартного заголовка и ширину рамки — сравнив *ClientSize* формы с *Size* всей формы (и вычтя высоту заголовка). Приходится проделывать больше работы, чем хотелось бы.

К счастью, класс *Form* содержит два метода, преобразующих координаты точек непосредственно между координатами экрана и координатами клиентской области:

Методы Form (выборочно)

Метод	Описание
`Point PointToClient(Point ptScreen)`	Преобразует экранные координаты в клиентские.
`Point PointToScreen(Point ptClient)`	Преобразует клиентские координаты в экранные.

Параметр *Point*, передаваемый в эти методы, не изменяется; методы возвращают объект типа *Point*, содержащий преобразованные точки. Например, вызов:

```
Point pt = PointToScreen(Point.Empty);
```

возвращает расположение левого верхнего утла клиентской области в экранных координатах.

Кроме того, класс *Form* поддерживает два дополнительных метода преобразования, работающих с объектами *Rectangle,* а не с объектами *Point.*

Эти методы не обеспечивают дополнительной информации по сравнению с методами *PointToClient* и *PointToScreen*, так как свойство *Size* объекта *Rectangle* не изменяется при преобразовании.

Методы *Form* (выборочно)

Метод	Описание
Rectangle RectangleToClient(Rectangle rectScreen)	Преобразует экранные координаты в клиентские.
Rectangle RectangleToScreen(Rectangle rectClient)	Преобразует клиентские координаты в экранные.

Структура цвета

Человеческий глаз воспринимает электромагнитное излучение в диапазоне 430-750 терагерц, что соответствует длине волны 700-400 нанометров. Электромагнитное излучение этого диапазона называется *видимым светом.* При не очень сильном освещении на интенсивность света реагируют 120 миллионов палочек сетчатой оболочки человеческого глаза, Более сильный свет воздействует на 7 миллионов колбочек, которые делятся на три различных типа, каждый из которых реагирует на различные диапазоны длин волн. Изменение степени возбуждения этих колбочек и есть явление, которое мы называем *цветом,* а три диапазона длин волн соответствуют нашему представлению красного, зеленого и синего,

Так как для определения цвета нужно очень мало данных, цвет претендует на то, чтобы быть структурой, а не классом, и в самом деле *Color* — еще одна важная структура пространства имен *System.Drawing.*

Цвета в Windows Forms основаны на модели ARGB (alpha-red-green-blue). Цвета определяются однобайтовыми значениями красного, зеленого и синего. Альфа-канал определяет прозрачность цвета. Значения alpha лежат в диапазоне от О для прозрачного и до OxFF для непрозрачного.

Структура *Color* имеет один конструктор по умолчанию, который можно использовать так:

```
Color color = new Color();
```

Однако вы почти наверняка не будете использовать код такого вида, так как он создает пустой цвет (прозрачный черный) и невозможно изменить свойства этого цвета. Вместо этого вы будете получать объекты типа *Color,* используя статические методы или свойства, определенные в *Color* для этих целей.

Статические свойства (не менее 141) в *Color* имеют довольно большое значение.

141 название для разных цветов

Структура *Color* имеет 140 статических неизменяемых свойств, которые являются именами цветов от *AliceBlue*[1] до *YellowGreen* (в алфавитном порядке). Только несколько имен (например, *Magenta* и *Fuchsia*) относятся к одинаковым цветам; большинство цветов уникальны. В классе *Color* также есть 141-е свойство — *Transparent,* означающее прозрачный цвет. В таблице показано несколько из 141 свой-

[1] Имя AliceBlue взято от Alice Roosevelt (1884–1980), которая была предприимчивым подростком, когда ее отец стал президентом в 1901, и чей любимый цвет был увековечен в моде и музыке. Более подробно см. *www.theodoreroosevelt.org/life/familytree/AliceLongworth.htm* и *www.theodoreroosevelt.org/life/aliceblue.htm*

ства класса *Color*. К сожалению, невозможно включить все свойства в таблицу, так как такой список занял бы 4 страницы.

Свойства Color

Тип	Свойство	Доступ
Color	*AliceBlue*	Чтение
Color	*AntiqueWhite*	Чтение
	⋮	
Color	*Yellow*	Чтение
Color	*YellowGreen*	Чтение
Color	Transparent	Чтение

Вы сможете найти полный список из 140 стандартных (и иногда имеющих причудливые названия) цветов в файле Cotors.pdf на компакт-диске, прилагаемом к этой книге.

Откуда взялись эти цвета? Они возникли в системе X Window, X11R3 (версия 11, переработка 3), которая являлась графическим пользовательским интерфейсом, разработанным Массачусетским технологическим институтом (MIT) для UNIX, Недавно рассматривался вопрос включения этих цветов в стандарт каскадных таблиц стилей (Cascading Style Sheets — CSS) World Wide Web Consortium (сокращенно — W3C), но до выхода спецификации эти названия цветов удалили. И все же эти 140 цветов фактически стали стандартом в HTML и поддерживаются последними версиями и Microsoft Internet Explorer, и Netscape Navigator.

Всякий раз, когда вам нужен объект *Color*, можете использовать *Color.Red* (или любой другой цвет), и это будет работать. Я уже делал так в нескольких показанных ранее программах, присваивая значение свойству *BackColor* формы и передавая цвет в качестве параметра методу *Clear* класса *Graphics*.

Для создания цвета, основанного на красном, зеленом, синем и альфа-канале, можно задействовать показанные ниже статические методы *Color.FromArgb* каждый из которых возвращает объект *Color*.

Статические методы *Color.FromArgb*

```
Color Color.FromArgb(int r, int g, int b)
Color Color.FromArgb(int a, int r, int g, int b)
Color Color.FromArgb(int a, Color color)
Color Color.FromArgb(int argbPacked)
```

Первый их этих методов используется в программе RandomClear, показанной ниже,

Перья и кисти

Объекты *Color* сами по себе почти не используются в Windows Forms. Вы уже видели, как можно присвоить свойству *BackColor* объект *Color*. Кроме того, аналогично можно присваивать значения свойству *ForeColor*. Метод *Clear* класса *Graphics* также принимает параметр *Color,* но это исключение.

Большинство других методов рисования *Graphics* не используют параметры типа *Color*. При рисовании линий или кривых (мы их рассмотрим в главе 5) использу-

ется объект типа *Pen,* при рисовании залитых областей и текста — объект типа *Brush.* Конечно, для самих перьев и кистей указываются цвета, но часто применяются и другие их свойства.

Перо создается одним из четырех конструкторов класса *Pen.* Простейший из них создает объект *Pen,* используя объект *Color.*

```
Pen pen = new Pen(clr);
```

Чтобы создать перо, использующее один из предопределенных цветов, нет нужды делать так:

```
Pen pen = new Pen(Color.RosyBrown);
```

Вместо этого лучше использовать класс *Pens.* (Обратите внимание на множественное число в имени класса.) *Pens* содержит 141 статическое неизменяемое свойство, и каждое возвращает объект типа *Pen.* Помимо возвращаемого типа, эти свойства идентичны 141 свойству *Color.*

Статические свойства *Pens*

Тип	Свойство	Доступ
Pen	*AliceBlue*	Чтение
Pen	*AntiqueWhite*	Чтение
	⋮	
Pen	*Yellow*	Чтение
Pen	*YellowGreen*	Чтение
Pen	*Transparent*	Чтение

Вы узнаете чуть больше о классе *Pen* в главе 5, а в главе 17 мы обсудим его подробно.

При рисовании текста или залитых областей указывается объект *Brush.* Сам *Brush* является *абстрактным* (abstract) классом, т. е. создать его экземпляр нельзя. Зато *Brush* — родитель пяти других классов: *SolidBrush, HatchBrush, TextureBrush, LinearGradientBrush* и *PathGradientBrush.* Мы более подробно рассмотрим кисти в главе 17. Кисть сплошного цвета можно создать так:

```
Brush brush = new SolidBrush(clr);
```

Можно присвоить результат объекту типа *Brush,* так как *SolidBrush* унаследован от *Brush.*

Как и с классом *Pen,* использование статических свойств *Color* в *SolidBrush* не необходимо, так как есть класс *Brushes* (снова обратите внимание на множественное число), содержащий 141 (да, снова 141!) статическое неизменяемое свойство, возвращающее объекты типа *Brush.*

Статические свойства *Brushes*

Тип	Свойство	Доступ
Brush	*AliceBlue*	Чтение
Brush	*AntiqueWhite*	Чтение
	⋮	
Brush	*Yellow*	Чтение
Brush	*YellowGreen*	Чтение
Brush	*Transparent*	Чтение

Этот класс мы использовали в главе 2, чтобы предоставить черную кисть (*Brushes.Black*) методу *DrawString*. Я упоминал, что следует использовать *Brushes.Black* для текста, только когда вы уверены, что *фон* формы также не является черным. Один из способов добиться этого — явным образом присвоить значение цвету фона формы:

```
BackColor = Color.White;
```

Системные цвета

Причина того, что *Brushes.Black* — не очень хорошая идея для вывода текста, в том, что пользователь Windows может запустить диалоговое окно Display Properties (Свойства: экран) (или из панели управления, или щелкнув правой кнопкой рабочий стол), выбрать вкладку Appearance (Оформление) и выбрать цветовую схему, в которой черные цвета фона окон и элементов управления, например High Contrast Black (Контрастная черная). Люди с плохим зрением или дальтоники часто задают такие высококонтрастные цветовые схемы, и вы определенно не поможете им, если будете отображать свой текст черным!

Добро пожаловать в мир *системных цветов*, который правильнее называть *цветами, выбираемыми пользователем* (user-preference colors). В диалоговом окне свойств дисплея, пользователи могут выбирать свои цветовые схемы. Сама Windows содержит 29 устанавливаемых пользователем цветов, применяемых в качестве цветов различных компонентов пользовательского интерфейса. 26 этих цветов доступны в среде Windows Forms,

Можно получить значения этих цветов из класса *SystemColors*, который состоит из 26 доступных только для чтения свойств, каждое из которых является объектом *Color*.

Статические свойства *SystemColors*

Тип	Свойство	Доступ	Комментарий
Color	*ActiveBorder*	Чтение	Рамка активного окна
Color	*ActiveCaption*	Чтение	Заголовок активного окна
Color	*ActiveCaptionText*	Чтение	Текст заголовка активного окна
Color	*AppWorkspace*	Чтение	Фон рабочего пространства в многодокументном интерфейсе
Color	*Control*	Чтение	Цвет фона элементов управления
Color	*ControlDark*	Чтение	Тень трехмерных элементов управления
Color	*ControlDarkDark*	Чтение	Темная тень трехмерных элементов управления
Color	*ControlLight*	Чтение	Выделение трехмерных элементов управления
Color	*ControlLightLight*	Чтение	Светлое выделение трехмерных элементов управления
Color	*ControlText*	Чтение	Цвет текста элементов управления
Color	*Desktop*	Чтение	Рабочий стол Windows
Color	*GrayText*	Чтение	Отключенный текст
Color	*Highlight*	Чтение	Цвет фона выделенного текста
Color	*HighlightText*	Чтение	Выделенный текст
Color	*HotTrack*	Чтение	След
Color	*InactiveBorder*	Чтение	Рамка неактивного окна

Статические свойства *SystemColors* *(продолжение)*

Тип	Свойство	Доступ	Комментарий
Color	*InactiveCaption*	Чтение	Заголовок неактивного окна
Color	*InactiveCaptionText*	Чтение	Текст заголовка неактивного окна
Color	*Info*	Чтение	Фон подсказки (ToolTip)
Color	*InfoText*	Чтение	Текст подсказки (ToolTip)
Color	*Menu*	Чтение	Фон меню
Color	*MenuText*	Чтение	Текст меню
Color	*ScrollBar*	Чтение	Фон полосы прокрутки
Color	*Window*	Чтение	Фон окна
Color	*WindowFrame*	Чтение	Тонкая рамка окна
Color	*WindowText*	Чтение	Текст окна

Можно создать *Pen* или *Brush* с одним из этих цветов таким образом:

```
Pen pen = new Pen(SystemColor.ControlText);
Brush brush = new SolidBrush(SystemColor.ControlText);
```

Однако обычно этого не делают, так как в пространство имен *System.Drawing* также входят классы *SystemPens* и *SystemBrusbes*. *SystemPens* содержит 15 статических неизменяемых свойств, возвращающих объекты типа *Pen:*

Свойства *SystemPens*

Тип	Свойство	Доступ
Pen	*ActiveCaptionText*	Чтение
Pen	*Control*	Чтение
Pen	*ControlDark*	Чтение
Pen	*ControlDarkDark*	Чтение
Pen	*ControlLight*	Чтение
Pen	*ControlLightLight*	Чтение
Pen	*ControlText*	Чтение
Pen	*GrayText*	Чтение
Pen	*Highlight*	Чтение
Pen	*HighlightText*	Чтение
Pen	*InactiveCaptionText*	Чтение
Pen	*InfoText*	Чтение
Pen	*MenuText*	Чтение
Pen	*WindowFrame*	Чтение
Pen	*WindowText*	Чтение

Класс *SystemBrusbes* содержит 21 статическое неизменяемое свойство, возвращающее объекты типа *Brush:*

Свойства *SystemBrushes*

Тип	Свойство	Доступ
Brush	*ActiveBorder*	Чтение
Brush	*ActiveCaption*	Чтение
Brush	*ActiveCaptionText*	Чтение

Свойства *SystemBrushes* *(продолжение)*

Тип	Свойство	Доступ
Brush	*AppWorkspace*	Чтение
Brush	*Control*	Чтение
Brush	*ControlDark*	Чтение
Brush	*ControlDarkDark*	Чтение
Brush	*ControlLight*	Чтение
Brush	*ControlLightLight*	Чтение
Brush	*ControlText*	Чтение
Brush	*Desktop*	Чтение
Brush	*Highlight*	Чтение
Brush	*HighlightText*	Чтение
Brush	*HotTrack*	Чтение
Brush	*InactiveBorder*	Чтение
Brush	*InactiveCaption*	Чтение
Brush	*Info*	Чтение
Brush	*Menu*	Чтение
Brush	*ScrollBar*	Чтение
Brush	*Window*	Чтение
Brush	*WindowText*	Чтение

Довольно странно, что не все системные цвета, относящиеся к тексту, включены в класс *SystemBrushes*. Однако, если нужны *Pen* или *Brush,* не входящие в классы *SystemPens* или *SystemBrushes,* их всегда можно создать, используя одно из свойств *SystemColors* в качестве параметра одного из этих статических методов:

```
Pen pen = SystemPens.FromSystemColor(SystemColor.ActiveBorder);

Brush brush = SystemBrushes.FromSystemColor(SystemColor.MenuText);
```

Известные цвета

Последний большой список цветов — перечисление *KnownColor*, содержащее все имена цветов и все системные цвета;

Перечисление *KnownColor*

Член	Значение	Член	Значение
ActiveBorder	1	*InactiveCaption*	17
ActiveCaption	2	*InactiveCaption Text*	18
ActiveCaptionText	3	*Info*	19
AppWorkspace	4	*InfoText*	20
Control	5	*Menu*	21
ControlDark	6	*MenuText*	22
ControlDarkDark	7	*ScrollBar*	23
ControlLight	8	*Window*	24
ControlLightLight	9	*WindowFrame*	25

Перечисление *KnownColor* *(продолжение)*

Член	Значение	Член	Значение
ControlText	10	*WindowText*	26
Desktop	11	*Transparent*	27
GrayText	12	*AliceBlue*	28
Highlight	13	*AntiqueWhite*	29
HighlightText	14	...	
HotTrack	15	*Yellow*	166
InactiveBorder	16	*YellowGreen*	167

Хотя *KnownColor* — это третье по величине перечисление .NET Framework, его используют не часто. В классе *Color* есть статический метод, позволяющий создавать цвета, используя значения *KnownColor,* но, если требуется что-то в этом роде, то, вероятно, имеет больший смысл использовать одно из статических свойств *Color* или *SystemColors.*

Какие цвета использовать?

Где-то в конструкторе для класса *Control,* вероятно, выполняется следующий код:

```
BackColor = SystemColors.Control;
ForeColor = SystemColors.ControlText;
```

Что из себя представляют системные цвета *Control* и *ControlText,* полностью зависит от пользователя. Обычно они соответственно серый и черный.

Когда себя рисует, например, кнопка, она задает цвета своего фона через свойство *BackColor,* а свой текст отображает через свойство *ForeColor.* Объект *Form* использует свойство *BackColor* для очистки фона клиентской области, но сам не использует свойство *ForeColor.* Это свойство сделано доступным для приложений, наследующих *Form* или создающих экземпляры *Form.*

Итак, возникает вопрос, какую кисть взять для рисования текста? Думаю, я достаточно убедительно обосновал, что эта кисть — не *Brushes.Black.* Гораздо лучше — *SystemBrushes.ControlText.*

Однако я не уверен, что и этот вариант оптимален. Подумаем; почему свойствам *BackColor* и *ForeColor Form* присваиваются значения системных цветов используемых элементами управления? Ответ таков; разработчики Windows Forms предполагали, что вы будете наносить на форму элементы управления или использовать форму в качестве диалогового окна,

Если вы не помещаете элементы управления на форму и если вы хотите видеть вашу форму похожей на обычные старые программы Windows, поместите следующие два оператора в конструктор формы:

```
BackColor = SystemColors.Window;
ForeColor = SystemColors.WindowText;
```

Б этом случае, чтобы быть последовательным, следует в вызовах *DrawString* в коде *OnPaint* использовать *SystemBrushes.WindowText.*

Но зачем писать код *OnPaint,* зависящий от способа установки значений *BackColor* и *ForeColor* в конструкторе? Чтобы написать универсальный код, в вызове *DrawString* надо использовать кисть:

```
new SolidBrush(ForeColor)
```

Это та кисть, которую я буду использовать в дальнейшем в этой книге, чтобы показывать текст в цвете, выбираемом пользователем при настройке Windows.

Кроме того, пока я не начал создавать элементы управления на формах, я буду присваивать свойствам *BackColor* и *ForeColor* значения *SystemColors.Windou* и *SystemColors.WindowText*, когда знаю, что буду рисовать что-либо зависящее от этих цветов.

Выполнение заливки для перерисовки

Как вы уже видели, фон формы автоматически окрашивается цветом *BackColor*. Кроме того, можно поменять цвет фона формы иначе — методом *Clear* класса *Graphics*. У *Clear* единственный параметр — объект *Color*.

Методы *Graphics* (выборочно)

```
void Clear(Color color)
```

Программа RandomClear случайным образом вычисляет новый цвет всякий раз, когда вызывается метод *OnPaint*, и с помощью метода *Clear* отображает текст новым цветом,

RandomClear.cs

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

class RandomClear: Form
{
     public static void Main()
     {
          Application.Run(new RandomClear());
     }
     public RandomClear()
     {
          Text = "Random Clear"!
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;
          Random   rand = new flandomO;

          grfx.Clear(Color.FromArgb(rand.Next(256),
                                    rand.Next(256),
                                    rand.Next(256)));
     }
}
```

Запустите эту программу и поэкспериментируйте, изменяя размер формы. Вы увидите: при увеличении размера формы, неокрашенные полосы в правой и нижней части формы будут изображаться другим цветом. Каждый новый цвет отображается новым вызовом *OnPaint*. Казалось бы, метод *Clear* должен очищать всю клиентскую область, но вообще-то его действие ограничивается областью, ставшей недействительной. (Заметьте: если уменьшить размер формы, цвета не изменятся, потому что нет части клиентской область, которая стала недействительной.)

Такое поведение не всегда желательно. Может оказаться, что ваша программа должна делать недействительной всю клиентскую область при изменении размера клиентской области. Ранее я показывал, как это сделать, в программе FormSize этой главы: переопределите метод *OnResize* и поместите в него вызов *Invalidate*.

Другое решение — присвоить значение *true* свойству *ResizeRedraw* в конструкторе формы:

```
ResizeRedraw = true;
```

Если *ResizeRedraw* равно *true*, всякий раз, когда размер клиентской области меняется, клиентская область становится недействительной. В следующей программе показаны эти различия,

RandomCiearResizeRedraw.cs

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

class RandomClearResizeRedraw: Form
{
     public static void Main()
     {
          Application. Run(new RandomClearResizeRedraw());
     }
     public RandomClearResizeRedraw()
     Л
          Text * "Random Clear with Resize Redraw";
          flesizeRedraw*true;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;
          flandoei rand = new Random();

          grfx.Clear(Color.FromArgb(rand.Next(256),
                                    rand.Next(256),
                                    rand.Next(256)));
     }
}
```

Я не совсем уверен, стоит ли рекомендовать помещать присваивание *ResizeRedraw* в каждую программу Windows Forms или даже присваивать это свойство для каждого элемента управления. Эта проблема, вероятно, больше проявляется в данной книге, чем в реальной жизни, так как я предпочитаю писать программы, изменяющие свое содержимое в зависимости от размера клиентской области.

Однако запомните: всякий раз, когда что-нибудь на вашей форме некорректно обновляется, надо подумать о присваивании *true* свойству *ResizeRedraw*. А если вы уже установили *ResizeRedraw,* то проблема заключается в чем-то еще.

Перед завершением работы с программой выполните маленькое упражнение. Поместите в RandomClearResizeRedraw пустое переопределение *OnResize*:

```
protected override void OnResize(EventArgs ea)
{
}
```

Вы обнаружите, что эта программа ведет себя в точности, как RandomClear. Очевидно, метод *OnResize,* реализованный в *Control* (и наследуемый *Form*), отвечает за то, чтобы делать элемент управления недействительным в зависимости от его стиля. Вероятно, *OnResize* содержит такой код:

```
if (ResizeRedraw)
    Invalidate();
```

По этой и по другим причинам всякий раз при переопределении метода *OnResize* нужно вызывать метод *OnResize*, реализованный в базовом классе:

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

    // Действия, которые нужно выполнить.
}
```

Центрирование «Hello World»

Где там умник с галерки, который спрашивал о центрировании текста в клиентской области программы? Мы прошли долгий путь и теперь готовы это сделать. Требуется ли в этой программе присваивать значение *true* свойству *ResizeRedraw?* Да, безусловно, потому что *центр* клиентской области зависит от ее общего *размера*.

Один из возможных подходов — изменить координаты точки в функции *DrawString*. Вместо:

```
grfx.DrawString ("Hello, world!", Font, brush, 0, 0);
```

следует использовать

```
grfx.DrawString ("Hello, world!", Font, brush,
                 ClientSize.Width / 2, ClientSize.Height / 2);
```

Это хорошо для начала, но не совсем правильно. Запомните, что координата точки, передаваемая *DrawString*, указывает позицию верхнего левого угла текстовой

строки, так что в центре клиентской области будет находиться левый верхний угол. Текст не будет располагаться в центре клиентской области — он появится в левом верхнем углу правого нижнего квадранта клиентской области.

Можно изменить это поведение по умолчанию, используя перегруженную версию *DrawString*, содержащую еще один параметр наряду с текстовой строкой, шрифтом, кистью и начальной позицией. Дополнительный параметр — объект типа *StringFormat*предназначен для более точного указания того, как отображать текст.

Мы подробно рассмотрим *StringFormat* в главе 9- Пока что сосредоточимся на наиболее часто используемой возможности *StringFormat*, изменяющей выравнивание текста по умолчанию. Выравнивание текста по умолчанию — правило, утверждающее, что координаты точки, передаваемые *DrawString*, указывают положение *левого верхнего* угла отображаемого текста.

Для изменения выравнивания текста по умолчанию сначала создается объект типа *StringFormat:*

```
StringFormat strfmt = new StringFormat();
```

Затем, чтобы указать расположение текста, присваиваются значения двум свойствам этого объекта:

Свойства *StringFormat* (выборочно)

Тип	Свойство	Доступ	Описание
StringAlignment	*Alignment*	Чтение/запись	Выравнивание по горизонтали.
StringAlignment	*LineAlignment*	Чтение/запись	Выравнивание по вертикали.

Оба этих свойства имеют тип *StringAlignment.* Это перечисление, состоящее из трех членов со странными названиями:

Перечисление *StringAlignment*

Член	Значение	Описание
Near	0	Обычно левый или верхний
Center	1	Всегда центр
Far	2	Обычно правый или нижний

Ну, ладно, здесь только два члена перечисления имеют странные имена. Windows Forms может показывать текст справа налево, что вполне нормально в некоторых языках, или вертикально, что также нормально в некоторых языках. *Near* и *Far* предназначены для того, чтобы быть нейтральными по отношению к ориентации, и означают «ближайший к началу текста» и «наиболее удаленный от начала текста».

Если известно, что текст имеет горизонтальную ориентацию и показывается слева направо, можно считать, что *StringAlignment.Near* означает «левый», а *StringAlignment.Far* — «правый» при использовании со свойством *Alignment* и что *StringAlignment.Near* означает «верхний», а *StringAlignment.Far* — «нижний» при использовании со свойством *LineAlignment,* Вот программа, где я применил все четыре комбинации этих параметров, чтобы показывать текстовые строки в четырех углах клиентской области.

FourCorners.cs

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

class FourCorners: Form
{
     public static void Main()
     {
          Application.Run(new FourCorners());
     }
     public FourCorners()
     {
          Text = "Four Corners Text Alignment";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw = true;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics     grfx   - pea.Graphics; •'
          Brush        brush  - new SolidBrush(ForeColor);
          StringFormat strfmt = new StringFormat();

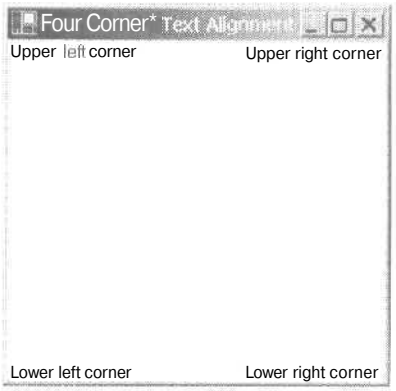
          strfmt.Alignment     = StringAlignment.Near;
          strfmt.LineAlignment = StringAlignment.Near;
          grfx.DrawString("Upper left corner", Font, brush, 0, 0, strfmt);

          strfmt.Alignment     - StringAlignment.Far;
          strfmt.LineAlignment - StringAlignment.Near;
          grfx.DrawString("Upper right corner", Font, brush,
                          ClientSize.Width, 0, strfmt);

          strfmt.Alignment     = StringAlignment.Near;
          strfmt.LineAlignment = StringAlignment.Far;
          grfx.DrawString("Lower left corner", font, brush,
                          0, ClientSize.Height, strfmt);

        • strfmt.Alignment     = StringAlignment.Far;
          strfmt.LineAlignment - StringAlignment.Far;
          grfx.DrawString("Lower right corner", Font, brush,
                          ClientSize.Width, ClientSize.Height, strfmt);
     }
}
```

Координаты точек, передаваемые в четырех вызовах *DrawString*, соответствуют четырем углам клиентской области. Вот как выглядит окно программы:



Если не использовать *StringFormat*, то только первый вызов *DrawString* покажет видимый текст. Текст, отображаемый тремя другими вызовами, окажется целиком за пределами клиентской области.

Если присвоить свойству *Alignment* объекта *StringFormat* значение *StringAlignmentCenter*, строка расположится так, что центр по горизонтали будет соответствовать координате *x*, передаваемой *в DrawString*. Если присвоить свойству *LineAlignment* значение *StringAlignment.Center*, координата *y*, передаваемая в *DrawString*, будет указывать точку, в которой расположится центр строки по вертикали,

Вот как используются эти два эффекта для центрирования текста в клиентской области,

HelloCenteredAlignment.cs

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

class HelloCenteredAlignment: Form
{
     public static void Main()
     {
          Application.Run(new HelloCenteredAlignment());
     >
     public ftelloCenteredAligraifentO
     {
          Text = "Hello Centered Using String Alignment";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
        , flesizeRedraw =    true;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics     grfx   = pea.Graphics;
```

см. след.стр

```
            StringFormat strfmt = new StringFormat();

            strfmt.Alignment     = StringAlignment.Center;
            strfmt.LineAlignment = StringAlignment.Center;

            grfx.DrawString("Hello, world!", Font, new SolidBrush(ForeColor),
                            ClientSize.Width / 2, ClientSize.Height / 2,
                            strfmt);
        }
    }
```

А вот как выглядит окно программы:



Я собираюсь предупредить вас об одной вещи, которая, возможно, звучит глупо, но мой опыт настойчиво требует это сделать. Часто, добавляя определение *StringFormat* в существующий код, я делаю все необходимое, кроме указания объекта в качестве последнего параметра *DrawString.* Так как *DrawString* не требует *StringFormat,* программа компилируется без ошибок, но наличие *StringFormat* не влияет на ее работу. На самом деле нужно добавить объект *StringFormat* в вызов *DrawString.* чтобы все работало правильно!

Измерение строки

Еще один способ центрирования текста — более общий метод позиционирования текста — не требует класса *StringFormat,* но вместо этого использует метод класса *Graphics — MeasureString.* У него семь версий, вызов самой простой из которых выглядит так:

```
SizeF sizefText = grfx.MeasureString(str, font);
```

MeasureString возвращает структуру *SizeF,* показывающую ширину и высоту строки в пикселах (как вы узнаете в главе 7. можно получить размер и в других единицах измерения). *MeasureString* — несомненно, второй по важности метод, используемый при отображении текста, — не настолько важный, как *DrawString,* но без него тоже не обойтись. Я расскажу подробнее о *MeasureString* в главе 9.

Представьте себе отображенную текстовую строку, потом — прямоугольник, нарисованный вокруг этого текста. Структура *SizeF,* возвращаемая *MeasureString,*

содержит ширину и высоту этого прямоугольника. Для данного шрифта независимо от строки символов свойство *Height* структуры *SizeF* всегда одно и то же. (На самом деле свойство *Height обычно* не зависит от строки символов. Если строка содержит символы перевода строки, *Height* является высотой нескольких строк и, следовательно, получается суммированием свойств *Height* отдельных строк.)

Свойство *Width* структуры *SizeF* зависит от символов, содержащихся в строке. Для всех шрифтов, кроме моноширинного, ширина строки текста «i» меньше, чем ширина «W», и *MeasureString* отражает это отличие,

Мы будем широко использовать *MeasureString* в этой книге. Чтобы центрировать текст в клиентской области, можно вычесть свойства *Width* и *Height* структуры *SizeF,* возвращаемые *MeasureString,* из ширины и высоты клиентской области. Эти две разности представляют собой общий горизонтальный и вертикальный отступ текста. Разделив каждое значение на 2, получаем положение левого верхнего угла строки. Вот полный код:

HetloCenleredMeasured.cs

```
//-------------------------
// HelloCenteredMeasured.cs © 2901 by Charles Petzold
//--          -------------
using System;
using System.Drawing;
using System.Windows.Forms;,

class HelloCenteredMeasured: Form
{
     public static void Main()
     {
          Application.Run(new HelloCenteredMeasured());
     }
     public HelloCenteredMeasured()
     {
          Text = "Hello Centered Using MeasureString";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw = true;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx      = pea.Graphics;
          string   str       = "Hello, world!";
          SizeF    sizefText = grfx.MeasureString(str, Font);

          grfx.DrawString(str, Font, new SolidBrush(ForeColor),
                          (ClientSize.Width  - sizefText.Width)  / 2,
                          (ClientSize.Height - sizefText.Height) / 2);
     }
}
```

Текст в прямоугольнике

Мы уже видели две версии метода *DrawString*. Всего их шесть:

Методы *Drawstring* класса *Graphics*

```
void DrawString(string str, Font font, Brush brush, PointF ptf)
void DrawString(string str, Font font, Brush brush, float x, float y)
void DrawString(string str, Font font, Brush brush, RectangleF rectf)
void DrawString(string str, Font font, Brush brush, PointF ptf,
                StringFormat sf)
void DrawString(string str, Font font, Brush brush, float x, float y,
                StringFormat sf)
void DrawString(string str, Font font, Brush brush, RectangleF rectf,
                StringFormat sf)
```

Можно видеть, что первые три параметра всегда одинаковы. Различия лишь в том, указываются ли координаты при помощи структуры *PointF,* двух значений *float* или *RectangleF* и имеется ли параметр типа *StringFormat.*

Использовать структуру *PointF* или значения *float* — дело вкуса. Эти два метода выполняют одинаковые функции. Применяйте тот, который считаете удобнее.

Однако версия с *RectangleF* немного отличается от этих двух методов. Метод *DrawString* ограничивает текст прямоугольником, и дополнительный параметр *StringFormat* определяет, как текст располагается внутри прямоугольника. Например, если *ClientRectangle* передастся функции *DrawString* и свойства *Alignment* и *LineAlignment StnngFormat* имеют значения *StringAlignmenCenter,* текст размещается по центру клиентской области. как продемонстрировано в следующей программе.

HelloCenteredRectangle.cs

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

class HelloCenteredRectangle: Form
{
     public static void Main()
     {
          Application.Run(new HelloCenteredRectangle());
     }
     public HelloCenteredRectangle()
     {
          Text = "Hello Centered Using fiectangle";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw * true;
     }
```

см. след. стр.

```
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics     grfx   = pea.Graphics;
        StringFormat strfmt = new StringFormat();

        strfmt.Alignment     = StringAlignment.Center;
        strfmt.LineAlignment = StringAlignment.Center;

        grfx.DrawString("Hello, world!", Font, new SolidBrush(ForeColor),
                        ClientRectangle, strfmt);
    }
}
```

Возможность выбора прямоугольника, а не одной координатной точки для расположения текста может вызвать удивление. Как вы считаете, может ли *DrawString переносить строки* внутри прямоугольника?

Есть только один путь это выяснить. Заменим короткую строку которую мы использовали, на что-нибудь более солидное — в качестве подходящего примера мне пришел в голову первый абзац книги Марка Твена «Приключения Гекльберри Финна» — и посмотрим, что получится,

HuckleberryFinn.cs

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

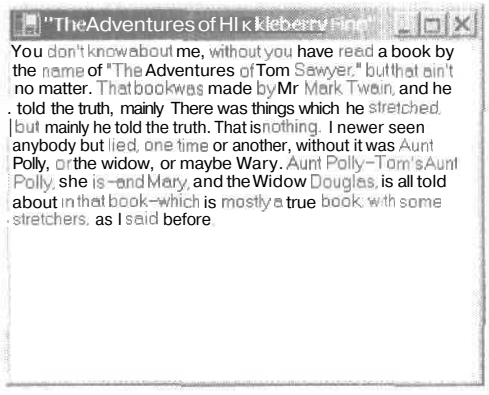
class HuckleberryFinn: Form
{
    public static void Main()
    {
        Application.Run(new HuckleberryFinn());
    }
    public HuckleberryFinn()
    {
        Text = "\"The Adventures of Huckleberry Finn\"";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        pea.Graphics.DrawString(
            "You don't know about me, without you " +
            "have read a book by the name of \"The " +
            "Adventures of Tom Sawyer,\" but that " +
```

см. след. стр.

```
                "ain't no matter. That book was made by " +
                "Mr. Mark Twain, and he told the truth, " +
                "mainly. There was things which he " +
                "stretched, but mainly he told the truth. " + - '
                "That is nothing, I never seen anybody " +
                "but lied, one time or another, without " *
                "it was Aunt Polly, or the widow, or " +
                "maybe Mary. Aunt Polly\x2014Tom's Aunt " +
                "Polly, she is\x2014and Mary, and the Widow " +
                "Douglas, is all told about in that book" +
                "\x2014which is mostly a true book; with " *
                "some stretchers, as 1 said before.",
                Font, new SolidBrush(ForeColor), ClientRectangle);
        }
    }
```

Обратите внимание на символы «\x2014» в строке. Это код символа Unicode *длинное тире* (em dash), которое часто используется для разделения фраз в предложении, некоторые писатели, пожалуй, им злоупотребляют, При вызове *DrawString* в этой программе не нужен параметр *StringFormat*, так как нас интересует обычное отображение текста.

И действительно, *DrawString* неплохо справляется с форматированием текста, заполняющего клиентскую область:



Что не делает метод *DrawString*, так это не создает пару полос прокрутки, если размер клиентской области недостаточен (хотя мы вправе ожидать этого от программы).

Ничего страшного. Вы узнаете, как создавать полосы прокрутки, в следующей главе.

ГЛАВА

4



Упражнение по выводу текста

Иногда клиентскую часть формы называют *презентационной областью,* В ней вы представляете пользователю внешний вид и стиль программы, се особенности, плюсы и минусы. Разумеется, вид клиентской области сильно зависит от предназначения программы. Одни программы — особенно клиентские части распределенных приложений — могут содержать только дочерние элементы управления, такие как кнопки и поля ввода. Другие — внутри своей клиентской части могут полностью прорисовывать интерфейс, управлять вводом с клавиатуры и обрабатывать мышь. Третьи же — как те, что мы рассмотрим в этой главе, — просто выводят на экран некоторую информацию.

Однако даже в случае простого отображения информации часто нужно обеспечить взаимодействие с пользователем. Кроме технологий форматирования текста в приятные глазу аккуратные колонки, мы обсудим способы создания и использования таких замечательных элементов пользовательского интерфейса, как *полосы прокрутки,*

Информация о системе

Когда я сталкиваюсь с новой ОС или средой разработки, например Windows Forms, я часто пишу программы, исследующие систему. Написание программ с единственной функцией — выводом информации (например, FormSize из главы 3) позволяет изучать программирование и саму систему одновременно.

Класс *SystemInformation* в пространстве имен *System.Windows.Forms* на данный момент содержит 60 статических неизменяемых свойств, предоставляющих сведения о компьютере, на котором выполняется программа, и некоторых показателях, используемых ОС для отображения элементов рабочего стола и вашей программы. *SystemInformation* содержит данные о количестве кнопок мыши, размере

значков рабочего стола и высоте строки заголовка формы. В нем также имеются сведения о подключении к сети и имени домена, в котором находится данный компьютер. Вся эта информация представляется данными различного типа — *int*, *bool*, *string*, *Size*, *Rectangle* — и несколькими перечислениями.

В этой главе моей задачей является написание программы, отображающей на экране все эти данные в удобном для восприятия виде. Поскольку вы, вероятно, будете часто обращаться к этой программе, то некоторые дополнительные усилия, требуемые для ее создания, вполне оправданны,

Разделение текстовых строк

В главе 3 вы увидели, что метод *DrawString* правильно отображает несколько строк текста, разделенных символом перевода строки, и вписывает их в прямоугольник. Но в программах, отображающих несколько строк текста, разбитых на колонки, бывает удобнее вызывать *DrawString* отдельно для каждой строки в каждой колонке. Это подразумевает задание конкретной координаты начала каждой строки текста при вызове *DrawString*.

В той же главе я рассказывал о методе *MeasureString* класса *Graphics*. Этот метод возвращает высоту символа в строке. Хотя он и годится для разделения последовательных строк текста, это не очень удобно. Для разделения, согласующегося с возможностью переноса слов в *DrawString*, надо использовать значение, несколько отличающееся от значения высоты символа, возвращаемого *MeasureString*. Здесь мы сталкиваемся со слегка запутанным вопросом, поскольку по именам свойств и методов, предоставляющих правильные значения межстрочных интервалов, можно подумать, что речь идет о высоте символов шрифта! Самым обобщенным методом для получения значения межстрочного интервала является метод *GetHeight* класса *Font:*

```
float cySpace = font.GetHeight(grfx);
```

В имени переменной я использовал префикс *cy* для обозначения количества *(count)* пикселов в направлении оси *y*. т. е. высоты. В данном операторе *font* — это объект типа *Font*, а *grfx* — объект типа *Graphics*. Я назвал этот метод самым обобщенным, так как аргумент *Graphics* позволяет применять его как для монитора, так и для принтера. Метод учитывает все преобразования. (Преобразования позволяют использовать при выводе графической информации не только пикселы, но и другие единицы измерения. Подробнее об этом см. главу 7.) Заметьте: возвращаемое значение имеет тип *float*. Для некоторых шрифтов значение, возвращаемое *GetHeight,* совпадает со значением *MeasureString*. Однако для большинства шрифтов оно немного меньше.

Другая версия метода *GetHeight* предоставляет значение межстрочного интервала, пригодное только для вывода на экран, но не на принтер. Этот метод следует применять только для вывода без использования преобразований:

```
float cySpace = font.GetHeight();
```

Если округлить это значение типа *float* до ближайшего большего целого, вы получите значение, возвращаемое свойством *Height* класса *Font:*

```
int cySpace = font.Height;
```

Если при выводе на экран в качестве единиц измерения используются пикселы (что имеет место по умолчанию), лучше задействовать свойство *Height* класса *Font.*

При использовании метода *GetHeight* или свойства *Height* со шрифтом формы по умолчанию можно просто указать свойство *Font* формы как объект;

```
float cySpace = Font.GetHeight();
int cySpace = Font.Height;
```

В этих операторах префикс *Font* ссылается на свойство *Font.* Класс *Form* содержит защищенное неизменяемое свойство *FontHeight* (унаследованное из класса *Control*), которое возвращает значение типа *int,* согласующееся со свойством *FontHeight.* Хотя теоретически вы можете изменять это свойство, это не приведет к изменению размера шрифта, применяемого в форме по умолчанию.

Значения свойств

Ниже показана первая — пока еще незавершенная — программа, выводящая на экран значения свойств класса *SystemInformation.*

SysEnfoFirst7ry.cs

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

class SysInfoFirstTry: Form
{
     public static void Main()
     {
          Application.Run(newSysInfoFirstTry());
     }
     public SysInfoFirstTry()
     {
          Text = "System Information: First Try";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx  = pea.Graphics;
          Brush    brush * new SolidBrush(ForeColor);
          int      y     = 0;

          grfx.DrawString("ArrangeDirection: " +
                          SystemInformation.ArrangeDirection,
                          Font, brush, 0, y);

          grfx.DrawString("ArrangeStartingPosition: " +
```

см. след. стр.

```
                    SystemInformation.ArrangeStartingPosition,
                    Font, brush, 0, y += Font.Height);

        grfx.DrawString("BootMode: " +
                    SystemInformation.BootMode,
                    Font, brush, 0, y += Font.Height);

        grfx.DrawString("Border3DSize: " +
                    SystemInformation.Border3DSize,
                    Font, brush, 6, y += Font.Height);

        grfx.DrawString("BorderSize: " +
                    SystemInformation.BorderSize,
                    Font, brush, 0, y += Font.Height);

        grfx.DrawString("CaptionButtonSize: " +
                    SystemInformation.CaptionButtonSize,
                    Font, brush, 0, y += Font.Height);

        grfx.DrawString("CaptionHeight: " +
                    SystemInformation.CaptionHeight,
                    Font, brush, 0, y += Font.Height);

        grfx.DrawString("ComputerName: " +
                    SystemInformation.ComputerName,
                    Font, brush, 0, y += Font.Height);

        grfx.DrawString("CursorSize: " +
                    SystemInformation.CursorSize,
                    Font, brush, 0, y += Font.Height);

        grfx.DrawString("DbcsEnabled: " +
                    SystemInformation.DbcsEnabled,
                    Font, brush, 0, y += Font.Height);
    }
}
```

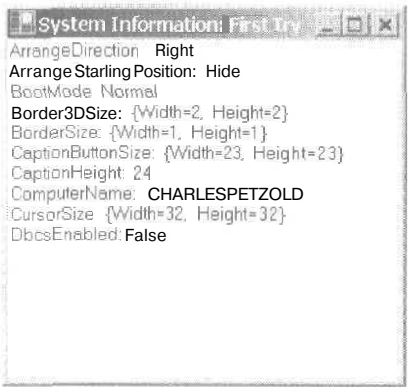
Что ж, я сдался после 10 элементов не потому, что устал печатать. Просто я осознал, что это был не лучший подход и, вероятно, мне придется переписать программу в каком-то другом, более универсальном виде. Хотя и это не так уж плохо для первой попытки (см. рис. на следующей странице).

Давайте взглянем на работу этой программы.

Каждую строку выводит метод *DrawString*. В качестве первого аргумента указывается имя свойства, объединенное с его значением. При конкатенации свойство *SystemInformation* автоматически преобразуется в строку. Для этого без участия программиста вызывается метод *ToString*. Обратите внимание на формат вывода свойств, связанных с размерами: здесь указана ширина и высота. *DrawString* также обеспечивает подходящий межстрочный интервал. При каждом вызове, начиная со второго, значение *y* увеличивается:

```
y += Font.Height
```

При этом следующее значение выводится строкой ниже.



Разбивка на колонки

По-моему, самый серьезный недостаток программы SysInfoFirstTry (не считая ее незавершенности) — форматирование. Такие данные было бы легче читать, если бы значения параметров выводились во второй колонке. Так что, прежде чем продолжить, давайте решим эту проблему.

Из десяти свойств, выводимых SysInfoFirstTry, самое длинное имя у свойства *ArrangeStartingPosition*. Перед выводом информации в этой программе вызывается *MeasureString* с этим именем в качестве аргумента (к имени добавлен один пробел, чтобы разделить колонки),

SysInfoColumns.cs

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

class SysInfoColumns: Form
{
     public static void Main()
     {
          Application.Run(new SysInfoColumns());
     }
     public SysInfoColumns()
     {
          Text = "System Information: Columns";
          BackColor * SystemColors.Window;
          ForeColor = SystemColors.WindowText;
     }
```

см. след. стр

```
protected override void OnPaint(PaintEventArgs pea)
{
     Graphics grfx   - pea.Graphics;
     Brush    brush * new SolidBrush(ForeColor);
     SizeF    sizef;
     float    cxCol, y * 0;
     int      cySpaee;

     sizef    = grfx.MeasureString("ArrangeStartingPosition ", Font);
     cxCol    = sizef.Width;
     cySpace = Font.Height;

     grfx.DrawString("ArrangeDirection", Font, brush, 0, y);
     grfx.DrawString(SystemInformation.ArrangeDirection.ToString(),
                     Font, brush, cxCol, y);
     y += cySpace;

     grfx.DrawString("ArrangeStartingPosition", Font, brush, 0, y);
     grfx.DrawString(
          SystemInformation.ArrangeStartingPosition.ToString(),
          Font, brush, cxCol, y);
     y += cySpace;

     grfx.DrawString("BootMode", Font, brush, 0, y);
     grfx.DrawString(SystemInformation.BootMode.ToString(),
                     Font, brush, cxCol, y);
     y += cySpace;

     grfx.DrawString("Border3DSize", Font, brush, 0, y);
     grfx.DrawString(SystemInformation.Border3DSize.ToString(),
                     Font, brush, cxCol, y);
     y += cySpace;

     grfx.DrawString("BorderSize", Font, brush, 0, y);
     grfx.DrawString(SystemInformation.BorderSize.ToString(),
                     Font, brush, cxCol, y);
     y += cySpace;

     grfx.DrawString("CaptionButtonSize", Font, brush, 0, y);
     grfx.DrawString(SystemInformation.CaptionButtonSize.ToString(),
                     Font, brush, cxCol, y);
     y += cySpace;

     grfx.DrawString("CaptionHeight", Font, brush, 0, y);
     grfx.DrawString(SystemInformation.CaptionHeight.ToString(),
                     Font, brush, cxCol, y);
     y += cySpace;

     grfx.DrawString("ComputerName", Font, brush, 0, y);
     grfx.DrawString(SystemInformation.ComputerName,
```

см. след. стр.

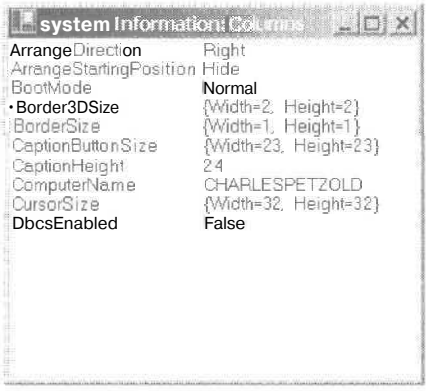
```
                    Font, brush, cxCol, y);
            y += cySpace;

            grfx.DrawString("CursorSize", Font, brush, 0, y);
            grfx.DrawString(SystemInformation.CursorSize.ToString(),
                    Font, brush, cxCol, y>;
            y += cySpace;

            grfx.DrawString("DbcsEnabled", Font, brush, 0, y);
            grfx.DrawString(SystemInformation.DbcsEnabled.ToString(),
                    Font, brush, cxCol, y);
        }
    }
```

Ширина строки сохраняется в переменной *cxCol* и используется при размещении второй колонки. Кроме того, в переменной *cySpace* запоминается значение *Height* объекта *Font,* которое служит для разделения соседних строк. Координаты, передаваемые методу *DrawString,* имеют тип *float.*

Теперь для вывода каждой строки требуется два вызова *DrawString:* первый — для отображения имени свойства, второй — для отображения значения этого свойства. Все эти свойства, кроме одного, теперь требуют явного вызова метода *ToString* для преобразования значений в строковые переменные. Вот как это выглядит:



Вокруг одни объекты

В программе вроде SysInfoColumns фрагмент кода, выводящий значения на экран, вероятно, следует оформить с применением цикла/or. Сами данные скорее всего стоит хранить в массиве и, возможно, отделить их от кода, осуществляющего вывод, так чтобы этот код можно было задействовать в других программах. В программе на C или C++ вы могли бы поместить данные в массив в заголовочном файле, однако в C# нет заголовочных файлов. Здесь надо вспомнить волшебное правило C#: «*вокруг одни объекты*», по крайней мере — классы со статическими методами и свойствами,

Вот как можно использовать класс, содержащий текстовые строки, которые мы хотим вывести на экран, и некоторую информацию о них.

SysInfoStrings.cs

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

class SysInfoStrings
{
     public static string[] Labels
     {
          get
          {
               return new string[]
               {
                    "ArrangeDirection",
                    "ArrangeStartingPosition",
                    "BootMode",
                    "Border3DSize",
                    "BorderSize",
                    "CaptionButtonSize",
                    "CaptionHeight",
                    "ComputerName",
                    "CursorSize",
                    "DbcsEnabled",
                    "DebugOS",
                    "DoubleClickSize",
                    "DoubleClickTime",
                    "DragFullWindows",
                    "DragSize",
                    "FixedFrameBorderSize",
                    "FrameBorderSize",
                    "HighContrast",
                    "HorizontalScrollBarArrowWidth",
                    "HorizontalScrollBarHeight",
                    "HorizontalScrollBarThumbWidth",
                    "IconSize",
                    "IconSpacingSize",
                    "KanjiWindowHeight",
                    "MaxWindowTrackSize",
                    "MenuButtonSize",
                    "MenuCheckSize",
                    "MenuFont",
                    "MenuHeight",
                    "MidEastEnabled",
                    "MinimizedWindowSize",
                    "MinimizedWindowSpacingSize",
                    "MinimumWindowSize",
```

см. след. стр.

```
                "MinWindowTrackSize",
                "MonitorCount",
                "MonitorsSameDisplayFormat",
                "MouseButtons",
                "MouseButtonsSwapped",
                "MousePresent",
                "MouseWheelPresent",
                "MouseWheelScrollLines",
                "NativeMouseWheelSupport",
                "Network",
                "PenWindows",
                "PrimaryMonitorMaximizedWindowSize",
                "PrimaryMonitorSize",
                "RightAlignedMenus",
                "Secure",
                "ShowSounds",
                "SmallIconSize",
                "ToolWindowCaptionButtonSize",
                "ToolWindowCaptionHeight",
                "UserDomainName",
                "UserInteractive",
                "UserName",
                "VerticalScrollBarArrowHeight",
                "VerticalScrollBarThumbHeight",
                "VerticalScrollBarWidth",
                "VirtualScreen",
                "WorkingArea",
            };
        }
    }
    public static string[] Values
    {
        get
        {
            return new string[]
            {
            SystemInformation.ArrangeDirection.ToString(),
            SystemInformation.ArrangeStartingPosition.ToString(),
            SystemInformation.BootMode.ToString(),
            SystemInformation.Border3DSize.ToString(),
            SystemInformation.BorderSize.ToString(),
            SystemInformation.CaptionButtonSize.ToString(),
            SystemInformation.CaptionHeight.ToString(),
            SystemInformation.ComputerName,
            SystemInformation.CursorSize.ToString(),
            SystemInformation.DbcsEnabled.ToString(),
            SystemInformation.DebugOS.ToString(),
            SystemInformation.DoubleClickSize.ToString(),
            SystemInformation.DoubleClickTime.ToString(),
            SystemInformation.DragFullWindows.ToString(),
```

см, след. стр.

```
            SystemInformation.DragSize.ToString(),
            SystemInformation.FixedFrameBorderSize.ToString(),
            SystemInformation.FrameBorderSize.ToString(),
            SystemInformation.HighContrast.ToString(),
            SystemInformation.HorizontalScrollBarArrowWidth.ToString(),
            SystemInformation.HorizontalScrollBarHeight.ToString(),
            SystemInformation.HorizontalScrollBarThumbWidth.ToString(),
            SystemInformation.IconSize.ToString(),
            SystemInformation.IconSpacingSize.ToString(),
            SystemInformation.KanjiWindowHeight.ToString(),
            SystemInformation.MaxWindowTrackSize.ToString(),
            SystemInformation.MenuButtonSize.ToString(),
            SystemInformation.MenuCheckSize.ToString(),
            SystemInformation.MenuFont.ToString(),
            SystemInformation.MenuHeight.ToString(),
            SystemInformation.MidEastEnabled.ToString(),
            SystemInformation.MinimizedWindowSize.ToString(),
            SystemInformation.MinimizedWindowSpacingSize.ToString(),
            SystemInformation.MinimumWindowSize.ToString(),
            SystemInformation.MinWindowTrackSize.ToString(),
            SystemInformation.MonitorCount.ToString(),
            SystemInformation.MonitorsSameDisplayFormat.ToString(),
            SystemInformation.MouseButtons.ToString(),
            SystemInformation.MouseButtonsSwapped.ToString(),
            SystemInformation.MousePresent.ToString(),
            SystemInformation.MouseWheelPresent.ToString(),
            SystemInformation.MouseWheelScrollLines.ToString(),
            SystemInformation.NativeMouseWheelSupport.ToString(),
            SystemInformation.Network.ToString(),
            SystemInformation.PenWindows.ToString(),
            SystemInformation.PrimaryMonitorMaximizedWindowSize.ToString(),
            SystemInformation.PrimaryMonitorSize.ToString(),
            SystemInformation.RightAlignedMenus.ToString(),
            SystemInformation.Secure.ToString(),
            SystemInformation.ShowSounds.ToString(),
            SystemInformation.SmallIconSize.ToString(),
            SystemInformation.ToolWindowCaptionButtonSize.ToString(),
            SystemInformation.ToolWindowCaptionHeight.ToString(),
            SystemInformation.UserDomainName,
            SystemInformation.UserInteractive.ToString(),
            SystemInfrotation.UserUame,
            SystemInformation.VerticalScrollBarArrowHeight.ToString(),
            SystemInformation.VerticalScrollBarThumbHeight.ToString(),
            SystemInformation.VerticalScrollBarWidth.ToString(),
            SystemInformation.VirtualScreen.ToString(),
            SystemInformation.WorkingArea.ToString(),
            };
        }
}
```

см. след. стр.

```
    public static int Count
    {
        get
        {
            return Labels.Length;
        }
    }
    public static float MaxLabelWidth(Graphics grfx, Font font)
    {
        return MaxWidth(Labels, grfx, font);
    }
    public static float MaxValueWidth(Graphics grfx, Font font)
    {
        return MaxWidth(Values, grfx, font);
    }
    static float MaxWidth(string[] astr, Graphics grfx, Font font)
    {
        float fMax = 0;

        foreach (string str in astr)
            fMax = Math.Max(fMax, grfx.MeasureString(str, font).Width);

        return fMax;
    }
}
```

У этого класса три неизменяемых статических свойства. Первое — *Labels* - возвращает массив строк, содержащий имена свойств *SystemInformation*.

Второе — *Values* — также возвращает массив символьных строк, каждая из которых соответствует своему элементу массива *Labels*. Однако на самом деле массив *Values* состоит из набора выражений для свойств *SystemInformation,* каждое из которых представляется в виде строки. Каждое выражение вычисляется в момент вызова свойства. В приведенном листинге вы можете видеть вызовы метода *ToString* для преобразования каждого из значений в строку.

Третье свойство возвращает количество элементов в массиве *Labels,* которое (если, конечно, я нигде не напутал) должно равняться числу элементов в массиве *Values*.

Кроме того, *SysInfoStrings*содержит два открытых метода: *MaxLabelWidth* и *MaxValueWidth* В качестве аргументов им передаются объекты *Graphics* и *Font,* и каждый возвращает максимальную ширину строки в соответствующем массиве, полученную с помощью метода *MeasureString*. В обоих случаях применяется закрытый метод *MaxWidth*, который в свою очередь использует статический метод *Math.Max*для сравнения двух значений и выбора максимального из них. (Класс *Math* в пространстве имен *System* представляет собой набор статических методов, реализующих математические функции. Подробнее о классе *Math* и других особенностях обработки чисел см. Приложение Б.)

Отображение информации о системе

Б проекте SysInfoList я создал файл SysInfoStrings.cs, а также показанный ниже файл SysInfoList.cs. Эта программа отображает сведения о системе, возвращаемые классом *SysInfoStrings*.

SysinfoLisl.cs

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

class SysInfoList: Form
{
     readonly float cxCol;
     readonly int   cySpace;
     public static void Main()
     {
          Application.Run(new   SysInfoListO);
     }
     public SysInfoListO
     {
          Text = "System Information: List";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

          Graphics grfx  - CreateGraphics();
          SizeF    sizef = grfx.MeasureString(" ", Font);
          cxCol  = sizef.Width + SysInfoStrings.MaxLabelWidth(grfx, Font);
          grfx.Dispose();
          cySpace = Font.Height;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx       = pea.Graphics;
          Brush    brush      = new SolidBrush(ForeColor);
          int      iCount     = SysInfoStrings.Count;
          string[] astrLabels = SysInfoStrings.Labels;
          string[] astrValues = SysInfoStrings.Values;
          for (int 1 = 0; i < iCount; i++)
          I
               grfx.DrawString(astrLabels[i], Font, brush,
                               0, i * cySpace);
               grfx.DrawString(astrValues[i], Font, brush,
                               cxCol, i * cySpace);

          1
     }
}
```

Эта программа *немного* оптимизирована, так как значения *cxCol* и *cySpace* вычисляются лишь однажды в конструкторе формы и затем сохраняются в качестве полей, используемых позже методом *OnPaint*. Однако это вычисление требует вызова метода *MeasureStnng,* для чего в свою очередь требуется объект *Graphics.* Как же получить этот объект вне события *Paint* или метода *OnPaint*?

Класс *Control* содержит наследуемый классом *Form* метод *CreateGrapbics,* позволяющий в любой момент получить доступ к объекту *Graphics*:

Методы *Control* (выборочно)

```
Graphics CreateGraphics()
```

Объект *Graphics* можно использовать для получения информации (как это сделано в SysInfoList) или для вывода графики в клиентской области программы. (Однако не стоит беспокоиться о выводе графики во время работы конструктора, так как в это время форма не отображается.)

До возврата управления от конструктора или события, в котором при помощи метода *CreateGrapbics* был создан объект *Graphics,* этот объект нужно уничтожить, вызвав метод *Dispose*:

Методы *Graphics* (выборочно)

```
void Dispose()
```

За созданием объекта *Graphics* методом *CreateGrapbics* в программе SysInfoList идет вызов *MeasureStnng* для вычисления ширины символа «пробел». Полученная величина прибавляется к значению *MaxLabelWidth*, возвращенному классом *SysInfoStrings*, а результат сохраняется в переменной *cxCol*. В методе *OnPaint* находится простой *цикл for,* внутри которого выводятся пары строк.

Мы определенно достигли некоторого прогресса:



Однако в зависимости от установленного на вашем компьютере разрешения экрана (которое мы будем подробно обсуждать в главе 9), вам, возможно, не удастся развернуть форму так, чтобы видеть все 60 значений. В этом случае программа окажется не слишком полезной, так как просмотреть значения, находящиеся в конце списка, невозможно,

Вы можете столкнуться с этой проблемой, даже если речь будет идти всего о 20 или 25 значениях. Возможно, худшей ошибкой в программировании для Windows является предположение о том, что некоторое количество текста будет умещаться на экранах всех пользователей программы. Пользователи — особенно со слабым зрением — могут установить крупный размер шрифта, уменьшив тем самым количество текста, умещающегося на экране. Поэтому в ваших программах надо предусматривать возможность работы при любом разрешении экрана и размере шрифта.

Отобразить больше текста, чем умещается на экране, помогут полосы прокрутки,

Windows Forms и полосы прокрутки

Полосы прокрутки — важная составляющая любой графической среды. Они улучшают наглядность и облегчают работу пользователя. Полосы прокрутки удобны при отображении любых объектов (текста, графики, таблиц, записей баз данных, картинок, Web-страниц), чьи размеры превышают размеры клиентской области программы.

Полосы прокрутки могут быть ориентированы вертикально (для перемещения вверх-вниз) или горизонтально (вперед-назад). Щелчок стрелок на концах полосы прокрутки перемещает документ на небольшое расстояние — обычно на одну строку вверх или вниз (для вертикальной полосы прокрутки). Щелчок области, расположенной между стрелками, вызывает перемещение на большее расстояние. *Ползунок* перемещается по длине полосы прокрутки, отображая примерное положение видимой части относительно общей длины документа. Для перехода в определенную область документа можно перетащить ползунок мышью. Относительно недавно размер ползунка на полосе прокрутки стал изменяться, отражая пропорцию видимой части ко всему документе

Полосы прокрутки можно добавить к форме двумя способами. Первый — создать элементов управления типа *VScrollBar* и *HScrollBar* и разместить их в любой части клиентской области программы. Работа полос прокрутки настраивается свойствами этих объектов. При использовании таких полос прокрутки инициируются определенные события. Мы будем работать с такими элементами управления в главе 12.

Второй способ проще. Его часто называют *автопрокруткой* (auto-scroll), и именно его мы используем в этой главе.

Средство автопрокрутки в первую очередь предназначено для программ, размещающих в своей клиентской части различные элементы управления (например, кнопки и текстовые поля). Включить автопрокрутку для формы можно, задав свойству *AutoScroll* значение *true*. Тогда, если размера клиентской области не хватит для отображения всех элементов управления, как по волшебству возникнут полосы прокрутки и позволят пользователю добраться до других элементов.

Включить автопрокрутку можно и без каких бы то ни было элементов управления- Я рассмотрю оба способа, а вы уж выбирайте.

Прокрутка панели

.NET Framework содержит массу интересных элементов управления: от кнопок, списков и текстовых полей, до календарей, деревьев и таблиц. Но элемент *«панель» не* входит в их число. Панель не имеет визуального представления и не сильно

влияет на интерфейс пользователя. Обычно панель применяют в «архитектурных» целях — чтобы сгруппировать элементы управления. Кроме того, панели бывают полезны, если нужен элемент управления без большой функциональной нагрузки.

В программе SysInfoPanel я создал панель, соответствующую по размерам отображаемой информации — т. е. достаточно большую для того, чтобы вместить 60 строк текста с информацией о системе. Я поместил этот элемент управления на клиентской части формы, позволив автопрокрутке разобраться со всем остальным.

SysInfoPanel.cs

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

class SysInfoPanel: Form
{
     readonly float cxCol;
     readonly int   cySpace;

     public static void Main()
     {
          Application.Run(new SysInfoPanel());
     }
     public SysInfoPanel()
     {
          Text = "System Information: Panel";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          AutoScroll = true;

          Graphics grfx  = CreateGraphics();
          SizeF    sizef = grfx.MeasureString(" ", Font);
          cxCol  * sizef.Width + SysInfoStrings.MaxLabelWidth(grfx, Font);
          cySpace = Font.Height;

               // Create a panel.

          Panel panel    - new Panel();
          panel.Parent   = this;
          panel.Paint  += new PaintEventHandler(PanelOnPaint);
          panel.Location = Point.Empty;
          panel.Size     = new Size<
              (int) Math.Ceiling(cxCol +
                                 SysInfoStrings.MaxValueWidth(grfx, Font»,
              (int) Math.Ceiling(cySpace * SysInfoStrings.Count));

          grfx.Dispose();
```

см. след. стр.

```
    }
    void PanelOnPaint(object obj, PaintEventArgs pea)
    {
        Graphics grfx      = pea.Graphics;
        Brush    brush     = new SolidBrush(ForeColor);
        int      iCount    = SysInfoStrings.Count;
        string[] astrLabels = SysInfoStrings.Labels;
        string[] astrValues = SysInfoStrings.Values;
        for (int i = 0; i < iCount; i++)
        {
            grfx.DrawString(astrLabels[i], Font, brush,
                            0, i * cySpace);
            grfx.DrawString(astrValues[i], Font, brush,
                            cxCol, 1 * cySpace);
        }
    }
}
```

Для работы этой программы нужен файл SysInfoStrings.cs. Хорошим способом совместного использования файлов различными проектами в Visual C# ,NET является выбор из меню Project команды Add Existing Item. (Ее также можно выбрать, щелкнув правой кнопкой имя проекта в окне Solution Explorer и раскрыв подменю Add.) Выберите имя существующего файла, который следует добавить к проекту, но вместо того, чтобы щелкнуть кнопку Open, щелкните расположенную рядом с ней стрелку и выберите команду Link File. Так вы избежите создания второй копии файла и проблем, которые могут возникнуть, если вы измените только одну из копий.

Взглянем на конструктор программы *SysInfoPanel*Для включения автопрокрутки следует установить свойству *AutoScroll* формы значение *true.* Пока все довольно просто. Затем программа вычисляет значения *cxCol* и *cySpace* тем же способом, что и SysInfoList. Но перед уничтожением объекта *Graphics* она создает панель:

```
Panel panel = new Panel();
```

Я присвоил ей имя *panel* и поместил ее на поверхности клиентской части формы. Поверхность, на которой расположен элемент управления, называется его *предком* (parent). Каждый элемент управления должен иметь предка. Выражение:

```
panel.Parent = this;
```

назначает предком панели форму программы. Ключевое слово *this* используется в методах или свойствах для ссылки на текущий экземпляр объекта; в данном случае слово *this* ссылается на созданный мною объект *Form.* Такое же значение передастся методу *ApplicationRun* в методе *Main.*

Я хочу иметь возможность выводить на панель графические данные. Для этого я назначаю обработчик событию *Paint* панели:

```
panel.Paint += new PaintEventHandler(PanelOnPaint);
```

Подобный код я использовал в главе 2, когда назначал обработчики события *Paint* для форм, являющихся экземплярами объекта *Form,* а не классов-наследников. Метод *PanelOnPaint* принадлежит классу *SysInfoPanel.*

Также следует указать расположение панели относительно родительской поверхности. Свойство *Location* класса *Panel* указывает положение верхнего левого угла панели. В этой программе я хочу поместить верхний левый угол панели в точке (0, 0) клиентской области, что позволяет сделать оператор:

```
panel.Location = Point.Empty;
```

Однако особой нужды в нем нет, так как верхний левый угол любых элементом управления по умолчанию помещается в точку (0, 0).

Для правильной работы программы размер панели должен соответствовать размеру выводимой информации:

```
panel.Size = new Size(
     (int) Math.Ceiling(cxCol + SysInfoStrings.MaxValueWidth(grfx, Font)),
     (int) Math.Ceiling(cySpace * SysInfoStrings.Count));
```

Переменная *cxCol* уже содержит значение ширины первой колонки с добавленным пробелом. Вызов метода *MaxValueWidth* класса *SysInfoStrings* возвращает максимальную ширину из значений свойства *SystemInformation*. Высота панели устанавливается равной межстрочному интервалу, умноженному на количество строк. Я использовал метод *Math.Ceiling* для дополнения значений до целого числа. (Как альтернативу можно было создать объект *SizeF*, ширина и высота которого выражаются числами с плавающей точкой, и использовать статический метод *Size.Ceiling* для преобразования его в объект *Size*.)

Вот и все. Метод *PanelOnPaint* отображает информацию на поверхности панели с помощью такого же кода, что и в методе *OnPaint* класса *SysInfoList*.

Но в этой программе, как только панель становится шире клиентской области, автоматически появляется горизонтальная полоса прокрутки. Если высота панели превышает высоту клиентской области, справа появляется вертикальная полоса прокрутки. Процесс динамичен: при изменении размеров клиентской области программы полосы прокрутки появляются и исчезают по мере надобности. Ползунки полос прокрутки изменяют свои размеры, отражая пропорции видимой части ко всему содержимому. Так, высота ползунка вертикальной полосы прокрутки определяется отношением высоты клиентской области к высоте панели



Поскольку элементы управления обычно наследуют цвет фона своих предков, а панели вообще тесно связаны с предками, трудно увидеть, что на самом деле

мы ввели еще один элемент управления. Для более ясного представления о том, что происходит в программе, можно явно указать цвет фона панели:

```
panel.BackColor - Color.Honeydew;
```

Если затем вы сделаете клиентскую часть программы больше самой панели — так, чтобы исчезли полосы прокрутки, — вы сможете увидеть панель медового цвета на (вероятно, белом) фоне формы. Другой способ увидеть панель — установить свойство *AutoScrollMargin* формы в конструкторе:

```
AutoScrollMargin = new Size{10, 10);
```

Если теперь вы сдвинете ползунок до упора вправо, то увидите область шириной в десять пикселов на правой стороне панели. Такую же область можно увидеть под панелью, сдвинув ползунок до упора вниз. Это фон клиентской части формы.

Выше я говорил, что более универсальный подход к прокрутке — использовать специальные элементы управления. Полосы прокрутки, созданные как отдельные элементы управления, имеют свойства *Minimum и Maximum,* которые позволяют задать числовые значения, соответствующие крайним положениям ползунка. Так устанавливается диапазон значений, в пределах которого может работать данная полоса прокрутки. В случае автопрокрутки в этих свойствах нет нужды. Диапазон значений задается разностью ширины и высоты клиентской части и ширины и высоты области расположения элементов управления (или, в нашем случае, единственного элемента *Panel*). К этому значению прибавляется разница между величиной *AutoScrollMargin* и шириной и высотой клиентской части,

Полосы прокрутки, созданные как отдельные элементы управления, генерируют событие *Scroll*, когда пользователь работает с полосой прокрутки. При автопрокрутке такое событие не возникает — по крайней мере получить к нему доступ из приложения нельзя.

Хотя программа SysInfoPanel напрямую не обрабатывает события *Scroll,* она обрабатывает события *Paint* для панели. Когда программа выводит информацию на элемент управления, прорисовывается только видимая область этого элемента. При каждой прокрутке SysInfoPanel генерирует событие *Paint*, так как в видимую область попадает непрорисованное пространство.

Наследование класса *ScrollableControl*

Что же происходит за кадром? Как я уже говорил в главе 2, среди многих классов, наследующих класс Form, есть *ScrollableControl*, и именно в нем реализована автопрокрутка. Мы уже столкнулись с двумя из шести свойств класса *ScrollableControl* и унаследованных классом *Form*.

Свойства класса *ScrollableControl* (выборочно)

Тип	Имя	Доступ	Описание
bool	*AutoScroll*	Чтение/запись	Включает автопрокрутку.
bool	*HScroll*	Чение/запись	Наличие горизонтальной полосы прокрутки.
bool	*VScroll*	Чтение/запись	Наличие вертикальной полосы прокрутки.
Size	*AutoScrollMargin*	Чтение/запись	Устанавливает отступ справа и снизу от элемента управления.

Свойства класса *ScrollableControl* (выборочно) *(продолжение)*

Тип	Имя	Доступ	Описание
Size	*AutoScrollMinSize*	Чтение/запись	Задает минимальную область прокрутки.
Point	*AutoScrollPosition*	Чтение/запись	Указывает положение полосы прокрутки,

Свойства *HScroll* и *VScroll* позволяют вам узнать, видима ли в данный момент горизонтальная или вертикальная полоса прокрутки. (Предполагается, что эти свойства можно использовать и для того, чтобы скрыть отображение полосы прокрутки, однако, по-видимому, это средство не очень хорошо работает.) Чуть позже я подробно опишу свойство *AutoScrollMinSize*.

AutoScrollPosition содержит текущее положение прокрутки в отрицательных координатах. В SysInfoPanel значение свойства *AutoScrollPosition* совпадает со значением свойства *Location* панели. Однако аксессоры *get* и *set* свойства *AutoScrollPosition* несовместимы. При чтении свойства получаемые координаты всегда меньше или равны 0. Между тем при его записи координаты должны быть положительны В главе 6 я приведу пример такой аномалии в программе SysInfoKeyboard, демонстрирующей работу с клавиатурой.

Очевидно, класс *ScrollableControl* имеет доступ к свойствам и событиям полос прокрутки; но он скрывает их от вас, предоставляя интерфейс более высокого уровня. При перемещении полосы прокрутки в SysInfoPanel код, содержащийся в классе *ScrollableControl*, изменяет для панели свойство *Location,* присваивая ему отрицательные значения. (Такое изменение легко проверить, добавив обработчик для события *Move* панели.) Отрицательные значения свойства *Location* означают что верхний левый угол панели располагается выше и левее верхнего левого угла клиентской области. Такое перемещение создает видимость прокрутки содержимого панели внутри формы.

В дальнейшем мы подробнее изучим средство автопрокрутки, когда будем создавать дополнительные элементы управления. Давайте посмотрим, сможем ли мы заставить автопрокрутку работать без создания дочерних элементов управления

Прокрутка при отсутствии элементов управления

Ключ к использованию автопрокрутки без создания дочерних элементов управления — в установке свойству *AutoScrollMinSize* значения отличного от (0, 0), устанавливаемого по умолчанию. Обычно область прокрутки основывается на положении и размере элементов управления в клиентской области. Однако *AutoScrollMinSize* задает минимальную область прокрутки независимо от наличия этих элементов. И, конечно, надо задать свойству *AutoScroll* значение *true,*

Обычно значение *AutoScrollMinSize* задается равным размерам, необходимым для отображения всей выводимой информации. В программах, отображающих сведения о системе, *свойство AutoScrollMinSize* должно быть установлено так, чтобы вместить высоту и ширину всех 60 строк текста. Такие же размеры мы задавали для панели в программе SysInfoPanel.

Программа SysInfoScroll по своей работе идентична SysInfoPanel, но она обеспечивает прокрутку содержимого при отсутствии дочерних элементов управления.

SysInfoScroll.cs

```
//-----------------------------------------
// SysInfoScroll.cs © 2001 oy Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoScroll: Form
{
     readonly float cxCol;
     readonly irtt  cySpace;

     public static void Main()
     {
          Application.Run(new SysInfoScroll());
     }
     public SysInfoScroll()
     {
          Text * "System Information: Scroll";
          BackColor * SystemColors.Window;
          ForeColor = SystemColors.WindowText;

          Graphics grfx  * CreateGraphics();
          SizeF    sizef = grfx.MeasureString(" ", Font);
          cxCol  = sizef.Width + SysInfoStrings.MaxLabelWidth(grfx, Font);
          cySpace = Font.Height;
               // Устанавливаем свойства автопрокрутки.

          AutoScroll = true;
          AutoScrollMinSize - new Slze(
              (int) Math.Ceiling(cxCol +
                                 SysInfoStrings.MaxValueWidth(grfx, Font)),
              (int) Math.Ceiling(cySpace * SysInfoStrings.Count));

          grfx.Dispose();
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx       - pea.Graphics;
          Brush    brush      = new SolidBrush(ForeColor);
          iRt      iCount     = SysInfoStrings.Count;
          string[] astrLabels = SysInfoStrings.Labels;
          string[] astrValues - SysInfoStrings.Values;
          Point    pt         " = AutoScrollPosition;

          for (int i = S; 1 < iCount; i++)
          {
               grfx.DrawString(astrLabels[i], Font, brush,
```

см. след. стр.

```
                                pt.X, pt.Y + i * cySpace);

                grfx.DrawString(astrValues[i], Font, brush,
                                •. pt.X + cxCol, pt.Y + 1 * cySpace);
            }
        }
    }
```

SysInfoPanel в своей клиентской части содержала панель. Вы можете спросить. размещаются ли полосы прокрутки в клиентской части такой программы, как SysInfoScroll? Нет! На самом деле клиентская область уменьшается в размерах на ширину полос прокрутки. Бывает, что ширина клиентской области лишь немного превышает значение *AutoScrollMinSize Width.* Однако при появлении вертикальной полосы прокрутки клиентская область сужается на ширину этой полосы; такое изменение может привести к тому, что ширина клиентской части станет меньше значения *AutoScrollMinSize.Width* и возникнет горизонтальная полоса прокрутки,

Устанавливая *значение AutoScrollMinSize,* вы *не* определяете что-либо подобное виртуальной области рисования. Независимо от того, насколько большое вы зададите значение *AutoScrollMinSize,* при обработке метода *OnPaint* вы работаете в границах физической клиентской области. На самом деле клиентская часть, вероятно, будет даже меньше обычных размеров из-за наличия полос прокрутки,

Любые действия с полосами прокрутки в SysInfoPanel приводили к отображению ранее скрытых областей панели и, следовательно, появлению события *Paint.* Та программа выводила всю информацию на панель, имеющую достаточные размеры для размещения этой информации. Прокрутка смещала панель относительно клиентской области, но координаты вывода информации на панель оставались неизменными. Например, вторая колонка третьего ряда текста всегда имела координаты *(cxCol, 2 ' cySpace).*

SysInfoScroll реагирует на изменение состояния полос прокрутки в переопределяемом методе *OnPaint* формы. Эта программа выводит информацию прямо в клиентскую область, а не на элемент управления, который затем смещается. Клиентская область недостаточно велика для размещения всей нужной информации, а объект *Graphics,* используемый в методе *OnPaint,* ничего не знает об автопрокрутке,

Это значит, что в методе *OnPaint* программы SysInfoScroll (или любой другой, использующей автопрокрутку и при этом выводящей данные прямо в клиентскую область) при вызове любой функции, осуществляющей вывод на экран, надо корректировать координаты вывода, основываясь на свойстве *AutoScrollPosition.* Как видите, в этой программе метод *OnPaint* получает значение *AutoScrollPosition,* сохраняет его в переменной типа *Point с* именем *pt* и прибавляет это значение к координатам вывода при вызове *DrawString.* Помните, что значения координат в свойстве *AutoScrollPosition* отрицательные. Если, скажем, вы прокрутили информацию вниз на 30 пикселов, первый вызов *DrawString* для первой строки текста получит координаты вывода (0, -30), данные в которых расположены выше видимой области, и не будет отображаться.

Такой метод перерисовки клиентской части кажется неэффективным: программа выводит 60 строк текста при каждой прорисовке, в то время как лишь небольшая

часть вызовов *DrawString* приводит к отображению какой-либо информации. Позже я еще вернусь к вопросу об эффективности.

Реальные числа

Прервемся на секунду и попытаемся получше понять, что происходит, взглянув на реальные числа. Помните только, что некоторые из этих чисел обусловлены параметрами моей системы и могут не совпадать с теми, что видите вы. (Например, в свойствах экрана у меня задан крупный шрифт, и это отражается на некоторых рассматриваемых параметрах.)

Допустим, вашей программе нужна клиентская область шириной в 400 и высотой в 1 600 пикселов. Вот как вы установите свойства *AutoScroll* и *AutoScrollMinSize* в конструкторе формы:

```
AutoScroll = true;
AutoScrollMinSize = new Size(400, 1600);
```

Мой опыт свидетельствует о том, что по умолчанию размер создаваемой формы составляет 300 x 300 пикселов. Каков тогда будет размер клиентской области? Что ж, у нас теперь есть две программы, позволяющих получить информацию о системе, так что вы можете выяснить размер клиентской области. У меня значение свойства *SystemInformation.CaptionHeight* равно 24. Это высота строки заголовка формы. Нормальная ширина границы окна содержится в свойстве *SystemInformatio.FrameBorderSize*. У меня она равна 4 пикселам, и не забывайте, что граница проходит по всем четырем сторонам формы. Таким образом, ширина клиентской области равна 300 минус два раза по 4, т. е. 292 пикселам. Высота клиентской области равна 300 минус два раза по 4 и минус 24, что составляет 268 пикселов. (Если вы не доверяете моим математическим навыкам, можете проверить эти значения при помощи свойства *ClientSize* формы.)

Поскольку высота клиентской области, равная *268* пикселам, меньше требуемых нам 1 600, программе потребуется вертикальная полоса прокрутки. Мое значение свойства *SystemInformation.VerticalScrollBarWidth* составляет 20 пикселов. Таким образом, ширина клиентской части уменьшается до 292 минус 20, что составляет 272 пиксела.

Ширина в 272 пиксела меньше 400, поэтому программе также потребуется и горизонтальная полоса прокрутки. Значение *SystemInformation.HorizontalScrollBarHeight* также равно 20 пикселам, и высота клиентской части составит 268 минус 20, или 248 пикселов.

Вероятно, диапазон значений для вертикальной полосы прокрутки включает в себя числа от 0 до 1 352. Второе значение получено вычитанием из 1 600 реальной высоты клиентской части, равной 248. Диапазон значений для горизонтальной полосы прокрутки включает в себя числа от 0 до 128 (400 минус 272),

Код, содержащийся в классе *ScrollableControl*, в ответ на щелчки пользователя по полосе прокрутки или перетаскивание ее ползунка выполняет два действия: изменяет значение свойства *AutoScrollPosition* и прокручивает содержимое клиентской части. Значение *AutoScrollPosition* сначала установлено в (0, 0). Когда пользователь передвигает горизонтальную полосу прокрутки, свойство *X* изменяется от О до *–128*, а свойство *Y* — от 0 до *–1352*.

Для прокрутки клиентской области системе нужно скопировать содержимое из одного положения клиентской области в другое. Б Win32 API есть функции *ScrollWindow*, *ScrollWindowEx* и *ScrollDC*, позволяющие программам прокручивать клиентские области. Хотя эти функции *не* открыты для библиотеки классов Windows Forms, очевидно, что класс *ScrollableControl* использует одну из них.

Когда код, содержащийся в классе *ScrollableControl,* прокручивает клиентскую область, он может прокрутить лишь ту часть, что в данный момент отображается на экране. Прокрутка «раскрывает» прямоугольник в клиентской области, делая эту часть клиентской области недействительной, что приводит к вызову метода *OnPaint*.

Так что, когда вы прокручиваете содержимое в окне SysInfoPanel или SysInfoScroll, методу *OnPaint* на самом деле надо перерисовать лишь небольшой прямоугольник клиентской области. Было бы нерационально использовать вызов *OnPaint* для прорисовки каждой строки выводимого текста.

Давайте решим эту проблему в два этапа.

Отслеживание изменений

Многие элементы, содержащиеся в классе *SystemInformation,* зависят от параметров, установленных пользователем компьютера, на котором выполняется программа. Сейчас это, вероятно, вы сами, но когда-нибудь вы можете захотеть опубликовать вашу программу, использующую свойства *SystemInformation*, чтобы с нею могли работать другие люди.

Некоторые свойства *SystemInformation* как будто высечены в камне. Например, для курсора мыши уже очень давно применяется квадрат со стороной в 32 пиксела, и единственное исключение составляет адаптер CGA фирмы IBM в режиме 640 на 200.[1]

Однако другие свойства могут изменяться даже во время работы программ SysInfoPanel или SysInfoScroll. Часто такие изменения производятся из Панели управления Windows.

Например, откройте Панель управления, дважды щелкните значок Display (Экран) и в диалоговом окне Display Properties (Свойства: Экран) перейдите на вкладку Effects (Эффекты). Если вы измените состояние флажка Show Window Contents While Dragging (Отображать содержимое окна при перетаскивании), это приведет к изменению свойства *SystemInformation.DragFullWindows*. Свойство *MouseButtonsSwapped* зависит от параметров, настраиваемых в диалоговом окне Mouse Properties (Свойства: Мышь) на вкладке Buttons (Кнопки мыши) в области Button Configuration (Конфигурация кнопок). Свойство *ShowSounds* зависит от параметров в окне Accessibility Options (Специальные возможности) на вкладке Sound (Звук) в области Show Sounds (Визуальное оповещение).

Изменение разрешения экрана затрагивает пять свойств класса *SystemInformation*: *MaxWindowTrackSize*, *PrimaryMonitorMaximizedWindowSize*, *PrimaryMonitorSize*, *VirtualScreen* и *WorkingArea*. Изменение системного шрифта в диалоговом окне Display Properties (Свойства: Экран) на крупный шрифт или как-то иначе затра-

[1] Charles Petzold, *Programming Windows* (Microsoft Press, 1988), с. 332.

гивает около двадцати свойств *SystemInformation,* но при таких значительных изменениях системе нужна перезагрузка. (Я еще расскажу о крупных и мелких шрифтах в главе 7.)

Проблема в том, что программы, о которых мы говорили до сих пор, получают свойства *SystemInformation* только при вызове метода *OnPaint.* Если вы измените одно из этих свойств в Панели управления, программа ничего не узнает об этих изменениях. Она даже не получит нового значения свойства до следующего вызова *OnPaint.* А так как метод *OnPaint* перерисовывает только недостоверную зону, то даже в этом случае отображение экранных элементов может остаться нескорректированным!

К счастью, приложение Windows Forms может быть проинформировано об изменениях в системе, влияющих на свойства *SystemInformation.*

Проинформировано? Но как? Разумеется, при помощи событий! Класс *SystemEvents* в пространстве имен *Microsoft.Win32* позволяет установить 12 событий, сообщающих вам об изменениях на уровне системы. Большая часть других компонентов *Microsoft.Win32* (включая делегаты, аргументы событий и перечисления) обеспечивают поддержку этих событий.

Нас интересуют два события: *UserPreferenceChanged* (отслеживает большинство изменений, которые пользователь может сделать в Панели управления) и *DisplaySettingsChanged* (инициируется при изменении параметров экрана). Добавление поддержки для этих событий требует модификаций в структуре программы. Ее измененная версия называется SysInfoUpdate.cs.

SysInfoUpdate.cs

```
//-----------------------------------------
// SysInfoUpdate.cs © 2001 by Charles Petzold
// ---------- -  ---------------------  . ___ .
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Windows.Forms;

class SysInfoUpdate: Form
{
     protected int      iCount;
     protected string[] astrLabels;
     protected string[] sstrValues;
     protected float    cxCol;
     protected in       cySpace;

     public static void Main()
     {
          Application.Run(new SysInfoUpdate());
     }
     public SysInfoUpdate()
     {
          Text = "System Information; Update"; .
```

см. след. стр.

```
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        AutoScroll = true;

        SystemEvents.UserPreferenceChanged +=
            new UserPreferenceChangedEventHandler(UserPreferenceChanged);

        SystemEvents.DisplaySettingsChanged +=
            new EventHandler(DisplaySettingsChanged);

        UpdateAllInfoO;
    }
    void UserPreferenceChanged(object obj,
                               UserPreferenceChangedEventArgs ea)
    {
        UpdateAllInfoO;
        Invalidate();
    }
    void DisplaySettingsChanged(object obj, EventArgs ea)
    {
        UpdateAllInfo();
        Invalidate();
    }
    void UpdateAllInfoO
    {
        iCount      = SysInfoStrings.Count;
        astrLabels = SysInfoStrings.Labels;
        astrValues = SysInfoStrings.Values;

        Graphics grfx  = CreateGraphics();
        SizeF    sizef * grfx.MeasureString(" ", Font);
        cxCol   = sizef.Width + SysInfoStrings.MaxLabelWidth(grfx, Font);
        cySpace = Font.Height;

        AutoScrollMinSize = new Size(
            (int) Math.Ceiling(cxCol +
                               SysInfoStrings.MaxValueWidth(grfx, Font»,
            (int) Math.Ceiling(cySpace * iCount));

        grfx.Dispose();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx  * pea.Graphics;
        Brush    brush = new SolidBrush(ForeColor);
        Point    pt    = AutoScrollPosition;

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

см. след. стр.

```
                grfx.DrawString(astrLabels[i], Font, brush,
                                pt.X, pt.Y + i * cySpace);

                grfx.DrawString(astrValues[i], Font, brush,
                                pt.X + cxCol, pt.Y + i * cySpace);
            }
        }
    }
```

Все, что зависит от сведений, полученных из класса *SysInfoStrings*, теперь объединено в модуле *UpdateAllInfo*. Первый раз *UpdateAUInfo* вызывается из конструктора программы, и там же устанавливаются обработчики событий. Они вновь вызывают метод *UpdateAUInfo* для получения новых данных, а затем вызывается метод *Invalidate* для сброса клиентской области. *Invalidate* инициирует вызов метода *OnPaint*.

Оптимизация программы

О программе, работающей не так быстро, как хотелось бы, пользователи говорят: «*тормозит*». Характеристика нелицеприятная, но зачастую справедливая.

Я уже немного улучшил эффективность программы, вызывая методы *SysInfoStrings* только при запуске программы или при изменении любого из свойств *SystemInformation*. Еще я избавился от трех вызовов *SysInfoStrings* при каждом вызове *OnPaint*.

Однако *OnPaint* продолжает выводить все 60 строк — и вызывает метод *DrawString* 120 раз — каждый раз, когда часть клиентской области становится недействительной. На большинстве компьютеров все 60 строк даже не будут видны. Более того, вертикальная прокрутка обычно продвигает текст на 1-2 строки; в этих случаях методу *OnPaint* требуется перерисовать лишь одну-две строки.

Windows сама предоставляет некоторую оптимизацию. Объект *Graphics*, который вы получаете в методе *OnPaint*, может выводить информацию только в недействительных участках пользовательской области. Это так называемая *вырезанная область* (clipping region), которая включает в себя только недействительные части пользовательской области и не позволяет выводить информацию за их пределами. Пример перерисовки только недействительных частей пользовательской области был приведен в программе RandomClear в главе 3. Однако факт остается фактом: вы все еще делаете 120 вызовов *DrawString*, и Windows все еще нужно проверять, относится ли каждый конкретный вызов *DrawString* к вырезанной области.

К счастью, нам поможет свойство *ClipRectangle* класса *PaintEventArgs* которое определяет наименьший прямоугольник в координатах клиентской области, в который умещается недействительная зона. (Как было показано в программе RandomClear, недействительная зона не обязательно имеет форму прямоугольника.) В качестве эксперимента можете добавить строку:

```
Console.WriteLine(pea.ClipRectangle);
```

в метод *OnPaint* и попробовать прокручивать информацию в окне или частично закрывать форму окнами других программ.

Программа SysInfoEffident наследует SysInfoUpdate и переопределяет метод *OnPaint* его более эффективной версией. При помощи пары довольно простых вычислений, основанных на свойстве формы *AutoScrollPosition* и свойстве *ClipRectangle* класса *PaintEventArgs*, вычисляются номера строк, заносимые в переменные *iFirst* и *iLast*, которые затем используются в цикле *for* для отображения минимального количества строк, требуемых для перерисовки клиентской области.

SysInfoEfficient.cs

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

class SysInfoEfficient: SysInfoUpdate
{
     public new static void Main()
     C
          Application.Run(newSysInfoEfficient());
     I
     public SysInfoEfficient() :,
     {
          Text = "System Information: Efficient";
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx  = pea.Graphics;
          Brush    brush = new SolidBrush(ForeColor);
          Point    pt    = AutoScrollPosition;

          int iFIrst = (int)((pea.ClipRectangle.Top    - pt.Y) / cySpace);
          int iLast  = (int)((pea.ClipRectangle.Bottom - pt.Y) / cySpaee);

          iLast = Math.Min(iCount - 1, iLast);

          for (int i = ifirst; i <= iLast; i++)
          {
               grfx.DrawString(astrLabels[i], Font, brush,
                               pt.X,  pt.Y + i * cySpace);

               grfx.DrawString(astrValues[i], Font, brush,
                               pt.X + cxCol, pt.Y 4 i * cySpaee);
          }
     }
}
```

Перед циклом *for* стоит выражение, которое с помощью метода *MathMin* предотвращает выход значения *iLast* за пределы общего количества отображаемых

строк. Предел может быть превышен, если окно программы выше, чем требуется для отображения всех строк,

Задел на будущее

Хотя сегодня .NET Framework может казаться верхом совершенства, существует вероятность (хоть и небольшая), что когда-нибудь грешные разработчики Microsoft будут вынуждены добавить в класс *SystemInformation* одно-два свойства. Тогда мой класс *SysInfoStrings* следует обновить, чтобы включить в него эти дополнительные свойства, и все программы этой главы надо будет перекомпилировать с учетом новой версии класса.

А можно ли написать программу, которая автоматически использует *все* свойства класса *SystemInformation*, даже те, которых и на свете не было во время ее написания?

Да, можно. Чтобы понять, как это сделать, подумаем, где размещается код класса *SystemInformation*. Согласно документации на этот класс код хранится в файле System.Windows.Forms.dll. При запуске любой программы из этой главы ОС связывает ее с файлом System.Windows.Forms.dll, и программа может вызывать класс *SystemInformation*.

Но DLL — это не просто набор кодов. Кроме них, она содержит *метаданные* в двоичном формате, подробно описывающие классы, включенные Б данный файл. Описываются все поля, свойства, методы и события этих классов. Компилятор C# использует эти данные при компиляции программ (вот почему надо устанавливать параметр Reference files). Документация .NET Framework также учитывает эти метаданные.

Так что имеет смысл ввести в программу возможность доступа к этим метаданным во время ее выполнения, *динамически* обновлять информацию о классах .NET и даже запускать некоторые методы этих классов. Это называется *отражением* (reflection), и концепция этого процесса заимствована из языка Java Вообще-то об отражении следует рассказывать в отдельной главе, но оно так здорово подходит для этого приложения, что я не могу удержаться.

В качестве первого шага перепишем класс *SysInfoStrings* так, чтобы для получения имен свойств и их значений использовалось отражение.

SysInfoReflectionStrings.cs

```
//-----------------------------------------------------
// SysInfoReflectionStrings.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using Microsoft.Win32;
using System;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;

class SysInfoReflectionStrings
{
                                                    // Поля
```

см. след. стр.

```
static bool     bValidInfo = false;
static int      iCount;
static string[] astrLabels;
static string[] astrValues;

                                                        // Конструктор
static SysInfoReflectionStrings()
{
    SystemEvents.UserPreferenceChanged +=
        new UserPreferenceChangedEventHandler(UserPreferenceChanged);

    SystemEvents.DisplaySettingsChanged +=
        new EventHandler(DisplaySettingsChanged);
}
                                                        // Свойства
public static string[] labels
{
    get
    {
        GetSysInfo();
        return astrLabels;
    }
}
public static string[] Values
{
    get
    {
        GetSysInfo();
        return astrValues;
    }
}
public static int Count
{
    get
    {
        GetSysInfo();
        return iCount;
    }
}
                                                // Обработчики событий
static void UserPreferenceChanged(object obj,
                                  UserPreferenceChangedEventArgs ea)
{
    bValidInfo = false;
}
static void DisplaySettingsChanged(object obj, EventArgs ea)
{
    bValidInfo = false;
}
                                                        // Методы
static void GetSysInfo()
```

см. след. стр.

```
{
     if(bValidInfo)
          return;

          // Получаем сведения о свойстве класса SystemInformation.
     Type type = typeof(SystemInformation);
     PropertyInfo[] apropinfo = type.GetProperties();

          // Счетчик числа статических считываемых свойств. ,

     iCount = 0;
     foreach (PropertyInfo pi in apropinfo)
     {
          if(pi.CanRead && pi.GetGetMethod().IsStatic)
               iCount++;
     }
          // Выделение строковых массивов.

     astrLabels = new string[iCount];
     astrValues = new string[iCount];

          // Снова проходим по всем свойствам.

     iCount = 0;
     foreach (PropertyInfo pi in apropinfo)
     {
          if(pi.CanRead && pi.GetGetMethod().IsStatic)
          {
                    // Получаем имена и значения свойств.

               astrLabels[iCount] = pi.Name;
               astrValues [iCount] = pi.GetValue(type, null).ToString();
               iCount++;
          }
     }
     Array.Sort(astrLabels, astrValues);
     bValidInfo = true;
}
public static float MaxLabelWidth(Graphics grfx, Font font)
(
     return MaxWidth(Labels, grfx, font);
}
public static float MaxValueWidth(Graphics grfx, Font font)
{
     return MaxWidth(Values, grfx, font);
}
static float MaxWidth(string[] astr, Sraphics grfx, Font font)
{
     float fMax = 0;
```

см. след. стр.

```
            GetSysInfo();

            foreach (string str in astr)
                fMax = Math.Max(fMax, grfx.MeasureString(str, font).Width);

            return fMax;
        }
}
```

Львиную долю работы в этом классе выполняет метод *GetSysInfo*. Он получает имена свойств и их значения при первом обращении и при каждом их изменении. Оператор *typeof* языка C# получает тип класса *SystemInformation*, который сохраняется в переменной типа *Type*. Одним из методов *Type* является метод *GetProperties*, возвращающий массив объектов *PropertyInfo*. Каждый объект в этом массиве является свойством класса *SystemInformation*. Цикл *foreach* подсчитывает все свойства, которые одновременно являются статическими и считываемыми. (Знаю, что на данный момент все свойства класса *SystemInformation* являются статическими и считываемыми, но я пытаюсь сделать программу универсальной.)

Затем программа создает массивы для свойств и их значений и еще раз проходит по массиву *PropertyInfo*. Свойство *Name* объекта *PropertyInfo* содержит имя свойства класса *SystemInformation*, в нашем случае свойство *Name* возвращает строки типа «*HighContrast*» и «*IconSize*». Метод *GetValue* получает значение каждого свойства. Статический метод *Sort* класса *Array* упорядочивает оба массива (с именами и значениями) по именам свойств.

Функционально программа, использующая класс *SysInfoReflectionStrings*, является сочетанием SysInfoUpdate и SysInfoEfficient.

SysInfoReflection.cs

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

class SysInfoReflection: Form
{
    protected int      iCount;
    protected string[] astrLabels;
    protected string[] astrValues;
    protected float    cxCol;
    protected int      cySpace;

    public static void Main()
    {
        Application.Run(new SysInfoReflection());
    }
```

см. след. стр.

```
public SysInfoReflection()
{
     Text = "System Information: Reflection";
     BackColor - SystemColors.Window;
     ForeColor = SystemColors.WindowText;
     AutoScroll - true;

     SystemEvents.UserPreferenceChanged +=
          new UserPreferenceChangedEventHandler(UserPreferenceChanged);
     SystemEvents.DisplaySettingsChanged +=
          new EventHandler(DisplaySettingsChanged);

     UpdateAllInfo();
}
void UserPreferenceChanged(object obj,
                           UserPreferenceChangedEventArgs ea)
{
     UpdateAllInfo();
     Invalidate();
}
void DisplaySettingsChanged(object Obj, EventArgs ea)
{
     UpdateAllInfo();
     InvalidateO;
}
void UpdateAUInfoO
{
     iCount     = SysInfoReflectionStrings.Count;
     astrLabels = SysInfoReflectionStrings.Labels;
     astrValues = SysInfoReflectionStrings.Values;

     firaphics grfx = CreateGraphics();
     SizeF    sizef = grfx.MeasureString(" ", Font);
     cxCol   = sizef.Width +
                  SysInfoReflectionStrings.MaxLabelWidth(grfx, Font);
     cySpace = Font.Height;

     AutoScrollMinSize = new Size(
         (int) Math.Ceiling(cxCol *
                  SysInfoReflectionStrings.MaxValueWidth(grfx, Font)),
         (int) Math.Ceiling(cySpace * iCount));

     grfx.Dispose();
}
protected override void OnPaint(PaintEventArgs pea)
(
     Graphics grfx  = pea.Graphics;
     Brush    brush = new SolidBrush(ForeColor);
     Point    pt    = AutoScrollPosition;
```

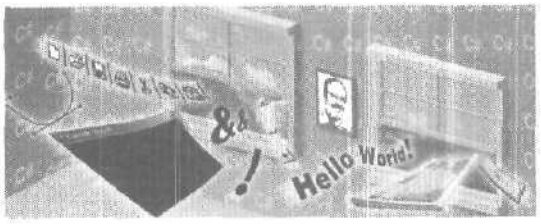
см. след. стр.

```
            int iFirst = (int)((pea.ClipRectangle.Top    - pt.Y) / cySpace);
            int Hast   * (int)((pea.ClipRectangle.Bottom - pt.Y) / cySpace);

            iLast = Math.Min(iCount - 1, iLast);
            for (int i = iFirst; i <= iLast; i++)
            {
                grfx.DrawString(astrLabels[i], Font, brush,
                                pt.X, pt.Y * i * cySpace);

                grfx.DrawString(astrValues[i], Font, brush,
                                pt.x + cxCol, pt.Y + i * cySpaee);
            }
        }
    }
```

Это моя самая последняя версия программы *SystemInformation* (до главы 6, в которой я добавлю в нее интерфейс для работы с клавиатурой).



ГЛАВА

5

Линии, кривые и заливка областей

Согласно Евклиду, «линия — это протяженность, лишенная ширины»[1]. Именно утверждение об «отсутствии ширины» делает это определение интересным. С одной стороны, оно свидетельствует о высокой степени абстракции, характерной для мышления математиков Древней Греции, а с другой — позволяет заключить, что эллины были в полном неведении о компьютерной графике. Иначе они бы осознавали, что на самом деле у пикселов есть ширина. С этим фактом связана одна из самых неприятных проблем компьютерной графики, известная как «ошибка смещения на 1 пиксел» (off-by-1-pixel error). В этой главе мы в подробностях рассмотрим эту проблему.

Мир компьютерной графики можно грубо разделить на две области:

- *векторная* графика — практическое применение аналитической геометрии, включает рисование прямых, кривых и заливку областей;
- *растровая* графика — включает битовые карты и изображения предметов реального мира.

За ними идет текст, занимающий собственное место в мире компьютерной графики. Однако в связи с ростом популярности контурных шрифтов (outline fonts) в последнее время с текстом часто обращаются как с частью векторной графики,

В этой главе мы познакомимся с векторной графикой в том виде в каком она реализована в Microsoft Windows Forms и GDI+. Все функции рисования, обсуждаемые в этой главе, являются методами класса *Graphics* и начинаются с префикса *Draw* или *Fill*. Методы *Draw* рисуют прямые и кривые; методы *Fill* служат для заливки областей (границы которых, естественно, определяются прямыми и кри-

[1] Sir Thomas L. Heath, ed., *The Thirteen Books of Euclid's Elements* (New York: Dover, 1956), 1: 153-

выми). Первым аргументом всех рассматриваемых в этой главе методов *Draw* является объект *Pen,* а первым аргументом методов *Fill* — объект *Brush.*

Как получить объект Graphics

Большинство функций рисования являются методами класса *Graphics* (кроме них, есть еще функции класса *ControlPaint*, но они довольно специфичны). Для рисования нужен объект типа *Graphics*. Но конструктор *Graphics* не является открытым, поэтому просто создать объект *Graphics*, как это показано ниже, не получится:

```
Graphics grfx = new Graphics();      // Не будет работать!
```

Кроме того, класс *Graphics* является *изолированным* (sealed), т. е. вы не можете создавать собственные классы, производные от *Graphics*:

```
class MyGraphics: Graphics           // Не будет работать!
```

Это делается иначе. Вот полный перечень способов получения объектов *Graphics,* начиная с самого распространенного.

- При переопределении метода *OnPaint* или установке обработчика события *Paint* в любом классе, производном от *Control* (например, *Form),* вы получаете объект *Graphics* как свойство класса *PaintEventArgs*.
- Для рисования на элементе управления или форме не только во время исполнения метода *OnPaint* или обработки события *Paint,* но и в другие моменты времени можно вызывать метод *CreateGraphics* этого элемента управления. Классы иногда вызывают метод *CreateGraphics* в своих конструкторах, чтобы получить нужные данные или выполнить инициализацию (как, скажем, это делали некоторые программы-примеры из главы 4). Хотя кое-какие классы не могут рисовать во время исполнения собственного конструктора, они могут делать это во время других событий. Для элемента управления или формы совершенно обычно рисовать что-то во время событий клавиатуры, мыши или таймера. Я еще продемонстрирую это в главах 6, 8 и 10. Важно, чтобы программа использовала объект *Graphics* только во время получающего его события (т. е. объект *Graphics* не должен храниться в поле класса). По завершении использования объекта *Graphics* программа также должна вызвать его метод *Dispose.*
- Можно получить объект типа *PrintPageEventArgs*, который содержит подходящий для данного принтера объект *Graphics*, установив обработчик события *PrintPage* при печати. Ниже я покажу, как это сделать.
- У некоторых элементов управления (меню, полей со списком, комбинированных полей и строк состояния) есть свойство *OwnerDraw*, позволяющее программе динамически рисовать на этом элементе управления. События *MeasureItem* и *DrawItem* предоставляют объекты типа *MeasureItemEventArgs* и *DrawItemEventArgs*, обладающие объектами *Graphics,* которые могут быть использованы обработчиком этого события;
- Чтобы рисовать по битовой карте или метафайлу, надо получить специальный объект *Graphics*, вызвав статический метод *GraphicsFromImage* (соответствующие методики я продемонстрирую в главах 11 и 23).

- Если нужно получить информацию из объекта *Graphics*, связанного с принтером, не распечатывая никаких данных, можно использовать метод *CreateMeasurementGraphics* класса *PrinterSettings*.
- При взаимодействии с кодом Win32 получить объект *Graphics* позволяют статические методы *GraphicsFromHwnd* и *GraphicFromHdc*.

Вкратце о перьях

На бумаге линии рисуют от руки — карандашами, мелками, перьевыми или шариковыми ручками, фломастерами и др. Тип инструмента определяет как минимум цвет и ширину линии. Эти и другие атрибуты линии находятся в ведении класса *Pen*, поэтому, чтобы нарисовать любую линию, надо задать объект *Pen*.

Однако сейчас мне не хочется вдаваться в подробное описание перьев. Дело в том, что они могут быть созданы на основе кистей, поэтому всерьез обсуждать перья нельзя в отрыве от этого предмета. В свою очередь кисти можно создать из растровых изображений и графических контуров (graphics paths), а это более сложные разделы программирования графики. Перьям и кистям посвящена глава 17.

Как сказано в главе 3, создать перо некоторого цвета можно так:

```
Pen pen = new Pen(color);
```

где *color* — объект типа *Color*. Кроме того, можно задействовать преимущества класса *Pens* со 141 статическим неизменяемым свойством, каждое из которых возвращает объект *Pen*. Так что *Pens.HotPink* — допустимый первый аргумент для любого метода рисования линий (хотя при его использовании надо соблюдать умеренность). Полный список имен цветов вы найдете и файле Colors.pdf на компакт-диске, прилагаемом к книге.

У класса *SystemPens* 15 статических неизменяемых свойств, также возвращающих объекты *Pen* на основе системных цветов. Но если нужно создать перо, гарантированно видимое на фоне, цвет которого выбран пользователем, следует поставить цвет пера в зависимость от текущего значения свойства *ForeColor*:

```
Pen pen = new Pen(ForeColor);
```

Толщина — еще один аспект перьев, который я хочу упомянуть здесь, — определяется значением изменяемого свойства:

Свойства Pen (выборочно)

Тип	Свойство	Доступ
float	*Width*	Чтение/запись

Кроме того, есть конструкторы *Pen*, включающие толщину пера как аргумент:

Конструкторы *Pen* (выборочно)

```
Pen(Color color)
Pen(Color color, float fWidth)
```

(Чтобы у вас не сложилось впечатления неполноты приводимой мной информации, отмечу, что, кроме показанных выше, есть еще два конструктора *Pen*. Они отличаются лишь первым аргументом, которым у них является объект *Brush*.) При

использовании первого конструктора создается перо с толщиной 1. Объекты *Pen*, доступные через классы *Pens* и *SystemPens*, также имеют толщину 1. Пока можно считать, что 1 — это 1 пиксел. Однако (как будет сказано в главе 7) реально толщина выражается в *глобальных координатах* (world coordinates) и подвержена влиянию *преобразований* (transforms).

Можно создавать перья, толщина которых всегда будет равна 1 пикселу независимо от преобразований. Для этого следует указать в конструкторе толщину 0:

```
Pen pen = new Pen(color, 0);
```

или установить свойство *Width* в 0:

```
pen.Width = 0;
```

Прямые линии

Для рисования одиночных прямых служит метод *DrawLine* класса *Graphics*. Есть четыре перегруженных версии *DrawLine*, но все они требуют один и тот же набор аргументов: координаты начальной и конечной точек линии, а также перо, которым она рисуется:

Методы *DrawLine* класса *Graphics*

```
void DrawLine(Pen pen, int x1, int y1, int x2, int y2)
void DrawLine(Pen pen, float x1, float y1, float x2, float y2)
void DrawLine(Pen pen, Point point1, Point point2)
void DrawLine(Pen pen, PointF point1, PointF point2)
```

Можно указывать координаты как четырьмя значениями типа *int* или *float*, так и парой структур *Point* или *PointF*.

Метод *DrawLine* чертит линию от первой точки до второй, *включая вторую точку в состав линии* (этим он несколько отличается от Win32 GDI, который проводит линию между двумя точками, *не включая конечную точку в состав линии*). ВОТ Пример:

```
grfx.Drawline(pen, 0, 0, 5, 5);
```

Эта команда закрашивает черным цветом 6 пикселов с координатами (0,0), (1,1), (2,2), (3,3), (4,4) и (5,5). Порядок указания начальной и конечной точек линии не имеет значения, поэтому вызов:

```
grfx.DrawLine(pen, 5, 5, 0, 0);
```

даст тот же результат. Вызов:

```
grfx.DrawLine(pen, 2, 2, 3, 3);
```

рисует 2 пиксела с координатами (2, 2) и (3, 3). Однако вызов:

```
grfx.DrawLine(pen, 3, 3, 3, 3);
```

не рисует ничего.

Как вам известно, определить ширину и высоту клиентской области позволяет свойство *ClientSize* формы. Число пикселов по горизонтали определяется зна-

чением *ClientSize.Width*; пикселы могут нумероваться от 0 до *ClientSize.Width* - 1. Аналогично пикселы клиентской области по вертикали нумеруются от 0 до *ClientSizeHeight* - 1.

Программа XMarksTheSpot рисует крестик в клиентской области.

XMarKsTheSpot.cs

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

class XMarksTheSpot: Form
{
     public static void Main()
     {
          Application.Run(new XMarksTheSpot());
     }
     public XMarksTheSpot()
     {
          Text = "X Marks The Spot";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw = true;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;
          Pen      pen  = new Pen(ForeColor);

          grfx.DrawLine(pen, 0, 0,
                             ClientSize.Width - 1, ClientSize.Height - 1);
          grfx.DrawLine(pen, 0, ClientSize.Height - 1,
                             ClientSize.Width - 1, 0);
     }
}
```

Первый вызов *DrawLine* рисует линию от верхнего левого до нижнего правого пиксела клиентской области. Второй вызов *DrawLine* ведет линию из нижнего левого пиксела с координатами (0. *ClientSize.Height* - 1) и доводит ее до верхнего правого пиксела с координатами *(ClientSize.Width* ~ 1, 0).

Кое-что о печати

В этой и следующих главах многие программы, демонстрирующие графические возможности, будут похожи на XMarksTheSpot. Большинство из них не такие неуклюжие, как XMarksTheSpot, хотя некоторые, увы, таковы. Но они лишь демонстрируют базовые методики программирования графики, показывая в клиентской области статические изображения, и ничего больше.

Возможность печати этих изображений может быть полезна даже сейчас, когда вы только начинаете изучать C#, чтобы, распечатав на принтере, с гордостью приклеить их на дверцу холодильника. Но, что важнее, печать изображений на принтере при изучении системы программирования графики позволяет лично убедиться в независимости этой системы от устройств.

В книгах по программированию описание вывода на печать обычно стараются загнать в самый конец или вовсе игнорируют эту тематику из-за ее чрезвычайной сложности. Главу 21 я целиком отвел описанию печати, чтобы полностью осветить все разнообразие доступных возможностей и механизмов вывода на печать. Однако наша текущая задача — распечатать одну страницу на стандартно настроенном принтере по умолчанию из приложения Windows Forms, — довольно проста.

Я колебался, стоит ли вообще знакомить читателя с печатью в самом начале. На самом деле единственной причиной моих сомнений была проблема с пользовательским интерфейсом, т. е. как дать пользователю возможность инициировать печать из программы. Как известно, у большинства программ, поддерживающих печать, в меню File (Файл) есть команда Print (Печать). Но время знакомства с меню пока не пришло — этому посвящена глава 14. Я думал реализовать простой интерфейс, использующий клавиатуру, например, через клавишу Print Screen (иногда ее подписывают как PrtScn) или клавиатурную комбинацию Ctrl+P. В конце концов я решил переопределить метод *OnClick*.

Метод *OnClick*, реализованный в классе *Control*, унаследовали все потомки этого класса, включая *Form*. Каждый раз, когда пользователь щелкает любой из кнопок мыши клиентскую область формы, вызывается метод *OnClick* — и это все, что я собираюсь сказать об использовании мыши до главы 8!

Чтобы печатать на принтере по умолчанию, нужно прежде создать объект типа *PrintDocument* — класса, определенного в пространстве имен *System.Drawing.Printing*:

```
PrintDocument prndoc = new PrintDocument();
```

В главе 21 этому классу будет уделено больше времени, а сейчас я упомяну лишь об одном свойстве, одном событии и одном методе этого класса.

Свойство *DocumentName* объекта *PrintDocument* следует установить как текстовую строку. Во время постановки графики в очередь печати текст этой строки идентифицирует задание в диалоговом окне принтера:

```
prndoc.DocumentName = "My print job";
```

Обычно программа, работающая с документом, заносит в нее его имя. В данной главе я буду для этих целей использовать текст заголовка окна программы.

Нужно создать в вашем классе метод, который будет исполнять вызовы на вывод графической информации. Он определяется как делегат *PrintPageEventHandler*. Этот метод можно назвать *PrintDocumentOnPrintPage*:

```
void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
{
     ⋮
}
```

Подключите этот обработчик к событию *PrintPage* объекта *PrintDocument*, например, так:

```
prndoc.PrintPage += new PrintPageEventHandler(PrintDocumentOnPrintPage);
```

Именно так были установлены обработчики события *Paint* в некоторых программах главы 2, а также в программе SysInfoPanel главы 4.

Для начала печати осталось лишь вызвать метод *Print* объекта *PrintDocument:*

```
prndoc.Print();
```

Этот метод *Print* возвращает управление не сразу — некоторое время отображается окно сообщения с именем документа и кнопкой, позволяющей отменить задание печати.

В результате исполнения метода также вызывается обработчик события *PrintPage* (названный здесь *PrintDocumentOnPrintPage*). Значением параметра *object* обработчика *PrintDocumentOnPrintPage* является ранее созданный объект *PrintDocument*. У параметра *PrintPageEventArgs* есть свойства, предоставляющие сведения о принтере, самое важное из которых — *Graphics* — аналогично одноименному свойству параметра *PaintEventArgs* за исключением того, что *PrintPageEventArgs* предоставляет объект *Graphics* для страницы принтера, а не для клиентской области формы.

Часто метод *PrintDocumentOnPrintPage* выглядит примерно так:

```
void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
{
     Graphics grfx = ppea.Graphics;
     ...
};
```

Здесь объект *Graphics* вызывает методы, выводящие графику на страницу принтера.

Если нужно напечатать несколько страниц, следует установить свойство *HasMorePages* объекта *PrintPageEventArgs* как *true*. Но поскольку печатается лишь одна страница, оставим значение свойства по умолчанию — *false*, и обработчик *PrintDocumentOnPrintPage* сразу вернет управление.

Если свойству *HasMorePages* объекта *PrintPageEventArgs* задано *false,* то после возврата управления обработчиком *PrintDocumentOnPrintPage* исходно вызванный метод *Print* объекта *PrintDocument* также возвращает управление. По завершении задания печати завершается и сама программа, передача графической информации на принтер — уже не наша проблема, как и застрявшая бумага, пустые картриджи, грязь от лишнего тонера и плохие кабели,

К машине может быть подключено несколько принтеров. Показанный выше подход использует принтер *по умолчанию.* Задать принтер по умолчанию можно, вызвав на панели управления или из меню Start|Settings (Пуск Настройка) диалоговое окно Printers (Принтеры) и выбрав из меню File (Файл) соответствующую команду.

Как известно, свойство *ClientSize* формы предоставляет размеры ее клиентской области, выраженные в пикселах. Для создания графического шедевра в пределах клиентской области этих сведений достаточно, но использовать аналогичное свойство для страницы принтера в силу некоторых причин проблематично.

Страница принтера определяется тремя разными областями. Во-первых, полным размером страницы. Эту информацию предоставляет свойство *PageBounds* класса *PrintPageEventArgs.* Его значением является структура *Rectangle*, чьи свойства X и Y равны 0, а свойства *Width* и *Height* предоставляют размеры бумаги по умолчанию, выраженные в сотых долях дюйма. Например, для листа размером 8,5

па 11 дюймов значение свойств *Width* и *Heigh* объекта *PageBounds* будет равно 850 и 1 100. Если в принтере по умолчанию задана альбомная, а не книжная ориентации страницы, то они будут равны 1 100 и 850 соответственно.

Во-вторых, есть *область печати* страницы (printable area). Ее размер обычно очень близок к полному размеру страницы, кроме полей, недоступных печатающей головке (или аналогичному узлу) принтера. Ширина верхнего, нижнего, левого и правого полей может быть разной. Значением свойства *VisibleClipBounds* класса *Graphics* является структура *RectangleF*, предоставляющая размер области печати страницы. Свойства *X* и У этой структуры устанавливаются в 0, а ее свойства *Width* и *Heigh* равны горизонтальному и вертикальному размерам области печати страницы, выраженным в тех же единицах, что будут использованы и для рисования на странице принтера.

Размер третьей области рассчитывается с учетом 1-дюймового отступа по периметру страницы и представляет собой границы, в пределах которых предпочитает печатать пользователь. Эта информация возвращается в виде структуры *Rectangle* свойством *MarginBounds* объекта *PrintPageEventArgs*.

В главе 21 мы разберем эти вопросы подробнее, а сейчас лучше всего без лишних слов применить свойство *VisibleClipBounds* класса *Graphics*. Объект *Graphics*, полученный из объекта *PrintPageEventArgs*. согласован с этим свойством, т. е. точка с координатами (0, 0) соответствует верхнему левому углу области печать страницы,

Конечно же, все. на что я так настойчиво обращал ваше внимание относительно использования видимых цветов на экране, неверно для принтера. Лучший цвет для принтера — *Color.Black*, лучшее перо — *Pens.Black*, а лучшая кисть — *Brushes.Black*. Это подходит для всех, кроме чудаков, непременно желающих печатать на черной бумаге.

Вот программа, которая выводит в клиентской области приглашение «Click to print» и начинает печать после щелчка этого приглашения.

HelloPrinter.cs

```
//-------------------------------------------
// HelloPrinter.cs • 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class HelloPrinter: Form
{
     public static void Main()
     {
          Application.Run(new HelloPrinter());
     }
     public HelloPrinter()
     {
          Text = "Hello Printer!";
          BackColor = SystemColors.Window;
```

см. след. стр.

```
            ForeColor = SystemColors.WindowText;
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics      grfx   = pea.Graphics;
            StringFormat strfmt = new StringFormatO;

            strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;
            grfx.DrawString("Click to print", Font, flew SolidBrush(ForeColor),
                            GlientRectangle, strfmt);
        }
        protected override void OnClick(EventArgs ea)
        {
            PrintDocument prndoc = new PrintDocument();

            prndoc.DocumentName = Text;
            prndoc.PrintPage +=
                  new PrintPageEventHandler(PrintDocumentOnPrintPage);
            prndoc.Print();
        }
        void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
        {
            Graphics grfx = ppea.Graphics;

            grfx.DrawString(Text, Font, Brushes.Black, 0, S);

            SizeF sizef = grfx.MeasureString(Text, Font);

            grfx.DrawLine(Pens.Black, sizef.ToPointF(),
                                      grfx.VisibleClipBounds.Size.ToPointF());
        }
}
```

Заметьте, что я использовал свойство *Text* формы как для имени печатаемого документа, так и в качестве аргумента *DrawString* и *MeasureString* в мегоде *PrintDocumentOnPrintPage*. Программа выводит текст «Hello Printer!» в верхнем левом углу области печати страницы, а затем рисует линию от правого нижнего угла текстовой строки до правого нижнего угла области печати. Этого примера должно быть достаточно, чтобы убедиться, что свойство *VisibleClipBounds* предоставляет сведения, согласованные с объектом *Graphics,*

Слышу, как смеются мои читатели, поскольку я легкомысленно использовал свойство *Font* формы при вызове методов *DrawString* и *MeasureString,* не думая, что у принтера может быть разрешение в 300, 600, 720, 1200, 1 400 и даже 2 400 или *2* 880 точек на дюйм. Шрифт, доступный через свойство *Font,* выбран системой как подходящий для экрана, разрешение которого скорее всего не превышает 100 точек на дюйм. В результате отпечатанный на принтере текст может выглядеть просто мизерным.

Ладно, продолжим: попробуйте исполнить программу. Принтер печатает текст вполне приличного размера — 8 пунктов. Кроме того, обратите внимание, что

толщина диагональной линии явно больше 1 пиксела. Линии толщиной в 1 пиксел едва ли будут различимы на современных принтерах, печатающих с высоким разрешением, вместо этого Windows Forms рисуют прекрасно видимую линию достаточной толщины. Причина столь радостного результата пусть пока останется тайной. Она раскроется в главах 7 и 9.

А теперь напишем программу, которая выводит в клиентской области формы и печатает на принтере одну и ту же информацию. Нет, мы не будем копировать код из *OnPaint* в метод *PrintDocumentOnPrintPage*. Давайте покажем, что мы кое-что смыслим в программировании и поместим код для вывода графической информации в отдельный метод. *DoPage*, вызываемый как обработчиком *OnPaint*, так и *PrintDocumentOnPrintPage*. Вот вариация программы XMarksTheSpot, которая работает именно так.

PrintableForm.cs

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

class PrintableForm: Form
{
     public static void Main()
     {
          Application.Run(new PrintableForm());
     }
     public  PrintableForm()
     {
          Text * "Printable Form";
          BackColor - SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw - true;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          DoPage(pea.Graphics, ForeColor,
                 ClientSize.Width, ClientSize.Height);
     }
     protected override void OnClick(EventArgs ea)
     {
          PrintDocument prndoc - new PrintDocument();

          prndoc.DocumentName = Text;
          prndoc.PrintPage +=
                 new PrintPageEventHandler(PrintDocumentOnPrintPage);
          prndoc.Print();
```

см. след. стр.

```
    }
    void PrintDocumentOnPrintPage(object obj, PrintPageEventArgs ppea)
    {
        Graphics grfx  = ppea.Graphics;
        SizeF    sizef = grfx.VisibleClipBounds.Size;
        DoPage(grfx, Color.Black, (int)sizef.Width, (int)sizef.Height);
    }
    protected virtual void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Pen pen = new Pen(clr);

        grfx.DrawLine(pen, 0,      0,  cx - 1, cy - 1);
        grfx.DrawLine(pen, cx - 1, 0,  0,      cy - 1);
    }
}
```

Метод *DoPage,* код которого расположен в конце листинга, выводит графическую информацию. Его аргументы — объект *Graphics*; цвет, подходящий для устройства, ширина и высота области вывода. *DoPage* вызывают два других метода: *OnPaint* и *PrintDocumentOnPrintPage*. При вызове из метода *OnPaint* трем последним аргументам метода *DoPage* присваиваются значения свойства *ForeColor*, а также ширины и высоты клиентской области формы соответственно. В случае вызова из *PrintDocumentOnPrintPage* аргументами становятся *Color.Black*, ширина и высота *VisibleClipBounds*.

Два последних аргумента метода *DoPage* я назвал *cx* и *cy*. Буква *c* — сокращенное слово *count* (число), а *x* и *y* — это обычно имена координат. Таким образом, *cx* и *cy* можно интерпретировать как «число координатных точек» или как ширину и высоту.

Занятно, что в случае объекта *Graphics для* клиентской области значения свойства *VisibleClipBounds* равны ширине и высоте именно этой области. Я мог бы просто передать аргументы *cx* и *cy* методу *DoPage,* после чего использовать *VisibleClipBounds* для вывода как на экран, так и на принтер. Но мне нравится, когда значения ширины и высоты заранее доступны в виде обычных переменных, особенно в контексте текущей задачи. Заметьте: я сделал метод *DoPage защищенным* (protected) и *виртуальным* (virtual), следовательно, его можно переопределить. Если понадобится написать программу, выводящую лишь один экран, заполненный графикой, можно породить нужные классы из *PrintableForm*, а не из *Form,* получив, таким образом, в своей программе встроенную поддержку печати,

Именно так я и буду поступать практически во всех программах-примерах *из* этой и других глав, посвященных программированию графики,

Свойства и состояние

В некоторые среды программирования графики включено понятие *текущей позиции* (current position). Эту координату среда удерживает в памяти и использует как начальную точку для функций рисования. В общем случае графическая система определяет пару функций: одну для установки текущей позиции, другую —

для рисования линии от текущей позиции до заданной точки. При этом функция рисования устанавливает текущую позицию в новую точку.

В GDI+ нет понятия текущей позиции. Это может несколько шокировать ветеранов программирования для Windows, поскольку для рисования линии в Windows GDI нужно вызвать две функции, для каждой из которых определена одна координата. Функция *MoveTo* устанавливает текущую позицию в заданную точку, а *LineTo* проводит линию до этой точки (но не включает ее в состав линии).

GDI+ также отличается от Windows GDI наличием у функций *DrawLine* и *DrawString* аргументов, задающих шрифт, кисть и перо. Если бы GDI+ был более похож на Windows GDI, то шрифт, кисть и перо были бы атрибутами объекта *Graphics*. Вспомните, что *StringFormat* задает некоторые подробности, касающиеся вывода текста. *StringFormat* тоже является аргументом метода *DrawString*, а не свойством объекта *Graphics*.

В силу этого архитекторы GDI+ назвали свое детище «средой программирования графики с *неопределенным состоянием*» (stateless graphics programming environment). Однако GDI+ не полностью соответствует такому названию. В таком случае у класса *Graphics* изменяемые свойства отсутствовали бы вовсе! А так у этого класса, помимо 6 неизменяемых свойств, еще и 12 изменяемых.

Я насчитал 4 изменяемых свойства класса *Graphics*, существенно влияющих на вид графических фигур:

- *PageScale* и *PageUnit* определяют единицы размеров при рисовании; по умолчанию единицей размера при рисовании на экране является пиксел (в главе 7 я расскажу о каждом из этих свойств подробнее);
- свойство *Transform* — это объект типа *Matrix*, определяющий матричное преобразование для всей выводимой графической информации; преобразование транслирует, масштабирует, обрезает и вращает точки координат (подробнее о матричном преобразовании см. главу 7);
- *Clip* — это вырезанная область; будучи установленной, она ограничивает область вывода для любой из вызываемых функций рисования (о вырезании см. главу 15).

Сглаживание

Помимо четырех вышеперечисленных, у класса *Graphics* есть и другие свойства, влияние которых менее заметно, например *SmoothingMode* и *PixelOffsetMode*.

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ	Описание
SmoothingMode	*SmoothingMode*	Чтение/запись	Сглаживание линий
PixelOffsetMode	*PixelOffsetMode*	Чтение/запись	Улучшенное сглаживание

Эти свойства обеспечивают поддержку методики визуализации графики, известную как *сглаживание* (anti-aliasing). Термин «alias» в данном контексте происходит из теории дискретизации. Сглаживание — это попытка скрыть резкие неровности выводимой графики с помощью полутонов.

Вот программа, рисующая небольшую линию. Среди прочего я включил в нее операторы, позволяющие устанавливать свойства *SmoothingMode* и *PixelOffsetMode*.

AntiAlias.cs

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

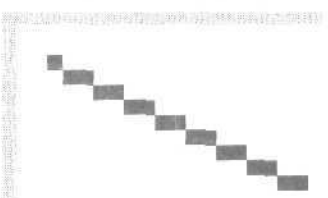
class AntiAlias: Form
{
     public static void Main()
     {
          Application.Run(new AntiAlias());
     }
     public AntiAliasO
     {
          Text = "Anti-Alias Demo";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx = pea.Graphics;
          Pen      pen  = new Pen(ForeColor);

          grfx.SmoothingMode  = SmoothingMode.None;
          grfx.PixelOffsetMode = PixelOffsetMode.Default;

          grfx.DrawLine(pen, 2, 2, 18, 10);
     }
}
```

Можно поэкспериментировать с различными комбинациями этих свойств. Для этого нужно перекомпилировать и исполнить программу, затем сделать снимок экрана и увеличить его в каком-нибудь графическом редакторе, чтобы полученные различия стали заметны. Есть и другой вариант: откиньтесь на спинку кресла и предоставьте все мне.

По умолчанию, линия визуализируется так:



Я специально включил в этот рисунок небольшую часть экрана за пределами клиентской области (наверху слева), чтобы хорошо было видно начало линии — пиксел с координатами (2, 2).

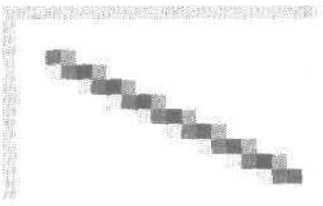
Перечисление *SmoothingMode* определяется в пространстве имен *System.Drawing.Drawing2D*:

Перечисление *SmoothingMode*

Член	Значение	Комментарии
Default	0	Без сглаживания.
HighSpeed	1	Без сглаживания.
HighQuality	2	Сглаживание включено.
None	3	Без сглаживания,
AntiAlias	4	Сглаживание включено.
Invalid	–1	Вызывает исключение.

Реально здесь только три возможности выбора: есть сглаживание, нет сглаживания или будет вызвано исключение. По умолчанию задано значение *None*.

После включения сглаживания (это можно сделать, установив свойство *SmoothingMode* равным *SmoothingMode.HighQuality* или *SmoothingMode.AntiAlias*), линия принимает такой вид:



Такая линия расплывается, если разглядывать ее вблизи, но на более далеком расстоянии она выглядит более гладко (некоторые не согласны с этим, считая, что сглаживание делает изображение «размытым»).

Идея здесь в том, что линия начинается в центре пиксела (2, 2) и кончается в центре пиксела (18, 10), ее толщина — 1 пиксел. Когда графическая система использует сглаживание, интенсивность закрашивания пиксела черным определяется его удаленностью от теоретической линии.

Можно немного улучшить сглаживание, установив для свойства *PixelOffsetMode* одно из значений перечисления *PixelOffsetMode* (подобно *SmoothingMode*, определяемого в пространстве имен *System.Drawing.Drawing2D*):

Перечисление *PixelOffsetMode*

Член	Значение	Комментарии
Default	0	Смещение пикселов не включено.
HighSpeed	1	Смещение пикселов не включено.
HighQuality	2	Включено смещение на половину пиксела.
None	3	Смещение пикселов не включено.
Half	4	Включено смещение на половину пиксела.
Invalid	–1	Вызывает исключение.

И здесь всего лишь три реальных возможности выбора, одна из которых бесполезна. Если свойство *PixelOffsetMode* установлено как *Half* или *HighQuality*, линия визуализируется так:



При включении смещения пиксела расчет координат осуществляется в соответствии с подходом, использующим аналитическую геометрию. Значение координат линии уменьшается на полпиксела. Предполагается, что линия начинается в точке, расположенной на расстоянии 2 пикселов от угла, которая на самом деле попадает в промежуток между соседними пикселами,

Сочлененные линии

Выше я сказал, что в некоторых средах программирования графики используется *текущая позиция*, что можно считать странным: ведь чтобы нарисовать одну линию, приходится вызывать дне функции. Однако текущая позиция серьезно помогает при рисовании нескольких соединенных друг с другом линий. позволяя задавать при вызове каждой функции координаты лишь одной точки. GDI+ не столь экономичен. Например, чтобы нарисовать прямоугольник по периметру клиентской области программы, приходится четырежды вызывать *DrawLine*:

```
grfx.DrawLine(pen, 0,      0,      cx - 1, 0);
grfx.DrawLine(pen, cx - 1, 0,      cx - 1, cy - 1);
grfx.DrawLine(pen, cx - 1, cy - 1, 0,      cy - 1);
grfx.DrawLine(pen, 0,      cy - 1, 0,      0);
```

Заметьте: конечная точка каждого предыдущего вызова должна повториться как начальная точка следующего.

В силу этих и некоторых других причин (которые мы обсудим чуть позже) в класс *Graphics* включен метод рисования нескольких соединенных линий, в совокупности обычно называемых *ломаной*. Есть две версии *DrawLines* (обратите внимание на множественное число в названии метода):

Методы *DrawLines* класса *Graphics*

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

Эти методы требуют массив целочисленных координат (*Point*) или координат с плавающей точкой (*PointF*).

Бот код, использующий метод *DrawLines* для очерчивания клиентской области:

BoxingTheClient.cs

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

см. след. стр.

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

class BoxingTheClient: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new BoxingTheClient());
     }
     public  BoxingTheClient()
     {
          Text = "Boxing the Client";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Point[] apt = {new Point(0,      0),
                         new Point(cx - 1, 0),
                         new Point(cx - 1, cy - 1),
                         new Point(0,      cy - 1),
                         new Point(0,      0)};

          grfx.DrawLines(new Pen(clr), apt);
     }
}
```

Используемый класс является производным от *PrintableForm*, так что программа поддерживает еще и печать.

Массив структур *Point* может быть определен прямо в методе *DrawLines*, и приведенная ниже программа так и делает. Она воплощает решение детской головоломки, в которой нужно нарисовать домик, не отрывая карандаша от бумаги,

DrawHouse.cs

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

class DrawHouse: PrintableForm
{
     public new Static void Main()
     {
          Application.Run(new DrawHouse());
     }
     public DrawHouse()
     {
          Text = "Draw a House in One Line";
     }
```

см. след. стр.

```
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawLines(new Pen(clr),
                       new Point[]
                       {
                         new Point(    cx / 4, 3 * cy / 4), // Нижний левый угол
                         new Point(    cx / 4,     cy / 2),
                         new Point(    cx / 2,     cy / 4), // Крыша
                         new Point(3 * cx / 4,     cy / 2),
                         new Point(3 * cx / 4, 3 * cy / 4), // Нижний правый угол
                         new Point(    cx / 4,     cy / 2),
                         new Point(3 * cx / 4,     ey / 2),
                         new Point(    cx / 4, 3 * cy / 4), // Нижний левый угол
                         new Point(3 * cx / 4, 3 * cy / 4)  // Нижний правый угол
                       });
    }
}
```

Но метод *DrawLines* создан не для решения детских головоломок. В главе 17 вы узнаете, как создавать перья на основе комбинации точек и штрихов, а также как при создании толстых перьев определять вид концов линии (закругленные, квадратные и т. д.) или вид двух соединенных вместе линий. Эти элементы линий называется *концами* и *соединениями*. Чтобы придать концам и соединениям правильный вид, GDI+ должен знать, какими являются линии, имеющие общие точки: отдельными или соединенными. Об этом можно сообщить GDI+, используя метод *DrawLines* вместо *DrawLine*.

Дополнительный довод в пользу *DrawLines* — большая производительность. Приведенные программы-примеры не продемонстрировали особого прироста производительности, да это для них и не важно. На самом деле мы еще не задействовали метод *DrawLines* в полной мере. Видите ли, *DrawLines* создан не для рисования прямых, его настоящее назначение — кривые. Хитрость *DrawLines* в том, что он способен нарисовать множество очень коротких отдельных линий, позволяя изобразить любую кривую, которая может быть задана математически.

Можно спокойно передавать сотни и даже тысячи структур *Point* или *PointF* при единственном вызове метода *DrawLines* — он создан именно для этого Даже если передать *DrawLines* миллион структур *Point* или *PointF,* их визуализация займет всего пару секунд.

Сколько потребуется точек, чтобы нарисовать кривую? Пожалуй, меньше миллиона. Максимальная гладкость кривой достигается, если число ее точек по крайней мере равно числу пикселов. Часто вы можете примерно аппроксимировать это число.

Этот фрагмент кода рисует один цикл синусоиды, вписанный в клиентскую область

SineCurve.cs

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

см. след. стр.

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

class SineCurve: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new SineCurve());
     }
     public SineCurve()
     {
          Text = "Sine Curve";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          PointF[] aptf - new PointF[cx];

          for (int i = 0; i < cx; i++)
          {
               aptf[i].X * i;
               aptf[i].Y = cy / 2 * (1 -(float)
                                   Math.Sin(i * 2 * Math.PI / (cx - 1)));
          Г
          grfx.DrawLines(new Pen(clr), aptf);
     }
}
```

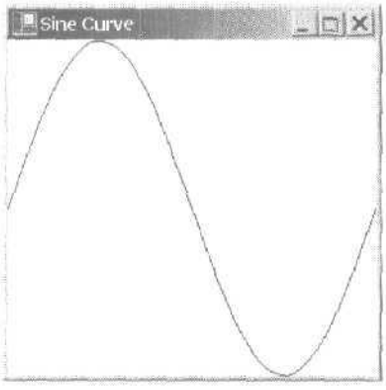
Это первая программа из этой книги, в которой применен тригонометрический метод из класса *Math* — очень важного класса, определяемого в пространстве имен *System*. Подробнее об этом классе см. приложение Б. Аргументы тригонометрических методов выражаются в радианах, а не в градусах. Класс *Math* также включает два поля типа *const* с именами *PI* и *E*, которые удобно использовать с методами этого класса. Теперь больше не надо помещать в начало программы ничего такого:

```
#define PI 3.14159      // Здорово, что теперь это можно убрать!
```

Однако замечу, что большинство методов класса *Math* возвращают значения типа *double*, которые нужно явно привести к типу *float*, прежде чем использовать в *Point!"* и подобных структурах.

Возможно, будет полезно разобрать оператор присваивания для свойства У массива *PointF* в подробностях. Аргумент функции *MathSin* выражается *в* радианах. Полный круг (360°) соответствует 2π радиан. Таким образом, аргумент может принимать значения от 0 (когда $z = 0$) до 2π (когда $i = ClientSize.Width - 1$). Значение метода *MathSin* находится в интервале от -1 до $+1$, поэтому обычно его нужно умножить на полувысоту клиентской области. Полученные значения будут изменяться от $-(ClientSizeHeight/2)$ до $+(ClientSizeHeight/2)$. Затем надо вычесть их из половины высоты клиентской области, чтобы результат варьировался от О до *ClientSize Height*. Но здесь я использовал более эффективный код. прибавляющий 1 к отрицательному результату метода *Sin,* получая в результате числа от С

до 2, которые умножаются на полувысоту клиентской области. Вот как выглядит результат выполнения этой программы:



Кривые и параметрические уравнения

Запрограммировать синусоиду довольно просто, поскольку ее значения *y* являются простой функцией *x*. Однако вообще программирование кривых сложнее. Например, уравнение единичной окружности (т. е. окружности с радиусом 1), с центром в начале координат обычно дают как:

$x? + y^2 = 1$

В более общем виде окружность с радиусом r может быть представлена как:

$x^2 + y^2 = r^2$

Но если попытаться представить это уравнение в виде зависимости *y* от *x,* то вот что получится:

$y = \pm \sqrt{r^2 - x^2}$

Здесь возникает целый ряд проблем. Во-первых, каждому значению *x* соответствуют пара значений *y*. Во-вторых, некоторые значения *x* недопустимы: *x* должен быть в интервале от -г до $+r$. И в-третьих, проблема практического толка возникает собственно при рисовании окружности на основе этого уравнения. Полученная функция нелинейна: при значениях *x,* близких к 0, изменения *x* дают относительно небольшие изменения *y*. Когда же *x* приближается к r или $-r$, изменения *x* дают гораздо большие изменения *y*.

Более общий подход к рисованию кривых применяет *параметрические* уравнения, в которых обе координаты каждой точки, *x* и *y,* вычисляются на основе значения третьей переменной, часто называемой t. Интуитивно *t* можно воспринимать как время или какой-то абстрактный показатель, нужный для исчерпывающего определения кривой. При программировании графики в Windows Forms можно считать, что переменная *t* изменяется от 0 до числа структур *PointF* в массиве минус 1.

Параметрические уравнения, определяющие единичную окружность, таковы:

$$x(t) = \cos(t)$$
$$y(t) = \sin(t)$$

Для t, изменяющегося от 0 до 2л радиан, эти уравнения задают окружность с центром в точке с координатами (0, 0) и радиусом 1.

Сходным образом может быть задан и эллипс:

$$x(t) = RX\cos(t)$$
$$y(t) = RY\sin(t)$$

Оси эллипса параллельны горизонтальной и вертикальной осям координат. Длина горизонтальной оси эллипса равна $2 \times RX$, вертикальной — $2 \times RY$. Эллипс также центрирован относительно начала координат. Чтобы переместить центр эллипса в точку с координатами *(CX, CY),* формулы должны принять такой вид:

$$x(t) = CX + RX\cos(t)$$
$$y(t) = CY + RY\sin(t)$$

А вот программа для рисования эллипса, вписанного в экран:

PolyEHipse.cs

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

class PolyEllipse: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new PolyEllipse());
     }
     public PolyEllipse()
     {
          Text = "Ellipse with DrawLines";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          int      iNum = 2 * (cx + cy);
          PointF[] aptf = new PointF[iNum];

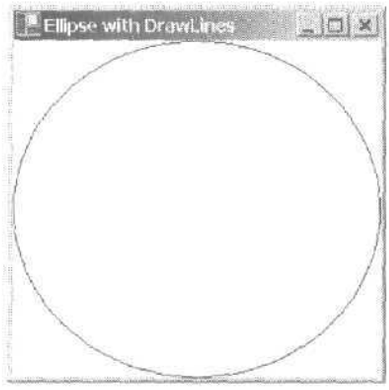
          for (int i = 0; i < iNum; i++)
          {
               double dAng = i * 2 * Math.PI / (iNum - 1);

               aptf[i].X = £cx - 1) / 2f * (1 + (float)Math.Cos(dAng));
               aptf[i].Y = (cy - 1) / 2f * <1 + (float)Math.Sin(dAng));
          }
```

см. след. стр.

```
            grfx.DrawLines(new Pen(clr), aptf);
        }
    }
```

Поскольку центр эллипса находится на пересечении прямых, делящих экран надвое по ширине и высоте, а ширина и высота эллипса равны ширине и высоте области экрана, я смог несколько упростить формулы. Я аппроксимировал число точек массива как число точек, достаточное для изображения прямоугольника, периметр которого равен периметру экрана,



Далее вы увидите, что в классе *Graphics* есть метод *DrawEllipse*. Возникает вопрос: зачем было рисовать эллипс «вручную»? Назовем это просто упражнением для подготовки к изучению следующей программы, рисующей фигуру, простого метода для изображения которой точно нет в классе *Graphics*,

Spiral.cs

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

class Spiral: PrintableForm
{
    public new static void ttain()
    {
        Application.Run(new Spiral());
    }
    public SpiralO
    {
        Text я "Spiral";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
```

см. след. стр.

```
    {
        const int iNumRevs   = 20;
        int       iNumPoints = iNumRevs * 2 * (cx + cy);
        PointF[]  aptf       = new PointF[iNumPoints];
        float     fAngle, fScale;
        for (int i = 0; i < iNumPoints; i++)
        {
            f Angle = (float)(i * 2 * Math.PI /(iNumPoints / iNumRevs));
            fScale я 1 - (float)i / iNumPoints;

            . aptf[i].X = (float)(cx / 2 * (1 * fScale * Math. Cos(f Angle)));
            aptf[i].Y = (float)(cy / 2 * (1 + fScale * Math.Sin(fAngle)));
        }
        grfx.DrawLines(new Pen(clr), aptf);
    }
}
```

А вот как выглядит эта фигура:



Вездесущий прямоугольник

Прямоугольники не так часто встречаются в природе, но определенно являются самой распространенной формой объектов, созданных человеком. Их можно встретить повсюду: страница, которую вы сейчас читаете — прямоугольник, абзацы ее текста имеют форму прямоугольника, расположенный перед этим разделом снимок экрана — тоже. Стол, за которым вы сидите, кровать, на которой вы отдыхаете, тоже скорее всего прямоугольные. И окно, куда вы посматриваете, когда надоедает читать эту книгу, вероятно, прямоугольное.

Несомненно, прямоугольник можно нарисовать при помощи методов *DrawLine* или *DrawLines* (что мы и делали, когда очерчивали клиентскую область), но проще сделать это методом *DrawRectangle*. В каждой из трех его версий прямоугольник определяется точкой, задающей положение верхнего левого угла, шириной и высотой. Аналогично определяется структура *Rectangle*, которую использует одна из версий *DrawRectangle*:

Методы *DrawRectangle* класса *Graphics*

```
void DrawRectangle(Pen pen, int x, int y, int cx, int cy)
void DrawRectangle(Pen pen, float x, float y, float cx, float cy)
void DrawRectangle(Pen pen, Rectangle rect)
```

Как ни странно, метод *DrawRectangle*, использующий структуру *RectangleF*, отсутствует. Возможно, программист просто забыл пометить его модификатором *public*. Надеюсь, мы все же увидим такой метод в следующем выпуске,

Ширина и высота прямоугольника должны быть больше 0. Прямоугольники с отрицательной шириной и высотой не вызовут исключения, но и не будут нарисованы.

При рисовании прямоугольников часто возникают ошибки смещения на 1 пиксел, поскольку толщина контура прямоугольника составляет минимум 1 пиксел. Входит ли вообще толщина контуров в ширину и высоту прямоугольника и, если да, то со всех сторон или с какой-то одной?

При использовании свойств пера по умолчанию (об этом я расскажу в главе 17) методу *DrawRectangle* передаются значения ширины и высоты, равные 3, в результате получается то, что показано ниже (конечно, в увеличенном виде):



В верхнем левом углу фигуры находится пиксел с координатами (x, y). Если задать высоте и ширине 2, будет нарисован прямоугольник размером 3 на 3 пиксела, внутренняя область которого состоит лишь из одного пиксела;



Если высота и ширина равны 1. рисуется квадрат размером 2×2 пиксела. Может возникнуть искушение поместить свойство формы *ClientRectangle* прямо в вызов *DrawRectangle*, чтобы очертить клиентский прямоугольник снаружи видимым контуром:

```
grfx.DrawRectangle(pen, ClientRectangle);  // Избегайте этого!
```

Это работать не будет! Правая и нижняя стороны прямоугольника станут невидимыми. Ниже показана программа, которая должным образом выводит целый прямоугольник как в клиентской области, так и на принтер. Я сделал прямоугольник красным, чтобы он был лучше виден на экране.

OutlineClientRectangle.cs

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

см. след. стр.

```
class OutlineClientRectangle: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new OutlineClientRectangle());
    }
    public OutlineClientRectangle()
    {
        Text = "Client Rectangle";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawRectangle(Pens.Red, 0, 0, cx - 1, cy - 1);
    }
}
```

Попробуйте указать в качестве последних двух аргументов метода *DrawRectangle* просто значения *cx* и *cy,* не вычитая из них 1, и вы увидите, что правая и нижняя стороны прямоугольника станут невидимыми в клиентской области. Это следствие той же проблемы, что возникает при использовании свойства *ClientRectangle* в вызове *DrawRectangle.*

Класс *Graphics* также включает два метода, позволяющих рисовать несколько прямоугольников:

Методы *DrawRectangles* класса *Graphics*

```
void DrawRectangles (Pen pen, Rectangle[] arect)
void DrawRectangles (Pen pen, RectangleF[] arectf)
```

Пользы от них намного меньше, чем от *DrawLines.* Но предположим, что есть структура *RectangleF* с именем, скажем, *rectf.* На основе этой структуры нужно нарисовать один прямоугольник. Помня об отсутствии перегруженных версий метода *DrawRectangle,* можно решить поставленную задачу методом *DrawRectangles:*

```
grfx.DrawRectangles(pen, new RectangleF[] { rectf });
```

Многоугольники

С точки зрения математики, многоугольник — это замкнутая фигура с тремя и более сторонами, например, треугольники, квадраты, пяти-, шестиугольники и т. *д.* Вот два метода класса *Graphics,* рисующих многоугольники:

Методы ***DrawPolygon*** **класса *Graphics***

```
void DrawPolygon(Pen pen, Point[] point)
void DrawPolygon(Pen pen, PointF[] point)
```

По функциональности *DrawPolygon* очень похож на *DrawLines* за исключением того, что он автоматически замыкает контур фигуры, соединяя ее первую и последнюю точки линией. Рассмотрим, например, следующий массив структур *Point:*

```
Point[] apt = {new Point (0, 0), new Point (50, 100), new Point (100, 0)};
```

Вызов:

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

рисует две линии в виде буквы V. а вызов;

```
grfx.DrawPolygon(pen, apt);
```

рисует треугольник

Иногда можно имитировать вызов *DrawPolygon* с помощью методов *DrawLines* и *DrawLine*:

```
DrawLines(pen, apt);
DrawLine(pen, apt[apt.Length-1], apt[0]);
```

Однако если приходится иметь дело с толстыми линиями с прорисованными концами и соединениями, такого же эффекта, как при использовании *DrawPolygon*, не получится.

Эллипсы: простой метод

Вы уже знаете, как нарисовать эллипс при помощи метода *DrawLines*, но есть и более простой подход, использующий те же аргументы, что и *DrawRectangle:*

Методы *DrawEllipse* класса *Graphics*

```
void DrawEllipse(Pen pen, int x, int y, int cx, int cy)
void DrawEllipse(Pen pen, float, x, float y, float cx, float cy)
void DrawEllipse(Pen pen, Rectangle rect)
void DrawEllipse(Pen pen, RectangleF rectf)
```

Методы *DrawEllipse* соответствуют методам *DrawRectangle*. Например, здесь нарисован эллипс с шириной и высотой, равными 3:



Если задать ширину и высоту равными 1, в результате будет нарисован закрашенный квадрат со стороной 2 пиксела.

Это означает (как и в случае *DrawRectangle*), что нужно уменьшить ширину и высоту эллипса на 1, чтобы он уместился в области размером *cx* на *cy*.

ClientEllipse.cs

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

см. след. стр.

```
class ClientEllipse: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new ClientEllipse());
    }
    public ClientEllipse()
    {
        Text = "Client Ellipse";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        grfx.DrawEllipse(new Pen(clr), 0, 0, cx - 1, cy - 1);
    }
}
```

Если два последних аргумента метода *DrawEllipse* приравнять *cx* и *cy,* то правая и нижняя стороны фигуры окажутся несколько обрезанными.

Дуги и секторы

Дуга (по крайней мере в случае Windows Forms) — это сегмент эллипса. Чтобы определить дугу, нужно указать те же сведения, что и для определения эллипса, плюс углы для начальной и конечной точек дуги. В силу этих причин каждая из четырех версий метода *DrawArc* требует на два аргумента больше, чем их нужно для *DrawEllipse*:

Методы *DrawArc* класса *Graphics*

```
void DrawArc(Pen pen, int x, int y, int cx, int cy,
             int iAngleStart, int iAngleSweep)
void DrawArc(Pen pen, float x, float y, float cx, float cy,
             float fAngleStart, float fAngleSweep)
void DrawArc(Pen pen, Rectangle rect,
             float fAngleStart, float fAngleSweep)
void DrawArc(Pen pen, RectangleF rectf,
             float fAngleStart, float fAngleSweep)
```

Пара дополнительных аргументов задает углы, определяющие начало дуги и ее длину. Эти углы (которые могут быть как положительными, так и отрицательными) измеряются по часовой стрелке, Б градусах и откладываются от горизонтальной оси справа от центра эллипса (стрелка часов в этом положении покажет 3 часа):



Вот программа, рисующая эллипс с пунктирным контуром. Угловой размер штрихов пунктира составляет 10°, промежутков между штрихами — 5.

DashedEllipse.cs

```
//-------------------------------------------
// DashedEllipse.cs © 2001 fcy Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DashedEllipse: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new DashedEllipse());
     }
     public DashedEllipse()
     {
          Text = "Dashed Ellipse Using DrawArc";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen       pen  = new Pen(clr);
          Rectangle rect = new Rectangle(0, 0, cx - 1, cy - 1);

          for (int iAngle = 0; iAngle < 360; iAngle += 15)
               grfx.DrawArc(pen, rect, iAngle, 10);
     }
}
```

Полученный эллипс выглядит так:



В Win32 API есть функция *RoundRect,* рисующая прямоугольник с закругленными углами. Эта функция принимает 4 аргумента, указывающих координаты верхнего левого и правого нижнего углов прямоугольника плюс еще два, определяющие ширину и высоту эллипса, используемого для скругления углов прямоугольника.

В классе *Graphics* метода *RoundRect* нет, но мы вполне можем попытаться имитировать его.

RoundRect.cs

```
//---------------------------------------
// fiountfRect.cs © 2061 by Charles Petzold
//---------------------------------------
using System;
«sing System.Drawing;
using System.Windows.Forms;

class RoundRect: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new RoundRect());
     }
     public RoundRect()
     {
          Text = "Rounded Rectangle";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          RoundedRectangle(grfx,Pens.fled,
                           new Rectangle(0, 0, cx - 1, cy - 1),
                           new Size(cx / 5, cy / 5));
     }
     void RoundedRectangle(Graphics grfx, Pen pen, Rectangle rect, Size size)
     {
```

см. след. стр.

```
            grfx.DrawLine(pen, rect.Left  + size.Width / 2, rect.Top,
                               rect.Right - size.Width / 2, rect.Top);

            grfx.DrawArc(pen, rect.Right - size.Width, rect.Top,
                              size.Width, size.Height, 270, 90);

            grfx.DrawLine(pen, rect.Right, rect.Top * size-Height / 2,
                               rect.Right, rect.Bottom - size.Height / 2);

            grfx.DrawArc(pen, rect.Right - size.Width,
                              rect.Bottom - size.Height,
                              size.Width, size.Height, 0, 90);

            grfx.DrawLine(pen, rect.Right - size.Width / 2, rect.Bottom,
                               rect.Left + size.Width / 2, rect.Bottom);

            grfx.DrawArc(pen, rect.Left, rect.Bottom - size.Height,
                              size.Width, size.Height, 90. 90);

            grfx.DrawLine(pen, rect.Left, rect.Bottom - size.Height / 2,
                               rect.Left, rect.Top + size.Height / 2);

            grfx.DrawArc(pen, rect.Left, rect.Top,
                              size.Width, size.Height, 180, 90);
        }
    }
```

Написанному мной методу *RoundedRectangle* требуются такие аргументы: *Rectangle*, указывающий расположение и размер прямоугольника и *Size*, задающий ширину и высоту эллипса для скругления углов. Я написал этот метод, чтобы сохранить согласованность с размерами прямоугольника, нарисованного методом *DrawRectangle*. То есть, когда ширина и высота фигуры устанавливаются равными соответствующим измерениям клиентской области минус 1, будет видима вся фигура. Мой метод по очереди вызывает *DrawLine* и *DrawArc*, прорисовывает сначала верхний контур фигуры и продолжает рисовать остальные части контура по часовой стрелке.



Однако я не *могу* без колебаний рекомендовать этот способ как общий подход к рисованию прямоугольников с закругленными углами. Отдельные линии и дуги здесь рисуют отдельные вызовы методов *DrawLine* и *DrawArc*. Это значит, что у каждой из восьми частей контура фигуры будут прорисованы концы, а соединения между ними — нет. Правильный путь объединения прямых и кривых в цельную фигуру состоит в использовании графического *контура* (path). Как это делается, я покажу в главе 15.

У методов *DrawPie* те же аргументы, что и у *DrawArc*, но они отличаются тем, что соединяют концы дуги с центром эллипса линиями, создавая замкнутую область.

Методы *DrawPie* класса *Graphics*

```
void DrawPie(Pen pen, int x, int y, int cx, int cy,
             int iAngleStart, int iAngleSweep)
void DrawPie(Pen pen, float x, float y, float cx, float cy,
             float fAngleStart, float fAngleSweep)
void DrawPie(Pen pen, Rectangle rect,
             float fAngleStart, float fAngleSweep)
void DrawPie(Pen pen, RectangleF rectf,
             float fAngleStart, float fAngleSweep)
```

В мире деловой графики очень уважают круговые диаграммы — это факт. Проблема в том, что если уж пришлось программировать круговую диаграмму, хорошо бы украсить ее трехмерными и другими эффектами. Здесь у метода *DrawPie* меньше возможностей, чем вы думаете. Так или иначе, вот программа, строящая круговую диаграмму на основе массива значений (хранимого как поле), который я создаю специально для этой цели.

PieChart.cs

```
//-------------------------------------
// PieChart.cs * 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PieChart: PrintableForm
{
     int[] aiValues = { 50, 100, 25, 150, 100, 75 };

     public new static void Main()
     {
          Application.Run(new PieChart());
     }
     public PieChart()
     {
          Text = "Pie Chart";
```

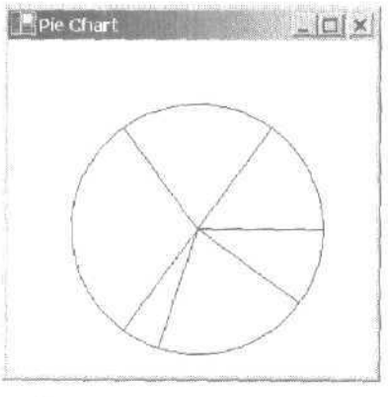
см. след. стр.

```
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Rectangle rect   = new Rectangle(50, 50, 200, 200);
        Pen       pen    = new Pen(clr);
        int       iTotal = fl;
        float     fAngle = 0, fSweep;

        foreach(int iValue in aiValues)
            iTotal += iValue;

        foreach(int lvalue in aiValues)
        {
            fSweep = 3SOf * iValue / iTotal;
            DrawPieSlice(grfx, pen, rect, fAngle, fSweep);
            fAngle += fSweep;
        I
    }
    protected virtual void DrawPieSlice(Graphics grfx, Pen pert,
                                        Rectangle rect,
                                        float fAngle, float fSweep)
    {
        grfx.DrawPie(pen, rect, fAngle, fSweep);
    }
}
```

Обратите внимание на определение *Rectangle* в методе *DoPage*. Это единственная программа в данной главе, использующая абсолютные координаты и размеры, так как круговые диаграммы в форме эллипса не слишком привлекательны. Метод *DoPage* суммирует значения массива и рассчитывает угол для каждого сектора, деля значение для этого сектора на сумму значений всех секторов и умножая частное на 360°. Вот результат:



Простите, но я просто не могу позволить вам думать, что это лучшая диаграмма, которую я способен нарисовать! К счастью, я оказался достаточно прозорливым, чтобы поместить вызов *DrawPie* в виртуальную функцию в классе *PieChart*. Это позволяет легко переопределить метод в программе BetterPieChart.

BetterPieChart.cs

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

class BetterPieChart: PieChart
{
     public new static void Main()
     {
          Application.Run(new BetterPieChart());
     }
     public BetterPieChart()
     {
          Text = "Better " + Text;
     }
     protected override void DrawPieSlice(Graphics grfx, Pen pen,
                                          Rectangle rect,
                                          float fAngle, float fSweep)
     {
          float fSlice = (float)(2 * Math.PI * (fAngle + fSweep / 2) / 360);

          rect.Offset((int)(rect.Width  / 10 * Math.Cos(fSlice)),
                      (int)(rect.Height / 10 * Math.Sin(fSlice)));

          base.DrawPieSlice(grfx, pen, rect, fAngle, fSweep);
     }
}
```

Переменная *fSlice* — это угол между горизонтальной осью и центром сектора, преобразованный в радианы. Это значение понадобилось мне для расчета смещения по осям *x* и *y* прямоугольников, определяющих размер и положение секторов диаграммы. В результате все секторы отошли от центра, и на изображении диаграммы они не соприкасаются:



На этом коллекция методов класса *Graphics*, рисующих линии, далеко не исчерпана. Методы *DrawBezier*, *DrawBeziers*, *DrawCurve* и *DrawClosedCurve* позволяют рисовать кривые посложнее, чем эллиптические дуги. Описание этих методов — в главе 11. Можно объединить набор прямых и кривых в графический контур и визуализировать его при помощи метода *DrawPath*. К этой теме мы обратимся в главе 15.

Заливка прямоугольников, эллипсов и секторов

Хотя некоторые из обсуждавшихся выше методов класса *Graphics* определяют замкнутые области, эти методы лишь прорисовывают заданным пером контуры области, не заполняя при этом ее внутреннюю часть. Этим методам, имена которых начинаются префиксом *Draw*, соответствуют методы с префиксом *Fill*, выполняющие заливку оконтуренных областей. Первый аргумент этих методов — объект *Brush*, используемый для заливки области.

Вот четыре версии метода *FillRectangle:*

Методы *FillRectangle* класса *Graphics*

```
void FillRectangle(Brush brush, int x, int y, int cx, int cy)
void FillRectangle(Brush brush, float x, float y, float cx, float cy)
void FillRectangle(Brush brush, Rectangle rect)
void FillRectangle(Brush brush, RectangleF rectf)
```

Ширина и высота результирующей фигуры определяется значениями аргументов метода. Например, если ширина и высота равны 3, вызов *FillRectangle* нарисует закрашенный квадрат со стороной 3 пиксела, верхний левый угол которого находится в пикселе с координатами (*x*, y). Если требуется нарисовать *и* залить некоторый прямоугольник, первым надо вызвать метод *FillRectangle,* чтобы при заливке не были затерты никакие линии,

Класс *Graphics* также включает два метода *FillRectangles*:

Методы *FillRectangles* класса *Graphics*

```
void FillRectangles(Brush brush, Rectangle[] arect)
void FillRectangles(Brush brush, RectangleF[] arect)
```

Вызов метода *FillRectangles* дает тот же результат, что несколько вызовов *FillRectangle*.

Имеется четыре версии метода *FillEllipse*, их аргументы не отличаются от аргументов *DrawEllipse*:

Методы *FillEllipse* класса *Graphics*

```
void FillEllipse(Brush brush, int x, int y, int cx, int cy)
void FillEllipse(Brush brush, float x, float y, float cx, float cy)
void FillEllipse(Brush brush, Rectangle rect)
void FillEllipse(Brush brush, RectangleF rectf)
```

Метод *FillEllipse* работает несколько иначе, чем все методы, о которых говорилось до сих пор. Предположим, что задан эллипс с центром в точке (0, 0), высо-

той и шириной 20. Как известно, *DrawEllipse* нарисует фигуру, состоящую по горизонтали и вертикали из пикселов от 0 до 20, Таким образом, действительная высота и ширина фигуры составит 21 пиксел.

Область, закрашиваемая *FillEllipse*, по сути включает в себя пикселы с 1 по 19 по горизонтали и вертикали, таким образом, эффективный размер закрашиваемой области равен 19 пикселам. Я сказал «по сути», поскольку считается, что слева остается пиксел, занимающий позицию 0! Кроме того, эллипс, нарисованный *DrawEllipse*, всегда немного перекрывается с областью, закрашенной *FillEllipse*. Если нужно получить закрашенный эллипс с контуром, сначала надо вызывать метод *FillEltipse*, а затем — *DrawEllipse*.

Также имеются три метода *FillPie*:

Методы *FillPie* класса *Graphics*

```
void FillPie(Brush brush, int x, int y, int cx, int cy,
             int iAngle, int iSweep)
void FillPie(Brush brush, float x, float y, float cx, float cy,
             float fAngle, float fSweep)
void FillPie(Brush brush, Rectangle rect,
             float fAngle, float fSweep)
```

Ошибка смещения на 1 пиксел

Теперь, когда мы изучили все методы рисования прямоугольников и эллипсов, пора сравнить их с точки зрения ошибки смещения на 1 пиксел. Следующая программа рисует прямоугольники и эллипсы размером 4 × 4 при помощи шести методов: *DrawRectangle*, *DrawRectangles*, *DrawEllipse*, *FillRectangle*, *FillRectangles* и *FillEllipse*.

FourByFours.cs

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

class FourByFours: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new FourByFours());
     }
     public FourByFours()
     {
          Text = "Four by Fours";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
```

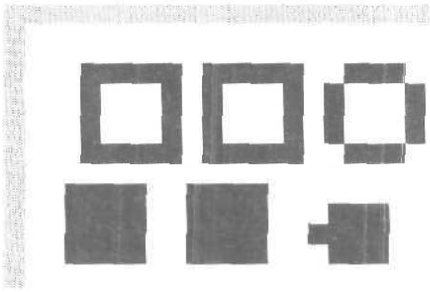
см. след. стр.

```
    {
        Pen   pen   = new Pen(clr);
        Brush brush = new SolidBrush(clr);

        grfx.DrawRectangle(pen,  new Rectangle(2,  2,  4,  4»;
        grfx.DrawRectangles(pen, new Rectangle[]
                                        {new Rectangle(8,  2,  4,  4)});
        grfx.DrawEllipse(pen,  new Rectangle(14,  2,  4,  4));

        grfx.FillRectangle(brush, new Rectangle(2, 8, 4, 4));
        grfx, FillRectangles(brush, new Rectangle[]
                                        {new fiectangleCS, 8,  4,  4)});
        grfx.FillEllipse(brush, new Rectangle(14, 8, 4, 4));
    }
}
```

Вот как выглядит результат, если увеличить его до размера, пригодного для анализа:



Как видно, особенность методов *DrawRectangle*, *DrawRectangles* и *DrawEllipse* в том, что визуализированные ими фигуры на один пиксел выше и шире, чем должны быть, исходя из заданного размера. Кроме небольшого утолщения слева, фигуры, нарисованные методом *FillEllipse*, на один пиксел уже и ниже, чем фигуры размера 4 × 4, нарисованные методами *FillRectangle* и *FillRectangles*

Многоугольники и режим заливки

В завершение (по крайней мере этой главы) нам осталось рассмотреть метод *FillPolygon*. От других закрашиваемых областей многоугольники отличаются тем, что определяющие их линии могут пересекаться и накладываться одна на другую. Это усложняет дело, так как есть два разных способа заливки многоугольников. Существует четыре метода *FillPolygon*:

Методы *FillPolygon* класса *Graphics*

```
void FillPolygon(Brush brush, Point[] apt)
void FillPolygon(Brush brush, PointF[] apt)
void FillPolygon(Brush brush, Point[] apt, FillMode fm)
void FillPolygon(Brush brush, PointF[] apt, FillMode fm)
```

Они во всем похожи на метод *DrawPolygon*, кроме одного необязательного аргумента. *FillMode* — это перечисление, определяемое в пространстве имен *System.Drawing.Drawing2D*. У него всего два возможных значения:

Перечисление *FillMode*

Член	Значение	Комментарий
Alternate	0	Задано по умолчанию, чередует закрашенные и незакрашенные области фигуры.
Winding	1	Закрашивает максимально возможное число внутренних областей фигуры,

Режим заливки имеет значение, лишь когда линии, определяющие многоугольник, пересекаются. Он определяет, какие из замкнутых областей будут закрашены, а какие — нет. Если режим заливки в методе *FillPolygon*не задан, по умолчанию используется *FillModeAlternate*. При этом замкттая область заполняется, только если она отделена от бесконечночти нечетным числом границ.

Классический пример — пятиконечная звезда. Ее внутренний пятиугольник закрашивается в режиме winding, но остается свободным в режиме alternate,

FillModesClassical.cs

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

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

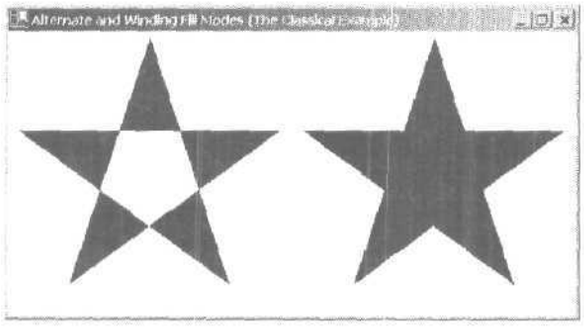
см. след. стр.

```
                                (int)(cx *(0.25 + 0.24 * Math.Cos(dAngle))),
                                (int)(cy *(0.50 + 0.48 * Math.Sin(dAngle))));
            }
            grfx.FillPolygon(brush, apt, FillMode.Alternate);

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

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

Первый цикл *for* определяет пять концов звезды, выводимой в левой половине клиентской области. Для такого многоугольника устанавливается режим заливки alternate. Второй цикл *for* сдвигает вершины звезды в правую часть клиентской области, где многоугольник закрашивается в режиме winding.



Обычно в режиме winding закрашиваются все замкнутые области. Но не всегда все так просто — есть и ряд исключений. Чтобы определить, будет ли закрашена область в режиме winding, мысленно проведите прямую из любой точки этой области в бесконечность. Если воображаемая линия пересекает нечетное число границ, область закрашивается (так же, как в режиме alternate). Если же воображаемая линия пересекает четное число границ, то область может быть закрашена, а может остаться и свободной. Такая область закрашивается, если число граничных линий, направленных по часовой стрелке, не равно числу линий, направленных против часовой стрелки,

Немного подумав, можно нарисовать фигуру, внутренняя область которой останется незакрашенной в режиме winding.

FillModesOddity.cs

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

см. след. стр.

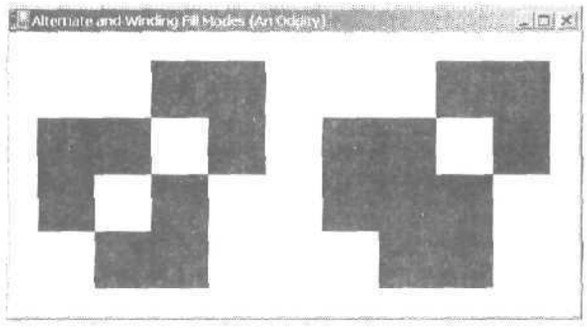
```
class FillModesOddity: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new FillModesOddity());
    }
    public FillModesQddityO
    {
        Text = "Alternate and Winding Fill Modes (An Oddity)";
        ClientSize = new Size(2 * ClientSize.Height, ClientSize.Height);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush   brush = new SolidBrush(clr);
        PointF[] aptf = { new PointF(0.1f, 0.7f), new PointF(0.5f, 0.7f),
                          new PointF(0.5f, 0.1f), new PointF(0.9f, 0.1f),
                          new PointF(0.9f, 0.5f), new PointF(0.3f, 0.5f),
                          new PointF(0.3f, 0.9f), new PointF(0.7f, 0.9f),
                          new PointF(0.7f, 0.3f), new PointF(0.1f, 0.3f)};

        for (int i = 0; i < aptf.Length; i++)
        {
            aptf[i].X *= cx / 2;
            aptf[i].Y *= cy;
        }
        grfx.FillPolygon(brush, aptf, FillMode.Alternate);

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

        grfx.FillPolygon(brush, aptf, FillMode.Winding);
    }
}
```

Ботрезультат:



В следующих главах мы обсудим методы *Fill* подробнее: *FillClosedCurve* — в главе 13, а *FillRegion* и *FillPath* — в главе 15.



ГЛАВА

6

Работа с клавиатурой

Несмотря на сложность интерфейсов современных графических сред, ориентированных на визуальную работу по принципу «указал и щелкнул», включающих работу с мышью, элементами управления, меню и диалоговыми окнами, для большинства приложений клавиатура остается главным устройством ввода. Кроме того, у клавиатуры самая длинная родословная из всех компонентов ПК. Она берег начало в 1874 г. от первой пишущей машинки системы Ремингтона. За несколько десятилетий эволюции возможности компьютерной клавиатуры существенно расширились по сравнению с пишущей машинкой: появились функциональные клавиши, клавиши перемещения курсора и отдельная цифровая клавиатура.

У большинства компьютеров клавиатура и мышь — единственные средства ввода информации пользователем. Несмотря на интенсивные исследования в области технологий ввода в компьютер алфавитно-цифровой информации при помощи распознавания речи и рукописного ввода, сейчас эти методы не обеспечивают (и вряд ли обеспечат в будущем) такую же точность, как клавиатура. И уж конечно, компьютер никогда не сможет разобрать плохой почерк или речь с сильным акцентом лучше, чем человек.

Игнорируемая клавиатура

Хотя в большинстве приложений клавиатура — важный источник информации, вводимой пользователем, нет никакой необходимости писать код, реагирующий абсолютно на все получаемые приложением события клавиатуры. Microsoft Windows и библиотеки классов Windows Forms обрабатывают многие функции клавиатуры самостоятельно.

Например, приложения обычно игнорируют нажатия клавиш при выборе пунктов меню. Программы не должны вести мониторинг клавиш, поскольку меню самостоятельно обрабатывает все действия клавиатуры, связанные с выбором пун-

ктов и команд, и посредством события сообщает программе о выборе того или иного пункта меню.

Программы Windows Forms часто определяют комбинации клавиш для быстрого доступа к часто используемым пунктам меню. В таких комбинациях часто участвует клавиша Ctrl в сочетании с буквой или функциональной клавишей (например, комбинация Ctrl+S сохраняет файл). И здесь приложение может не заботиться о трансляции нажатых клавиш в команды меню: меню сделает это самостоятельно.

У диалоговых окон также есть интерфейс, использующий клавиатуру. Но программам обычно не требуется вести мониторинг клавиатуры, пока активно диалоговое окно. В это время программа уведомляется о *результатах* нажатия любых клавиш через события. Разместив элементы управления на форме, можно не беспокоиться о навигации по ним при помощи клавиш, таких как Tab или клавиши управления курсором (достаточно проследить во время разработки, чтобы порядок перехода фокуса по клавише Tab был логичным). Навигацию пользователя по элементам управления полностью обрабатывает за вас система. Вводимую с клавиатуры информацию можно также обрабатывать, применяя элементы управления вроде *TextBox* и *RichTextBox*. По завершении ввода пользователем они передают результирующую текстовую строку программе.

Несмотря на все вспомогательные средства, остается множество приложений, где нужно напрямую обрабатывать вводимую с клавиатуры информацию. А если вы собираетесь писать собственные элементы управления, то вам обязательно понадобятся кое-какие сведения о клавиатуре.

У кого фокус?

Клавиатура в Windows — разделяемый ресурс. Вводимую информацию все приложения получают с одной и той же клавиатуры, хотя у некоторых клавиш должен быть единственный получатель. В случае программ Windows Forms этим получателем должен быть объект типа *Control* (класса, в котором реализована обработка клавиатуры) или его потомок, например *Form*.

Объект, получающий событие клавиатуры, обладает *фокусом ввода* (input focus). Это понятие тесно связано с понятием *активной формы* (active form). Обычно активную форму легко определить. Часто это расположенная поверх остальных на рабочем столе форма с подсвеченным заголовком (если таковой есть). Если вместо заголовка у активной формы рамка диалогового окна, то подсвечивается она. Если активная форма в данный момент свернута, ее окно на панели задач отображается как нажатая кнопка.

Активная форма доступна из единственного статического свойства, реализованного в *Form*:

Статическое свойство объекта *Form*

Тип	Свойство	Доступ
Form	*ActiveForm*	Чтение

Однако это свойство возвращает объект, а не *null*, только если активная форма - часть нашего приложения, Оно не может получать объекты, созданные другими приложениями!

Форма может попытаться сделать себя активной, вызвав следующий метод:

Методы Form (выборочно)

```
void Activate()
```

Обычно, если эта форма не самая верхняя на рабочем столе, данный метод заставит Windows сделать так, чтобы окно этой формы на панели задач стало мигать для привлечения внимания пользователя, который должен расположить эту форму поверх остальных окон и сделать ее активной. Кроме того, в классе реализованы два события, связанные с активной формой:

События *Form* (выборочно)

Событие	Метод	Делегат	Аргумент
Activated	*OnActivated*	*EventHandler*	*EventArgs*
Deactivate	*OnDeactivate*	*EventHandler*	*EventArgs*

Я решил упомянуть об этих свойствах, методах и событиях именно сейчас, так как они требуются не часто: программам редко приходится вмешиваться в активизацию собственных форм.

Фокус ввода — совсем другое дело. Если у активной формы есть дочерние элементы управления (расположенные в ее клиентской области подобно элементу управления *Panel* в примере из главы 4), то либо один из этих элементов управления, либо сама форма должна быть в фокусе. Элементы управления по-разному показывают, что они в фокусе: вокруг текста кнопки выводится точечная линия, а в текстовом поле — мигающая каретка. Ниже (в этой главе и следующих) мы еще вернемся к обсуждению вопросов, связанные с фокусом ввода.

Клавиши и символы

Клавиатура всегда генерирует некоторые числовые коды, но ее можно рассматривать по-разному;

- как совокупность отдельных физических клавиш;
- как средство генерации кодов символов,

В первом случае любой генерируемый ею код должен идентифицировать клавишу и указывать, нажата она или отпущена. Во втором — код, генерируемый при нажатии некоторой клавиши, идентифицирует уникальный символ из набора символов. Традиционно этим набором символов был ASCII, но в среде Windows Forms поддерживается набор символов Unicode.

У многих клавиш современного компьютера нет кодов символов. Ни функциональные клавиши, ни клавиши управления курсором их не генерируют. Поэтому программам, использующим ввод информации с клавиатуры нетривиальным способом, обычно приходится иметь дело с клавиатурой и как с совокупностью клавиш, и как с генератором символов *одновременно*.

Клавиши можно разделить на четыре общие группы,

- **Клавиши-переключатели** — Caps Lock, Num Lock, Scroll Lock и в некоторых случаях Insert. При нажатии такой клавиши включается ее состояние, а при

повторном нажатии — выключается. Состояние клавиш Caps Lock, Num Lock и Scroll Lock является общесистемным, т. е. если на компьютере работает несколько программ одновременно, не может быть, чтобы клавиша Caps Lock была включена для одной программы и в то же время отключена для другой. Как правило, на клавиатуре есть световые индикаторы состояния клавиш-переключателей

- **Клавиши управления регистром** — Shift, Ctrl и Alt. Будучи нажатой, такая клавиша меняет интерпретацию нажатия других клавиш. В библиотеке классов Windows Forms они называются *модификаторами*.
- **клавиши, не имеющие символов** — функциональные клавиши, клавиши управления курсором, Pause, Delete и в некоторых случаях Insert.
- **символьные клавиши** — буквы, цифры и другие символы, пробел, Tab, Backspace, Esc и Enter (клавиши Tab, Backspace, Esc и Enter также можно рассматривать как клавиши без символов),

Часто одна и та же физическая клавиша может генерировать различные коды символов в зависимости от состояния клавиш-переключателей и модификаторов. Скажем, клавиша A генерирует код символа A в верхнем *(A)* или в нижнем *(a)* регистре в зависимости от состояния клавиш Caps Lock и Shift. Иногда две разные физические клавиши (например, две клавиши Enter, имеющиеся на большинстве клавиатур) могут генерировать одинаковые коды символов,

Программы Windows Forms получают информацию, вводимую с клавиатуры, в виде событий. Сначала я опишу, как обращаться с клавиатурой как с совокупностью клавиш, а затем — как с генератором кодов символов.

Нажатые и отпущенные клавиши

Значительная часть функциональности клавиатуры реализована в классе *Control*. Он поддерживает следующие события и методы для обработки нажатия и отпускания клавиш:

События *Control* (выборочно)

Событие		Метод _ Делегат _ Аргумент	
KeyDown	*OnKeyDown*	*KeyEventHandler*	*KeyEventArgs*
KeyUp	*OnKeyUp*	*KeyEventHandler*	*KeyEventArgs*

Как и в любом классе, производном от *Control* (например, в *Form*), методы *OnKeyDown* и *OnKeyUp* можно переопределить:

```
protected override void OnKeyDown(KeyEventArgs kea)
{
    ⋮
}
protected override void OnKeyUp(KeyEventArgs kea)
{
    ⋮
}
```

Это обычный способ обработки событий клавиатуры в классе, производном от *Form*.

События, возникающие при нажатии и отпускании клавиш, также можно обрабатывать в объектах, порожденных от класса *Control* или его потомков. Но сначала нужно определить методы как делегаты *KeyEventHandler*:

```
void MyKeyDownHandler(object objSender, KeyEventArgs kea)
{
    ⋮
}
void MyKeyUpHandler(object objSender, KeyEventArgs kea)
{
    ⋮
}
```

Затем — зарегистрировать обработчики событий клавиш:

```
cntl.KeyDown += new KeyEventHandler(MyKeyDownHandler);
cntl.KeyUp   += new KeyEventHandler(MyKeyUpHandler);
```

Независимо от того, как это сделано, при нажатии или отпускании клавиши вы получите объект *KeyEventArgs* с такими свойствами:

Свойства объекта *KeyEventArgs*

Тип	Свойство	Доступ	Комментарий
Keys	*KeyCode*	Чтение	Идентифицирует клавишу.
Keys	*Modifiers*	Чтение	Идентифицирует состояния регистра.
Keys	*KeyData*	Чтение	Комбинация *KeyCode* и *Modifiers,*
bool	*Shift*	Чтение	Равно *true,* если нажата клавиша Shift,
bool	*Control*	Чтение	Равно *true,* если нажата клавиша Ctrl.
bool	*Alt*	Чтение	Равно *true,* если нажата клавиша Alt.
bool	*Handled*	Чтение/запись	Устанавливается обработчиком события (изначально равно *false)* .
int	*KeyValue*	Чтение	Возвращает *KeyData* в виде целочисленного значения.

Здесь куча избыточной информации. Из всех этих свойств нужны только *KeyData* и *Handled,* а значения остальных свойств можно вывести из *KeyData.* Но эта избыточность удобна: вероятно, вы обнаружите, что чаще всего используете именно свойства *KeyCode, Shift, Control* и *Alt.*

Первые три свойства относятся к одному типу — очень важному перечислению *Keys.* Свойство *KeyCode* сообщает, какая клавиша нажата (и это самое важное), свойство *Modifiers* — нажаты ли также клавиши Alt, Ctrl или Shift. Свойство *KeyData* комбинирует значения *KeyCode* и *Modifiers,* а *Shift, Control* и *Alt* дублируют данные свойства *Modifiers* булевыми значениями. Иногда элемент управления устанавливает свойство *Handled* как *true,* чтобы указать родительскому объекту, что событие клавиатуры используется им и не должно передаваться родительскому объекту. *KeyValue* возвращает ту же информацию, что и *KeyData,* но в виде целых чисел, а не значений перечисления *Keys.*

Перечисление *Keys*

Три свойства объекта *KeyEventArgs*являются свойствами типа *Keys*. *Kevs* — это большое перечисление, второе по величине из всех перечислений .NET Framework. В нем есть клавиши, точно отсутствующие на моей клавиатуре (скорее всего и на вашей). Ветераны программирования для Windows могут заметить, что значения этого перечисления соответствуют виртуальным кодам клавиш, определенным в заголовочных файлах Windows.

Давайте разберем перечисление *Keys,* разделив его на логические группы. Во-первых, *26* членов этого перечисления независимо от состояния модификаторов идентифицируют клавиши букв:

Перечисление *Keys* (буквы)

Член	Значение	Член	Значение
A	65	*N*	78
B	66	*O*	79
C	67	*P*	30
D	68	*Q*	81
E	69	*R*	82
F	70	*S*	83
G	71	*T*	84
H	72	*U*	85
I	73	*V*	86
J	74	*W*	87
K	75	*X*	88
L	76	*Y*	89
M	77	*Z*	90

Заметьте: значения перечисления для букв в верхнем регистре совпадают со значениями ASCII-кодов для этих символов (которые также совпадают с кодами Unicode). Коды символов, генерируемые этими клавишами, зависят и от состояния клавиш Ctrl, Shift и Caps Lock.

Чтобы не отходить слишком далеко от практики, рассмотрим программу, использующую одно из значений перечисления *Keys*.

ExitOnX.cs

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

class ExitOnX: Form
{
     public static void Main()
```

```
    {
        Application.Run(new ExitOnX());
    }
    public ExitOnX()
    {
        Text = "Exit on X";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        if (kaa.KeyCode == Keys.X)
            Close();
    }
}
```

Эта программа закрывает сама себя при нажатии клавиши X. При этом клавиша X может быть нажата одновременно с любыми комбинациями клавиш Shift, Ctrl и Alt. Из-за совпадения значений перечисления с некоторыми кодами Unicode оператор *if* можно изменить:

```
if (kea.KeyCode == (Keys)(int)'X')
```

Следующая группа значений *Keys* независимо от состояния модификаторо и соответствует цифровым клавишам, расположенным в виде горизонтального ряда над клавишами букв:

Перечисление Keys (цифровые клавиши)

Член	Значение
D0	48
D1	49
D2	50
D3	51
D4	52
D5	53
D6	54
D7	55
D8	56
D9	57

И вновь обратите внимание на связь между значениями *Keys* и ASCII-кодами для цифровых клавиш. Коды символов, генерируемые этими клавишами, также зависят от состояния модификаторов,

В общей сложности перечисление *Keys* включает значения для 24 функциональных клавиш:

Перечисление *Keys* (функциональные клавиши)

Член	Значение	Член	Значение
F1	112	*F13*	124
F2	113	*F14*	125
F3	114	*F15*	126
F4	115	*F16*	127
F5	116	*F17*	128
F6	117	*F18*	129
F7	118	*F19*	130
F8	119	*F20*	131
F9	120	*F21*	132
F10	121	*F22*	133
F11	122	*F23*	134
F12	123	*F24*	135

Знаю, что на моей клавиатуре (да и на вашей) не найдется 24 функциональных клавиш, и это меня вполне устраивает,

Цифровая клавиатура впервые появилась в оригинальной IBM PC. Следующие клавиши, расположенные на цифровой клавиатуре, независимо от состояния Num Lock генерируют одни и те же коды:

Перечисление *Keys* (знаки действий на цифровой клавиатуре)

Член	Значение	Описание
Multiply	106	* на цифровой клавиатуре
Add	107	+ на цифровой клавиатуре
Subtract	109	- на цифровой клавиатуре
Divide	111	/ на цифровой клавиатуре

Остальные клавиши цифровой клавиатуры генерируют различные коды в зависимости от состояния клавиши Num Lock. Вот коды, генерируемые при включенной клавише Num Lock (я построил эту таблицу в соответствии с расположением клавиш на цифровой клавиатуре):

Перечисление *Keys* (клавиши цифр на цифровой клавиатуре)

Член	Значение	Член	Значение	Член	Значение
NumPad7	103	*NumPad8*	104	*NumPad9*	105
NumPad4	100	*NumPad5*	101	*NumPad6*	106
NumPad1	97	*NumPad2*	98	*NumPad3*	99
NumPad0	96			*Decimal*	110

Эти клавиши генерируют коды 10 цифр и символа-разделителя десятичных разрядов. В раскладке клавиатуры одних стран разделителем является точка, в других — запятая. Независимо от раскладки этой клавише соответствует значение *KeysDecimal* в перечислении *KeyCode*. А следующее значение перечисления, похоже, не используется:

Перечисление Keys (неиспользуемое значение цифровой клавиатуры)

Член	Значение
Separator	108

А вот коды, генерируемые при выключенной клавише Num Lock:

Перечисление *Keys* (клавиши цифровой клавиатуры, управляющие курсором)

Член	Значение	Член	Значение	Член	Значение
Home	36	*up*	38	*PageUp* или *Prior*	33
left	37	*Clear*	12	*Right*	39
End	35	*Down*	40	*PageDown* или *Next*	34
Insert	45			*Delete*	46

Заметьте, что значения *KeysPrior* и *KeysNext* дублируют *KeysPageUp* и *KeysPageDown*. На многих клавиатурах 10 цифровых клавиш (кроме Clear) дублируют клавиши управления курсором, генерируя те же самые коды.

Следующие 6 клавиш я поместил в отдельную группу, так как они тоже генерируют коды символов. Значение перечисления *Keys* для них совпадают с названиями генерируемых ими символов:

Перечисление Keys (управляющие клавиши ASCII)

Член	Значение
Back	8
Tab	9
LineFeed	10
Enter *Return*	13
tiscape	27
Space	32

Back — это клавиша Backspace. В настоящее время в цифровой клавиатуре есть вторая клавиша Enter (или Return), независимо от состояния Num Lock генерирующая тот же код, что и обычная клавиша Enter (Return),

Следующая таблица содержит значения перечисления *Keys* для клавиш Shift, Ctrl и Alt (которая здесь называется клавишей *Menu,* поскольку обычно она активизирует меню). В нижнем ряду на современной клавиатуре располагается по паре клавиш Shift. Ctrl и Alt. Из таблицы следует, что правые и левые версии этих клавиш генерируют разные коды:

Перечисление *Keys* (клавиши-модификаторы)

Член	Значение	Член	Значение	Член	Значение
ShiftKey	16	*LShiftKey*	160	*RshiftKey*	161
ControlKey	17	*LControlKey*	162	*RcontrolKey*	163
Menu	18	*Lmenu*	164	*Rmenu*	165

Однако в реальных объектах *KeyEventArgs* я никогда не видел членов перечисления с именами, начинающимися с буквы *L* или *K*.

А вот клавиши, встречающиеся на клавиатуре Microsoft Natural Keyboard и ее клонах:

Перечисление *Keys* (клавиши Microsoft)

Член	Значение	Описание
Lwin	91	Левая клавиша с логотипом Windows
Rwin	92	Правая клавиша с логотипом Windows
Apps	93	Клавиша контекстного меню

В этом случае левая и правая клавиши действительно генерируют *разные* коды.

В следующей таблице описана подборка клавиш и их комбинаций, выполняющих различные функции:

Перечисление *Keys* (прочее)

Член	Значение	Описание
Cancel	3	Pause/Break, если нажата клавиша Ctrl.
Pause	19	Pause/Break, если клавиша Ctrl не нажата,
Capital CapsLock	20	Клавиша Caps Lock.
Snapshot PrintScreen	44	Клавиша Print Scrn.
NumLock	144	Клавиша Num Lock.
Scroll	145	Клавиша Scroll Lock.

Пять значений перечисления *Keys* на самом деле соответствуют кнопкам мыши:

Перечислений *Keys* (кнопки мыши)

Член	Значение
LButton	1
RButton	2
MButton	4
XButton1	5
XButton2	6

Но вы не увидите эти значения в событиях *KeyDown* и *KeyUp*. А вот группа особых клавиш:

Перечисление *Keys* (специальные клавиши)

Член	Значение
Select	41
Print	42
Execute	43
Help	47
ProcessKey	229
Attn	246
Crsel	247
Exsel	248
EraseEof	249
Play	250

Перечисление *Keys* (специальные клавиши) *(продолжение)*

Член	Значение
Zoom	251
NoName	252
Pa1	25 5
OemClear	254

Понятия не имею, что делать с этими клавишами, если мне придется сесть за клавиатуру, на которой все они будут.

Следующие 12 значений *Key* применяются лишь в Windows 2000 и выше. Эти клавиши также генерируют коды символов:

Перечисление *Keys* (символы)

Член	Значение
OemSemicolon	186
Oemplus	187
Oemcomma	188
OemMinus	189
OemPeriod	190
OemQuestion	191
Oemtilde	192
OemOpenBrackets	219
OemPipe	220
OemCloseBrackets	221
OemQuotes	*222*
Oem8	223
OemBackslash	*226*

Например, когда пользователь нажимает и отпускает клавишу, на которой нарисованы двоеточие и точка с запятой, генерируется код клавиши *OemSemicolon*.

Эти коды клавиш предназначены для специальных клавиатур (например, Microsoft Natural Keyboard Pro или Microsoft Internet Keyboard Pro), поддерживающих браузер и проигрыватель в Windows 2000 и выше:

Перечисление Keys (коды управления браузером и проигрывателем)

Член	Значение
BrowserBack	166
BrowserForward	167
BrowserRefresh	168
BrowserStop	169
BrowserSearch	170
BrowserFavorites	171
BrowserHome	172
VolumeMute	173
VolumeDown	174
VolumeUp	175

Перечисление *Keys* (коды управления браузером и проигрывателем) *(продолжение)*

Член	Значение
MediaNextTrack	176
MediaPreviousTrack	177
MediaStop	178
MediaPlayPause	179
LaunchMail	180
SelectMedia	181
LaunchApplication1	182
LaunchApplication2	183

Очевидно, что многие приложения просто игнорируют эти коды.

Следующие коды клавиш генерируются в связи *с* работой IME (Input Method Editor), используемого для ввода иероглифов китайского, японского и корейского языков:

Перечисление *Keys* (IME)

Член	Значение
HanguelMode *HangulMode*	21
KanaMode	
JunjaMode	23
FinalMode	*24*
KanjiMode *HanjaMode*	*25*
IMEConvert	28
IMENonconvert	29
IMEAceept	30
IMEModeChange	31

В общем случае приложениям «интересны» лишь коды символов Unicode, которые являются результатами использования IME.

Все перечисленные до сих пор члены *Keys* были кодами клавиш, т. е. каждый из них идентифицировал нажимаемую или отпускаемую клавишу и заносился *A* свойство *KeyCode* объекта *KeyEventArgs*, предоставляемого с событием *KeyDown* или *KeyUp*.

Перечисление *Keys* также включает следующие коды модификаторов:

Перечисление *Keys* (клавиши-модификаторы)

Член	Значение
None	0x00000000
Shift	0x00010000
Control	0x00020000
Alt	0x00040000

Заметьте: это битовые значения. Коды модификаторов показывают, нажаты ли во время события key-down или key-up клавиши Shift, Ctrl и Alt. Вспомните, что раньше

уже приводились коды для *ShiftKey*, *ControlKey* и *Menu*, отражающие состояние реальных клавиш Shift, Ctrl или Alt.

Каждая нажатая/отпущенная клавиша генерирует одно событие. Все три неизменяемых свойства объекта *KeyEventArgs* — *KeyCode*, *Modifiers* и *KeyData* — относятся к типу *Keys:*

- *KeyCode* — идентифицирует нажатую иди отпущенную клавишу, в числе которых могут быть Shift (*Keys.ShiftKey*) Ctrl (*Keys.ControlKey*) или Alt (*Keys.Menu*);
- *Modifiers* — указывает состояние клавиш Shift, Ctrl и Alt во время нажатия или отпускания данной клавиши; значением свойства *Modifiers* может быть любая комбинация *Keys.Shift*, *Keys.Control* или *Keys.Alt*, или, если не нажата ни одна из клавиш-модификаторов, у *Modifier* будет значение *Keys.None*, определяемое просто как 0;
- *KeyData* — это комбинация *KeyCode* и *Modifiers.*

Пусть, например, вы нажали клавишу Shift, потом D, после чего отпустили D, а затем — Shift. Ниже в таблице показаны четыре события и свойства объекта *KeyEventArgs*, связанного с этими манипуляциями:

Действия над клавишами и связанные с ними свойства *KeyEventArgs*

		Свойства		
Действие	Событие	***KeyCode***	*Modifiers*	*KeyData*
Нажата клавиша Shift	*KeyDown*	*Keys.ShiftKey*	*KeysSbift*	*KeysShift* \ *KeysShiftKey*
Нажата клавиша D	*KeyDown*	*Keys.D*	*KeysSbift*	*KeysSbift* \ *Keys.D*
Отпущена клавиша D	*KeyUp*	*Keys.D*	*Keys.Shift*	*KeysShift* *Keys.D*
Отпущена клавиша Shift	*KeyUp*	*KeysShiftKey*	*Keys.None*	*Keys.ShiftKey*

Перечисление *Keys* также определяет две маски для работы со свойством *KeyData*, позволяющие различать коды клавиш и коды модификаторов:

Перечисление Keys (битовые маски)

Член	**Значение**	**Комментарий**
KeyCode	*0x0000FFFF*	Битовая маска для кодов клавиш
Modifiers	*0xFFFF0000*	Битовая маска для клавиш-модификаторов

Имена этих членов перечисления совпадают с именами соответствующих свойств класса *KeyEventArgs*. Если объект *KeyEventArgs* назван *kea*, то выражение:

```
kea.KeyData & Keys.KeyCode
```

эквивалентно *kea.KeyCode*, а выражение:

```
key.KeyData & Keys.Modifiers
```

тождественно *kea.Modifiers*.

Проверка клавиш-модификаторов

Вовсе не обязательно обрабатывать событие *KeyDown* или *KeyUp*, чтобы определить, нажата ли одна из клавиш Shift, Ctrl или Alt. Текущее состояние трех клавиш-модификаторов можно получить при помощи статического свойства *ControlModifierKeys*:

Статические свойства объекта *Control* (выборочно)

Тип	Свойство	Доступ	Описание
Keys	*ModifierKeys*	Чтение	**Указывает** состояние клавиш Shift, Ctrl и Alt.

Допустим, программа должна выполнить другое действие, если нажата клавиша Shift, Ctrl или обе эти клавиши, но не Alt. Прежде всего надо вызвать статическое свойство *ModifierKeys*:

```
Keys keysMod = Control.ModifierKeys;
```

Затем — проверить возможные комбинации клавиш, представляющие интерес для программы:

```
if (keysMod == (Keys.Shift | Keys.Control))
{
    // Нажаты клавиши Shift и Ctrl.
}
else if (keysMod == Keys.Shift)
{
    // Нажата только клавиша Shift.
}
else if (keysMod == Keys.Control)
{
    // Нажата только клавиша Ctrl,
}
```

Не исключено, что свойство *ModifierKeys* придется использовать не во время обработки событий *KeyDown* или *KeyUp*, а, скажем, при обработке события мыши. Все мы видели программы, по-разному интерпретирующие щелчки и движения мыши в зависимости от нажатий Shift или Ctrl. В такой ситуации полезно свойство *ModifierKeys*. Примером может служить программа CanonicalSpline из главы 13.

К сожалению, проверить состояние клавиш Caps Lock, Num Lock и Scroll Lock невозможно.

Не теряйте связи с реальностью

Хотя программы Windows Forms, несомненно, получают много информации о нажатых клавишах, чаще всего они могут игнорировать большинство событий клавиатуры. Например, при обработке события *KeyDown* обычно можно не беспокоиться о событии *KeyUp*.

Программы Windows Forms часто игнорируют события с участием клавиш, генерирующих символы (ниже я расскажу об этом подробнее). Из этого можно сделать неправильный вывод, что всю информацию о клавиатуре можно извлекать из событий *KeyDown* и *KeyUp* и генерировать собственные коды символов.

Предположим, ваш обработчик события получает объект *KeyEventArgs*, у которого свойство *KeyCode* равно *Keys.D3*, а свойство *Modifiers* — *Keys.Shift*. Вы знаете, что это за символ, не так ли? Это знак фунта (#), кодируемый в ASCII и Unicode значением 0x0023. Допустим, это так, но в раскладке клавиатуры для Великобритании клавиша 3 в верхнем регистре генерирует другой знак фунта — £.

Более серьезная проблема возникает с клавишей Caps Lock. Как сказано выше, в Windows Forms нет средств для проверки состояния Caps Lock. Можно узнать, когда пользователь нажимает или отпускает Caps Lock, но что, если на момент начала работы вашей программы эта клавиша уже включена?

Событие *KeyDown* особенно полезно для обработки клавиш управления курсором, функциональных клавиш, а также Insert и Delete. Однако функциональные клавиши часто используются как клавиши быстрого доступа к меню. Поскольку нажатия таких клавиш транслируются в события меню автоматически, нет нужды обрабатывать их самостоятельно. Часто функциональные клавиши также дублируют функциональность элементов меню. Программы, в которых определены функциональные клавиши, не дублирующие элементы меню, трудно назвать дружественными к пользователю (особенно когда эти клавиши используются в сочетании с Shift и Ctrl, как в великом множестве старых программ для MS-DOS, таких как WordPerfect или Microsoft Word).

Подытожим сказанное: чаще всего придется обрабатывать события *KeyDown* для клавиш управления курсором, Insert и Delete. При использовании этих клавиш можно проверить состояние модификаторов свойством *Modifiers* объекта *KeyEventArgs*. Программы часто используют клавишу Shift в сочетании с клавишами управления курсором, например, в текстовых процессорах для расширения области выделения в документе, а Ctrl часто служит для изменения значений клавиш управления курсором, Например, в сочетании со «стрелкой-вправо» клавиша Ctrl перемещает каретку вправо не на один символ, а на целое *слово*.

Клавиатурный интерфейс для программы SysInfo

Полагаю, вы помните программы-примеры из главы 4, которые показывали сведения о системе. Последняя из них — SysInfoReflection — была значительно усовершенствована по сравнению с первым «сырым* вариантом, но одна небольшая проблема еще оставалась: у этой программы не было интерфейса, использующего клавиатуру.

Пришло время добавить к ней такой интерфейс, и это еще один пример того, как наследование полностью оправдывает себя. Класс *SysInfoKeyboard* является производным от *SysInfoReflection* и включает переопределенный метод *OnKeyDown*.

SysInfoKeyboard.cs

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

class SysInfoKeyboard: SysInfoReflection
{
     public new static void Main()
```

aw. *след. стр.*

```
    {
        Application.Run(new SysInfoKeyboard());
    }
    , public SysInfoKeyboardO
    {
        Text = "System Information: Keyboard";
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        Point pt = AutoScrollPosition;

        pt.X * -pt.X;
        pt.Y = -pt,¥;

        switch(kea.KeyCode)
        {
        case Keys.Right: -
            if ((kea.Modifiers 4 Keys.Control) == Keys.Control)
                pt.X += ClientSize.Width;
            else
                pt.X += Font.Height;
            break;

        case Keys.Left:
            if ((kea.Modifiers & Keys.Control) == Keys.Control)
                pt.X -= ClientSize.Width;
            else
                pt.X -= Font.Height;
            break;

        case Keys.Down:     pt.Y += Font.Height;        break;
        case Keys.Up:       pt.Y -= Font.Height;        break;
        ease Keys.PageDown:
            pt.Y += Font.Height * (ClientSize.Height / Font.Height);
            break;
        case Keys.PageUp:
            pt.Y -= Font.Height * (ClientSize.Height / Font.Height);
            break;
        ease Keys.Home:     pt  - Point.Empty;          break;
        case Keys.End:      pt.Y = 1000000;             break;
        }
        AutoScrollPosition = pt;
    }
}
```

Как сказано в главе *4*. изменяемое свойство *AutoScrollPosition* реализовано в объекте *ScrollableControl* (потомком которого является *Form*) как средство поддержки автоматической прокрутки. Значением *AutoScrollPosition* является структура *Point*, указывающая положение двух полос прокрутки.

При получении значения *AutoScrollPosition* значения координат отрицательны и указывают положение виртуальной клиентской области относительно верхне-

го левого угла физической клиентской области. Однако при установке *AutoScrollPosition* координаты должны быть положительными. Для этого нужно добавить к программе две строки:

```
pt.X = -pt.X;
pt.Y = -pt.Y;
```

А можно и иначе: просто настроить координаты на основе какой-либо клавиши управления курсором. При нажатии левой или правой стрелки одновременно с Ctrl я сдвигаю клиентскую область на ее ширину или, если клавиша Ctrl не нажата, — на величину высоты символа. Действие остальных клавиш управления курсором я сделал независимым от состояния модификаторов. Клавиша Home возвращает курсор в начало списка, End — перемещает его в конец списка, не меняя его положения по горизонтали.

Метод *KeyPress* для символов

Многие клавиши генерируют коды символов. Чтобы получить их, следует установить обработчик события *KeyPress* или (если это возможно) переопределить метод *OnKeyPress*.

События *Control* (выборочно)

Событие	Метод	Делегат	Аргумент
KeyPress	*OnKeyPress*	*KeyPressEventHandler*	*KeyPressEventArgs*

У класса *KeyPressEventArgs* всего два свойства:

Свойства *KeyPressEventArgs*

Тип	Свойство	Доступ	Комментарий
char	*KeyChar*	Чтение	Код символа Unicode.
bool	*Handled*	Чтение/запись	Устанавливается обработчиком события (изначально равно *false*).

Значение с типом данных *char* — это, конечно, 16-разрядный символ Unicode.

Вспомните таблицу событий, сопутствующих нажатиям клавиш D и Shift. В самой середине процесса — между парами событий *KeyDown* и *KeyUp* — происходит событие *KeyPress* со свойством *KeyChar,* равным 0x0044, соответствующим символу D верхнем регистре (если включена клавиша Caps Lock, вы получите код 0x0064 — это тот же символ, но в нижнем регистре: d).

Естественно, я исходил из того, что на вашем компьютере установлена раскладка клавиатуры для американского варианта английского языка. Если задана греческая раскладка клавиатуры, вы получите код 0x0394, соответствующий символу д, а если русская — 0x0412, или «В». Ниже я объясню, как установить иностранную раскладку клавиатуры,

Управляющие символы

Нажимая Ctrl одновременно с другими клавишами, можно генерировать управляющие символы, о которых сообщает событие *KeyPress*. При нажатии Ctrl в сочетании с буквами от A до Z независимо от состояния модификаторов получаются

коды символов от 0x0001 до 0x001A. Вот еще несколько управляющих символов, которые можно генерировать с клавиатуры.

Управляющие символы, генерируемые клавиатурой

Клавиша	Управляющий символ
Shift+Ctrl @	0x0000
Backspace	0x0008
Tab	0x0009
Ctrl+Enter	0x000A
Enter	0x000D
Esc	0x001B
Ctrl [	
Ctrl \	0x001C
Ctrl]	0x001D
Shift+Ctrl ^	0x001E
Shift+Ctrl _	0x001F
Ctrl+Backspace	0x007F

Программы часто используют комбинацию Shift+Tab для перехода на одну позицию табуляции назад (или к предыдущему элементу формы). Для этой комбинации нет специального кода — ее придется обрабатывать самостоятельно.

Иногда события *KeyDown* и *KeyUp* предоставляют коды перечисления *Keys*, соответствующие кодам клавиш, которые сообщает *KeyPress*. Какое из событий выбрать для обработки клавиш Enter, Backspace и Esc?

Я ломал голову над этим вопросом последние 15 лет, но сейчас я предпочитаю обрабатывать эти клавиши как управляющие символы Unicode, а не нажатия соответствующих клавиш. Однако я могу назвать лишь одну серьезную причину применения этого подхода: есть пользователи, издавна привыкшие нажимать Ctrl+H вместо Backspace и Ctrl+I — вместо Tab. Обрабатывая Backspace и Tab при помощи события *KeyPress*, можно обеспечить корректную работу этих комбинаций.

Просмотр клавиш

Возможно, вас заинтересует, откуда я получал информацию о событиях клавиатуры, которой делился с вами, поскольку ее источник явно не *в* документации пс Windows Forms. Большую часть того, что я знаю о клавиатуре, мне помогла узнать следующая программа. Она выводит сведения о клавишах при их нажатии и запоминает в массиве последние 25 событий клавиатуры (*KeyDown*, *KeyUp* и *KeyPress*).

KeyExamine.cs

```
//-----------------------------------------
// KeyExamine.cs © 2001 fay Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
```

см. след. стр.

```
public class KeyExamine: Form
{
     public static void Main()
     {
          Application.Run(new KeyExamine());
     }
          // Определение enum и struct для хранения событий клавиатуры

     enum EventType
     {
          None,
          KeyDown,
          KeyUp,
          KeyPress
     }
     struct KeyEvent
     {
          public EventType evttype;
          public EventArgs evtargs;
     }
          // Хранилище событий клавиатуры

     const int iNumLines    = 25;
     int       iNumValid    = 0;
     int       iInsertIndex = 0;
     KeyEvent[] akeyevt = new KeyEvent[iNumLines];

          // Позиционирование текста

     int xEvent, xChar, xCode, xMods, xData,
         xShift, xCtrl, xAlt, xRight;

     public KeyExamine()
     {
          Text = "Key Examine";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

          xEvent = 0;
          xChar  = xEvent + 5 - Font.Height;
          xCode  = xChar  + 5 * Font.Height;
          xMods  = xCode  + 8 * Font.Height;
          xData  = xMods  + 8 * Font.Height;
          xShift * xData  + 8 * Font.Height;
          xCtrl  = xShift + 5 * Font.Height;
          xAlt   = xCtrl  + 5 * Font.Height;
          xRight = xAlt   + & * Font.Height;

          ClientSize  = new Size(xRight, Font.Height * (iNumLines * 1));
          FormBorderStyle = FormBorderStyle.Fixed3D;
```

см. след. стр.

```
        MaximizeBox = false;
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        akeyevt[iInsertIndex].evttype = EventType.KeyDown;
        akeyevt[iInsertIndex].evtargs = kea;
        OnKey();
    }
    protected override void OnKeyUp(KeyEventArgs kea)
    {
        akeyevt[iInsertIndex].evttype = EventType.KeyUp;
        akeyevt[iInsertIndex].evtargs = kea;
        OnKey();
    }
    protected override void OnKeyPress(KeyPressEventArgs kpea)
    {
        akeyevt[iInsertIndex].evttype = EventType.KeyPress;
        akeyevt[iInsertIndex].evtargs = kpea;
        OnKey();
    }
    void OnKeyO
    {
        if(iNumValid < iNumLines)
        {
            Graphics grfx = CreateGraphics();
            DisplayKeyInfo(grfx, iInsertIndex, iInsertIndex);
            grfx.Dispose();
        }
        else
        {
            ScrollLines();
        }
        iInsertIndex = (iInsertIndex + 1) X iNumLines;
        iNumValid = Math.Min(iNumValid * 1, iNumLines);
    }
    protected virtual void ScrollLinesO
    {
        Rectangle rect = new Rectangle(0, Font.Height,
                                       ClientSize.Width,
                                       ClientSize.Height - Font.Height);

        // А здесь хотелось бы иметь возможность прокрутки!

        Invalidate(rect);
    }
    protected override void OnPaint(PaintEventArgs pea)
    <
        Graphics grfx = pea.Graphics;

        BoldUnderline(grfx, "Event",     xEvent, 0);
```

см. след. стр.

```
        BoldUnderline(grfx, "KeyChar",   xChar,   0);
        BoldUnderline(grfx, "KeyCode",   xCode,   0);
        BoldUnderline(grfx, "Modifiers", xMods,   0);
        BoldUnderline(grfx, "KeyData",   xData,   0);
        BoldUnderline(grfx, "Shift",     xShift,  0);
        BoldUnderline(grfx, "Control",   xCtrl,   8);
        BoldUnderline(grfx, "Alt",       xAlt,    0);

        if(iNumValid < iNumLines)
        {
            for (int i = 0; i < iNumValid; i++)
                DisplayKeyInfo(grfx, 1, 1);
        }
        else
        {
            for (int 1 = 0; 1 < iNumLines; i++)
                DisplayKeyInfo(grfx, i, (iInsertIndex + i) X iNumLines);
        }
    }
    void BoldUnderline(Graphics grfx, string str, int x, int y)
    {
            // Нарисовать текст полужирным шрифтом.

        Brush brush - new SolidBrush(ForeColor);
        grfx.DrawString(str, Font, brush, x, y);
        grfx.DrawString(str, Font, brush, x + 1, y>;

            // Подчеркнуть нарисованный текст.

        SizeF sizef = grfx.MeasureString(str, Font);
        grfx.DrawLine(new Pen(ForeColor), x, y + sizef.Height,
                                    x + sizef.Width, y + sizef.Height);

    }
    void DisplayKeyInfo(Graphics grfx, int y, int i)
    {
        Brush br = new SolidBrush(ForeColor);
        y = (1 + y) * Font.Height;    // Преобразовать y в пикселы.

        grfx.DrawString(akeyevt[i].evttype.ToString(),
                        Font, brush, xEvent, y);

        if(akeyevt[i].evttype == EventType.KeyPress)
        {
            KeyPressEventArgs kpea =
                            (KeyPressEventArgs) akeyevt[i].evtargs;

            string str = String.Format("\x202D{0} (0x{1:X4})",
                                        kpea.KeyChar, (int) kpea.KeyChar);
            grfx.DrawString(str, Font, br, xChar, y);
```

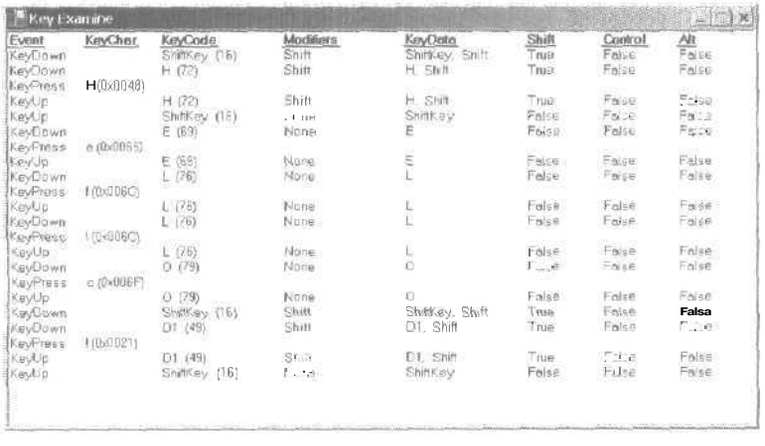
см. след, стр.

```
            }
            else
            {
                KeyEventArgs kea = (KeyEventArgs) akeyevt[i].evtargs;

                string str * String.Format("{0} ({1})",
                                           kea.KeyCode, (int) kea.KeyCode);
                grfx.DrawString(str, Font, br, xCode, y);
                grfx.DrawString(kea.Modifiers.ToString(), Font, br, xMods, y);
                grfx.DrawString(kea.KeyData.ToString(), Font, br, xData, y);
                grfx.DrawString(kea.Shift.ToString(), Font, br, xShift, y);
                grfx.DrawString(kea.Control.ToString(), Font, br, xCtrl, y);
                grfx.DrawString(kea.Alt.ToString(), Font, br, xAlt, y);
            }
        }
    }
```

Довольно большая программа для этой книги. В начале листинга класса находятся определения закрытого перечисления *EventType* и *структуры KeyEvent*, используемых для хранения данных *KeyEventArgs и KeyPressEventArgs*, связанных :: каждой нажатой клавишей. Программа создает массив из 25 подобных структур, Целочисленные поля с префиксом *x* используются для размещения информации по столбцам.

По мере того как происходят события *KeyDown*, *KeyUp* и Keypress, данные каждого из них записываются в массив и выводятся в клиентской области программы при помощи *DisplayKeyInfo*— самого длинного метода класса *KeyExamine*. Метод *OnPaint* также использует *DisplayKeyInfo* и выводит заголовки столбцов полужирным подчеркнутым шрифтом. В главе 9 я покажу другой, гораздо лучший способ получения полужирного подчеркнутого шрифта, но эта программа просто дважды прорисовывает текст, второй раз — смещая его на один пиксел относительно текста, нарисованного в первый раз. Затем при помощи метода *DrawLine* программа рисует линию ниже текста. Вот как выглядит результат работы программы, если набрать на клавиатуре «Hello!»:



Проблема здесь в том, что по достижении нижнего края клиентской области программа должна прокрутить ее содержимое до конца вверх. Если бы я писал Win32-код, то вызвал бы для этого *ScrollWindow*. Однако в Windows Forms нет ничего подобного этой функции. Поэтому вместо прокрутки программа просто делает часть клиентской области ниже заголовков столбцов недействительной, заставляя метод *OnPaint* перерисовывать все строки. Мне стыдно применять этот далеко не лучший прием, но не так, как должно быть субъекту из Microsoft, забывшему реализовать *ScrollWindow* в Windows Forms!

Вызов API Win32

Но что делать, если позарез нужна функция API Win32, которой в .NET Framework просто нет?

В таком случае можно прибегнуть к службам *PInvoke* (Platform Invocation Services). PInvoke — это общий механизм вызова функций, экспортируемых из DLL. Функция *ScrollWindow* находится в динамически подключаемой библиотеке User32.dll, поэтому ее можно точно квалифицировать. Но у этого подхода есть минус: после того как программист воспользовался им, его код перестает быть управляемым и уж подавно — независимым от платформы.

В документации по API Win32 приводится такой синтаксис *ScrollWindow*:

```
BOOL ScrollWindow(HWND hWnd, int XAmount, int YAmount,
                  CONST RECT *lpRect, CONST RECT *lpClipRect);
```

В заголовочных файлах C для Windows *BOOL* определяется просто как *int*, а *HWND* (описатель окна) — как указатель на *void*, но реально это просто 32-разрядное значение.

Но откуда взять описатель окна в Windows Forms? У класса *Control* есть свойство *Handle*, документированное как описатель *HWND* для экземпляра этого класса. Тип свойства *Handle* — определяемая в пространстве имен *System* структура *IntPtr* - свидетельствует, что это указатель. Типы данных *int* и *IntPtr* могут быть легко преобразованы друг в друга. Пока мы не встретили никаких трудностей при переходе от типов данных C# к аргументам функции *ScrollWindow* и возвращаемым ей значениям,

Трудности связаны с двумя последними аргументами *ScrollWindow* — указателями на структуру Windows *RECT*, определяемую в заголовочном файле Windows так:

```
typedef struct tagRECT
{
     LONG left;
     LONG top;
     LONG right;
     LONG bottom;
} RECT;
```

Тип данных *LONG* определяется в заголовочном файле Windows как *long*, но это не 64-разрядный *long* из C#, а всего лишь 32-разрядный *long* из C, поэтому он совместим с типом данных *int* из C#.

Чтобы вызвать *ScrollWindow* из программы C#, надо определить структуру *struct,* набор и порядок полей которой соответствуют таковым структуры Windows *RECT,* и предварить ее атрибутом;

```
[StructLayout(LayoutKind.Sequential)]
```

StructLayout — это атрибут C#, основанный на классе *StructLayoutAttribute,* определенном в пространстве имен *System.Runtime.InteropServices.* Кроме того, нужно объявить *ScrollWindow* как внешнюю функцию *extern,* предварив ее атрибутом:

```
[DllImport("user32.dll")]
```

Возможно, вы заметили, что класс *KeyExamine* содержит метод *ScrollLines,* отвечающий за прокрутку содержимого клиентской области. Метод *ScrollLines* в *KeyExamine* просто делает недействительной часть клиентской области ниже заголовков столбцов. В классе *KeyExamineWithScroll,* порожденном от *KeyExamine,* определена структура *KECT.* объявлена функция *ScrollWindow* и переопределен метод *ScrollLines* в *KeyExamine.* Модифицированная версия *ScrollLines* вызывает функцию Windows *ScrollWmdow.*

KeyExamineWithScroll.cs

```
//-----------------------------------------------
// KeyExamineWithScroll.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
«sing System.Runtime.InteropServices;
using System.Windows.Forms;

class KeyExamineWithScroll: KeyExamine
{
    public new static void Main()
    {
        Application.Run(new KeyExamineWithScroll());
    }
    public KeyExamineWithScroll()
    {
        Text += " With Scroll";
    }
        // Определить структуру RECT в стиле Win32.

    [StructLayout(LayoutKind.Sequential)]
    public struct RECT
    {
        public int left;
        public int top;
        public int right;
        public int bottom;
    I
        // Объявить функцию ScrollWindow.
```

см. след. стр.

```
    [DllImport("user32.dll")]
    public static extern int ScrollWindow(IntPtr hwnd, int cx, int cy,
                                          ref RECT rectScroll,
                                          ref RECT rectClip);

        // Переопределить методы в KeyExamine.

    protected override void ScrollLines()
    {
        RECT rect;

        rect.left   =0;
        rect.top    = Font.Height;
        rect.right  =ClientSize.Width;
        rect.bottom = ClientSize.Height;

        ScrollWindow(Handle, 0, -Font.Height, ref rect, ref rect);
    }
}
```

Эта версия намного лучше: программа работает более гладко и эффективно, когда ей не приходится перерисовывать все выводимые строки,

Обработка информации при вводе с иноязычных клавиатур

Всегда полезно протестировать свои программы на разных типах клавиатур, отличных от установленной на вашей машине, в частности, на *иноязычных* клавиатурах[1]. При этом запустить программу с русской раскладкой клавиатуры намного легче, чем съездить в Москву. Для такого тестирования можно установить иноязычные *раскладки клавиатуры* — небольшие файлы, позволяющие с имеющейся клавиатуры генерировать коды символов других языков.

Ниже вы найдете инструкции по установке иноязычных раскладок клавиатуры в английской версии Windows 2000,

Запустите Regional Options (Язык и стандарты) из Панели управления и перейдите на вкладку General (Общие). В списке Language Settings For The System (Языковые параметры системы) скорее всего будет отмечена лишь Western Europe and United States (Западная Европа и США) как раскладка по умолчанию. Установите флажки для Arabic, Cyrillic, Greek и Hebrew (Арабский, Кириллица, Греческий. Иврит) и щелкните OK. Система попросит вставить диск Windows 2000 CD. после чего последует ее перезагрузка.

После перезагрузки вызовите Панель управления и откройте Keyboard (Клавиатура). Перейдите на вкладку Input Locales (Языки и раскладки). В разделе Installed Input Locales (Установленные языки ввода и раскладки клавиатуры) щелкните

[1] Диаграммы многих иноязычных клавиатур приводятся в книге Nadine Капо «Developing International Software for Windows 95 and Windows NT». Эта книга должна вскоре выйти в издательстве Microsoft Press, но ее электронная версия уже доступна на Web-узле библиотеки MSDN [http://msdn.microsoft.com/library, раздел Development (General) and Internationalization],

кнопку Add (Добавить) и по очереди добавьте следующие языки ввода, если они еще не установлены: Arabic (Egypt), English (United Kingdom), English (United States). German (Germany), Greek, Hebrew и Russian. В результате будут также установлены раскладки клавиатуры, ассоциированные с этими региональными параметрами.

Если раньше вы не устанавливали дополнительные раскладки клавиатуры и оставили в диалоговом окне Keyboard Properties (Свойства: Клавиатура) флажок Enable Indicator On Taskbar (Отображать индикатор на панели задач), то в системном меню вы увидите новый значок в виде квадратика с буквами *EN* (что означает *English*). Щелкнув его, можно переключиться на альтернативную английскую, арабскую, немецкую, греческую, русскую клавиатуру или клавиатуру для иврита. Эти изменения повлияют лишь на текущее активное приложение.

А теперь поэкспериментируем с программами KeyExamine и KeyExamineWithScroll. Запустите одну из них и переключитесь на раскладку English (United States), если она еще не включена. Нажмите Shift+3- Вы получите код *KeyChar* 0x0023 и символ #. А теперь переключитесь на раскладку English (United Kingdom) и нажмите ту же комбинацию клавиш. Теперь вы получите код OxOOA3 и символ £.

Переключитесь на раскладку German (Germany), нажмите Y и Z. Заметьте, что как коды *KeyCode,* так и *KeyCbar* показывают символ Z, когда вы нажимаете Y, и наоборот. Это так, поскольку на немецкой клавиатуре эти символы поменялись местами.

Не переключая раскладку клавиатуры, нажмите клавишу +/= (при этом значение *KeyCode* будет равно 221, что соответствует *Keys.OemCloseBrackets*), затем клавишу A. В результате появляется символ *а* с акутом (острым ударением): á, соответствующий символу Unicode OxOOE1. В немецкой клавиатуре клавиша +/= называется *мертвой*. Если после мертвой клавиши нажать подходящий символ, получится символ со знаком ударения. С мертвой клавишей можно использовать любые гласные в верхнем или нижнем регистре: *а. е, г, о, и* или *у* (в немецкой раскладке последний символ водится клавишей Z). Если нажать +/= в верхнем регистре, затем гласную в любом регистре (в этом случае *а, е, г,* о, и, но не *у*), получится буква со знаком грависа, например: à.

Если после мертвой клавиши нажать согласную букву, напечатается отдельно знак ударения (' или ˋ), за которым следует сама буква. Чтобы напечатать только знак ударения, после мертвой клавиши нажмите пробел.

Аналогично, если на немецкой клавиатуре после клавиши ~/´ нажать *а, е. i,* о или *и,* получится буква со знаком циркумфлекса: â (клавиша ~/´, нажатая вместе с Shift, не является мертвой — она генерирует символ °). Умляуты существуют и немецком языке только в нижнем регистре: ä, ö и ü. Эти символы генерируются клавишами "/', :/; и {/[соответственно.

До сих пор мы работали со значениями *KeyCbar,* сопутствовавшими событиям *KeyPress,* из 8-разрядного диапазона. Это символы, определяемые одним или двумя стандартами, первый из которых — это ANSI X3.4-1986 или 7-разрядный ASCII

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
2-		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	л	_
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7-	p	q	r	s	t	и	v	w	x	y	z	{	l	}	-	

Второй — это расширение ASCII, документированное в ANSI/ISO 8859-1-1987, обычно его называют просто «Latin 1»:

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
A-		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B-	°	±	²	³	´	µ	¶	·	¸	1	º	»	¼	½	¾	¿
C-	À	Á	Â	Ã	Ä	Л	Æ	g	Ë	Ë	Ë	Ë	Ì	Í	Î	Ï
D-	Ð	Ñ	Ò	6	6	6	Ö	×	O	Ù	Ú	Û	Ü	Ý	Þ	ß
E-	à	á	â	ã	ä	å	æ	ę	ë	ë	ë	ë	]	í	T	T
F-	б	ñ	ò	6	6	6	6	÷	o	й	ú	û	й	ý	þ	ÿ

Эти наборы символов годятся лишь для языков, использующих только латинский алфавит. Для поддержки алфавитов других языков (включая китайские, японские и корейские иероглифы) была разработана 16-разрядная кодировка — Unicode. В общем случае программы Windows Forms, написанные на C#, не требуют дополнительных мер для поддержки Unicode. Например, длина типа данных *char* в C# равна 16 разрядам.

Если переключиться в русскую раскладку и нажать несколько клавиш, будут набраны кириллические буквы, коды символов для которых находятся в диапазоне 0x0410—0x044F, определенном в стандарте Unicode как Основной русский алфавит. Аналогично можно переключиться на арабскую, греческую, еврейскую клавиатуру и набирать символы алфавитов этих языков,

Если раньше вы не сталкивались с этой проблемой, у вас может возникнуть вопрос: а как иноязычные алфавиты и клавиатуры работали до Unicode, когда коды символов были только 8-разрядными? Если коротко: с этим была полная неразбериха.

Чтобы информировать программу об изменении раскладки клавиатуры пользователем, можно установить обработчики событий *InputLanguageChanging* и *InputLanguageChanged* или переопределять методы *OnInputLanguageChanging* и *OnInput-*

LanguageChanged, показанные в таблице (имена событий в методах, делегатах и аргументов событий заменены многоточием):

События *Form* (выборочно)

Событие	Метод	Делегат	Аргумент
InputLanguageChanging	*On...*	*...EventHandler*	*...EventArgs*
InputLanguageChanged	*On...*	*...EventHandler*	*...EventArgs*

Первым происходит событие *InputLanguageChanging*, при этом объект *InputLanguageChangingEventArgs* содержит сведения о языке, на который хочет переключиться пользователь. Если программа не допускает такое переключение, установите свойство *Cancel* объекта *InputLanguageChangingEventArgs* как *true*, иначе вскоре будет получено событие *InputLanguageChanged*.

Чтобы лучше разобраться в том, как это работает, взгляните на класс *InputLanguage* в пространстве имен *System.Windows.Forms* и класс *CultureInfo* в пространстве имен *System.Globalization*.

Фокус ввода

Фокус ввода определяет, какой из элементов управления получает данные, вводимые с клавиатуры. Некоторые клавиши (например, Tab и клавиши управления курсором) позволяют передавать фокус от элемента к элементу диалогового окна.

Класс *Form* наследует три неизменяемых свойства, имеющих отношение к фокусу ввода:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
bool	*CanFocus*	Чтение.
bool	*ContainsFocus*	Чтение,
bool	*Focused*	Чтение,

Элемент управления (или форма) не может получить фокус, если он отключен или невидим. Это состояние позволяют проверить свойства *CanFocus*. *ContainsFocus* возвращает *true*, если фокус находится у элемента управления (формы) или одного из его дочерних объектов, а *Focused* возвращает *true*, если в фокусе находится сам элемент управления (форма).

Программа может передать фокус ввода одному из своих элементов управления при помощи метода *Focus*.

Методы *Control* (выборочно)

```
bool Focus()
```

Возвращаемое методом значение показывает, успешно ли был передан фокус, Метод закончится неудачей, если целевой элемент управления не является дочерним объектом активной формы,

Наконец, два события сообщают элементу управления (форме) о получении и потере им фокуса:

События *Control* (выборочно)

Событие	Метод	Делегат	Аргумент
GotFocus	*OnGotFocus*	*EventHandler*	*EventArgs*
LostFocus	*OnLostFocus*	*EventHandler*	*EventArgs*

Если элемент управления (форма) получил событие *GotFocus,* в конечном счете он обязательно получит и событие *LostFocus.* В главе 12, когда мы приступим к созданию элементов управления, я расскажу о фокусе ввода подробнее.

Пропавшая каретка

Элементы управления или формы, получающие вводимую с клавиатуры информацию, обычно как-то показывают, что они находятся в фокусе. Например, текст кнопки, находящейся в фокусе, обводится точечной линией. Элементы управления и формы, позволяющие вводить текст, обычно отмечают место появления следующего символа на экране короткой горизонтальной (вертикальной) линией или квадратиком. Этот индикатор, может быть, знаком вам под названием *курсор,* но в Windows правильнее называть его *кареткой.* Слово *курсор* зарезервировано для растровой картинки, отражающей положение указателя мыши на экране.

Если вы создадите элемент управления *TextBox* или *RichTextBox* (как это делается, я покажу в главе 18), то за формирование и вывод каретки будет отвечать он. Во многих случаях функциональности этих элементов управления вполне хватает программе. В частности, *RicbTextBox* построен на основе того же элемента управления Windows, что и Microsoft WordPad, и обладает достаточно широкими возможностями.

Если же эти элементы управления не соответствуют вашим задачам и нужно написать собственный код для ввода текста, возникает небольшая проблема. Среди функций, которых нет в библиотеках классов Windows Forms, пожалуй, самым загадочным образом пропала каретка.

Боюсь, чтобы решить нашу проблему, снова придется писать неуправляемый код, копающийся в DLL Windows, Если вы захотите использовать мой класс *Caret* в своих программах, учтите, что он определяется в моем собственном пространстве имен. Он основан на API каретки Windows и начинается объявлением пяти внешних функций, расположенных в библиотеке User32.dll.

Caret.cs

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

namespace Petzold.ProgrammingWindowsWithCSharp
{
     class Caret
```

см. след. стр.

```
{
    [DLLImport("user32.dll")]
    public static extern int CreateCaret(IntPtr hwnd, IntPtr hbm,
                                         int cx, int cy);
    [DLLImport("user32.dll")]
    public static extern int DestroyCaret();

    [DLLImport("user32.dll")]
    public static extern int SetCaretPos(int x, int y);

    [DLLImport("user32.dll")]
    public static extern int ShowCaret(IntPtr hwnd);

    [DLLImport("user32.dll")]
    public static extern int HideCaret(IntPtr hwnd);
                                                    // Поля
    Control ctrl;
    Size    size;
    Point   ptPos;
    bool    bVisible;
                                                    // Конструкторы
        // Конструктор по умолчанию недоступен.
    private Caret()
    {
    }
        // Только у допустимого конструктора есть аргумент Control.

    public Caret(Control ctrl)
    {
        this.ctrl = ctrl;
        Position  = Point.Empty;
        Size    . * new Size(1, ctrl.Font.Height);

        Control.GotFocus  += new EventHandler(ControlOnGotFocus);
        Control.LostFocus += new EventHandler(ControlOnLostFocus);

            // Создать каретку, если элемент управления уже в фокусе. ,

        if (ctrl.Focused)
            ControlOnGotFocus(ctrl, new EventArgs());
    }
                                                - -  '  // Свойства
    public Control Control
    {
        get
        {                                  ..
          - ; return ctrl;    :
        }
     \
    public Size Size
```

см. след. стр.

```
{
    . get
:   {  '
        return  size;
    } ,
    set
     {
        size * value;
    }
}
public Point Position
 {
    get
    {
        return ptPos;
    }
    set ;
     {
        ptPos = value;
        SetCaretPos(ptPos.X, ptPos.Y);
    }
}
public bool Visibility
i
   . get
    {
        return bVisible;
    }
    set
    {
        if (bVisible = value)
            ShowCaret(Control.Handle);
        else
            HideCaret(Control.Handle);
    }
}
                                                •        // Методы
public void Show()
i
    Visibility * true;
}
public void Hide()
{
    Visibility  *  false;
}
public void Dispose()
{
        // Уничтожить каретку при потере фокуса.

    if (ctrl.Focused)
```

см. след. стр.

```
                ControlOnLostFocus(ctrl, new EventArgs());

            .; Control.GotFocus  -= new EventHandler(ControlOnGotFocus);
               Control.LostFocus -= new EventHandler(ControlOnLostFocus);
            }
                                                        // Обработчики  событий
            void ControlOnGotFocus(object obj, fiverrtArgs ea)
            {
                CreateCaret(Control.Handle, IntPtr.Zero,
                            Size.Width, Size.Height);
                SetCaretPos(Position.X, Position.Y);
                Show();
            }
            void ControlOnLostFocus(object obj, EventArgs sa)
            {
                Hide();
                DestroyCaret();
            }
        }
    }
```

Чтобы создать каретку в форме (или любом другом объекте, производном от *Control),* используйте конструктор:

```
Caret caret = new Caret(form);
```

Класс *Caret* определяет конструктор по умолчанию как закрытый, поэтому надо включить аргументы в конструктор. У *Caret* четыре свойства:

Свойства *Caret*

Тип	Свойство	Доступ	Описание
Control	*Control*	Чтение	Объект *Control*, с которым связана каретка.
Size	*Size*	Чтение/запись	Размер каретки в пикселах.
Point	*Position*	Чтение/запись	Положение каретки относительно начала координат формы.
bool	*Visibility*	Чтение/запись	Видимость каретки,

В средах, функционирующих в текстовом режиме, каретка часто принимает вид символа подчеркивания или квадратика. Однако такие каретки не очень хорошо работают с текстом переменной ширины — здесь лучше использовать вертикальную линию. В общем случае программа, использующая класс *Caret* со шрифтом формы по умолчанию, может установить размер каретки так:

```
caret.Size = new Size(2, Font.Height);
```

Свойство *Position* указывает позицию каретки относительно левого верхнего угла клиентской области.

Свойство *Visibility* позволяет прятать и снова показывать каретку При рисовании на форме каретку нужно каждый раз скрывать. Это надо делать до или после, но не во время события *Paint*! В качестве альтернативы свойству *Visibility* годятся

методы *Hide* и *Show*. *Dispose* — единственный открытый метод, поддерживаемый классом *Caret:*

Методы Caret

Метод	Описание
void Hide()	Скрывает каретку.
void Show()	Показывает каретку
void Dispose()	Деактивизирует каретку.

Вызывать метод *Dispose* обычно не требуется. Он нужен лишь для того, чтобы прекратить ввод текста с клавиатуры в форму, осуществляемый с использованием каретки,

Caret — хороший пример класса, который должен подключить обработчики событий к форме, с которой он связан. *Caret* устанавливает обработчики событий *GotFocus* и *LostFocus*. Он создает каретку, когда форма получает фокус, и уничтожает ее, когда форма теряет фокус, согласно рекомендациям по обработке каретки в программировании Win32. Метод *Dispose* просто отключает обработчики событий, после чего каретка больше не создается,

Но имейте в виду, что форма, использующая класс *Caret* и самостоятельно переопределяющая его методы *OnGotFocus* и *OnLostFocus*, рискует отключить обработчики событий в *Caret*! Если пришлось переопределить эти методы, обязательно вызывайте методы из базового класса:

```
protected override void OnGotFocus(EventArgs ea)
{
     base.OnGotFocus(ea);
     :
}
protected override void OnLostFocus(EventArgs ea)
{
     base.OnLostFocus(ea);
     :
}
```

Методы *OnGotFocus* и *OnLostFocus* из базового класса вызывают установленные обработчики событий так же, как методы *Caret.*

Отображение набираемых символов на экране

А теперь рассмотрим программу, которая позволяет вводить и редактировать текст при помощи класса *Caret.* По функциональности она очень близка к элементу управления *TextBox,* работающему в режиме обработки одной строки.

TypeAway.cs

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

см. след. стр.

```
using System.Drawing.Text;
using System.Windows.Forms;
using Petzold.ProgrammingWindowsWithCSharp;

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

     protected Caret  caret;
     protected string strText = "";
     protected int    ilnsert * 0;

     public TypeAway0
     {
          Text = "Type Away";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

          FontHeight = 24;

        ' caret = newCaret(this);
          caret.Size = new Size(2, Font.Height);
          caret.Position = new Point(0, 0);
     }
     protected override void OnKeyPress(KeyPressEventArgs kpea)
     {
          caret.Hide();
          Graphics grfx = CreateGraphics();
          grfx.FillRectangle(new SolidBrush(BackColor),
                             new RectangleF(Point.Empty,
                             grfx.MeasureString(strText, font,
                             Point.Empty, StringFormat.GenericTypographic)));
          switch(kpea.KeyChar)
          {
          case '\b':
               if (ilnsert > 0)
               {
                    strText = strText.Substring(0, Ilnsert - 1) +
                                                   strText.Substring(iInsert);
                    ilnsert -= 1; =
               }
               break;

          case '\r':
          case '\n':
               break; "
```

см. след. стр.

```
        default:
            if (iInsert == strText.Length)
                strText += kpea.KeyChar;
            else
                strText = strText.Substring(0, iInsert) +
                          kpea.KeyChar +
                          strText.Substring(iInsert);
            iInsert++;
            break;
        }
        grfx.TextRenderingHint = TextRenderingHint.AntiAlias;
        grfx.DrawString(strText, Font, new SolidBrush(ForeColor),
                        0, 0, StringFormat.GenericTypographic);
        grfx.Dispose();

        PositionCaret();
        caret.Show();
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        switch (kea.KeyData)
        {
        case Keys.Left:
            if (iInsert > 0)
                iInsert--;
            break;

        case Keys.Right:
            if (iInsert < strText.Length)
                iInsert++;
            break;

        case Keys.Home:
            iInsert = 0;
            break;

        case Keys.End:
            iInsert = strText.Length;
            break;

        case Keys.Delete:
            if (iInsert < strText.Length)
            {
                iInsert++;
                OnKeyPress(new KeyPressEventArgs('\b'));
            }
            break;

        default:
            return;
```

см. след. стр.

```
            }
            PositionCaret();
        }
        protected void PositionCaret()
        {
            Graphics grfx = CreateGraphics();
            string str = strText.Substring(0, iInsert);
            StringFormat strfmt = StringFormat.GenericTypographic;
            strfmt.FormatFlags |= StringFormatFlags.MeasureTrailingSpaces;
            SizeF sizef = grfx.MeasureString(str, Font, Point.Empty, strfmt);
            caret.Position = new Point((int)sizef.Width, 0);
            grfx.Dispose();
        }
        protected override void OnPaint(PaintEventArgs pea)
        {
            Graphics grfx = pea.Graphics;
            grfx.TextRenderingHint - TextRenderingHint.AntiAlias:
            grfx.DrawString(strText, Font, new SolidBrush(ForeColor),
                            0, 0, StringFormat.GenericTypographic);
        }
}
```

Класс *TypeAway* создает объект типа *Caret* в своем конструкторе и инициализирует его размер и положение. При рисовании на форме программа должна лишь спрятать каретку до события *Paint* и вновь показать ее после этого события, а также установить позицию каретки в клиентской области.

Вводимая и редактируемая пользователем строка символов хранится в поле *strText*. Поле *iInsert* хранит точку вставки Б эту строку. Например, после набора трех символов значение *iInsert* равно 3. Если затем нажать «стрелку-влево», это значение станет равным 2. Метод *PositionCaret* программы *TypeAway* отвечает за преобразование позиции символа в позицию пикселов. Результат этого преобразования служит для установки свойства *Position* объекта *Caret*.

Посмотрим, как *TypeAway* обрабатывает метод *OnKeyPress*. Кажется, в большинстве случаев программе достаточно просто вывести на форму новый символ, позиция пикселов которого соответствует текущей точке вставки, и добавить этот символ к значению поля *strText*. Но вместо этого программа стирает текст формы при помощи методов *MeasureString* и *FillRectangle*! Возможно, это выглядит как крайность, но это необходимо, если точка вставки находится не в конце строки или при выводе текста на языке, отличном от английского (в этом мы еще убедимся).

Метод *OnKeyPress* обрабатывает нажатие клавиши Backspace, удаляя из строки символ, расположенный перед точкой вставки. Этот метод игнорирует управляющие символы возврата каретки и перевода строки, и помещает все остальные символы в строку в точке вставки. После он выводит всю строку целиком и вызывает метод *PositionCaret* (я опишу его чуть ниже). Заметьте, что метод прячет каретку во время рисования на форме.

Метод *OnKeyDown* обрабатывает лишь клавиши управления курсором, перемещая точку вставки, и клавишу Delete, имитируя действие Backspace. Он тоже вызывает *PositionCaret*.

Метод *PositionCaret* отвечает за преобразование позиции точки вставки (*insert*) в координаты пикселов каретки. Он делает это при помощи *MeasureString*. К сожалению, версия ***MeasureString*** по умолчанию не обеспечивает точности, нужной для приложений такого рода, но хуже всего то, что при расчете длины строки данная версия *MeasureString* обычно не учитывает концевые пробелы. Чтобы выйти из ситуации, программа использует версию ***MeasureString*** *с* аргументом *StringFormat* и заносит в свойство *FormatFlag* аргумента ***StringFormat*** значение перечисления *StringFormatFlags.MeasureTrailingSpaces*. Без этих изменений каретка двигалась бы при наборе букв слова, но оставалась бы на месте при вводе пробела, разделяющего слова.

Но одних этих мер недостаточно, чтобы точно выровнять каретку и выводимый текст. В силу причин, обсуждаемых в главе 9, в методах ***MeasureString*** и *DrawString* имеются встроенные средства компенсации искажений, связанных с независимой от устройства растеризацией контурных шрифтов. Чтобы избежать применения этих средств, что происходит по умолчанию, программа использует объект *StringFormat*, получаемый из статического свойства *StringFormat.GenericTypographic*. Кроме того, в программе применяется сглаживание выводимого текста при помощи свойства *TextRenderingHint* объекта *Graphics* (полностью это решение освещается в главе 9).

Проблема с текстом, выводимым справа налево

Выше я сказал, что по функциональности программа TypeAway почти полностью соответствует элементу управления *TextBox*, работающему в режиме обработки одной строки, кроме двух моментов. Первая проблема в том, что TypeAway не поддерживает буфер обмена, вторая — TypeAway некорректно показывает каретку при наборе текста, который пишется справа налево, например, на арабском и иврите.

Рассмотрим эти проблемы. Запустите TypeAway и переключитесь на раскладку для иврита. Мы наберем фразу «Доброе утро» на иврите: בֹּ Т ט 1 р ב, латинская транслитерация которой выглядит как «*boker tov*». Чтобы сделать этот маленький трюк на английской клавиатуре, надо знать, какие символы иврита соответствуют ее клавишам.

Алфавит **иврита**

Код Unicode	Буква	Начертание	Клавиша	Код Unicode	Буква	Начертание	Клавиша
0x05D0	Алеф	א	t	0x05DE	Мэм	מ	n
0x05D1	Бэт	ב	c	0x05DF	Нун конечная	ן	i
0x05D2	Гимел	ג	d	0x05E0	Нун	נ	b
0x05D3	Далет	ד	s	0x05E1	Самех	ס	x
0x05D4	Хэй	ה	v	0x05E2	Аин	ע	g
0x05D5	Вав	ו	u	0x05E3	Пэй конечная	ף	;
0x05D6	Заин	ז	z	0x05E4	Пэй	פ	p
0x05D7	Хэт	ח	j	0x05E5	Цади конечная	ץ	.
0x05D8	Тэт	ט	y	0x05E6	Цади	צ	m

Алфавит иврита *(продолжение)*

Код Unicode	Буква	Начертание	Клавиша	Код Unicode	Буква	Начертание	Клавиша
0x05D9	йод	י	h	0x05E7	Коф	ק	e
0x05DA	Каф конечная	ך	l	0x05E8	Рейш	ר	r
0x05DB	Каф	כ	f	0x05E9	Шин	ש	A
0x05DC	Ламед	ל	k	0x05EA	Тав	ת	,
0x05DD	Мэм конечная	ם	o				

Названия этих букв я нашел в описании третьей версии стандарта Unicode. Как видите, в название некоторых из них входит слово «конечная»: такие буквы пишутся иначе, если стоят в конце слова,

Также следует знать, что на иврите пишут справа налево. Таким образом, чтобы набрать вышеуказанную фразу на иврите в TypeAway, надо вводить буквы в таком порядке: *бет* (клавиша с), *коф* (е), *реш* (г), пробел, *тэт* (у), *вав* (и) и снова *бет* (с). TypeAway хранит символы Unicode в символьной строке в порядке их ввода, и это правильно. Метод *DrawString* выводит эти символы справа налево, что тоже правильно. Поэтому методу *DrawString* надо отдать должное и поздравить его с тем, что смог узнать и правильно вывести текст, читающийся справа налево.

Теперь вы знаете, почему TypeAway должна полностью стирать ранее нарисованную текстовую строку: вводимый текст вовсе не обязательно добавляется к концу строки. При наборе на иврите новый текст должен выводиться в начале, а не в конце строки. Набор арабских текстов еще сложнее: в арабском письме соседние символы часто объединяются и образуют символы с другим начертанием. Чтобы корректно обработать такую ситуацию, *DrawString* должен рисовать всю строку сразу, а не отдельные символы.

Позиционирование каретки — именно то, с чем не справляется TypeAway. При вводе текста справа налево каретка не соответствует точке вставки. Решение этой проблемы нетривиально, особенно когда приходится иметь дело со строками, включающими текст, читающийся в разных направлениях, Похоже, что в Windows Forms не хватает средств для решения этих проблем. Если вам интересно, как они решаются при помощи API Win32, прочитайте статью «Supporting Multilanguage Test Layout and Complex Scripts wirh Windows NT 5.0» в ноябрьском выпуске журнала *Microsoft Systems Journal* за 1998 г.



ГЛАВА

7

Страницы и преобразования

Главная цель любой среды программирования графики — достичь независимости от устройства. Работа программ на разных типах видеоадаптеров независимо от их разрешения не должна отличаться. Программы также должны печатать текст и графику на разных принтерах, не требуя специализированного драйвера или отдельного кода для работы с графикой на каждом принтере,

В главе 5 я продемонстрировал создание кода, способного выводить графику как на экран, так и на принтер. Но до сих пор я использовал в качестве основной единицы измерений при рисовании (по крайней мере на экране) пикселы. Однако принтер несколько более сложная штука, да и сами пикселы едва ли можно назвать независимыми от устройства.

Достижение независимости от устройства при помощи текста

Приложив немного усилий, можно добиться независимости от устройства, даже используя пикселы. Первый способ — сделать так, чтобы вывод графики зависел от кегля шрифта по умолчанию, связанного с формой (т. е. от значения свойства *Font* формы). Этот подход особенно полезен при комбинировании текста и простой графики.

Допустим, вы программируете простое приложение для обработки баз данных, использующее метафору каталожной карточки. Для вывода каждой записи программа имитирует каталожную карточку размером 3 на 5 дюймов. Каков должен быть размер карточки, выраженный в пикселах? Вспомним пишущую машинку. Пишущая машинка с гарнитурой «пика» печатает 6 строчек на дюйм, поэтому на нашей карточке уместится до 18 машинописных строк. Таким образом, высоту карточки можно сделать равной 18 x *FontHeigh*пикселов, а ширину — 5/3 от этого числа.

Ширина карточки, равная 5/3 ее высоты, предполагает, что разрешение устройства вывода (число пикселов на единицу размера, в данном случае на дюйм) по горизонтали и вертикали одинаковое. Когда разрешение устройства графического вывода по горизонтали и вертикали одинаково, говорят, что у этого устройства *квадратные пикселы*. У самых первых дисплеев, бывших в ходу на момент выхода первой версии Windows в 1985 г., пикселы были не квадратные. Так продолжалось, пока созданный IBM в 1987 г. видеоадаптер VGA (Video Graphics Array), обладающий квадратными пикселами, не стал стандартом видеоадаптеров, совместимых с ПК,

Сегодня вполне допустимо считать пикселы дисплея, на котором работают программы Windows Forms, квадратными. Я сказал «вполне допустимо», так как квадратные пикселы не являются обязательным требованием для работы Windows. Кроме того, всегда есть возможность, что кто-нибудь напишет драйвер устройства Windows для какого-нибудь специализированного видеоадаптера с формой пикселов, отличной от квадратной.

Пикселы современных принтеров, как правило, *не* квадратные, а разрешение по одному измерению вдвое больше разрешения по другому

А сколько это в твердой валюте?

Рассмотрим связь между пикселами и размерами предметов реального мира. Допустим, вы рисуете квадрат со стороной 100 пикселов, расположенный на расстоянии 100 пикселов от верхнего левого угла клиентской области (или области печати страницы принтера).

HundredPixelsSquare.cs

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

class HundredPixelsSquare: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new HundredPixelsSquare());
     }
     public HundredPixelsSquare()
     {
          Text = "Hundred Pixels Square";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.FillRectangle(new SolidBrush(clr), 100, 100, 100, 100);
     }
}
```

Какой размер будет у этого квадрата на экране? А на принтере? И вообще, будет ли эта фигура квадратом?

Ясно, что пока программа не запущена, представление о размерах этого квадрата на экране (по крайней мере относительно размеров самого экрана) останется весьма туманным. Обычно наименьший размер современных дисплеев — 640 пикселов по горизонтали на 480 по вертикали (или 640 x 480). На таком дисплее относительные размеры квадрата будут равны примерно 1/6 ширины и 1/5 высоты экрана. Однако размер современного дисплея может достигать и 2 048 X 1 536 пикселов, и тогда размер квадрата относительно полного размера экрана будет существенноменьше,

Было бы неплохо знать разрешение дисплея, выраженное в общих единицах измерения, например в точках на дюйм (dpi), Но если для принтеров этот параметр определен совершенно точно (обычно его можно найти даже на упаковке принтера), то для мониторов он остается довольно расплывчатым. Реальное разрешение дисплея в dpi определяется двумя параметрами: физическим размером экрана (обычно его измеряют в дюймах по диагонали) и соответствующим числом пикселов,

Сильно сбивает с толку то, что именно последний параметр часто называют *разрешением* дисплея. В классе *SystemInformation*этому параметру соответствует элемент *PrimaryMonitorSize.*Если вызвать диалоговое окно Display Properties (Свойства экрана) из Панели управления [или щелкнув Рабочий стол правой кнопкой и выбрав из контекстного меню Properties (Свойства)] и перейти на вкладку Settings (Настройка), вы увидите, что размеры экрана в пикселах там названы *областью экрана,* Этот термин я и буду использовать.

Современные видеоадаптеры поддерживают 5-6 вариантов параметров размера области экрана, диагональ экрана мониторов также бывает разной. Вот небольшая таблица, в которой показано примерное разрешение дисплея в точках на дюйм для разных сочетаний размеров экрана монитора и области экрана:

Реальное разрешение дисплея (в точках на дюйм)

	Размер экрана монитора по диагонали (дюймов)			
Область экрана	15	17	19	21
640 X 480	57	50	44	40
800 X 600	7]	63	;б	50
1 024 X 768	91	80	71	64
1 152 X 870	103/104	90/91	80/81	72/73
1 280 X 1024	114/122	100/107	89/95	80/85
1 600 X 1 200	143	125	111	[illegible]
2 048 X 1 536	183	160	i 12	:28

Предполагается, что размер видимой области экрана на один дюйм меньше объявленного, а стандартные пропорции экрана — 4:3. Например, реальный размер экрана 21-дюймового монитора по диагонали составляет 20 дюймов, или (спасибо Пифагору) 16 дюймов по горизонтали и 12 по вертикали. Для области экрана 1 152 X 870 и 1 280 X 1 024 отношение размеров по горизонтали и вертикали отличается от 4:3. Отсюда следует, что разрешение по горизонтали и вертикали тоже отличается, но эти отличия достаточно малы, и ими можно пренебречь.

Итак, если вы работаете на 21-дюймовом мониторе в режиме 1 600 x 1 200, то размер квадрата со стороной 100 пикселов на экране будет около 1 дюйма. Но при использовании других режимов экрана он может быть как меньше 1/2 дюйма, так и больше 2. Конечно, мало кто использует режим 640 x 480 на 21-дюмовом мониторе, как и 2 048 X 1 536 на 15-дюймовом. Значения разрешения дисплея, которые применяются чаще всего, занимают в приведенной выше таблице диагональ от левого верхнего угла к правому нижнему.

Поскольку Windows не знает размера монитора, она не может сообщить реальное разрешение экрана. Даже обладая такими сведениями, ОС окажется в затруднении. В самом деле, *как* должна действовать ОС, если в результате подключения к компьютеру видеопроектора создан экран шириной около 2 метров? Должна ли ОС считать, что разрешение экрана существенно снизилось, поскольку увеличился его размер? Почти уверен, что вам бы этого не хотелось,

Самая важная проблема с дисплеем в том, чтобы получить на нем читабельный текст. Шрифт по умолчанию должен быть достаточно велик, чтобы текст бмл пригоден для чтения, но не слишком велик, поскольку хотелось бы также, чтобы экран вмещал как можно больше текста.

По этой причине Windows в основном игнорирует область экрана и размер монитора, делегируя выбор разрешения экрана очень важной персоне — вам!

Я уже упоминал диалоговое окно Display Properties, в котором на вкладке Settings можно менять параметры экрана (заметьте: здесь описано диалоговое окно Display Properties в Windows 2000, в других версиях Windows оно может несколько отличаться). На вкладке Settings есть кнопка Advanced (Дополнительно). В сущности это обходной путь, следуя которым, *вы сами* выбираете разрешение экрана в точках на дюйм. Я сказал «обходной», так как реально вы выбираете при этом размер системного шрифта Windows, удобный для чтения. Предполагается, что кегль системного шрифта равен 10 пунктам (кегль шрифта измеряют в пунктах, а пункт в компьютерной типографии равен 1/72 дюйма). Размер шрифта с кеглем 10, выраженный в пикселах, зависит от разрешения дисплея, измеряемого в точках на дюйм.

Например, системный шрифт по умолчанию называется Small Fonts (Мелкий шрифт). Высота символов этого шрифта составляет 13 пикселов. Если допустить, что кегль шрифта равен 10 пунктам, то 13 пикселов будут равны 10/72 дюйма, а разрешение дисплея составит (округленно) 96 dpi.

Высота символов другого варианта системного шрифта — Large Fonts (Крупный шрифт) — составит 16 пикселов. Если принять 16 пикселов равными 10/72 дюйма, то в этом случае разрешение дисплея (тоже округленно) составит 120 dpi.

Кстати, системный шрифт Windows — это вовсе не шрифт по умолчанию, доступный в программе Windows Forms через свойство *Font*. Windows Forms устанавливают несколько более мелкий шрифт по умолчанию с кеглем около 8 пунктов.[1]

[1] Ветеранам программирования для Windows будет любопытно узнать, откуда берутся мои числа. Здесь я могу сослаться на значения полей TextMetric: tmHeigh (равное 16 для Small Fonts и 20 для Large Fonts) и меньшее по значению tmInternalLeading (равно соответственно 3 и 4). Значение tmHeight подходит для межстрочного интервала, а меньшее значение tmInternalLeading показывает, как размер межстрочного интервала в пунктах преобразуется в высоту символа в пикселах (13 Для Small Fonts и *16 для* Large Fonts). Может сбить с толку то, что у шрифта по умолчанию в Windows Forms есть свойство Font.Height, сообщающее близкие значения: 13 для Small Fonts и 15 для Large Fonts. Но это значение межстрочного интервала, сравнимое со значением tmHeight. Кегль системного шрифта Windows составляет 10, а шрифта по умолчанию в Windows Forms — около 8 пунктов.

Шрифты Large Fonts и Small Fonts — не единственно возможные. При выборе из списка пункта Other (Другой...) выводится специальная линейка. Настраивая ее вручную, можно выбрать по-настоящему большой (с разрешением 480 dpi) или маленький (около 19 dpi) шрифт.

Сказанное можно кратко подытожить так: следует выбирать такой размер системного шрифта, который вы находите удобным для чтения. Windows считает подходящим шрифт с кеглем 10 пунктов. При выборе размера учитывается разрешение дисплея, измеряемое в точках на дюйм.

Обычно физический размер символов выбранного системного шрифта несколько больше, чем можно предположить исходя из его кегля, выраженного в пунктах. Дело в том. что расстояние между глазами и бумагой при чтении напечатанного текста равно примерно 30 см, а расстояние между глазами пользователя и экраном монитора — 60 см или около того.

Сколько точек в одном дюйме?

У объекта *Graphics* есть пара свойств, указывающих разрешение устройства вывода графики в точках на дюйм:

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ	Описание
float	*DpiX*	Чтение	Разрешение по горизонтали, точек на дюйм
float	*DpiY*	Чтение	Разрешение по вертикали, точек на дюйм

Вот короткая программа, которая выводит эти значения без лишнею шума.

DotsPerInch.cs

```
//-----------------------------------------
// DotsPerInch.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class DotsPerInch: PrintableForm
{
     public new static void Main()
     {
          Application.Run(newDotsPerInch());
     }
     public DotsPerInch()
     {
          Text = "Dots Per Inch";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.DrawString(String.Format("DpiX * {0}\nDpiY = {1}",
                                        grfx.DpiX, grfx.DpiY),
                          Font, new SolidBrush(clr), 0, 0);
     }
}
```

Значения, которые эта программа выводит в своей клиентской области, соответствуют параметрам в диалоговом окне Display Properties. Это будет 96 dpi, если выбран шрифт Small Fonts, 120 dpi в случае Large Fonts или иное значение, если выбран шрифт другого размера,

Если щелкнуть клиентскую область, будет напечатано соответствующее значение, показывающее разрешение принтера (может, вы и так его знаете или запросто можете найти в инструкции к принтеру). Разрешение современных принтеров составляет 300, 600, I 200 и 2 400 dpi или 720, 1 440 и 2 880 dpi.

А что с принтером?

В начале этой главы я привел программу HundredPixelsSquare, которая выводит квадрат со стороной 100 пикселов. Обсуждая ее, я задался вопросом: а какой размер будет у этого квадрата на экране? В действительности же ответ таков: понятие физического размера для такого объекта неуместно. Никто не думает, что, измеряя объекты приложенной к экрану линейкой, можно извлечь хоть немного полезной информации. Важно другое: горизонтальная и вертикальная линейки. выводимые *на экране,* должны быть согласованы друг с другом. В этом смысле экранные размеры квадрата со стороной 100 пикселов, выраженные в дюймах, равняются:

```
100 / grfx.DpiX
100 / grfx.DpiY
```

т. е. 1,04 дюйма, если выбран системный шрифт Small Fonts, или 0,83 дюйма, если выбран шрифт Large Fonts, или иному значению, если выбран другой размер системного шрифта,

Теперь о принтере... наверное, вам стоит попробовать сделать это самостоятельно. Программа HundredPixelsSquare рисует на вашем принтере прямоугольник со стороной, в точности равной 1 дюйму. Давайте попробуем еще что-нибудь, А вот эта программа пытается нарисовать эллипс диаметром 1 дюйм, заданный свойствами *DpiX* и *DpiY* объекта *Graphics:*

TryOneInchEllipse.cs

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

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

см. след. стр

```
            Text = "Try One-Inch Ellipse";
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            grfx.DrawEllipse(new Pen(clr), 0, 0, grfx.DpiX, grfx.DpiY);
        }
}
```

На экране эллипс выглядит практически безупречно. Однако мой принтер с разрешением 600 dpi напечатал эллипс диаметром 6 дюймов.

Ясно, что в случае дисплея функциям рисования объекта *Graphics* передаются координаты, исчисляемые в пикселах. Но для принтера, кажется, все иначе. В этом случае функциям рисования объекта *Graphics* передаются координаты, исчисляемые в единицах по 0,01 дюйма независимо от вида и модели принтера. Чуть ниже мы увидим, как это работает. Изящность решения в том, что разрешение дисплея где-то около 100 dpi и принтер рассматривается как устройство с разрешением 100 dpi. Это значит, что в случае крайней необходимости можно использовать одни и те же координаты для просмотра графики на дисплее и печати ее на принтере, получая приблизительно одинаковые результаты.

Ручное преобразование координат

При желании можно настраивать координаты, передаваемые функциям рисования объекта *Graphics,* через свойства *DpiX* и *DpiY.* Допустим, нужно что-то нарисовать в миллиметровых координатах с плавающей точкой. Для этого потребуется метод, преобразующий миллиметры в пикселы:

```
PointF MMConv(Graphics grfx, PointF pointf)
{
    pointf.X *= grfx.DpiX / 25.4f;
    pointf.Y *= grfx.DpiY / 25.4f;

    return pointf;
}
```

Метод получает координаты точек, выраженные в желаемых миллиметрах. Путем деления на 25,4 эти координаты преобразуются в дюймы (кстати, это точный пересчет). После умножения результата на разрешение устройства (в точках на дюйм) получается число пикселов.

Проверим наши расчеты, нарисовав 10-сантиметровую линейку,

TenCentimeterRuler.cs

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

см. след, стр.

```
class TenCentimeterRuler: PrintableForm
{
    public new static void Main()
    {
        Application.flun(ftew TenCentimeterRuler());
    }
    public TenCentimeterRuler()
    {
        Text = "Ten-Centimeter Ruler";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int ey)
    i
        Pen       pen   = new Pen(clr);
        Brush     brush = new SolidBrush(clr);
        const int xOffset = 10;
        const int yOffset * 10;

        grfx.DrawPolygon(pen, new PointF[]
            {
                MMConv(grfx, new PointF(xOffset,       yOffset)),
                MMConv(grfx, new PointF(xOffset + 100, yOffset)),
                MMConv(grfx, new PointF(xOffset + 100, yOffset + 10)),
                MMConv(grfx, new PointF(xOffset,       yOffset + 10))
            });

      , StringFormat strfmt = new StringFormat();
        strfmt.Alignment = StringAlignment.Center;

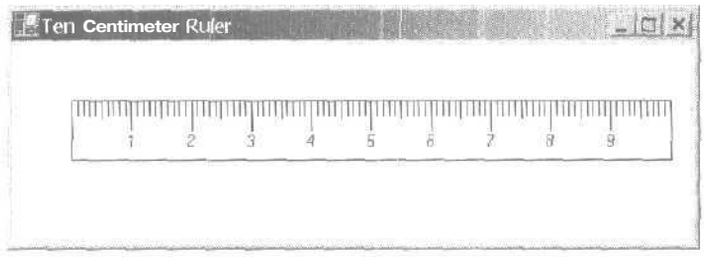
        for (int i = 1; i < 100; i++)
        {
            if (i % 10 == 0)          // Сантиметровые деления
            {
                grfx.DrawLine(pen,
                    MMConv(grfx, new Poi ntF(xOf fset + i, yOffset)),
                    MMConv(grfx, new PointF(xOffset + i, yOffset + 5)));

                grfx.DrawString((i/10).ToString(), Font, brush,
                    MMConv(grfx, new PointF(xOffset + i, yOffset + 5)),
                    strfmt);
            }
            else if (i % 5 == 0)      // Деления по 5 мм
            {
                grfx.DrawLine(pen,
                    MMConv(grfx, new PointF(xOffset + i, yOffset»,
                    MMConv(grfx, new PointF(xOffset + I. yOffset + 3)));
            } ,
            else                      // Миллиметровые деления
            {
                grfx.DrawLine(pen,
                  MMConv(grfx, new Point F(xOff set + i, yOffset)),
```

см. след. стр.

```
                    MMConv(grfx, new PointF(xOffset + 1, yOffset + 2.5f)));
                }
            }
        }
        PointF MMConv(Graphics grfx, PointF pointf)
        {
            pointf.X *= grfx.DpiX / 25.4f;
            pointf.Y *= grfx.DpiY / 25.4f;
            return pointf;
        }
    }
```

Вот как выглядит эта линейка на экране:



Кроме рисунка, здесь выводится текст. Почему я был уверен, что размер текста будет правильным? Просто мне было известно, что свойством *Font* задан шрифт с кеглем около 8 пунктов, высота его символов будет около 3 мм, что примерно соответствует нужному размеру.

Я сделал класс *TenCentimerRuler* потомком *PrintableForm*, чтобы раз и навсегда стало ясно: эта методика не будет работать на принтере. Мой принтер с разрешением 600 dpi печатает эту линейку в шесть раз больше, чем она выглядит на экране,

Единицы измерений и масштабирование страницы

Чтобы избежать написания таких методов, как *MMConv*, в GDI+ включен механизм автоматического масштабирования до заданного размера. В сущности этот механизм масштабирует координаты, передаваемые функциям рисования объекта *Graphics*, при помощи констант, как метод *MMConv*. Но при этом коэффициенты масштабирования задают не напрямую, а через свойства *PageUnit* и *PageScale*:

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ
GraphicsUnit	*PageUnit*	Чтение/запись.
float	*PageScale*	Чтение/запись.

В свойство *PageUnit* заносится одно из значений перечисления *GraphicsUnit*:

Перечисление *GraphicsUnit*

Член	Значение	Описание
World	0	Не используется со свойством *PageUnit*.
Display	1	То же, что *Pixel* для дисплея; для принтера равно 1/100 дюйма (задано по умолчанию для принтера).

Перечисление *GraphicsUnit* *(продолжение)*

Член	Значение	Описание
Pixel	2	Единицы измерения — пикселы (задано по умолчанию для дисплея).
Point	3	Единицы измерения — пункты (1/72 дюйма),
Inch	4	Единицы измерения — дюймы.
Document	5	Единицы измерения - 1/300 дюйма.
Millimeter	6	Единицы измерения — миллиметры.

Например, если вы хотите рисовать в координатах, исчисляемых сотыми долями дюйма, этой паре свойств надо задать такие значения:

```
grfx.PageUnit = GraphicsUnit.Inch;
grfx.PageScale = 0.01f;
```

что эквивалентно указанию считать одно деление координатной оси равным 0,01 дюйма. После этого следующий вызов метода *DrawLine* нарисует линию длиной в 1 дюйм:

```
grfx.DrawLine(pen, 0, 0, 100, 0);
```

Если отпечатать такую линию на принтере, вполне можно ее измерить и убедиться, **что ее** длина действительно равна 1 дюйму; на дисплее этот дюйм равен *grfx.DpiX* пикселов. Аналогичный результат можно получить так:

```
grfx.PageUnit = GraphicsUnit.Document;
grfx.PageScale = 3;
```

или:

```
grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 0.254f;
```

или:

```
grfx.PageUnit = GraphicsUnit.Point;
grfx.PageScale = 0.72f;
```

По умолчанию для дисплея задано значение *GraphicsUnitPixel*а для принтера — *GraphicsUnitDisplay* для обоих устройств значение *PageScale* равно 1. Кстати, смысл значения *GrapbicsUnitDisplay* для дисплея и для принтера разный. Для дисплея *это* то же самое, что *GraphicsUnitPixel,* но для принтера оно задает единицы измерения, равные 1/100 дюйма.

Итак, чтобы заставить программу TenCentimeterRuler работать с принтером, достаточно установить свойство *PageUnit* как *GraphicsUnitPixel*, и все будет хорошо. Новая версия метода *OnPage* устанавливает свойство *PageUnit* и вызывает метод *DoPage* из базового класса,

PrintableTenCentimeterRuler.cs

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

см. след. стр.

```
using System;
using System.Drawing;
using System.Windows.Forms;
class PrintableTenCentimeterRuler: TenCentimeterRuler
{
     public new static void Main()
     {
          Application.flun(rvew PrintableTenCentimeterRuler());
     }
     public PrintableTenCentimeterRuler()
     {
          Text = "Printable " + Text;
     }
    ' protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.PageUnit = GraphicsUnit.Pixel;
          base.DoPage(grfx, clr, cx, cy);
     }
}
```

В этой программе метод *DoPage* не использует аргументы *cx* и *cy*. Размеры, определяемые этими аргументами (размеры клиентской области формы и области печати страницы принтера), исчисляются в единицах, совместимых со значением *PageUnit* по умолчанию. В общем случае при изменении значения *PageUnit* размер страницы устройства вывода тоже должен быть пересчитан в идентичные единицы. Чуть ниже мы обсудим этот вопрос.

Теперь, даже если рисовать на принтере, задавая координаты в пикселах, вид шрифтов не исказится, а у шрифта, доступного через свойство формы *Font,* будет кегль 8 как на дисплее, так и на принтере. В главе 9 мы увидим, как это работает.

Однако в этой программе остается проблема с пером, определяемым версией метода *DoPage*, которая используется в программе TenCentimeterRuler:

```
Pen pen = new Pen(clr);
```

Толщина этого пера по умолчанию — 1. Для дисплея это означает 1 пиксел, а для принтера это обычно 1/100 дюйма. Но если изменить значение *PageUnit* на *GraphicsUnit.Pixel*, то толщина 1 будет интерпретирована как 1 пиксел. На некоторых принтерах с высоким разрешением такая линия будет почти невидимой,

Чтобы не возиться с исходной версией программы, рисующей 10-сантиметровую линейку, задействуем преимущество свойств *PageUnit* и *PageScale* и избавимся от ручного преобразования координат.

TenCentimeterRulerAuto.cs

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

см. след. стр.

```
class TenCentimeterRulerAuto: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new TenCentimeterRulerAuto());
    > •
    public TenCentimeterRulerAuto()
    {
        Text = "Ten-Centimeter Ruler (Auto)";
    }
    protected override void DoPage(Graphics grfx, Color clr, int ox, int cy)
    {
        Pen       pen   = new Pen(clr, 0.25f);
        Brush     brush = new SolidBrush(clr);
        const int xOffset = 10;
        const int yOffset = 10;

        grfx.PageUnit = GraphicsUnit.Millimeter;
        grfx.PageScale = 1;
        grfx.DrawRectangle(pen, xOffset, yOffset, too, 10);

        StringFormat strfmt = new StringFormat();
        strfmt.Alignment - StringAlignment.Center;

        for (int 1 * 1; i < 100; i++)
        {
            if (i % 10 == 0)         // Сантиметровые делений
            {
                grfx.DrawLine(pen,
                              new PointF(xOffset + i, yOffset),
                              new PointF(xOffset + i, yOffset + 5));

                grfx.DrawString((i/10).ToString(), Font, brush,
                                new PointF(xOffset + i, yGffset + 5),
                                strfmt);
            }
            else if (i % 5 == 0)     // Деления по 5 им
            {
                grfx.DrawLine(pen,
                              new PointF(xOffset + i, yOffset),
                              new PointF(xOffset + i, yOffset + 3»;
            }
            else                     // Миллиметровые деления
            {
                grfx.DrawLine(pen,
                              new PointF(xOffset + i, yOffset),
                              new PointF(xOffset * i, yOffset + 2.5f));
            }
        }
    }
}
```

Кроме удаления метода *MMConv*, почти ничего не изменилось. Мой метод *MMConv* работал только со структурами *PointF*, поэтому в ранних версиях программы для рисования контуров линейки использовался метод *DrawPolygon*, а не *DrawRectangle*. Поскольку GDI+ одинаково масштабирует координаты и размеры, здесь можно применить метод *DrawRectangle*. Другое изменение коснулось метода *DoPage*. Теперь в начале *DoPage* программа создает перо толщиной 0,25 единицы:

```
Pen pen = new Pen(clr, 0.25f);
```

Программа также настраивает объект *Graphics* для рисования в миллиметровых координатах:

```
grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 1;
```

Возможно, вам любопытно знать, есть ли разница между установкой свойств *PageUnit* и *PageScale* перед созданием пера и созданием пера с заданной толщиной перед установкой этих свойств. Когда создано перо: до установки этих свойств или после нее — не имеет никакого значения, так как объекты *Pen* независимы от устройства! Пока не вызван ни один из методов рисования линий, объект *Pen* не связан ни с одним конкретным объектом *Graphics*. Лишь тогда толщина пера интерпретируется в зависимости от текущих значений свойств *PageUnit* и *PageScale*. В данном случае толщина пера интерпретируется как 0,25 мм или около 1/100 дюйма. Чтобы увидеть различия на печати, можно попробовать меньшее значение (например, 0,10 мм).

Если толщина пера не включена в его конструктор, создается перо толщиной в 1 единицу. В данном случае это значит, что будет создано перо толщиной ровно 1 мм и все деления сольются в одно большое пятно (убедитесь сами!).

Толщина пера

Перо какой толщины подходит для принтера? Намек на ответ дает PostScript — широко известный и весьма уважаемый язык описания страниц, применяемый в большинстве профессиональных принтеров. Можно считать, что толщина обычного пера по умолчанию — 1 пункт (иначе 1/72 дюйма, или 1/3 мм). Лично я не считаю этот пример самым изящным, зато его легко запомнить.

Вот программа, которая выводит несколько перьев с разной толщиной, выраженной в пунктах.

PenWidths.cs

```
//-------------------------------------
// PenWidths.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class PenWidths: PrintableForm
{
     public new static void Main()
```

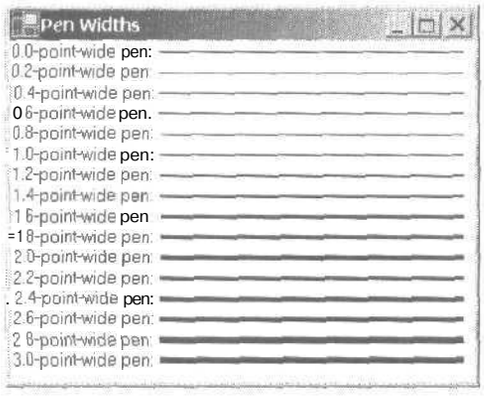
см. след. стр.

```
{
    Application.Run(new PenWidths());
}
public PenWidths()
{
    Text = "Pen Widths";
}
protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
{
    Brush brush = new SolidBrush(clr);
    float y = 0;

    grfx.PageUnit = GraphicsUnit.Point;
    grfx.PageScale = 1;
    for (float f = 0; f < 3.2; f += 0.2f)
    {
        Pen pen = new Pen(clr, f);
        string str = String.Format("{0:F1} point wide pen: ", f);
        SizeF sizef = grfx.MeasureString(str, Font);
        grfx.DrawString(str, Font, brush, 0, y);
        grfx.DrawLine(pen, sizef.Width, y + sizef.Height / 2,
                           sizef.Width + 144, y + sizef.Height / 2);
        y += sizef.Height;
    }
}
}
```

Хотя число пикселов толщины пера можно задать дробным числом, перо визуализируется лишь целым числом пикселов, При выводе на экран толщина большинства перьев, созданных этой программой, округляется до соответствующих значений:

Можно не беспокоиться, что при выводе на экран толщина пера будет округлена до 0 и перо исчезнет. Перья всегда рисуются толщиной минимум 1 пиксел. Действительно, если установить Б конструкторе *Pen* толщину пера 0, то независимо от значений свойств *PageUnit и PageScale* все линии будут нарисованы толщиной в 1 пиксел.



Хотя перья с толщиной 0 прекрасно подходят для дисплея, никогда не используйте их для принтера. На современном лазерном принтере, печатающем с высоким разрешением, линия толщиной в 1 пиксел будет практически невидимой.

Вот программа для рисования *шестидюймовой* линейки с делениями, *использующая* единицы измерения 1/64 дюйма и создающая перо толщиной 1/128 дюйма.

SixInchRuler.cs

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

class SixInchRuler: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new SixInchRuler());
     }
     public SixInchRuler()
     {
          Text = "Six-Inch Ruler";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush brush = new SolidBrush(clr);
          Pen pen = new Pen(clr, 0.5f);
          const int xOffset = 16;
          const int yOffset = 16;

          grfx.PageUnit = GraphicsUnit.Inch;
          grfx.PageScale = 1 / 64f;
          grfx.DrawRectangle(pen, xOffset, yOffset, 8 * 64, 64);

          StringFormat strfmt = new StringFormat();
          strfmt.Alignment = StringAlignment.Center;

          for (int i = 0; i < 8 * 16; i++)
          {
               int x = xOffset + i * 4;
               int y = yOffset;
               int dy;

               if(i % 16 == 0)          // Дюймовые деления
               {
                    dy = 32;
                    grfx.DrawString((i / 16).ToString(), Font, brush,
                                                  x, y * dy, strfmt);
```

См. след. стр.

```
            }
            else if(i * 8 == 0)          // Деления а полдюйма .
                dy = 24;

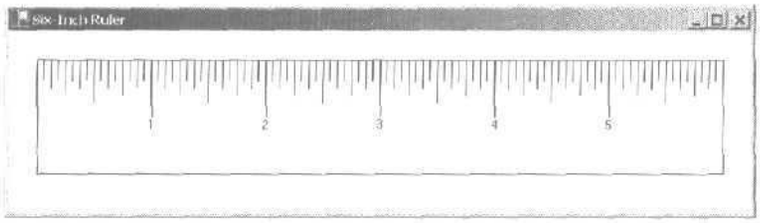
            else if<i X 4 == 0)          // Деления в четверть дюйма
                dy = 20;

            else if(i % 2 == 0)          // Деления в одну восьмую дюйма
                dy = 16;

            else                         // Деления в одну шестнадцатую дюйма
                dy = 12;

            grfx.DrawLine(pen, x, y, x, y + dy);
        }
    }
}
```

Программа рисует такую линейку:



Возможно, вы заметили, что ни одно из значений *PageUnit* и *PageScale*, по-видимому не влияет на текст. Дело в том, что кегль шрифта, доступного через свойство *Font* формы, составляет около 8 пунктов и не меняется. В главе 9 я покажу, как создать шрифт, размер которого *зависит* от значений свойств *PageUnit* и *PageScale*.

Преобразования страницы

Устанавливая значения свойств *PageScale* и *PageUnit,* в действительности вы задаете параметры *преобразования страницы* (page transform), которое может быть представлено парой простых формул. Допустим, координаты, передаваемые графическим методам объекта *Graphics,* — это *координаты на странице* (page coordinates). Как вы еще убедитесь, это допущение практически соответствует истине, но только когда установлены свойства *PageScale* и *PageUnit.* В пространстве координат страницы любая точка может быть представлена как (x_{page}, y_{page}).

Говорят, что координаты, исчисляемые в пикселах (начало координат расположено в левом верхнем углу клиентской области или области печати страницы), — это *координаты на устройстве* (device coordinates). В этом пространстве координат точка может быть задана как (x_{device}, y_{device}). Параметрами преобразования страницы являются значения свойств *PageUnit, PageScale, DpiX* и *DpiY.*

Формулы преобразования страницы

Значение *PageUnit*	Формула преобразования
GraphicsUnit.Pixel	$x_{device} = x_{page} \times PageScale$ $y_{device} = y_{page} \times PageScale$
GraphicsUnit.Display (для дисплея)	$x_{device} = x_{page} \times PageScale$ $y_{device} = y_{page} \times PageScale$
GraphicsUnit.Display (для принтера)	$x_{device} = x \times PageScale \times DpiX / 100$ $y_{device} = y_{de} * PageScale \times DpiY / 100$
GraphicsUnit.Inch	$x_{device} = x_{page} \times PageScale \times DpiX$ $y_{device} = y_{page} \times PageScale \times DpiY$
GraphicsUnit.Millimeter	$x_{device} = x_{page} \times PageScale \times DpiX / 25.4$ $y_{device} = y_{page} \times PageScale \times DpiY / 25.4$
GraphicsUnit.Point	$x_{device} = x_{page} \times PageScale \times DpiX / 72$ $y_{device} = y_{page} \times PageScale \times DpiY / 72$
GraphicsUnit.Document	$x_{device} = x_{page} \times PageScale \times DpiX / 300$ $y_{device} = y_{page} \times PageScale \times DpiY / 300$

В общем случае:

$x_{device} = x_{page} \times PageScale \times DpiX /$ (*GraphicsUnit*, единиц на дюйм)
$y_{device} = y_{page} \times PageScale \times DpiY /$ (*GraphicsUnit*, единиц на дюйм)

Остерегайтесь действий над целыми числами. Можно попытаться задать коэффициент масштабирования страницы в 1/64 дюйма так:

```
grfx.PageScale = 1 / 64;
```

Но C# выполняет целочисленное деление *перед* автоматическим преобразованием в *float*! Здесь нужно помочь компилятору, указав ему. что от него требуется:

```
grfx.PageScale = 1f / 64;
```

Преобразование страницы влияет на все координаты всех функций рисования из класса *Graphics*, о которых шла речь до сих пор. Оно также влияет на сведения, возвращаемые *MeasureString* и версией метода *GetHeight*, реализованной в классе *Font* и принимающей объект *Graphics* как аргумент.

Преобразование страницы — атрибут, характерный только для класса *Graphics*, Оно влияет только на объекты-члены класса *Graphics* или принимающие таковые в качестве аргумента (как *GetHeight*) и не влияет ни на что другое. Так, преобразование страницы не влияет на данные, предоставляемые *ClientSize*: значение *ClientSize* всегда выражается в пикселах,

Сохранение состояния объекта Graphics

Изменение значений свойств *PageUnit* и *PageScale* объекта *Graphics* существенно влияет на вид выводимой графики. Иногда значение этих или других свойств объекта *Graphics* меняют на время, чтобы нарисовать несколько фигур или получить какие-то данные, после чего возникнет необходимость вернуться к исходным значениям свойств.

У класса *Graphics* есть два метода, *Save* и *Restore*, которые служат именно этой цели: сохраняют свойства объекта *Graphics* и восстанавливают их впоследствии.

Эти методы используют класс *GraphicsState*, определяемый в пространстве имен *System.Drawing.Drawing2D*.

Методы *Graphics* (выборочно)

GraphicsState Save()
void Restore(GraphicsState gs)

У класса *GraphicsState* нет открытых атрибутов, представляющих хоть какой-нибудь интерес, — это просто черный ящик! При вызове:

```
GraphicsState gs = grfx.Save();
```

текущие значения всех изменяемых свойств объекта *Graphics* сохраняются в объекте *GraphicsState,* после чего можно изменять свойства объекта *Graphics,* Для восстановления сохраненных свойств служит вызов:

```
grfx.Restore(gs);
```

Программисты с опытом использования Win32, вероятно, привыкли думать о подобных механизмах (к которым относятся функции *SaveDC* и *RestoreDC*) как о стеке магазинного типа (который работает по принципу «последним пришел – первым ушел»). Но в Windows Forms они реализованы более гибко. Например, можно начать обработку *OnPaint* с сохранения трех состояний объекта *Graphics*:

```
GraphicsState gs1 = grfx.Save();

    // Изменение некоторых свойств.
    ...

GraphicsState gs2 = grfx.Save();

    // Изменение некоторых свойств.
    ...

GraphicsState gs3 = grfx.Save();
```

После этого сохраненные данные о состоянии можно задействовать в любое время и в любом порядке, вызывая метод *Restore*.

Аналогичный механизм предоставляют методы *BeginContainer* и *EndContainer* класса *Graphics,* использующие класс *GraphicsContainer* из пространства имен *System.Drawing.Drawing2D*.

Метрические размеры

Размеры клиентской области формы доступны через ее свойство *ClientSize* и всегда исчисляются в пикселах. Задав другое преобразование страницы, лучше выражать размеры клиентской области не в пикселах, а в единицах, используемых для методов рисования.

Существует минимум два способа получения размеров клиентской области в метрических единицах. Возможно, самый удобный — использование свойства *VisibleClipBounds* объекта *Graphics*. Оно всегда возвращает размеры клиентской об-

ласти в единицах, соответствующих текущим значениям свойств *PageUnit* и *PageScale*. Вот программа, которая при помощи *VisibleClipBounds* показывает размер клиентской области, выраженный во всевозможных единицах.

WhatSize.cs

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

class WhatSize: PrintableForm
{
     public new static void Main()
     (
          Application.Run(new WhatSize());
     >
     public WhatSize()
     {
          Text = "What Size?";
     >
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush brush = new SolidBrush(clr);
          int   y     * 0;

          DoIt(grfx, brush, ref y, GraphicsUnit.Pixel);
          DoIt(grfx, brush, ref y, GraphicsUnit.Display);
          DoIt(grfx, brush, ref y, GraphicsUnit.Document);
          DoIt(grfx, brush, ref y, GraphicsUnit.Inch);
          DoIt(grfx, brush, ref y, GraphicsUnit.Millimeter);
          DoIt(grfx, brush, ref y, GraphicsUnit.Point);
     }
     void DoIt(Graphics grfx, Brush brush, ref int y, GraphicsUnit gu)
     {
          GraphicsState gs * grfx.Save();

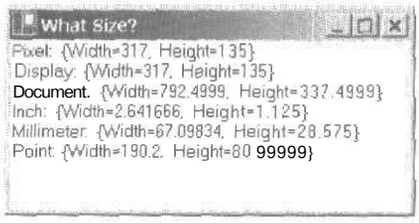
          grfx.PageUnit  * gu;
          grfx.PageScale = 1;

          SizeF sizef = grfx.VisibleClipBounds.Size;

          grfx.Restore(gs);

          grfx.DrawString(gu+ ": " + sizef, Font, brush, 0, y);
          y +« (int) Math.Ceiling(Font.GetHeight(grfx));
     }
}
```

Метод *DoIt* в этой программе использует возможности методов *Save* и *Restore*, поэтому различные значения *PageUnit* в действительности не мешают выводу информации при вызове методов *DrawString* и *GetHeight*. Вот типичный вид программы WhatSize:



К сожалению, в случае принтера все иначе. Конструкция свойства *VisibleClipBounds* такова, что для принтера оно возвращает значения в единицах 1/100 дюйма независимо от заданного преобразования страницы. Но если значение свойства *PageUnit* для принтера задает измерения в пикселах. *VisibleClipBounds* также возвращает размер области печати в пикселах.

Историческое замечание: я написал свою первую статью о программировании для Windows в 1986 г. для декабрьского выпуска *Microsoft Systems Journal*. WSZ («what size»), программа-пример для этой статьи, выводила размер клиентской области программы в пикселах, дюймах и миллиметрах. Приведенная здесь программа WhatSize — существенно упрощенная и сокращенная версия своей бабушки.

Другой подход к определению размеров области экрана включает использование метода *TransformPoints*, реализованного в классе *Graphics*:

Методы *TransformPoints* класса *Graphics*

```
void TransformPoints(CoordinateSpace csDst, CoordinateSpace csSrc,
                     Point[] apt)
void TransformPoints(CoordinateSpace csDst, CoordinateSpace csSrc,
                     PointF[] aptf)
```

Перечисление *CoordinateSpace* определяется в пространстве имен *System.Drawing.Drawing2D*:

Перечисление *CoordinateSpace*

Член	Значение
World	0
Page	1
Device	2

Пока нам известны два пространства координат: *Device* (где размеры исчисляются в пикселах относительно левого верхнего угла клиентской области) и *Page* (в нем размеры исчисляются в дюймах, миллиметрах, пунктах и т. п.). При наличии массива структур *PointF*, выраженных в единицах устройства, можно пересчитать их в единицы страницы, вызвав метод:

```
grfx.TransformPoints(CoordinateSpace.Page, CoordinateSpace.Device, apt);
```

О пространстве координат *World* мы поговорим чуть ниже.

Вот другая версия программы WhatSize, использующая для расчета размера клиентской области метод *TransformPoints*.

WhatSizeTransform.cs

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

class WhatSizeTransform : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new WhatSizeTransform());
    }
    public WhatSizeTransform()
    {
        Text = "What Size? With TransformPoints";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush brush = new SolidBrush(clr);
        int y = 0;
        Point[] apt = { new Point(cx, cy) };

        grfx.TransformPoints(CoordinateSpace.Device,
                             CoordinateSpace.Page, apt);

        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Pixel);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Display);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Document);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Inch);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Millimeter);
        DoIt(grfx, brush, ref y, apt[0], GraphicsUnit.Point);
    }
    void DoIt(Graphics grfx, Brush brush, ref int y,
              Point pt, GraphicsUnit gu)
    {
        GraphicsState gs = grfx.Save();

        grfx.PageUnit = gu;
        grfx.PageScale = 1;

        PointF[] aptf = { pt };

        grfx.TransformPoints(CoordinateSpace.Page,
```

см. след. стр.

```
                                CoordinateSpace.Device, aptf);

            SizeF sizef = new SizeF(aptf[0]);
            grfx.Restore(gs);

            grfx.DrawString(gu + ": " + sizef, Font, brush, 0, y);
            y += (int) Math.Ceiling(Font.GetHeight(grfx));
        }
```

В этой программе я добавил к методу *DoIt* еще один аргумент — структуру *Point,* содержащую значения ширины и высоты области экрана в пикселах. Поскольку значения аргументов *cx* и *cy* метода *DoPage* уже выражены в пикселах, это не проблема для дисплея, но проблема для принтера. В силу этих причин метод *DoPage* прибавляет значения *cx и cy к* структуре *Point,* создает массив структур *Point* из одного элемента и передает этот массив методу *TransformPoints* для преобразования значений в единицы устройства. Заметьте, что целевым пространством координат при вызове *TransformPoints* является *CoordinateSpaceDevice*. Затем *DoIt* осуществляет преобразование из единиц устройства к пространству координат *CoordinateSpace.Page* с помощью метода *TransformPoints,*

Произвольные координаты

Некоторые программы в предыдущих главах масштабировали выводимую ими графику по размерам клиентской области или области печати страницы принтера. Программы же этой главы задают размеры в миллиметрах или дюймах.

Иногда требуется жестко запрограммировать набор координат и избежать *при* этом их масштабирования в явном виде. Допустим, нужно запрограммировать вывод графики в некоторой системе координат, скажем, размером 1 000 X 1 000 единиц по горизонтали и вертикали. Это пространство координат нужно сделать как можно большим, но не больше клиентской области или страницы принтера.

Вот как это делается:

ArbitraryCoordinates.cs

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

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

см. след, стр.

```
    public ArbitraryCoordinates()
    {
        Text = "Arbitrary Coordinates";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy>
    {
        grfx.PageUnit = GraphicsUnit.Pixel;
        SizeF sizef = grfx.VisibleClipBounds.Size;

        grfx.PageUnit  = GraphicsUnit.Inch;
        grfx.PageScale = Math.Min(sizef.Width  / grfx.DpiX / 1000,
                                  sizef.Height / grfx.DpiY / 1000);

        grfx.DrawEllipse(new Pen(clr), 0, 0, 990, 990);
    }
}
```

Сначала метод *DoPage* устанавливает свойство *PageUnit* как *GraphicsUnit.Pixel* с единственной целью — получить значение свойства *VisibleClipBounds*, указывающее размер клиентской области или страницы принтера в пикселах,

Далее *DoPage* устанавливает *PageUnit* для измерений в дюймах:

```
grfx.PageUnit = GraphicsUnit.Inch;
```

Выше я показал, что при этом применяются следующие формулы преобразования:

$$x_{device} = x_{page} \times PageScale \times DpiX$$
$$y_{device} = y_{page} \times PageScale \times DpiY$$

Значения x и y должны варьироваться в пределах 0 до 1000, а x_{device} и y_{device} — от 0 до значений *Width* и *Height* свойства *VisibleCUpBounds* соответственно. Иным словами:

$$Width = 1000 \times PageScale \times DpiX$$
$$Height = 1000 \times PageScale \times DpiY$$

Однако эти уравнения дадут два различных коэффициента масштабирования, а нужен лишь один. Надо выбрать меньшее из двух рассчитанных значений:

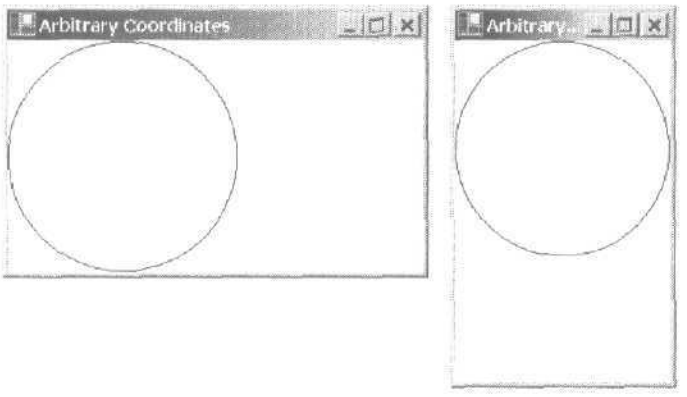
```
grfx.PageScale = Math.Min(sizef.Width  / grfx.DpiX / 1000,
                          sizef.Height / grfx.DpiY / 1000);
```

Далее программа рисует эллипс шириной и высотой в 990 единиц (использование значений 1 000 или 999 для ширины и высоты иногда приводит к обрезанию фигуры при больших размерах окна). Результирующая фигура — окружность, которая рисуется слева, если ширина клиентской области больше ее высоты, или сверху, если наоборот (см. рис. на следующей странице).

Окружность можно напечатать, при этом она будет вписана по ширине в область печати страницы.

Но с этой программой есть маленькая проблема, Попробуйте как можно сильнее уменьшить размер окна. Вы увидите, что уменьшать ширину окна можно лишь до некоторого предела, в то время как высоту окна можно уменьшать хоть до О,

В этот момент возникает исключение, поскольку метод *DoPage* присваивает свойству *PageScale* недействительное значение 0.



Есть два пути решения этой проблемы. Самый очевидный, вероятно, — просто прервать исполнение метода *DoPage* при достижении нулевой высоты клиентской области:

```
if (cy == 0)
     return;
```

Это вполне нормально, поскольку рисовать что-то в таком окне нет смысла.

Не находите ли вы, что вызывать метод *OnPaint* при нулевом размере клиентской области, несколько странно? Поэтому не повредит добавить в начало метода *OnPaint* оператор:

```
if (pea.ClipRectangle.IsEmpty)
     return;
```

эквивалентный оператору:

```
if (grfx.IsVisibleClipEmpty)
     return;
```

Очень специализированное решение — вызвать метод *MathMax* для расчета свойства *PageScale*, чтобы исключить появление значения 0:

```
grfx.PageScale = Math.Min(sizef.Width                / grfx.DpiX / 1000,
                          Math.Max(sizef.Height, 1) / grfx.DpiY / 1000);
```

Или, чтобы показать всем, что вам кое-что известно об обработке исключений в C#, можно поместить проблемный оператор в блок *try:*

```
try
{
     grfx.PageScale = Math.Min(sizef.Width  / grfx.DpiX / 1000,
                               sizef.Height / grfx.DpiY / 1000);
}
catch
{
     return;
}
```

Но есть и не столь очевидный метод, суть которого — в предотвращении уменьшения размера клиентской области до 0. Статическое свойство *SystemInformation.MinimumWindowSize* возвращает размер, равный сумме высоты заголовка и удвоенной толщины границы окна. Чтобы при минимальной ширине окна была видна часть его заголовка, ширина окна должна быть гораздо больше минимальной.

Можно использовать свойство *MinimumSize*, чтобы поддерживать размер окна не меньше заданного. Попробуйте поместить в конструктор ArbitraryCoordinates следующее:

```
MinimumSize = SystemInformation.MinimumWindowSize + new Size(0, 1);
```

Ограничения преобразования страницы

Некоторые действия в преобразовании страницы недопустимы. Во-первых, нельзя заносить в свойство *PageScale* отрицательные значения, т. е. нельзя увеличивать координаты по оси *x* влево от начала координат (что иногда пытаются делать) или распространять координаты *y* за пределы верхней границы экрана (люди, склонные мыслить математически, считают это полезным).

Во-вторых, нельзя вводить разные единицы для измерений по горизонтали и вертикали. Свойства *PageScale* и *PageUnit* применимы в равной мере к обеим осям. Функция типа:

```
grfx.DrawEllipse(pen, 0, 0, 100, 100);
```

всегда будет рисовать окружность независимо от преобразования страницы, кроме единственного случая, когда свойство *PageUnit* задано как *GraphicsUnit.Pixel*, а разрешение устройства вывода по горизонтали и вертикали отличается. Эта проблема редко возникает с дисплеем, но довольно обычна для принтеров.

Наконец, нельзя менять положение начала координат. Точка с координатами (0, 0) всегда — в левом верхнем углу клиентской области или области печати страницы принтера.

К счастью, GDI+ поддерживает еще одно преобразование, позволяющее решать эти и множество других задач.

Знакомство с глобальным преобразованием

Еще одно преобразование, поддерживаемое GDI+, называется *глобальным преобразованием* (world transform). Оно использует традиционную матрицу 3×3, но это не обязательно, и матрицу можно заменить любым удобным методом. Для начала взгляните на эту программу, которая выводит первый абзац книги Г. Мелвилла «Моби Дик».

MobyDick.cs

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

см. след. стр.

```
class MobyDick: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new MobyDick());
    }
    public MobyDick()
    {
        Text = "Moby-Dick by Herman Melville";
    }

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
            // Сюда можно вставить вызовы ScaleTransform,
            // RotateTransform, TranslateTransform и других функций.

        grfx.DrawString("Call me Ishmael. Some years ago\x2014never " +
                        "mind how long precisely\x2014having little " +
                        "or no money in my purse, and nothing " +
                        "particular to interest me on shore, I " +
                        "thought I would sail about a little and " +
                        "see the watery part of the world. It is " +
                        "a way I have of driving off the spleen, " +
                        "and regulating the circulation. Whenever " +
                        "I find myself growing grim about the " +
                        "mouth; whenever it is a damp, drizzly " +
                        "November in my soul; whenever I find " +
                        "myself involuntarily pausing before " +
                        "coffin warehouses, and bringing up the " +
                        "rear of every funeral I meet; and " +
                        "especially whenever my hypos get such an " +
                        "upper hand of me, that it requires a " +
                        "strong moral principle to prevent me " +
                        "from deliberately stepping into the " +
                        "street, and methodically knocking " +
                        "people's hats off\x2014then, I account it " +
                        "high time to get to sea as soon as I " +
                        "can. This is my substitute for pistol " +
                        "and ball. With a philosophical flourish " +
                        "Cato throws himself upon his sword; I " +
                        "quietly take to the ship. There is " +
                        "nothing surprising in this. If they but " +
                        "knew it, almost all men in their degree, " +
                        "some time or other, cherish very nearly " +
                        "the same feelings towards the ocean with " +
                        "me.",
                        Font, new SolidBrush(clr),
                        new Rectangle(0, 0, cx, cy));
    }
}
```

Для вас здесь ничего нового. Я лишь отметил место, куда можно вставить пару строк кода, после чего, перекомпилировав программу, можно посмотреть на результат.

Начать можно с этого:

```
grfx.RotateTransform(45);
```

Проследите, чтобы этот вызов был вставлен *перед* вызовом *DrawString.* В результате текст будет повернут на 45° по часовой стрелке:



Довольно просто, не так ли? Заметьте: текст по-прежнему внутри прямоугольника, заданного вызовом функции *DrawString,* хотя этот прямоугольник был повернут вместе с текстом. Повернутый текст также можно напечатать, но должен вас предупредить, что создание файла задания печати займет больше времени, чем обычно,

На что действует *RotateTransform?* На все упомянутые выше функции рисования объекта *Graphics.* Аргументом *RotateTransform* является значение *float,* которое может быть как положительным, так и отрицательным.

Теперь попробуйте вставить такой код:

```
grfx.RotateTransform(-45);
```

Текст поворачивается на 45° против часовой стрелки. Угол может быть и больше 360°. и меньше -360°. В нашем примере любое значение, которое не разрешается в угол между -90° и 90°, вызовет поворот текста вправо с выходом за пределы видимой области окна.

Последовательные вызовы *RotateTransform* имеют кумулятивный эффект, Б результате вызовов:

```
grfx.RotateTransform(5);
grfx.RotateTransform(10);
grfx.RotateTransform(-20);
```

текст повернется на 5° против часовой стрелки.

А теперь попробуйте это:

```
grfx.ScaleTransform(1, 3);
```

Эта функция увеличивает значения координат и размеры выводимой графики. Первый аргумент действует на координаты и размеры по горизонтали, а второй —

по вертикали. Вызов этой функции в программе MobyDick втрое увеличивает высоту символов, оставляя их ширину прежней. Вызов;

```
grfx.ScaleTransform(3,1);
```

не влияет на высоту символов, но делает их втрое шире. Аналогично увеличиваются размеры прямоугольника, в котором выводится текст, поэтому позиции обрывов строк не меняются. Эти эффекты можно комбинировать:

```
grfx.ScaleTransform(3, 3);
```

И в этом случае аргументами являются значения *float*, которые действуют взаимодополняюще. Масштабирование по горизонтали и вертикали с коэффициентом 3 можно выполнить этими двумя вызовами, сделанными последовательно

```
grfx.ScaleTransform(3, 1);
grfx.ScaleTransform(1, 3);
```

или:

```
grfx.ScaleTransform((float)Math.Sqrt(3), (float)Math.Sqrt(3));
grfx.ScaleTransform((float)Math.Sqrt(3), (float)Math.Sqrt(3));
```

Но что *самое* интересное: после масштабирования текст не становится рваным. Эффект скорее напоминает текст, напечатанный шрифтом большего размера, чем результат увеличения размера уже существующего шрифта.

Могут ли коэффициенты масштабирования быть отрицательными? Да. Однако, если попытаться использовать отрицательный коэффициент прямо сейчас, вы ничего не увидите. Но скоро мы доберемся до места, когда вы сможете применить отрицательные коэффициенты масштабирования и увидеть их потрясающий эффект. Коэффициент масштабирования не может равняться 0, при этом функция вызовет исключение.

Ну, а самое скучное я приберег на конец. Вызов *TranslateTransform* просто сдвигает координаты по горизонтальной и вертикальной осям. Например, вызов:

```
grfx.TranslateTransform(100, 50);
```

добавленный в программу MobyDick, сдвигает начало текста на 100 пикселов вправо и на 50 пикселов вниз от начала клиентской области. Если напечатать эту версию, текст окажется сдвинутым на 1 дюйм вправо и на 1/2 дюйма вниз относительно начала области печати страницы. Отрицательное значение первого аргумента сдвигает текст за левую границу клиентской области, а отрицательное значение *y* — за верхнюю границу этой области,

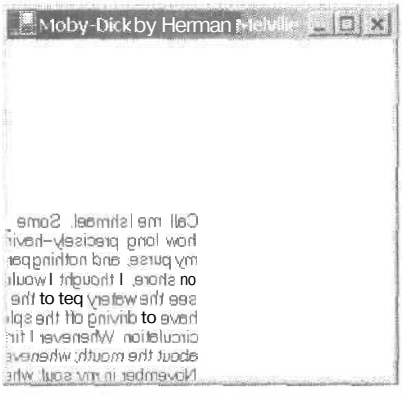
Смещая текст, можно продемонстрировать другие методики. Вставьте в программу строку:

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

В результате текст начнется в центре клиентской области или страницы принтера, что само по себе не очень интересно. А теперь вставьте такой вызов *TranslateTransform:*

```
grfx.ScaleTransform(-1, 1);
```

Теперь все стало намного интереснее, правда? А случилось вот что: текст был отражен по вертикальной оси и теперь выводится в зеркально отраженном виде в левой нижней четверти клиентской области:



А теперь замените вызов *ScaleTransform* следующим:

```
grfx.ScaleTransform(1, -1);
```

Заметьте, что текст отражается по горизонтальной оси и кажется перевернутым. Эти два эффекта также можно комбинировать:

```
grfx.ScaleTransform(-1, -1);
```

Теперь ясно, почему нельзя просто вызвать метод *ScaleTransform* с отрицательными аргументами: текст выйдет за пределы видимой части клиентской области. Чтобы наблюдать эффект, нужно отодвинуть текст подальше от левого и верхнего краев клиентской области.

Ладно, а теперь поэкспериментируем с порядком вызова *TranslateTransform* и *ScaleTransform*:

```
grfx.ScaleTransform(-1, 1);
grfx.TranslateTransform(cx / 2, cy / 2);
```

Сейчас в клиентской области ничего не видно. Наверное, вы догадались, что текст почему-то переместился за пределы клиентской области. Есть два способа вернуть его назад. Первый — сделать первый аргумент *TranslateTransform* отрицательным:

```
grfx.ScaleTransform(-1, 1);
grfx.TranslateTransform(-cx / 2, cy / 2);
```

Теперь текст вновь выводится в центре клиентской области, отраженный по вертикальной оси. Между прочим, я вовсе не надеюсь, что вы сейчас поймете, почему это работает. Действительно, небольшая путаница была бы сейчас кстати.

Чтобы еще больше запутать вас, скажу, что есть и другой способ решения этой задачи. Оставьте первый аргумент как есть, но вызовите перегруженный метод *TranslateTransform*:

```
grfx.ScaleTransform(-1, 1);
grfx.TranslateTransform(cx / 2, cy / 2, MatrixOrder.Append);
```

Все три рассмотренных выше метода (*RotateTransform*, *ScaleTransform* и *TranslateTransform*) являются перегруженными, чтобы можно было применить последний аргумент — *MatrixOrder*. Это перечисление, определяемое в пространстве имен *System.Drawing.Drawing2D* (именно поэтому я для удобства поместил в начале программы MobyDick дополнительный оператор *using*).

Вот формальные определения методов класса *Graphics*, как уже обсуждавшихся, так и одного нового:

Методы *Graphics* (выборочно)

```
void TranslateTransform(float dx, float dy)
void TranslateTransform(float dx, float dy, MatrixOrder mo)
void ScaleTransform(float sx, float sy)
void ScaleTransform(float sx, float sy, MatrixOrder mo)
void RotateTransform(float fAngle)
void RotateTransform(float fAngle, MatrixOrder mo)
void ResetTransform()
```

Метод *ResetTransform* возвращает все в исходное состояние. Перечисление *MatrixOrder* состоит всего из двух членов:

Перечисление *MatrixOrder*

Член	Значение	Описание
Prepend	0	Задано по умолчанию,
Append	1	Меняет порядок применения преобразований.

До конца этой главы функции значений этого перечисления станут ясны.

Глобальное преобразование

Координаты, передаваемые различным методам рисования объекта *Graphics*, называются *глобальными координатами* (world coordinates). Глобальные координаты первыми подвергаются глобальному преобразованию, с которым мы уже немного поэкспериментировали, вызывая методы *TranslateTransform*, *ScaleTransform* и *RotateTransform*. Чуть ниже я попытаюсь формализовать понятие глобального преобразования.

Глобальное преобразование трансформирует глобальные координаты в координаты на странице. Преобразование страницы — это преобразование, определенное свойствами *PageUnit* и *PageScale* объекта *Graphics*. Оно преобразует координаты на странице в координаты на устройстве, исчисляемые в пикселах относительно верхнего левого угла области печати страницы принтера.



В случае таких методов, как *MeasureString* класса *Graphics* или метод *GetHeight* класса *Font*, это процесс выполняется в обратном порядке. Координаты на устройстве преобразуются в координаты на странице, которые затем преобразуются в глобальные координаты и возвращаются методом.

Линейные преобразования

Рассмотрим действие различных методов преобразования с математической точки зрения. По-видимому, простейшим из этих методов является;

```
grfx.TranslateTransform(dx, dy);
```

где символы d_x и d_y обозначают координаты (*d* — это *дельта*, или *приращение* в математической символике). Глобальное преобразование, осуществляемое при вызове этого метода, можно представить так:

$$x_{page} = x_{world} + d_x$$
$$y_{page} = y_{world} + d_y$$

Все довольно просто. Как видите, вызов *TranslateTransform* приводит к смещению всех координат.

А вот похожий вызов *ScaleTransform*:

```
grfx.ScaleTransform(sx, sy);
```

Здесь s означает *масштаб* (scale), а глобальное преобразование осуществляется путем умножения, а не сложения:

$$x_{page} = s_x \cdot x_{world}$$
$$y_{page} = s_y \cdot y_{world}$$

Действие масштабирования очень похоже на преобразование страницы.

Не буду заставлять вас гадать, что происходит при вызове:

```
grfx.RotateTransform(α);
```

с заданным углом a. Результирующее преобразование несколько сложнее и выглядит так:

$$x_{page} = x_{world} \cdot \cos(\alpha) + y_{world} \cdot \sin(\alpha)$$
$$y_{page} = -x_{world} \cdot \sin(\alpha) + y_{world} \cdot \cos(\alpha)$$

Эта небольшая таблица синусов и косинусов поможет убедиться, что эти формулы действительно работают:

Угол a	Синус	Косинус
0	0	1
45	$\sqrt{2}$	$\sqrt{2}$
90	1	0
135	$\sqrt{2}$	$-\sqrt{2}$
180	0	-1
225	$-\sqrt{2}$	$-\sqrt{2}$
270	-1	0
315	$-\sqrt{2}$	$\sqrt{2}$
360	0	1

Кстати, если вы уже встречались с вращением во время работы в других средах программирования графики: эти две формулы вращения могут отличаться от виденных вами раньше. Дело в том, что GDI+ осуществляет вращение по часовой стрелке. В более математически ориентированных средах вращение выполняется против часовой стрелки. В этом случае член первой формулы, содержащий синус, является отрицательным, а аналогичный член второй формулы — положительным.

Все три вышеперечисленных преобразования можно обобщить двумя формулами:

$$x_{page} = s_x \cdot x_{world} + r_x \cdot y_{world} + d_x$$
$$y_{page} = r_y \cdot x_{world} + s_y \cdot y_{world} + d_y$$

где s_x, s_y, r_x, r_y, d_x и d_y — константы, определяющие некоторое преобразование, С коэффициентами масштабирования s_x и s_y, как и со значениями смещения d_x и d_y вы уже знакомы. Вы могли заметить, что некоторые комбинации s_x, s_y, r_x и r_y — комбинации, определяемые тригонометрическими функциями определенных углов, — задают вращение. У факторов r_x и r_y есть собственный смысл, их действие на графику мы скоро выясним.

Вместе эти две формулы называются *«общее линейное преобразование плоскости».*[2] Хотя x и y одновременно являются функциями x_{world} и y_{world}, в формулах линейного преобразования они не подвергаются возведению в степень или чему-то подобному. Линейность глобального преобразования предполагает существование у него некоторых ограничений:

- при глобальном преобразовании прямые всегда преобразуются в прямые и никогда — в кривые;
- параллельные линии никогда не становятся сходящимися;
- пара объектов с равными размерами никогда не будет преобразована в пару объектов, размеры которых не равны;
- параллелограммы (включая прямоугольники) всегда преобразуются в параллелограммы, а эллипсы — Б эллипсы.

Преобразование, действующее на входе в событие *Paint* или *PrintPage* нового, еще не измененного класса *Graphics,* называется *единичным*: значения коэффициентов s_x и s, этого преобразования равны 1, а остальные параметры — 0. Метод *ResetTransform* возвращает объект *Graphics* к использованию единичного преобразования.

Как вы уже видели, действие последовательных вызовов *TranslateTransform*, *ScaleTransform* и *RotateTransform* суммируется. Однако результирующее глобальное преобразование может быть различным в зависимости от порядка вызова этих методов. Причину этого продемонстрировать несложно, но я немного боюсь, поэтому не обижусь, если вы закроете глаза и пропустите самое страшное.

Во-первых, предположим, что имеется глобальное преобразование, назовем его T_1:

$$rf = s_{x1} \cdot x + r_{x1} \cdot y_s + d_{x1}$$
$$y_s' = r_{y1} \cdot x + s_{y1} \cdot y + d_{y1}$$

[2] Строгое изложение математического подхода см. в разделе 3-7 главы 3 книги Anthony J. Pettofrezzo, *Matrices and Transformations* (New York: Dover, 1978).

Чтобы не использовать индексы для обозначения глобальных координат и координат на странице, я обозначу первые просто x и y, а вторые — x' и y'. Введем второе преобразование T_2 с другими коэффициентами:

$$x' = s_{x2} \cdot x + r_{x2} \cdot y + d_{x2}$$
$$y' = r_{y2} \cdot x + s_{y2} \cdot y + d_{y2}$$

Применив к глобальным координатам преобразование T_1, а к его результату — преобразование T_2, получаем следующее преобразование:

$$x' = s_{x2} \cdot s_{x1} \cdot x + s_{x2} \cdot r_{x1} \cdot y + s_{x2} \cdot d_{x1} + r_{x2} \cdot r_{y1} \cdot x + r_{x2} \cdot s_{y1} \cdot y + r_{x2} \cdot d_{y1} + d_{x2}$$
$$y' = r_{y2} \cdot s_{x1} \cdot x + r_{y2} \cdot r_{x1} \cdot y + r_{y2} \cdot d_{x1} + s_{y2} \cdot r_{y1} \cdot x + s_{y2} \cdot s_{y1} \cdot y + s_{y2} \cdot d_{y1} + d_{y2}$$

После приведения подобных получаем:

$$x' = (s_{x2} \cdot s_{x1} + r_{x2} \cdot r_{y1}) \cdot x + (s_{x2} \cdot r_{x1} + r_{x2} \cdot s_{y1}) \cdot y + (s_{x2} \cdot d_{x1} + r_{x2} \cdot d_{y1} + d_{x2})$$
$$y' = (r_{y2} \cdot s_{x1} + s_{y2} \cdot r_{y1}) \cdot x + (r_{y2} \cdot r_{x1} + s_{y2} \cdot s_{y1}) \cdot y + (r_{y2} \cdot d_{x1} + s_{y2} \cdot d_{y1} + d_{y2})$$

Если применить сначала T_2, а потом T_1, результат будет другим:

$$x' = s_{x1} \cdot s_{x2} \cdot x + s_{x1} \cdot r_{x2} \cdot y + s_{x1} \cdot d_{x2} + r_{x1} \cdot r_{y2} \cdot x + r_{x1} \cdot s_{y2} \cdot y + r_{x1} \cdot d_{y2} + d_{x1}$$
$$y' = r_{y1} \cdot s_{x2} \cdot x + r_{y1} \cdot r_{x2} \cdot y + r_{y1} \cdot d_{x2} + s_{y1} \cdot r_{y2} \cdot x + s_{y1} \cdot s_{y2} \cdot y + s_{y1} \cdot d_{y2} + d_{y1}$$

После приведения подобных получаем:

$$x' = (s_{x1} \cdot s_{x2} + r_{x1} \cdot r_{y2}) \cdot x + (s_{x1} \cdot r_{x2} + r_{x1} \cdot s_{y2}) \cdot y + (s_{x1} \cdot d_{x2} + r_{x1} \cdot d_{y2} + d_{x1})$$
$$y' = (r_{y1} \cdot s_{x2} + s_{y1} \cdot r_{y2}) \cdot x + (r_{y1} \cdot r_{x2} + s_{y1} \cdot s_{y2}) \cdot y + (r_{y1} \cdot d_{x2} + s_{y1} \cdot d_{y2} + d_{y1})$$

В этом-то, друзья мои, и причина разных результатов в зависимости о того, какой из методов — *ScaleTransform* или *TranslateTransform* — будет вызван первым.

Немного о матрицах

Для математического подхода к решению запутанных задач (например, похожих на только что показанную) характерно не упрощение условий путем удаления их части, а наоборот, их усложнение путем ввода новых понятий. Здесь весьма полезно ввести понятие *матрицы,* хотя бы потому, что матричная алгебра достаточно известна (по крайней мере математикам). Линейное преобразование может быть представлено в виде матрицы, а применение нескольких преобразований эквивалентно умножению соответствующих матриц.

Матрица — это прямоугольный массив чисел. Вот массив, состоящий из трех столбцов и двух строк:

$$\begin{vmatrix} 27 & 9 & 14 \\ 3 & 0 & 88 \end{vmatrix}$$

Матрицы обычно обозначают заглавными буквами. Умножение матриц можно записать так:

$$A \times B = C$$

Число столбцов матрицы *A* должно равняться числу строк матрицы *B,* а число строк произведения *C* — таковому матрицы Л. Число столбцов в матрице *C* равно числу столбцов матрицы *B.* Число в i-й строке *j*-го столбца матрицы *C* равно сумме произведений чисел 2-й строки матрицы *A* на соответствующие числа *j*-го столбца

матрицы B.[1] Умножение матриц не коммутативно: произведение A×B не обязательно равняется произведению B×A.

Если бы мы не имели дело со смещением, глобальные координаты (x, y) можно было представить в виде матрицы 1X2, а матрицу преобразования — как матрицу 2x2. Эти матрицы следует перемножить и выразить результирующие координаты на странице (x', y') в виде другой матрицы 1X2:

$$\begin{vmatrix} x & y \end{vmatrix} \times \begin{vmatrix} s_x & r_y \\ r_x & s_y \end{vmatrix} = \begin{vmatrix} x' & y' \end{vmatrix}$$

Применение правил умножения матриц дает формулы:

$x' = s_x \cdot x + r_x \cdot y$
$y' = r_y \cdot x + s_y \cdot y$

Но это не совсем полные формулы. Глобальное преобразование также включает значения смещения. Чтобы умножение матриц было корректным, нужно расширить матрицы глобальных координат и координат на странице до размера 1×3, а преобразование будет представлять собой матрицу 3X3:

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} s_x & r_y & 0 \\ r_x & s_y & 0 \\ d_x & d_y & 1 \end{vmatrix} = \begin{vmatrix} x' & y' & 1 \end{vmatrix}$$

А вот результирующие формулы:

$x' = s_x \cdot x + r_x \cdot y + d_x$
$y' = r_y \cdot x + s_y \cdot y + d_y$

Тип преобразований, который можно представить подобной матрицей, часто называют *матричным преобразованием*. У матричного преобразования, не изменяющего преобразуемый объект, коэффициент масштабирования равен 1, а параметры r И d — 0:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Такая матрица называется *единичной*.

Класс Matrix

Матричное преобразование инкапсулировано в классе *Matrix*, определенном в пространстве имен *System.Drawing.Drawing2D*. Объект *Matrix* может быть создан с помощью одного из четырех конструкторов, два из которых таковы:

Конструкторы *Matrix* (выборочно)

```
Matrix()
Matrix(float sx, float ry, float rx, float sy, float dx, float dy)
```

[1] См. примеры из разделов 1-2 книги Pettofrezzo.

Второй конструктор позволяет задавать все шесть констант, определяющих матричное преобразование. Коэффициенты масштабирования *sx* и *sy* должны быть отличными от 0, иначе вы получите исключение!

У класса *Graphics* есть изменяемое свойство *Transform*, куда заносится объект *Matrix*:

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ
Matrix	*Transform*	Чтение/запись

Любой вызов методов *TranslateTransform*, *ScaleTransform*, *RotateTransform* и *ResetTransform* действует на свойство *Transform*. Это свойство можно установить и напрямую. Вызов:

```
grfx.Transform = new Hatrix(1, 0, 0, 1, O, O);
```

действует, как метод *ResetTransform*.

У класса *Matrix* 5 свойств, и все неизменяемые:

Свойства ***Matrix***

Тип	Свойство	Доступ	Описание
float[]	*Elements*	Чтение	Шесть констант преобразования.
float	*OffsetX*	Чтение	Константа d_x.
float	*OffsetY*	Чтение	Константа d_y.
bool	*Isldentity*	Чтение	Показывает, является ли матрица единичной,
bool	*Islnvertible*	Чтение	Показывает, обратима ли матрица.

А теперь рассмотрим пример объединенных преобразований. Пусть сначала был вызван метод:

```
grfx.ScaleTransform(2, 2);
```

После этого программа может изучить результирующую матрицу, вызвав:

```
float[] afElements = grfx.Transform.Elements;
```

В результате вы увидите значения 2, 0, 0, 2, O, O, которые можно представить матрицей:

$$\begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ O & 0 & 1 \end{vmatrix}$$

Теперь вызовем:

```
grfx.TranslateTransform(100, 100);
```

Если вызвать его отдельно, этот метод даст такую матрицу:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & 100 & 1 \end{vmatrix}$$

Однако новое преобразование — это объединенный результат вызова двух методов. Матрица, определяемая вторым методом, умножается на текущее значение свойства *Transform* (определенное первым методом), в итоге значение этого свойства изменится:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & 100 & 1 \end{vmatrix} \times \begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 200 & 200 & 1 \end{vmatrix}$$

А теперь вызовем методы *ScaleTransform* и *TranslateTransform* в обратном порядке:

```
grfx.TranslateTransform(100, 100);
grfx.ScaleTransform(2, 2);
```

И в этом случае результирующее преобразование рассчитывается путем умножения второй матрицы на первую:

$$\begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & 100 & 1 \end{vmatrix} = \begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 100 & 100 & 1 \end{vmatrix}$$

Это преобразование также можно получить, вызвав:

```
grfx.ScaleTransform(2, 2);
grfx.TranslateTransform(100, 100, MatrixOrder.Append);
```

Аргумент *MatrixOrder.Append* указывает, что следует объединить новое преобразование с существующим. По умолчанию задан аргумент *MatrixOrder.Prepend*.

У класса *Graphics* есть еще один метод преобразования глобальных координат:

Методы *Graphics* (выборочно)

```
void MultiplyTransform(Matrix mat)
void MultiplyTransform(Matrix mat, MatrixOrder mo)
```

Этот метод позволяет множить существующую матрицу на новую.

В главе 15 мы продолжим обсуждение *класса Matrix*.

Сдвиг и подобные ему преобразования

Вернемся к программе MobyDick и вставим в нее оператор:

```
grfx.Transform = new Matrix(1, 0, 0, 3, 0, 0);
```

Он действует аналогично такому:

```
grfx.ScaleTransform(1, 3);
```

С коэффициентами r_x и r_y мы пока не экспериментировали отдельно, поэтому рассмотрим такой вызов:

```
grfx.Transform = new Matrix(1, 0, 0.5f, 1, 0, 0);
```

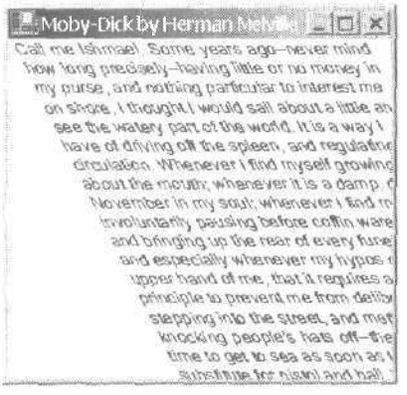
Он дает следующую матрицу преобразования:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

А вот соответствующие формулы преобразования:

$$x' = x + 0.5 \cdot y$$
$$y' = y$$

Заметьте, что значения координат по оси *x* увеличиваются в зависимости от значения *y*. Если *y* равен 0 (верхний край клиентской области), преобразования не будет. По мере возрастания значения *y* по направлению к нижнему краю клиентской области *x* также *возрастает*. В результате возникает эффект, называемый *сдвигом* (shear),



В частности, показанный здесь эффект называется *горизонтальный сдвиг* или *x-сдвиг*. К сожалению, слово *shear* (сдвиг) начинается с той же буквы, что и *scale* (масштабирование), поэтому коэффициенты сдвига в формулах преобразования я решил обозначить последней буквой слова *shear*.

Также можно задавать коэффициент *вертикального сдвига* или *y-сдвига*:

```
grfx.Transform = new Matrix(1, 0.5f, 0, 1, 0, 0);
```

В результате получится такая матрица:

$$\begin{vmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Формулы преобразования в этом случае примут следующий вид:

$$x' = x$$
$$y' = 0.5 \cdot x + y$$

Заметьте: все строки текста по-прежнему начинаются у левого края клиентской области:



Вообще-то вращение — это комбинация горизонтального и вертикального сдвигов. Однако такие комбинации, как эта, работать не будут:

```
grfx.Transform = new Matrix(1, 1, 1, 1, 0, 0);
```

Эта комбинация определяет преобразование:

$$x' = x + y$$
$$y' = x + y$$

Такое преобразование вызвало бы сжатие изображения в линию, но прежде оно генерирует исключение. Но такой вызов работает:

```
grfx.Transform = new Matrix(1, 1, -1, 1, 0, 0);
```

и производит такой эффект:



Если присвоить первым четырем аргументам значение корня квадратного из 1/2:

```
grfx.Transform = new Matrix(0.707f, 0.707f, -0.707f, 0.707f, 0, 0);
```

эффект будет тот же, что и у вызова, с которого мы начали исследование преобразований:

```
grfx.RotateTransform(45);
```

Комбинирование преобразований

Теоретически преобразовывать страницу не нужно. Преобразование лишь масштабирует изображение, что вполне можно сделать при глобальном преобразовании. Однако часто удобно объединить два преобразования, особенно если требуется нарисовать фигуры определенного размера, которые затем подвергаются глобальному преобразованию.

Эта программа рисует 36 дюймовых квадратов, повернутых вокруг центра области экрана.

HotateclHectangles.cs

```
//-------------------------------------------------------------------- v
// RotatedRectangles.cs © 2001 by Charles Petzold
//--------------------------------------------------------------------
using System;
«sing System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class RotatedRectangles: PrintableForm
{
     public new static void Main()
     {
          Application.flun(nexRotatedRectangles());
     }
     public RotatedRectangles()
     {
          Text * "Rotated flectangles";
     )
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen      pen   = new Pen(clr);
          grfx.PageUnit  = GraphicsUnit.Pixel;
          PointF[] aptf  = { (PointF) grfx.VisibleClipBounds.Size };
          grfx.PageUnit  * GraphicsUnit.Inch;
          grfx.PageScale = 0.01f;

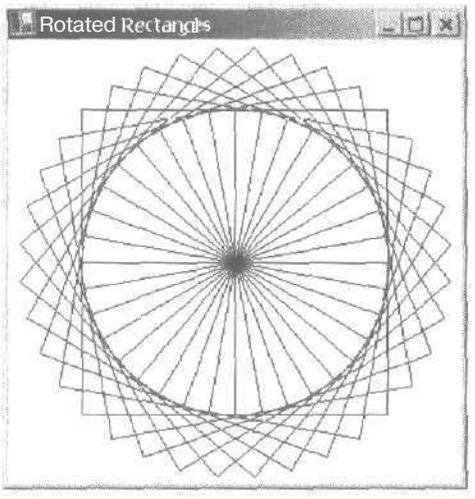
          grfx.TransformPoints(CoordinateSpace.Page,
                               CoordinateSpace.Device, aptf);

          grfx.TranslateTransform(aptf[0].X / 2, aptf[0].Y / 2);

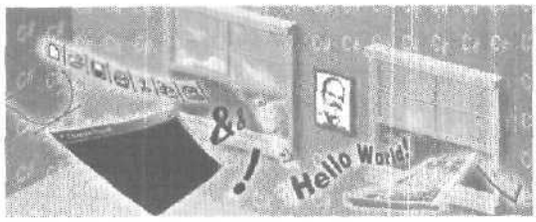
          for (int 1 =0; i < 36; i++)
          {
               grfx.DrawRectangle(pen, 0, 0, 100, 100);
               grfx.RotateTransform(10);
          }
     }
}
```

Самое сложное здесь — расчет аргументов метода *TranslateTransform* для сдвига начала глобальных координат в центр области экрана. Метод *OnPage* задает измерения страницы в пикселах, чтобы получить значение свойства *VisibleClipBounds* в пикселах. После этого *OnPage* устанавливает измерение страницы в единицах 1/100 дюйма, затем преобразует ширину и высоту области экрана к координатам страницы. Метод *TranslateTransform* использует эти значения вполовину уменьшенными,

Самое простое в этой программе — цикл *for*: он рисует прямоугольник шириной в 100 единиц, расположенный сверху в точке (0, 0). После этого метод *RotateTransform* поворачивает его на 10°, готовясь к следующему обороту цикла. Вот результат:



Умение вращать объекты вокруг начала координат пригодится нам в главе 10 — в программе, рисующей аналоговые часы,



ГЛАВА

8

Приручение мыши

Изобретение, на которое был выдан патент США за номером 3 541 541 от 21 июня 1967 г., описано как «двухкоординатный указатель позиции на дисплейных системах».[1] Изобретателем значится некто Дуглас Энгельбарт, сотрудник Стэнфордского научно-исследовательского института. Конечно же, нигде в оригинальном патенте не упоминается слово *мышь,* но ясно как день, что патент описывает именно ее.

Дуглас Энгельбарт (род. в 1925 г.) основал в Стэнфордском институте исследовательский центр, где занимались совершенствованием аппаратного и программного обеспечения компьютеров, преследуя весьма амбициозную целы создать инструменты, помогающие работе человеческого интеллекта. В 1986 г. Энгельбарт будет вспоминать: «Мы хотели начать эксперименты с выбором объектов на экране. Для активной работы во взаимодействии с дисплеем нужно сообщить компьютеру, на чем сосредоточено внимание пользователя в данный момент, поэтому мы испытывали потребность в устройстве для выбора объектов на экране. Было множество доводов в пользу световых перьев и трэкболов, но ни одно из этих устройств не соответствовало нашим задачам безоговорочно. Я хотел найти наилучшее решение для работы с любыми объектами, нужными в данный момент — с текстом, структурными элементами, интерактивными командами... Недавно я откопал в своих бумагах описание прототипа, ставшего в конечном счете самой первой мышью...»[2]

---

[1] Патенты США доступны для просмотра по адресу *http://www.uspto.gov/pat*Для просмотра патентов, выданных после 1976 г., потребуется программа для просмотра TIFF-файлов.

[2] Адель Голдберг (Adele Goldberg), *A History of Personal Workstations* (New York: ACM Press, 1988), с. 194–195. Эта книга представляет собой сборник материалов конференции ACM (Association for Computing Machinery), посвященной истории развития персональных рабочих станций, проводившейся 9-Ю января 1986 г. в калифорнийском городе Пало-Альто. Всестороннее обсуждение мыши см. в книге Тьери Бардини (Thierry Bardini. *Bootstrapping: Douglas Engelbart, Coevolution, and the Origins of the Personal Computer,* Stanford, CA: Stanford University Press. 2000).

К 1972 г. мышь пробралась в исследовательский центр компании Xerox в Пало Альто (PARC), где стала частью компьютера Alto, который до сих пор считают первой машиной с графическим интерфейсом пользователя и предшественником ПК Но лишь после выпуска компанией Apple в 1983 г. компьютера Apple Lisa (не очень удачного) и, годом позже, модели Macintosh (которая имела больший успех), мышь стала обязательным аксессуаром любого нормально оснащенного компьютера,

Хотя для ввода алфавитно-цифровой информации и ограниченного передвижения *курсора* вполне достаточно клавиатуры, мышь обеспечивает более тесное взаимодействие между пользователем и объектами на экране. Мышь — это продолжение пальцев пользователя, она позволяет ему выбирать, брать и перемеш ать объекты. Кроме того, она неплохо приспособлена для работы с новыми типами приложений. Хотя первыми испытателями мыши стали компьютерные игроки и художники, в последнее время мышь зарекомендовала себя как бесценное средство для навигации в гипертекстовых средах, таких как Web,

Темная сторона мыши

В 1985 г., когда вышла первая версия Microsoft Windows, мышь еще была довольно редким зверем в мире IBM-совместимых компьютеров. Создатели первых версий Windows не хотели делать покупку мыши непременным условием работы с их продуктом, поэтому мышь сделали необязательным устройством для работы Windows. Во всех маленьких Windows-программах функции, выполняемые с помощью мыши, были снабжены клавиатурными эквивалентами (они остались до сих пор: заглянув в справку по Калькулятору Windows, вы увидите, что каждой из его кнопок скрупулезно назначен клавиатурный эквивалент). Вслед за Microsoft и другие разработчики программ стали предоставлять в своих программах интерфейс, использующий клавиатуру.

Хотя найти компьютер без мыши теперь трудно, Windows получила в наследство возможность работы в системах без мыши. Я сам сторонник идеи предоставления клавиатурных эквивалентов действий программ, выполняемых при помощи мыши (если это возможно в принципе). Тот, кто печатает вслепую, предпочитает вовсе не убирать рук с клавиатуры. Думаю, аналогичная привычка есть у каждого, кто «терял» мышь на столе, заваленном всякой всячиной, или у кого забарахлила мышь, забитая грязью. Клавиатурные эквиваленты обычно не требуют от программиста больших усилий мысли и затрат энергии, но расширяют функциональность его программы для пользователей, предпочитающих работать с клавиатурой.

Есть ряд веских причин, по которым клавиатурные эквиваленты следует считать необходимыми любому приложению для Windows. Большая часть этих причин касается эргономики, С увеличением среднего возраста пользователей компьютеров некоторым из них, включая меня, пришлось страдать от мучительных изматывающих болей в запястье и во всей руке — прямого результата чрезмерного использования мыши. Иногда даже можно точно назвать приложение, ставшее причиной заболевания. Мне пришлось потратить достаточно усилий на поиск клавиатурных эквивалентов в программах, с которыми я работаю. Разочарование — неизменный результат работы с программой, чьи создатели, по-видимому, просто махнули рукой на реализацию гармоничного пользовательского интерфейса в своих творениях.

Игнорируемая мышь

С главы 2 и до сих пор мы писали программы, реагирующие на действия мыши. Стандартная форма поддерживает интерфейс, позволяющий при помощи мыши перемещать форму, «зацепив» ее заголовок: изменять размеры формы, перетаскивая ее границы; входить в управляющее поле (также известное как системное меню), выбирать элементы и нажимать кнопки максимизации, минимизации и закрытия окна. Все это происходит без малейших усилий с вашей (программиста) стороны. Очевидно, в этом случае Windows самостоятельно обрабатывает информацию, вводимую при помощи мыши.

Как стало ясно в главе 4, программа Windows Forms, использующая полосы прокрутки, может не беспокоиться о мыши: код полосы прокрутки самостоятельно обрабатывает события мыши и адекватно реагирует на них.

В главе 12 я начну рассказ о многочисленных готовых элементах управления, доступных в Windows Forms, а потом речь пойдет о меню и диалоговых окнах. Вы узнаете, что все эти средства улучшения пользовательского интерфейса сами обрабатывают информацию, вводимую мышью. Действительно, главное назначение элементов управления — инкапсуляция низкоуровневого интерфейса клавиатуры и мыши и предоставление интерфейса более высокого уровня, с которым программист может иметь дело.

В этой главе мы рассмотрим ряд ситуаций, требующих напрямую обрабатывать вводимую мышью информацию, что, конечно, нужно далеко не каждому приложению. Тот, кто планирует украшать клиентскую область своих программ лишь готовыми элементами управления, возможно, никогда не столкнется с прямой обработкой активности мыши. Но тому, кто собрался писать собственные элементы управления, солидная подготовка в этой области необходима.

Несколько быстрых определений

Мышь — это указательное устройство с одной или несколькими кнопками. Собственно *мышь* — это предмет, лежащий на вашем столе. Когда вы двигаете мышь, среда Windows передвигает по экрану небольшую растровую картинку — *курсор* мыши (в некоторых графических средах и даже в части документации по Windows Forms его называют *указателем*).

У курсора мыши есть *горячая точка* (hot spot) — пиксел, соответствующий положению курсора на экране. Например, горячая точка стандартного курсора в виде стрелки находится на кончике стрелки. Когда говорят о *позиции* курсора мыши, имеют в виду позицию горячей точки. Надеюсь, вы не слишком встревожитесь, если иногда, отступив от правил, я скажу «позиция *мыши*», а не «позиция *курсора*». Не бойтесь: мышь все еще у вас на столе, а не бегает по экрану.

Щелкнуть мышью значит нажать и отпустить ее кнопку. *Тащить* мышь — передвигать мышь, держа нажатой ее левую кнопку. *Двойной щелчок* — это два щелчка, выполненных подряд. Чтобы действие было квалифицировано как двойной щелчок, оба щелчка должны произойти в течение заданного промежутка времени, причем положение курсора на экране за это время не должно существенно измениться. Если понадобится узнать соответствующие значения (что маловероятно), можно обратиться к статическим неизменяемым свойствам класса *SystemInformation:*

Статические свойства *SystemInformation* (выборочно)

Тип	Свойство	Доступ	Описание
int	*DoubleClickTime*	Чтение	Время в миллисекундах
Size	*DoubleClickSize*	Чтение	Размер области двойного щелчка в пикселах

Пользователь может регулировать эти параметры при помощи значка Mouse (Мышь) на Панели управления.

Сведения о мыши

Можно ли работать на компьютере без мыши? Давайте попробуем. Выключите компьютер, отключите мышь, затем снова включите компьютер и наблюдайте за происходящим. Похоже, Windows 2000 ни на что не жалуется. Нажав Ctrl+Esc (или клавишу Windows на некоторых клавиатурах), можно вызвать меню Start (Пуск) и при помощи клавиш управления курсором просматривать списки программ, документов, избранного.

Программе Windows Forms может потребоваться определить, установлена ли мышь и. если да, сколько у нее кнопок. И здесь на помощь приходит класс *System-Information*:

Статические свойства *SystemInformation* (выборочно)

Тип	Свойство	Доступ	Описание
bool	*MousePresent*	Чтение	Указывает, установлена ли мышь.
int	*MouseButtons*	Чтение	Указывает число кнопок мыши.
bool	*MouseButtonsSwapped*	Чтение	Указывает, используется ли правая кнопка как левая и наоборот.

Если мышь установлена, свойство *MousePresent* возвращает *true,* а *MouseButtons* указывает число кнопок мыши. Обычно *MouseButtons* возвращает число кнопок от одной до пяти, но в настоящее время на машинах с Windows чаще всего встречаются мыши с двумя или тремя кнопками.

Свойство *MouseButtonsSwapped* возвращает *true,* если пользователь поменял функции правой и левой кнопок мыши, щелкнув значок Mouse (Мышь) на Панели управления. Обычно так делают пользователи-левши, располагающие мышь слева от клавиатуры и желающие выполнять самые распространенные действия мыши при помощи указательного пальца левой руки.

Обычно программисту не обязательно знать об обмене функциями между правой и левой кнопками мыши. Но, допустим, нужно написать программу для компьютерного обучения с анимацией, показывающей нажатые кнопки мыши. В этом случае неплохо было бы порадовать пользователя, показывая кнопки именно в той конфигурации, которую он выбрал. (Конечно же, ничто не мешает пользователю просто положить мышь слева от клавиатуры, не меняя функции кнопок — я специально так делал, чтобы поменьше работать с мышью одной рукой, — или, поменяв функции кнопок, продолжать работать с мышью правой рукой),

Менялись функции кнопок или нет, *главной* считается та, что называется *левой*. Она выполняет самые распространенные действия: выбор предметов, перетаскивание значков и инициирование действий.

Правая кнопка предназначена для вызова *контекстного меню*. Оно появляется там, где находится курсор мыши, и набор его функций соответствует возможностям, доступным в этой области. Например, если в Internet Explorer расположить курсор мыши над рисунком Web-страницы и нажать правую кнопку мыши, то появится меню с рядом функций, среди которых будет команда сохранения рисунка в файле. Если мышь располагается не над рисунком, а над другой областью страницы, команда сохранения рисунка исчезнет, зато появится команда печати страницы. Способы создания контекстных меню мы обсудим в главе 14.

Колесико мыши

«Сделай мышеловку лучше, чем у других, и люди проложат тропу к дверям твоего дома»,— говорила мне мать, перефразируя Р. Эмерсона, сама того не зная/ Но в наши дни больше толку будет, если сделать не мышеловку, а лучшую *мышь*.

Трехкнопочная мышь не отличалась особой популярностью у пользователей Windows, пока Microsoft не выпустила IntelliMouse. Хотя эта мышь не отличается интеллектом в прямом смысле этого слова, у нее есть одно реальное улучшение в виде небольшого колесика, расположенного между кнопок. При нажатии на это колесико оно функционирует как третья кнопка (в интерфейсах программирования она называется *средней*), кроме того, его можно крутить, и программы, поддерживающие эту функцию, будут реагировать прокруткой или масштабированием документа.

Хотя на первый взгляд это новшество совершенно бесполезно, оказалось, что к нему быстро привыкают, особенно если приходится часто читать длинные документы или Web-страницы. Большое преимущество в том, что, чтобы прокручивать текст, нет нужды постоянно держать курсор мыши над полосой прокрутки — он в это время может быть в любом месте документа,

И в этом случае информацию о колесике предоставляет класс *SystemInformation,*

Статические свойства *SystemInformation* (выборочно)

Тип	Свойство	Доступ	Описание
bool	*MouseWheelPresent*	Чтение	Возвращает *true,* если имеется колесико.
int	*MouseWheelScrollLines*	Чтение	Число строк, прокручиваемых за оборот колесика.
bool	*NativeMouseWheelSupport*	Чтение	Не имеет значения доя приложений.

Колесико вращается не плавно — чувствуется явное пощелкивание. Чтобы приложения правильно реагировали на действия колесика, предполагается, что при каждом таком щелчке (в документации по.NET Framework он называется *detent)*

[5] А может, и нет, поскольку полностью это высказывание звучит так: «Если человек пишет книги, проповедует или просто делает мышеловки лучше, чем другие, то даже если он построит свой дом в глухом лесу, мир все равно проложит тропу к его дверям». Считается, что эта цитата взята из лекции Эмерсона, но в его письменных сочинениях этого изречения нет. См. книгу Bartlett's Familiar Quotations, 16th ed. (Boston: Little, Brown, 1992), с. 430. Однако в наши дни мало кто полностью согласится с этим очаровательным мнением: для коммерческого успеха мышеловок, как и любого другого товара широкого потребления, нужна еще и хорошая маркетинговая политика.

приложение должно прокручивать определенное число строк текста. Свойство *MouseWheelScrollLines* указывает это число, для Microsoft IntelliMouse оно равно 3. Однако в будущем наверняка появятся более совершенные модели с большим разрешением колесика, тогда *MouseWheelScrollLmes* однажды может вернуть значение 2 или даже 1,

Если вам захочется добавить поддержку колесика к одной из программ SysInfo, написанных в главе 4, чтобы расширить функциональность полос прокрутки, — не беспокойтесь: полосы с автопрокруткой автоматически реагируют на активность колесика.

Четыре основных события мыши

Приложение Windows Forms получает сведения об активности мыши *в* виде событий. Класс *Control* определяет 9 событий мыши и 9 соответствующих защищенных методов. Все классы, порожденные от *Control* (включая *Form),* также наследуют эти 9 методов.

Хотя мы подробно обсудим элементы управления в главе 12, полезно было бы иметь какое-то представление о принципах взаимодействия мыши с элементами управления уже сейчас. Представьте себе форму или диалоговое окно с множеством кнопок, текстовых меток и полей. Эти элементы управления считаются *дочерними объектами* формы, а форма — *их родительским объектом (предком).* Мы уже встречались с такой нотацией в программе SysInfoPanel из главы 4, когда в свойство *Parent* элемента управления *Panel* заносился объект *Form.*

Любое событие мыши может быть получено только одним элементом управления. Элемент управления получает события мыши, только когда он активизирован (включен) и видим, т. е. его свойства *Enabled* и *Visible* равны *true.* Обычно события мыши получает только элемент управления, находящийся прямо под курсором мыши,

Если провести курсор над дочерним элементом управления, который активизирован и видим, то событие мыши получит именно этот дочерний объект, а не его родительский объект, Если дочерний элемент управления деактивизирован (отключен) или невидим, то событие мыши получает его родительский объект. При этом все происходит так, как если бы дочерний объект был прозрачным. Если несколько элементов расположены «стопкой» в одной и той же физической точке, то событие мыши достанется активизированному и видимому элементу управления, занимающему высшую позицию в *Z-порядке* (первому сверху в «стопке»), иначе говоря, тому, что будет виден сверху всех перекрывающихся элементов управления. С этим мы разберемся в главе 12.

Любой объект, являющийся потомком или экземпляром *Form,* получает события мыши, только когда курсор находится над клиентской областью формы. Объект *Form* не получает события мыши, если курсор находится над границей, заголовком, системным меню, управляющими кнопками окна или полосами прокрутки.

Однако, как я еще покажу, иногда элемент управления или форма получает события мыши, даже если курсор мыши *не* расположен над ним. Эта функция называется *захват* (capturing) мыши и помогает формам и элементам управления отслеживать движения мыши. Ниже я еще много буду говорить об этом.

Вот четыре основных события мыши:

События *Control* (выборочно)

Событие	Метод	Делегат	Аргумент
MouseDown	*OnMouseDown*	*MouseEventHandler*	*MouseEventArgs*
MouseUp	*OnMouseUp*	*MouseEventHandler*	*MouseEventArgs*
MouseMove	*OnMouseMove*	*MouseEventHandler*	*MouseEventArgs*
MouseWheel	*OnMouseWheel*	*MouseEventHandler*	*MouseEventArgs*

Как можно догадаться по их именам, события *MouseDown* и *MouseUp* говорят о том, что кнопка мыши нажата или отпущена. *MouseMove* сигнализирует о движении мыши, а *MouseWheel* происходит при вращении колесика мыши.

Только эти четыре события связаны с объектами типа *MouseEventArgs*. У класса *MouseEventArgs* пять неизменяемых свойств:

Свойства *MouseEventArgs*

Тип	Свойство	Доступ	Описание
int	*X*	Чтение	Позиция мыши по горизонтали
int	*Y*	Чтение	Позиция мыши по вертикали
MouseButtons	*Button*	Чтение	Кнопка (кнопки) мыши
int	*Clicks*	Чтение	В случае двойного щелчка возвращает 2
int	*Delta*	Чтение	Вращение колесика мыши

X и Y — целочисленные значения, указывающие координаты горячей точки курсора, исчисляемые в пикселах относительно верхнего левого угла клиентской области. Эта пара свойств действительна для всех четырех событий мыши.

Свойство *Button* указывает кнопку или кнопки, участвующие в данном событии. Это свойство не действительно для событий *MouseWheel*. В свойстзо *Button* заносится одно из значений перечисления *MouseButtons*:

Перечисление *MouseButtons*

Член	Значение
None	0x00000000
Left	0x00100000
Right	0x00200000
Middle	0x00400000
XButton1	0x00800000
XButton2	0x01000000

В этом перечислении *Left* следует понимать как главную кнопку, заданную пользователем, т. е. кнопку, которая вызывает меню приложений и позволяет изменять размеры форм и перемещать их. Свойство *Right* указывает кнопку, вызывающую контекстное меню. *XButton1* и *XButton2* — это кнопки мыши IntelliMouse Explorer, у которой пять кнопок.

Для событий *MouseDown* и *MouseUp* свойство *Button* указывает, какая именно кнопка нажата или отпущена.

Для события *MouseMove* свойство *Button* указывает кнопку или кнопки, нажатые *в текущий момент*. Заметьте, что значениями этих свойств являются битовые флаги, которые можно комбинировать. Скажем, если нажаты одновременно

левая и правая кнопки мыши, то значение свойства *Button* равно 0x00300000. Для объекта *MouseEventArgs*с именем *mea* следующее выражение вернет *true,* если нажата *только* правая кнопка мыши, и никаких других:

```
(mea.Button == MouseButtons.Right)
```

А это выражение всегда будет равно *true,* если нажата правая кнопка мыши, невзирая на другие кнопки:

```
(mea.Button & MouseButtons.Right 1= 0)
```

Свойство *Clicks* действительно лишь для событий *MouseDown* и обычно равно 1. Этому свойству задается 2, если за одним событием *MouseDown* достаточно быстро следует другое такое же, чтобы эта пара событий была квалифицирована как двойной щелчок.

Свойство *Delta* действительно лишь для событий *MouseWheel.* Если повернуть колесико мыши на один щелчок вперед, свойство *Delta* в типичном случае станет равно 120, а если на один щелчок назад, то ему будет присвоено -120.

Работа с колесиком

Давайте сразу разберемся с колесиком, чтобы потом спокойно сосредоточиться на более привычных аспектах работы с мышью. Абзацем выше я упоминал число 120. Это довольно редкий пример числа, которое очень важно для программирования Windows Forms (по крайней мере для обработки событий колесика мыши), но не связано ни с одним статическим свойством или значением перечисления. В заголовочных файлах Win32 идентификатор *WHEEL_DELTA* определен как 120. В программах Windows Forms, использующих колесико, придется жестко запрограммировать это значение или определить собственную переменную *const,*

Получив событие *MouseWheel,* надо рассчитать число строк для прокрутки, скажем, так:

```
mea.Delta * SystemInformation.MouseWheelScrollLines / 120
```

В данный момент результатом этого вычисления будет 3 или -3, но включение в расчет константы *SystemInformation* позволяет программе лучше приспособится к будущим устройствам с большим числом градаций вращения колесика. Положительные значения свидетельствуют о том. что пользователь вращает колесико вперед, на что программа должна ответить прокруткой текста к началу документа. Отрицательные значения говорят, что пользователь вращает колесико назад и программа должна прокрутить текст в направлении конца документа.

Следующая программа демонстрирует использование колесика мыши, выводя (и прокручивая) текст душещипательного стихотворения Эдгара Алана По «АннабельЛи»,

Заметьте: программа проверяет, есть ли колесико мыши, и сообщает пользователю, если не может найти таковое. Я поместил эту проверку в *Main,* но это не единственный способ проверки условий, в которых программа не должна работать. В качестве альтернативы можно переопределить метод *OnLoad* класса *Form* и выполнять такую проверку при загрузке программы. Событие *Load* происходит после исполнения кода конструктора, но до того, как форма становится видимой на экране. Если программа определила, что не должна запускаться, можно выве-

сти сообщение и вызвать метод *Close*, чтобы предотвратить появление ее: формы на экране. Во время исполнения конструктора формы прервать выполнение программы и предотвратить вывод формы на экран нельзя, так как в этот момент не работают ни *Close*, ни статический метод *ApplicationExit*.

PoePoem.cs

```
//---------------------------------
// PoePoem.cs © 2061 by Charles Petzold
//---------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PoePoem: Form
{
    const string strAnnabelLee *

          "It was raany and many a year ago,\n"                          +
          "   In a kingdom by the sea,\n"                                +
          "That a maiden there lived whom you may know\n"                +
          "   By the name of Annabel Lee;\x2014\n"                       *
          "And this maiden she lived with no other thought\n"            +
          " Than to love and be loved by me .\n"                         +
          "\n"                                                           +
          "I was a child and she was a child\n"                          +
          "   In this kingdom by the sea An"                             +
          "But we loved with a love that was more than love\x2014\n"     +
          "   I and ray Annabel Lee\x2014\n"                             +
          "With a love that the wingйd seraphs of Heaven\n"              +
          "   Coveted her and me.\n"                                     +
          "\n"                                                           +
          "And this was the reason that, long ago,\n"                    +
          "   In this kingdom by the sea,\n"                             +
          "A wind blew out of a cloud, chilling\n"                       +
          "   My beautiful Annabel Lee;\n"                               +
          "So that feer highborn kinsmen came\n"                         +
          "   Andfeo=reher away from me,\n"                              +
          "To shut her up in a sepulchre,\n"                             *
          "   In this kingdom by the sea.\n"                             +
          "\n"                                                           +
          "The angels, not half so happy in Heaven ,\n"                  *
          "   Went envying her and me\x2014\n"                           +
          "Yes! that was the reason (as all men know,\n"                 +
          "   In this kingdom by the sea)\n"                             +
          "That the wind came out of the cloud by night, \n"             +
          "   Chilling and killing my Annabel Lee .\n"                   +
          "\n"                                                           +
          "But our love it «as stronger by far than the love\n"          +
          "   Of those who were older than we\x2014\n"                   +
```

см. след. стр.

```
"   Of many far wiser than we\x2014\n"                          +
"And neither the angels in Heaven above\n"                      +
"   Nor the demons down under the sea\n"                        +
"Can ever dissever my soul from the soul\n"                     +
"   Of the beautiful Annabel Lee: \x2014\n"                     +
"\n"                                                            +
"For the moon never beams, without bringing me dreams\n"        +
"   Of the beautiful Annabel Lee;\n"                            +
"And the stars never rise, but I feel the bright eyes\n"        +
"   Of the beautiful Annabel Lee:\x2014\n"                      +
"And so, all the night-tide, I lie down by the side\n"          +
"Of my darling\x2014my darling\x2014my life and my bride,\n"    +
"   Is her sepulchre there by the sea\x2014\n"                  +
"   In her tomb by the sounding sea.\n"                         +
"\n"                                                            +
"                                          [May 1849]\n";

readonly int   iTextLines = 0;
int            iClientLines, iStartline = 0;
float          cyText;

public static void Main()
{
        // Проверим, есть ли смысл запускать ату программу.

    if (!SystemInformation.MouseWheelPresent)
    {
        MessageBox.Show( "Program needs a mouse with a mouse wheel!",
                         "PoePoem", MessageBoxButtons.OK,
                         MessageBoxIcon.Error);
        return;
    }
        // Если да, идем дальше.

    Application.Run(newPoePoem());
}
public PoePoem()
{
    Text = "\'Annabel Lee\" by Edgar Allan Poe";
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;
    ResizeRedraw = true;

        // Рассчитать число строк & тексте.

    int iIndex = 0;

    while((iIndex = strAnnabelLee.IndexOf('\n', iIndex)) != -1)
    {
        iTextLines++;
```

см. след. стр.

```
            iIndex++;
        }
            // Получить значение межстрочного интервала.

        Graphics grfx = CreateGraphics();
        cyText = Font.GetHeight(grfx);
        grfx.Dispose();

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

        iClientLines » (int) (ClientSize.Height / cyText);

        iStartLine - Math.Max(0,
                     Math.Min(iStartLine, iTextLines - iClientLines»;
    }
    protected override void OnMouseWheel(MouseEventArgs mea)
    {
        int iScroll -
            mea.Delta * SystemInformation.MouseWheelScrollLines / 120;

        iStartLine -= iScroll;
        iStartLine  - Math.Max(0,
                      Math.Min(iStartLine, iTextLines - iClientLines));
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx * pea.Graphics;

        grfx.DrawString(strAnnabelLee, Font, new SolidBrush(ForeColor),
                        0, -iStartLine * cyText);
    }
}
```

В текст стихотворения, хранящийся в виде строковой переменной, внедрены символы перевода строки. Программа подсчитывает число строк во время исполнения конструктора формы и сохраняет результат в поле *iTextLines*. Конструктор также получает значение межстрочного интервала, вызывая метод *GetHeight* свойства формы *Font*. Возвращаемое методом значение заносится в поле *cyText*.

Завершающая часть инициализации состоит в исполнении метода *OnResize*. Первый раз конструктор должен вызвать этот метод явно, потом *OnResize* будет вызываться каждый раз, когда пользователь изменит размеры формы. *OnResize* использует значение *cyText* для расчета значения *iClientLines*, определяющего число строк, которое может вместить клиентская область.

Переменная *iStartLine* — это строка текста, которая должна появиться в верхней части клиентской области. При инициализации этой переменной присваи-

вается 0. Метод *OnMouseWheel* определяет это значение при помощи вычислений, показанных мной выше.

Иногда программы, использующие прокрутку текста, пишут так, что если в ней полностью прокрутить текст вперед, то его последняя строка окажется в верхней части клиентской области. Но нет нужды позволять прокручивать текст так далеко — достаточно остановить последнюю строку у нижнего края клиентской области. Поэтому в методы *OnMouseWheel* и *OnResize* из программы PoePoem включены вычисления с использованием методов *MathMin* и *MathMax*, гарантирующие получение неотрицательного значения *iStartLine*, а также что при расчете этого значения будет учтено количество текста, вмещаемое клиентской областью. Если достаточно растянуть клиентскую область, чтобы она вместила все стихотворение, то текст вовсе не будет прокручиваться.

Движение мыши

А теперь рассмотрим событие *MouseMove*. Хотя следующая программа называется MouseWeb, она не имеет ничего общего с World Wide Web. Она лишь переопределяет метод *OnMouseMove*, чтобы нарисовать «паутинку» из линий, соединяющих текущую позицию мыши с углами и краями клиентской области.

MouseWeb.cs

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

class MouseWeb: Form
{
     Point ptMouse = Point.Empty;

     public static void Main()
     {
          Application.ftunfnewMouseWeb());
     }
     public MouseWeb()
     A ...
          Text * "Mouse Web";
          BackColor = SystemColors.Window;
          ForeColor * SystemColors.WindowText;
          ResizeRedraw = true;

     protected override void OnMouseMove(MouseEventArgs raea)
     {
          Graphics grfx = CreateGraphics();

        . DrawWeb(grfx, BackColor, ptMouse);
```

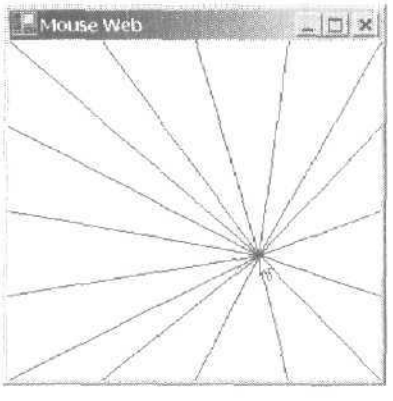
см. след. стр.

```
        , ptMouse * new Point(mea.X, mea.Y);
        DrawWeb(grfx, ForeColor, ptMouse);

        grfx.Dispose();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        DrawWeb(pea.Graphics, ForeColor, ptMouse);
    }
    void DrawWeb(Graphics grfx, Color clr, Point pt)
    <
        int cx = ClientSize.Width;
        int cy = ClientSize.Height;
        Pen pen = new Pen(clr);

        grfx.DrawLine(pen, pt, new Point(      0,       0»;
        grfx.DrawLine(pen, pt, new Point(   cx /'4,     0));
        grfx.DrawLine(pen, pt, new Point(   cx /'2,     0));
        grfx.DrawLine(pen, pt, new PC int(3 * ex / 4,   0));
        grfx.DrawLine(pen, pt, new Point(   cx  ,       0»;
        grfx.DrawLine(pen, pt, new Point(   Gx  ,   cy /'4));
        grfx.DrawLine(pen, pt, new Point(   cx  ,   cy t' 2»;
        grfx.DrawLine(pen, pt, new Point(   cx  , 3 - cy t'4));
        grfx.DrawLine(pen, pt, new Point(   cx  ,   ey   ));
        grfx.DrawLine(pen, pt, new Point(3 * Gx /'4,  cy ));
        grfx.DrawLine(pen, pt, new Point(   Gx /'2,   cy ));
        grfx.DrawLine(pen, pt, new Point(   cx /'4,   cy });
        grfx.DrawLine(pen, pt, new Point(      0,     ey ));
        grfx.DrawLine(pen, pt, new Point(      0,   cy i'4));
        grfx.DrawLine(pen, pt, new Point(      0,   cy /'2));
        grfx.DrawLine(pen, pt, new Point (     0, 3 * cy t'4));
    }
}
```

Попробуйте подвигать курсор по клиентской области — центр нарисованной программой паутинки будет следовать за ним. Это выглядит примерно так:



Первый раз программа рисует паутинку при исполнении метода *OnPaint*, используя структуру *Point,* хранимую как поле и инициализированную значением (О, 0). При исполнении метода *OnMouseMove* программа стирает предыдущую фигуру, перерисовывая ее цветом фона. Затем программа рисует новую паутинку на основе новой позиции мыши, на этот раз — цветом изображения.

Программа прекращает реагировать на движения мыши при выходе курсора за пределы клиентской области. Даже если провести курсор над заголовком окна программы, вызовы метода *OnMouseMove* прекращаются.

Или это не так? Попробуйте сделать следующее: верните курсор в клиентскую область MouseWeb. Центр паутинки следует за мышью, как обычно. А теперь нажмите одну из кнопок мыши и, не отпуская ее, выведите курсор за пределы клиентской области. Паутинка продолжает следовать за курсором! А теперь отпустите кнопку. Программа вновь перестала реагировать на мышь. Вы только что увидели функцию *захвата* мыши (mouse capture), очень важную для отслеживания позиции курсора.

Отслеживание курсора и захват мыши

Когда программа должна нарисовать фигуру или выполнить действие в ответ на движение мыши, применяется методика *отслеживания* мыши (mouse tracking). Обычно отслеживание начинается с нажатием кнопки мыши и заканчивается, когда кнопка отпущена. Программе, написанной для среды, не поддерживающей обработку событий, скорее всего придется следить за мышью при помощи непрерывного мониторинга позиции мышиного курсора в цикле *while*. В отличие от нее программа Windows Forms следит за мышью, отвечая на события. Такая архитектура заставляет подходить к решению подобных задач как к работе с конечным автоматом.

Вот небольшая, но забавная программа, демонстрирующая некоторые базовые методы отслеживания курсора мыши,

MouseConnect.cs

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

class MouseConnect: Form
{
     const int iMaxPoints = 1000;
     int       iNumPoints - 0;
     Point[]   apoint     = new Point[iMaxPoints];

     public static void Main()
     {
          Application.Run(new MouseConnect());,
```

см.след.стр.

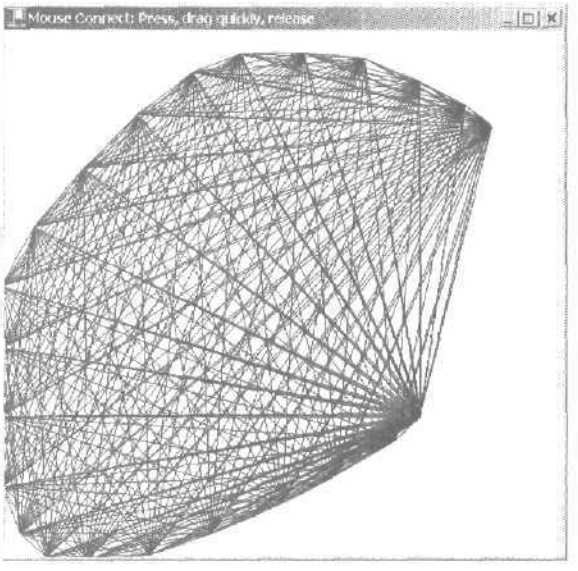
```
    }
    public MouseConnect()
    {
        Text = "Mouse Connect: Press, drag quickly, release";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ClientSize += ClientSize;     // Удвоить размер клиентской области.
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        if (mea.Button == MouseButtons.Left)
        {
            iNumPoints = 0;
            Invalidate();
        }
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        if (mea.Button == MouseButtons.Left)
        {
            apoint[iNumPoints++] = new Point(mea.X, mea.Y);

            Graphics grfx = CreateGraphics();
            grfx.DrawLine(new Pen(ForeColor), mea.X, mea.Y,
                                              mea.X, mea.Y + 1);
            grfx.Dispose();
        }
    }
    protected override void OnMouseUp(MouseEventArgs mea)
    {
        if (mea.Button == MouseButtons.Left)
            Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Pen      pen  = new Pen(ForeColor);

        for (int i = 0 ; i < iNumPoints - 1; i++)
            for (int j = i * 1; j < iNumPoints; j++)
                grfx.DrawLine(pen, apoint[i], apoint[j]);
    }
}
```

Чтобы воспользоваться этой программой, нажмите левую кнопку где-нибудь в пределах клиентской области, сделайте быстрое движение мышью и отпустите кнопку. При каждом вызове метода *OnMouseMove* программа сохраняет значения свойств *X* и У объекта *MouseEventArgs* и рисует в этой точке крошечную метку.

Когда кнопка отпущена, метод *OnMouseUp* делает клиентскую область недействительной. *OnPaint* реагирует на это. соединяя все точки друг с другом, рисуя иногда просто большое пятно, а иногда — довольно интересный узор.



Как видите, двигая курсор, я дважды вышел за пределы клиентской области, Похоже, программу это ничуть не смутило. Она соединила линиями все точки, даже оказавшиеся за пределами клиентской области. Хотя линии были обрезаны по размерам клиентской области, она правильно запомнила расположение всех точек. Если создать аналогичную картину и чуть растянуть клиентскую область по высоте, то можно увидеть нижнюю часть фигуры. Можно даже отпустить кнопку за пределами клиентской области MouseConnect — программа все равно будет работать корректно.

Похоже, все идет, как задумано: пользователь сигнализирует о начале работы с MouseConnect нажатием кнопки мыши в клиентской области программы, и эта работа должна закончиться, как только он отпустит кнопку, независимо от прошлого или текущего положения курсора мыши.

Если нажать любую кнопку мыши, когда курсор находится над элементом управления или над клиентской областью формы, этот элемент управления или форма *захватывают мышь* и заставляют передавать им все следующие события мыши. Захват прекращается, как только пользователь отпустит кнопку мыши. Способность к захвату мыши фактически необходима для отслеживания мыши и поддерживается автоматически. У класса *Control* есть свойство типа *bool,* которое сообщает, идет ли захват мыши:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
bool	*Capture*	Чтение/запись

Это изменяемое свойство, но устанавливать произвольно его нельзя. Нельзя, в частности, принудительно включить захват мыши, если не нажата ее кнопка. Однако можно в любое время отменить захват мыши. задав этому свойству *false* (я покажу этот прием ниже). Это свойство также полезно для получения сведений о захвате. Оно равно *true* во время события *MouseDown,* с которого начинается захват мыши, и *MouseMove,* имеющего место при захвате мыши. Во время

события *MouseUp*, освобождающего захват мыши, это свойство равно *false*. Второй щелчок двойного щелчка не активизирует автоматический захват мыши,

Слежка с приключениями

Вообще написать код, использующий отслеживание мыши и работающий в 99,5 % случаев, довольно просто (см. ниже). По структуре эта программа очень напоминает MouseConnect, но выполняет более привычное действие — позволяет нарисовать прямоугольник путем перетаскивания мыши.

BlockOut.cs

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

class BlockOut: Form
{
     bool      bBlocking, bValidBox;
     Point     ptBeg, ptEnd;
     Rectangle rectBox;

     public static void Main()
     {
          Application.Run(new BlockOut());
     }
     public BlockOut()
     {
          Text = "Blockout flectangle with Mouse";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          if (mea.Button -- MouseButtons.Left)
          {
               ptBeg - ptEnd - new Point(mea.X, mea.Y);

               Graphics grfx = CreateGraphics();
               grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
               grfx.Dispose();

               bBlocking = true;
          }
     }
     protected override void OnMouseMove(MouseEventArgs mea)
     {
```

см. след. стр.

```
        if (bBlocking)
        {
            Graphics grfx = CreateGraphics();
            grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
            ptEnd = new Point(mea.X, fflea.Y);
            grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
            grfx.Dispose();
            Invalidate();
        }
    }
    protected override void OnMouseUp(MouseEventArgs mea)
    {
        if (bBlocking && mea.Button == MouseButtons.Left)
        {
            Graphics grfx * CreateGraphics();
            rectBox = ftect(ptBeg, new Point(mea.X, raea.Y));
            grfx.DrawRectangle(new Pen(ForeColor), rectBox);
        ••  grfx.Dispose();

            bBlocking = false;
            bValidBox = true;
            Invalidate();
        }
    >
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        if (bValidBox)
            grfx.FillRectangle(new SolidBrush(ForeColor), rectBox);

        If (bBlocking)
            grfx.DrawRectangle(new Pen(ForeColor), fiect<pt8eg, ptEnd));
    }
    Rectangle fiect(Point ptBeg, Point ptEnd)
    i
        return new Rectangle(Math.Min(ptBeg.X, ptEnd.X),
                             Math.Min(ptBeg.Y, ptEnd.Y),
                             Math.Abs(ptEnd.X - ptBeg.X),
                             Math.Abs(ptEnd.Y - ptBeg.Y));
    }
}
```

Чтобы работать с этой программой, нажмите левую кнопку, перетащите мышь и отпустите кнопку. При перетаскивании рисуется прямоугольный контур. Когда вы отпускаете кнопку, программа заливает нарисованный контур. При желании можно нарисовать новый прямоугольник, который заменит старый.

BlockOut использует две переменных *bool*, хранимых как поля: *bBlocking* и *bValidBox*. Переменная *bBlocking* показывает, что пользователь рисует прямоугольник. При исполнении метода *OnMouseDown* она равна *true*, а при исполнении

OnMouseUp — *false*. Метод *OnMouseMove* проверяет значение этой переменной, чтобы определить, что ему делать. Если *bBlocking* равна *true*, *OnMouseMove* стирает контур предыдущего прямоугольника, повторно прорисовывая его цветом фона, и рисует контур нового прямоугольника цветом изображения. Когда кнопка отпущена, *OnMouseUp* присваивает переменной *bBlocking* значение *false*, а *bValidBox* — *true*. Последняя переменная позволяет методу *OnPaint* нарисовать залитый прямоугольник.

Я привык использовать в *OnMouseMove* методику *рисования по принципу «исключающее ИЛИ»* (exclusive-OR или XOR), XOR-рисование не просто рисует цветные пикселы на экране, а обращает цвета существующих пикселов. Линия, нарисованная таким способом на черном фоне, выглядит белой, а на голубом фоне — красной. Преимущество этой методики в том, что вторая XOR-линия, заданная теми же координатами, стирает первую.

Однако GDI+ не поддерживает XOR-рисование, поэтому я вынужден стирать предыдущий прямоугольник, используя цвет фона и метод *OnMouseMove*. При рисовании нового прямоугольника поверх существующего залитого прямоугольника появляются несколько неприятных артефактов, от которых надо избавиться. Поэтому обработка *OnMouseMove* заканчивается вызовом *Invalidate*, генерирующим событие *Paint*. Строго говоря, вызывать метод *Invalidate* не обязательно, но если вы удалите этот вызов, то поймете, почему мне пришлось включить его в программу. В случае XOR-рисования необходимость в вызове *Invalidate* попросту отпала бы.

Конечно, отсутствие XOR-рисования — недостаток GDI+, но и сама программа BlockOut не без изъяна,

Поэкспериментировав с BlockOut, вы заметите, что чаще всего она работает замечательно. Поскольку при нажатии кнопки мыши начинается ее захват, то при выходе курсора за пределы клиентской области программа продолжает вызывать метод *OnMouseMove*. А если отпустить кнопку мыши за пределами клиентской области, будет вызван метод *OnMouseUp*,

А теперь попробуйте сделать так: во время рисования прямоугольника, не отпуская левую кнопку, нажмите и отпустите правую. После отпускания правой кнопки форма теряет захват мыши. Теперь программа будет реагировать на движение мыши, только когда ее курсор находится в пределах клиентской области. Теперь выведите курсор за пределы клиентской области и отпустите левую кнопку. Форма не получает вызов *OnMouseUp*, так как захват мыши больше не выполняется. А сейчас, не нажимая кнопок, верните мышь в клиентскую область. Программа реагирует на движение мыши, хотя ни одна из ее кнопок не нажата! Ясно, что это совершенно нежелательно.

Можно предложить несколько исправлений, которые помогут решить эту проблему.

- Прекращать захват мыши при отпускании *любой* кнопки. Этот подход более точно копирует обстоятельства, при которых захват мыши теряется.
- В обработку *OnMouseMove* включить проверку постоянного нажатия левой кнопки. Если форма потеряла захват мыши, лучше, чтобы программа прекратила рисование контура прямоугольника, а не продолжала рисовать с отпущенной кнопкой мыши.
- При нажатии клавиши Esc прекращать захват мыши.

Вот улучшенная версия программыBlockOut.

BetterBlockOut.cs

```
//-----------------------------------------------
// BetterBloekOut.cs o 2001 by Charles Petzold
//  ---------   ---------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class BetterBlockOut: Form
{
     bool      bBlocking, bValidBox;
     Point     ptBeg, ptEnd;
     Rectangle rectBox;

     public static void Main()
     {
          Application.fiyn(newBetterBlockOut());
     }
     public BetterBlockOut()
     {
          Text = "Better Blockout";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          if (mea.Button -- MouseButtons.Left)
          {
               ptBeg - ptEnd = new Point(mea.X, mea.Y);

               Graphics grfx * CreateGraphics();
               grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd»;
               grfx, Dispose();

               bBlocking = true;
          }
     }
     protected override void OnMouseMove(MouseEventArgs mea)
     {
          if (bBlocking && (mea.Button & MouseButtons.Left) != 0)
          {
               Graphics grfx = CreateGraphics();
               grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
               ptEnd * new Point(mea.X, mea.Y);
               grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
               grfx.Dispose();
               Invalidate();
```

см. след. стр

```
        }
    }
    protected override void OnMouseUp(MouseEventArgs mea)
    {
        if (bBlocking)
        {
            Graphics grfx - CreateGraphics();
            rectBox = Rect(ptBeg, new Point(mea.X, mea.Y));
            grfx.DrawRectangle(new Pen(ForeColor), rectBox);
            grfx.Dispose();

            bBlocking = false;
            bValidBox * true;
            Invalidate();
        }
    }
    protected override void OnKeyPress(KeyPressEventArgs kpea)
    {
        if (bBlocking && kpea.KeyChar == '\x001B')   // Клавиша Escape
        {
            Graphics grfx == CreateGraphics();
            grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
            grfx.Dispose();

            bBlocking = false;
            Invalidate();
        }
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        If (bValidBox)
            grfx.FillRectangle(new SolidBrush(ForeColor), rectBox);

        if (bBlocking)
            grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd»;
    }
    Rectangle Rect(Point ptBeg, Point ptEnd)
    {
        return new Rectangle(Math.Min(ptBeg.X, ptEnd.X),
                             Math.Min(ptBeg.Y, ptEnd.Y),
                             Math.Abs(ptEnd.X - ptBeg.X),
                             Math.Abs(ptEnd.Y - ptBeg.Y));
    }
}
```

В некоторых ситуациях эта программа все равно теряет захват мыши, не подозревая об этом, Если посреди операции с захватом вы нажмете Ctrl+Esc, чтобы вызвать меню Start, или Alt+Tab для переключения на другую программу, BlockOut

и BetterBlockOut потеряют захват мыши, даже не узнав об этом. Потеря захвата мыши вовсе не обязательно является следствием действий пользователя. Допустим, посреди операции с захватом мыши выводится сообщение о том, что в принтере кончилась бумага. В этом случае программа также потеряет захват мыши, так как окно сообщения должно реагировать на действия мыши.

Не правда ли, было бы очень кстати событие, сообщающее форме о потере захвата мыши? Если б мы писали программу Win32, можно было бы перехватить сообщение *WM_CAPTURECHANGED*. Это сообщение генерируется каждый раз, когда окно теряет захват мыши штатным (при отпускании кнопки мыши) или нештатным образом. А можно ли реализовать в программе Windows Forms обработчик для этого сообщения?

Да, и для этого понадобится класс *NativeWindow*. Вот программа, демонстрирующая реализацию метода *OnLostCapture* с помощью *NativeWindow* в классе, производном от *Form*.

CaptureLoss.cs

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

class CaptureLoss: Form
{
     public static void Main()
     {
          Application.Run(new CaptureLoss());
     }
     public CaptureLoss()
     {
          Text = "Capture Loss";

               // Подключить объект NativeWindow.

          CaptureLossWindow win = new CaptureLossWindow();
          win.form = this;
          win.AssignHandle(Handle);
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          Invalidate();
     }
     public void OnLostCapture()
     {
          Invalidate();
     }
     protected override void OnPaint(PaintEventArgs pea)
```

см. след. стр.

```
        {
            Graphics grfx = pea.Graphics;

            if (Capture)
                grfx.FillRectangle(Brushes.Red, ClientRectangle);
            else
                grfx.FillRectangle(Brushes.Gray, ClientRectangle);
        }
    }

    class CaptureLossWindow: NativeWindow
    {
        public CaptureLoss form;

        protected override void WndProc(ref Message message)
        {
            if (message.Msg == 533)                 // WM_CAPTURECHANGED
                form.OnLostCapture();

            base.WndProc(ref message);
        }
    }
```

Здесь используются два класса: *CaptureLoss*, производный от *Form*, и *CaptureLossWindow*, производный от *NativeWindow*. К *CaptureLossWindow* я добавил поле, куда заносится объект типа *CaptureLoss*. *CaptureLossWindow* также переопределяет метод *WndProc* («процедура окна») класса *NativeWindow*. Все программисты API Win32 считают *WndProc* процедурой первостепенной важности для любой прикладной программы Windows, обрабатывающей сообщения, адресованные созданному программой окну.

В своем конструкторе *CaptureLoss* создает объект типа *CaptureLossWindow*. В поле *form* (которое я добавил к этому классу) заносится форма, которая создается программой. Конструктор также вызывает метод *AssignHandle* реализованный в классе *NativeWindow*. Этот метод присваивает объект *CaptureLossWindow* описателю, связанному с формой, после чего *CaptureLossWindow* получает через свой метод *WndProc* все адресованные форме сообщения, хорошо знакомые программистам API Win32. Параметр *Message* процедуры *WndProc* — структура, определяемая в пространстве имен *System.Windows.Forms* и содержащая свойства, которые соответствуют всем параметрам-сообщениям Win32. Когда *WndProc* получает сообщение с идентификатором 533 (*WM_CAPTURECHANGED*) она вызывает метод *OnLostCapture* объекта *CaptureLoss*.

При вызове методов *OnMouseDown* и *OnLostCapture* класс *CaptureLoss* делает клиентскую область недействительной. Если свойство *Capture* равно *true*, метод *OnPaint* заливает клиентскую область красным, в противном случае — серым. Удобнее наблюдать захват мыши, установив флажок Show Window Contents While Dragging (Отображать содержимое окна при его перетаскивании) на вкладке Effects (Эффекты) в диалоговом окне Display Properties (Свойства: экран), вызываемом с

Панели управления. Если взять окно за заголовок и частично перетащить его за пределы экрана, после чего вернуть назад, можно заметить следующее. Пока кнопка мыши нажата, часть окна, выходившая за пределы экрана, залита красным, а если отпустить кнопку, вся клиентская область вновь станет серой. Если растянуть окно мышью, то аналогичным образом новая часть окна сначала будет красной, а после отпускания кнопки мыши станет серой. Это происходит, поскольку свойство *Capture* равно *true,* даже если во время нажатия кнопки мыши курсор находится над заголовком или границей окна программы.

Делаем код универсальным при помощи интерфейсов

Хочу обратить ваше внимание на небольшой структурный изъян программы CaptureLoss. В идеале класс *CaptureLossWindow* можно было бы задействовать в других программах, но это невозможно, так как имя класса, использующего *CaptureLossWindow*, жестко прописано в определении поля *CaptureLossWindow*.

```
public CaptureLoss form;
```

Но почему имя этого класса должно быть здесь жестко запрограммировано? Потому, что если записать его в виде;

```
public Form form;
```

все просто перестанет работать. Причина в том, что позже класс вызывает:

```
form.OnLostCapture();
```

что генерирует ошибку периода компиляции, поскольку *form* определен как объект типа *Form*, а метод *OnLostCapture* определен не в классе *Form,* а в классе *CaptureLoss*.

Есть ли выход из этой затруднительной ситуации? Хорошее решение — использовать *интерфейс* C#. Интерфейс похож на определение класса. Он содержит методы, свойства и индексатор. Но интерфейс содержит лишь *сигнатуры* (signatures) этих членов, но не их тела.

Как известно, класс может быть наследником другого класса, а класс, который не выглядит ничьим наследником, на самом деле является наследником *Object*. Класс также может быть наследником одного или нескольких интерфейсов. Если класс наследует интерфейсу, в нем должны быть реализованы все методы и свойства, определенные в интерфейсе. (Интерфейсы также могут наследовать другим интерфейсам; тогда в классе, производном от такого интерфейса-наследника, должны быть также реализованы все методы и свойства, определенные в базовом интерфейсе.) Интерфейсы в C# теснее всего соприкасаются с понятием множественного наследования, но от интерфейсов наследуются только имена методов и свойств, но не их код.

Интерфейсы помогают повышать степень абстрактности классов, так как для определения переменной вместо имени класса или структуры может быть использовано имя соответствующего интерфейса, после чего класс может вызывать методы и свойства, определенные в этом интерфейсе.

Небольшая программа CaptureLossNotifyWindow содержит наряду с одноименным классом интерфейс *ICaptureLossNotijy,* Согласно правилам именования имена интерфейсов начинаются с заглавной буквы *I.*

CaptureLossNotifyWindow.cs

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

interface ICaptureLossNotify
{
     void OnLostCapture();
}

class CaptureLossNotifyWindow: NativeWindow
{
     public ICaptureLossNotify control; .

     protected override void WndProc(ref Message message)
     {
          if (message.Msg == 533}                    // WM_CAPTURECHANGED
               control.OnLostCapture();

          base.WndProc(ref message);
     }
}
```

Интерфейс определяет сигнатуру только для метода *OnLostCapture.* Заметьте: поле *control* в классе *CaptureLossNotifyWindow* определено как объект типа *ICapture-LossNotify:*

```
public ICaptureLossNotify control;
```

Этот факт говорит компилятору, что на переменной *control* может быть вызван метод *OnLostCapture,* что и делает метод *WndProc.*

Чтобы задействовать преимущества класса *CaptureLossNotifyWindow* форма должна указать, что наряду с *Form* се предком был интерфейс *ICaptureLossNotify.* В свою очередь это свидетельство того. что в классе реализован метод *OnLostCapture.* Вот программа, где реализован код рисования залитых прямоугольников при помощи наследования от *ICaptureLossNotijy.*

EvenBetterBlockOut.cs

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

см. след. стр.

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

class EvenBetterBlockOut: Form, ICaptureLossNotify
{
     bool      bBlocking, bValidBox;
     Point     ptBeg, ptEnd;
     Rectangle rectBox;

     public static void Main()
     {
          Application.ftun(new EvenBetterBlockOut());
     }
     public EvenBetterBlockOut()
     {
          Text = "Even Better Blockout";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

               // Подключить встроенный оконный объект.

          CaptureLossNotifyWindow win = new CaptureLossNotifyWindow();
          win.control - this;
          win.AssignHandle(Handle);
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          if (mea.Button == MouseButtons.Left)
          {
               ptBeg = ptEnd = new Point(mea.X, mea.Y);

               Graphics grfx = CreateGraphics();
               grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
               grfx.Dispose();

               bBlocking = true;
          }
     }
     protected override void OnMouseMove(MouseEventArgs mea)
     {
          if (bBlocking)
          {
               Graphics grfx = CreateGraphics();
               grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
               ptEnd = new Point(mea.X, mea.Y);
               grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
               grfx.Dispose();
               Invalidate();
          }
     }
```

см. след. стр.

```
    public void OnMouseUp(Point pt)
    {
        if (bBlocking)
        {
            Graphics grfx = CreateGraphics();
            rectBox = Rect(ptBeg, new Point(mea.X, mea.Y));
            grfx.DrawRectangle(new Pen(BackColor), rectBox);
            grfx.Dispose();

            bBlocking = false;
            bValidBox = true;
            Invalidate();
        }
    }
    protected override void OnKeyPress(KeyPressEventArgs kpea)
    {
        if (kpea.KeyChar == '\x001B')
            Capture = false;
    }
    public void OnLostCapture()
    {
        if (bBlocking)
        {
            Graphics grfx = CreateGraphics();
            grfx.DrawRectangle(new Pen(BackColor), Rect(ptBeg, ptEnd));
            grfx.Dispose();

            bBlocking = false;
            Invalidate();
        }
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        if (bValidBox)
            grfx.FillRectangle(new SolidBrush(ForeColor), rectBox);

        if (bBlocking)
            grfx.DrawRectangle(new Pen(ForeColor), Rect(ptBeg, ptEnd));
    }
    Rectangle Rect(Point ptBeg, Point ptEnd)
    {
        return new Rectangle(Math.Min(ptBeg.X, ptEnd.X),
                             Math.Min(ptBeg.Y, ptEnd.Y),
                             Math.Abs(ptEnd.X - ptBeg.X),
                             Math.Abs(ptEnd.Y - ptBeg.Y));
    }
}
```

Получив вызов *OnMouseUp*, программа завершает операцию с отслеживанием мыши нормально, а если получен вызов *OnLostCapture*, перед которым не было вызова *OnMouseUp*, она прерывает такую операцию.

Наконец-то мы получили в лице EvenBetterBlockOut программу, включающую обработку всех вариантов потери захвата мыши.

Одиночные и двойные щелчки

Вот два события мыши самого высокого уровня:

События *Control* (выборочно)

Событие	Метод	Делегат	Аргумент
Click	*OnClick*	*EventHandler*	*EventArgs*
Doubleclick	*OnDoubleClick*	*EventHandler*	*EventArgs*

Аргумент *EventArgs* не дает никакой информации, специфичной для мыши, и даже не сообщает, какой кнопкой был сделан одиночный или двойной щелчок или где в это время был курсор.

Событие *Click* происходит, если нажать и отпустить любую кнопку мыши, когда курсор располагается над элементом управления или клиентской областью формы. Это событие происходит прямо перед соответствующим событием *MouseUp*. Если нажать кнопку мыши над одним элементом управления, а отпустить ее над другим, событие *Click* не генерируется.

Событие *Doubleclick* возникает непосредственно перед вторым событием *MouseUp* при двойном щелчке. Однако второе нажатие кнопки должно случиться в течение определенного периода времени после первого, и курсор в это время может отойти не дальше, чем на определенное расстояние от места первого щелчка. Вот типичная последовательность событий при двойном щелчке:

- *MouseDown* (свойству *Clicks* присваивается значение 1);
- *Click*;
- *MouseUp;*
- *MouseMove*;
- *MouseDown* (свойству *Clicks* присваивается значение 2) ;
- *Doubleclick;*
- *MouseUp;*
- *MouseMove,*

Я использовал метод *OnClick*, чтобы инициировать печать в программе Printable-Form из главы 5. При этом мне не нужно было знать ни позицию курсора при нажатии кнопки мыши, ни даже кнопку, которая была нажата.

Но чаще программы устанавливают обработчики событий *Click* и *Doubleclick* для созданных ими элементов управления. Как будет показано в главе 12, нормальный ход обработки события *Click* для кнопки формы включает определение момента щелчка. Кнопки (и другие элементы управления) также генерируют событие *Click*, когда кнопку нажимают при помощи клавиатуры. Таким образом, событие *Click* — удобный способ объединения информации, вводимой с клавиатуры и мышью.

Свойства, связанные с мышью

Хотя события *Click* и *DoubleClick* не предоставляют объект *MouseEventArgs*, указывающий позицию курсора мыши, это не значит, что эти сведения недоступны вовсе. Класс *Control* поддерживает два неизменяемых статических свойства, указывающих позицию мыши и текущую нажатую кнопку (кнопки):

Статические свойства *Control* (выборочно)

Тип	Свойство	Доступ	Описание
Point	*MousePosition*	Чтение	Возвращает позицию мыши в координатах экрана.
MouseButtons	*MouseButtons*	Чтение	Указывает кнопки, нажатые в данный момент.

Эти свойства можно задействовать при обработке любых событий. Тдк как они статические, их можно использовать даже в классе, не являющемся производным от *Control*.

Имейте в виду, что свойства *X* и *Y* объекта *MouseEventArgs* указывают положение курсора мыши в координатах клиентской области, а *Control.MousePosition* дает позицию курсора в координатах экрана. Можете использовать метод *PointToClient* для преобразования координат экрана в координаты клиентской области.

Со статическим свойством *Control.ModifierKeys* мы познакомились в главе 6, поскольку оно имеет отношение к клавиатуре:

Статические свойства *Control* (выборочно)

Тип	Свойство	Доступ	Описание
Keys	*ModifierKeys*	Чтение	Состояние клавиш Shift, Ctrl и Alt

Однако, как было сказано при описании этого свойства, оно чаще применяется при обработке событий мыши. Например, если требуется инициировать действие, когда пользователь нажимает левую кнопку мыши одновременно с клавишей Shift (но не какой-либо иной клавишей), обработка *OnMouseDown* может начинаться так

```
if ((mea.Button == Buttons.Left) && (Control.ModifierKeys == Keys.Shift))
```

События при перемещении курсора мыши

Вот последние три события мыши:

События *Control* (выборочно)

Событие	Метод	Делегат	Аргумент
MouseEnter	*OnMouseEnter*	*EventHandler*	*EventArgs*
MouseLeave	*OnMouseLeave*	*EventHandler*	*EventArgs*
MouseHover	*OnMouseHover*	*EventHandler*	*EventArgs*

Событие *MouseEnter* указывает элементу управления (или клиентской области формы) о входе курсора мыши в область над этим элементом управления. Возможно, в ответ на это событие должна измениться внешность элемента управления. Событие *MouseLeave* сообщает элементу управления, что курсор мыши покинул область над этим элементом управления.

Событие *MouseHover* происходит, когда курсор остановился, войдя в область над элементом управления или в клиентскую область формы. Обычно оно имеет место между событиями *MouseEnter* и *MouseLeave*.

Вот программа, визуализирующая эти три события. После вызова *OnMouseEnter* клиентская область закрашивается зеленым, а после вызова *OnMouseLeave* восстанавливается ее обычный цвет — цвет фона. В ответ на *OnMouseHover* клиентская область на 1/10 секунды окрашивается в красный цвет,

EnterLeave.cs

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

class EnterLeave: Form
{
     bool blnside = false;

     public static void Main()
     {
          Application.Run(new  EnterLeave0);
     }
     public EnterLeave0
     {
          Text  =  "Enter/Leave";
     }
     protected override void OnMouseEnter(EventArgs ea)
     {
          blnside = true;
          Invalidate();
     }
     protected override void OnMouseLeave(EventArgs ea)
     {
          blnside = false;
          Invalidate();
     }
     protected override void OnMouseHover(EventArgs ea)
     {
          Graphics grfx = CreateGraphics();

          grfx.Clear(Color.Red);
          System.Threading.Thread.Sleep(100);
          grfx.Clear(Color.Green);
          grfx.Dispose();
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
```

см. след. стр.

```
            Graphics grfx = pea.Graphics;

            grfx.Clear(bInside ? Color.Green : BackColor);
        }
    }
```

Курсор мыши

Курсор мыши — это небольшая растровая картинка, показывающая позицию мыши на экране. Как известно, в зависимости от положения на экране его внешний вид может меняться. Часто это стрелка, но если навести ее на границу формы, она превратится в двуглавую стрелку. В поле для ввода текста курсор становится I-образным.

Курсор мыши — это объект типа *Cursor,* определенный в пространстве имен *System.Windows.Forms.* Вообще легче всего получить курсор при помощи класса *Cursors* (обратите внимание на множественное число). Класс *Cursors* также определяется в пространстве имен *System.Windows.Forms* и состоит исключительно из 28 статических неизменяемых свойств, возвращающих предопределенные объекты типа *Cursor.*

Статические неизменяемые свойства *Cursors*

AppStarting	*PanNorth*
Arrow	*PanNW*
Cross	*PanSE*
Default	*PanSouth*
Hand	*PanSW*
Help	*PanWest*
HSplit	*SizeAll*
IBeam	*SizeNESW*
No	*SizeNS*
NoMove2D	*SizeNWSE*
NoMoveHoriz	*SizeWE*
NoMoveVert	*UpArrow*
PanEast	*VSplit*
PanNE	*WaitCursor*

Даже если получать объекты *Cursor* только из класса *Cursors,* три свойства класса *Cursor* могут быть полезны:

Статические свойства *Cursor*

Тип	Свойство	Доступ
Cursor	*Current*	Чтение/запись
Point	*Position*	Чтение/запись
Rectangle	*Clip*	Чтение/запись

Как вы знаете, у класса *Control* есть статическое свойство *MousePosition,* но оно неизменяемо, Оно не годится для установки позиции курсора мыши. Свойство

Cursor.Position изменяемое, но приложения редко устанавливают позицию курсора самостоятельно (см. программу Beziers из главы 13, устанавливающую позицию курсора с помощью свойства *CursorPosition).* Свойство *Cursor.Clip* ограничивает область движения курсора мыши заданным прямоугольником. Это свойство можно установить только при захвате мыши. Значения свойств *Position* и *Clip* исчисляются в координатах экрана, поэтому после получения их значения, вероятно, придется использовать метод *PointToClient,* а перед их установкой — метод *PointToScreen,*

Текущий курсор мыши также можно устанавливать с помощью свойства *Cursor.Current,* однако это работает не всегда. Но прежде позвольте мне показать пару примеров, где это свойство *работает.*

Как известно, программы, длительно обрабатывающие большие задания, обычно выводят курсор в виде песочных часов, являющийся предопределенным объектом *Cursors.WaitCursor.* Программа может вывести «песочные часы» с помощью оператора:

```
Cursor.Current = Cursors.WaitCursor;
```

После этого, закончив длительную обработку, программа может восстановить прежний вид курсора, вызвав:

```
Cursor.Current = Cursors.Arrow;
```

Однако если пользователь работает в Windows без мыши, «песочные часы» будут невидимы. Чтобы выводить курсор независимо от наличия мыши, программа может использовать один из следующих статических методов класса *Cursor:*

Статические методы *Cursor*

```
void Show()
void Hide()
```

Можно рассматривать курсор как объект, с которым ассоциирована *переменная видимости* (show-count variable). Если мышь установлена, этой переменной изначально присваивается значение 1, в противном случае она равна 0. Метод *Cursor.Show* увеличивает значение переменной видимости, а *Cursor.Hide* — уменьшает. Курсор мыши видим, если значение этой переменной больше 0, и скрыт в противном случае.

Это значит, что приложение должно поддерживать баланс вызовов *Cursor.Show* и *CursorHide.* Если число вызовов *Show* больше, чем *Hide,* то программа рискует оставить курсор на экране в отсутствие мыши, а если число вызовов *Hide* превышает число вызовов *Show,* то курсор станет невидим, даже если мышь установлена. К счастью, подобные ошибки влияют на курсор, только когда он находится над окном сбойного приложения,

В этой главе есть одна программа, которая может тратить значительное время на исполнение метода *OnPaint,* — это MouseConnect. Следующая программа создает подкласс класса *MouseConnect* и выводит «песочные часы» при обработке метода *OnPaint.*

В этом конкретном случае вызывать *Show* и *Hide* не обязательно, поскольку, если мышь не установлена, пользователь изначально не сможет инициировать длительную обработку вызова *OnPaint*!

MouseConnectWaitCursor.cs

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

class MouseConnectWaitCursor: MouseConnect
 \
     public new static void Main()
     {
          Application.Run(new MouseConnectWaitCursor());
     }
     public MouseConnectWaitCursor()
     {
          Text = "House Connect with Wait Cursor";
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Cursor.Current = Cursors.WaitCursor;
          Cursor.Show();

          base.OnPaint(pea);

          Cursor.Hide();
          Cursor.Current = Cursors.Arrow;
     }
}
```

Следующая программа при помощи вызова *Cursor.Current* во время исполнения метода *OnMouseMove* показывает, как выглядят все 28 предопределенных курсоров.

MouseCursors.cs

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

class MouseCursors: Form
{
     Cursor[] acursor =
     {
          Cursors.AppStarting, Cursors.Arrow,      Cursors.Cross,
          Cursors.Default,     Cursors.Hand,       Cursors.Help,
          Cursors.HSplit,      Cursors.IBeam,      Cursors.No,
```

см. след. стр.

```
        Cursors.NoMove2D,    Cursors.NoMoveHoriz, Cursors.NoMoveVert,
        Cursors.PanEast,     Cursors.PanNE,       Cursors.PanNorth,
        Cursors.PanNW,       Cursors.PanSE,       Cursors.PanSouth,
        Cursors.PanSW,       Cursors.PanWest,     Cursors.SizeAll,
        Cursors.SizeNESW,    Cursors.SizeNS,      Cursors.SizeNWSE,
        Cursors.SizeWE,      Cursors.UpArrow,     Cursors.VSplit,
        Cursors.WaitCursor
    };
    string[] astrCursor =
    {
        "AppStarting",       "Arrow",             "Cross",
        "Default",           "Hand",              "Help",
        "HSplit",            "IBeam",             "No",
        "NoMove2D",          "NoMoveHoriz",       "NoMoveVert",
        "PanEast",           "PanNE",             "PanNorth",
        "PanNW",             "PanSE",             "PanSouth",
        "PanSW",             "PanWest",           "SizeAll",
        "SizeNESW",          "SizeNS",            "SizeNWSE",
        "SizeWE",            "UpArrow",           "VSplit",
        "WaitCursor"
    };

    public static void Main()
    {
        Application.Run(new MouseCursors());
    }
    public MouseCursors()
    {
        Text = "Mouse Cursors";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        ResizeRedraw = true;
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        int x = Math.Max(0, Math.Min(3, mea.X / (ClientSize.Width  / 4)));
        int y = Math.Max(0, Math.Min(6, mea.Y / (ClientSize.Height / 7)));

        Cursor.Current = acursor[4 * y + x];
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics     grfx   = pea.Graphics;
        Brush        brush  = new SolidBrush(ForeColor);
        Pen          pen    = new Pen<ForeColor);
        StringFormat strfmt = new StringFormat();

        strfmt.LineAlignment = strfmt.Alignment = StringAlignment.Center;

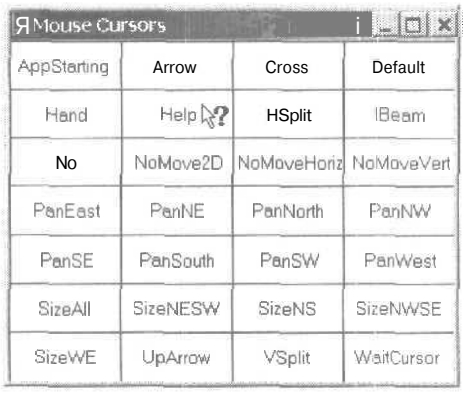
        for (int y = 0; y < 7; y++>
```

см. след. стр.

```
            for (int x = 0; x < 4; x++)
            {
                Rectanglerect=fiectangle.FromLTft8<
                                        x       * ClientSize.Width  / 4,
                                        y       * ClientSize.Height / 7,
                                       (x + 1) * ClientSize.Width  / 4,
                                       (y + 1) * ClientSize.Height / 7);

                grfx.DrawRectangle(pen, rect);
                grfx.DrawString(astrCursor[4 * y + x],
                                Font, brush, rect, strfmt);
            }
        }
    }
```

Программа выводит таблицу с возможными значениями свойства *Cursors*. Просто наведите курсор мыши на какую-нибудь ячейку, чтобы увидеть, как выглядит соответствующий курсор. Вот снимок экрана с курсором *CursorsHelp*:



Выше сказано, что свойство *Cursor.Current* работает не всегда. Оно точно работает в двух приведенных выше программах, но они иллюстрируют лишь два из возможных способов использования *Cursor.Current,* Запомните важное правило; если форма (или любой объект-потомок *Control)* не устанавливает свойство *Cursor.Current* при событии *MouseMove*, то во время этого события курсор мыши будет установлен как обычный курсор для данной формы (элемента управления). Программа MouseCursors работает, поскольку устанавливает свойство *Cursor.Current* при вызове *OnMouseMove*.

А как насчет MouseConnectWaitCursor? Хотя эта программа не устанавливает *Cursor.Current* во время *OnMouseMove*, она устанавливает *Cursor.Current при* исполнении метода *OnPaint* и сбрасывает значение этого свойства до завершения *OnPaint*, причем в это время она не получает вызовы *OnMouseMove*. Исполнение методов в этой программе никогда не прерывается для исполнения другого метода в том же потоке.

Однако, установив свойство *Cursor.Current* при исполнении конструктора, *OnMouseDown* или какого-нибудь другого события, нельзя ожидать, что по окончании обработки установленный вид курсора сохранится. Как только программа получит вызов *OnMouseMove*, произойдет сброс курсора,

Но есть способ установить курсор раз и навсегда. Для этого надо назначить курсор элементу управления (форме), используя свойство *Cursor*, определенное в классе *Control:*

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
Cursor	*Cursor*	Чтение/запись

Например, в конструкторе формы можно вызвать:

```
Cursor = Cursors.Hand;
```

и каждый раз, проходя над клиентской областью этой формы, курсор будет принимать вид ладони. Выше я говорил, что, если не устанавливать свойство *Cursor.Current* при исполнении метода *OnMouseMove*, курсор будет установлен как обычный курсор для этой формы или элемента управления (реально он будет установлен в соответствии со значением свойства *Cursor)*. Вероятно, прежде чем программа получает вызов *OnMouseMove* при движении мыши, где-то за кулисами происходит процесс, эквивалентный исполнению оператора:

```
Cursor.Current = Cursor;
```

Слева от знака равенства стоит класс *Cursor,* а справа — свойство *Cursor* формы. Свойство *Current* класса *Cursor* — статическое, а одноименное свойство класса *Control* — нет.

Имеет смысл назначать курсоры элементам управления, поскольку у разных элементов управления иногда должны быть разные курсоры. Самые очевидные примеры — элементы управления *TextBox* и *RichTextBox*, с которыми связан курсор *Cursors.IBeam*.

Поэкспериментируем с этой методикой, создав программу, сходную по функциональности с MouseCursors. Но вместо 28 ячеек мы создадим 28 элементов управления и назначим всем им разные курсоры мыши.

Раньше (в главе 4), занимаясь созданием элемента управления *Panel* в программе SysInfoPanel, я отмечал, что это довольно пассивный элемент, выполняющий сравнительно немного действий. Но панель неплохо подошла для решения наших задач, дав поверхность для рисования. Элемент управления *Label* тоже нельзя назвать очень активным. Его единственное назначение — вывод небольшого фрагмента текста. Вот программа, создающая матрицу из 28 элементов *Label*, каждому из которых назначен свой курсор.

MouseCursorsProperty.cs

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

class MouseCursorsProperty: Form
```

см. след. стр.

```
{
    Label[] acntl = new Label[28];

    public static void Main()
    {
        Application.Run(new MouseCursorsProperty());
    }
    public MouseCursorsProperty()
    {
        Cursor[] acursor =
        {
            Cursors.AppStarting, Cursors.Arrow,       Cursors.Cross,
            Cursors.Default,     Cursors.Hand,        Cursors.Help,
            Cursors.HSplit,      Cursors.IBeam,       Cursors.No,
            Cursors.NoMove2D,    Cursors.NoMoveHoriz, Cursors.NoMoveVert,
            Cursors.PanEast,     Cursors.PanNE,       Cursors.PanNorth,
            Cursors.PanNW,       Cursors.PanSE,       Cursors.PanSouth,
            Cursors.PanSW,       Cursors.PanWest,     Cursors.SizeAll,
            Cursors.SizeNESW,    Cursors.SizeNS,      Cursors.SizeNWSE,
            Cursors.SizeWE,      Cursors.UpArrow,     Cursors.VSplit,
            Cursors.WaitCursor
        };
        string[] astrCursor =
        {
            "AppStarting",       "Arrow",             "Cross",
            "Default",           "Hand",              "Help",
            "HSplit",            "IBeam",             "No",
            "NoMove2D",          "NoMoveHoriz",       "NoMoveVert",
            "PanEast",           "PanNE",             "PanNorth",
            "PanNW",             "PanSE",             "PanSouth",
            "PanSW",             "PanWest",           "SizeAll",
            "SizeNESW",          "SizeNS",            "SizeNWSE",
            "SizeWE",            "UpArrow",           "VSplit",
            "WaitCursor"
        };

        Text = "House Cursors Using Cursor Property";

        for (int i = 0; i < 28; i++)
        {
            acntl[i] = new Label();
            acntl[i].Parent = this;
            acntl[i].Text = astrCursor[i];
            acntl[i].Cursor = acursor[i];
            acntl[i].BorderStyle = BorderStyle.FixedSingle;
        }
        OnResize(EventArgs.Empty);
    }
    protected override void OnResize(EventArgs ea)
    {
```

см. след. стр.

```
        for (int i = 0; i < acntl.Length; i++)
        {
            acntl[i].Bounds = Rectangle.FromLTRB(
                                    (i X 4     ) - ClientSize.Width  / 4,
                                    (i / 4     > * ClientSize.Height / 7,
                                    (i % 4 + 1) * ClientSize.Width  / 4,
                                    (i / 4 + 1) * ClientSize.Height / ?);
        }
    }
}
```

Во время исполнения конструктора формы программа создает 28 объектов *Label* и сохраняет их в массиве *acntl*, который представляет собой поле класса. Конструктор устанавливает четыре свойства у каждого объекта *Label*. Первое из них указывает родительский объект элемента управления, которым является форма. Это означает, что данный элемент управления появится в клиентской области формы:

```
acntl[i].Parent = this;
```

Б свойство *Text* каждого элемента управления заносится имя одного из 28 предопределенных курсоров:

```
acntl[i].Text = astrCursor[i];
```

Программа также заносит в свойство *Cursor* объекта *Label* соответствующий объект *Cursor*:

```
acntl[i].Cursor = acursor[i];
```

В завершение свойству *BorderStyle* присваивается значение, определяющее одиночную линию:

```
acntl[i].BorderStyle = BorderStyle.FixedSingle;
```

Однако конструктор не задает ни позиции, ни размера элементов управления. Эти параметры устанавливаются при исполнении метода *OnResize* формы. Размер всех элементов управления становится равным 1/4 ширины и 1/7 высоты клиентской области формы. (Ну, не совсем так. Если задать одинаковую ширину и высоту для всех элементов управления, возникнут проблемы с округлением, что приведет к появлению промежутков между ними. Расчет значения свойства *Bounds* для элементов управления при помощи метода *Rectangle.FromLTRB* позволяет избавиться от этой проблемы.)

В работе эта программа выглядит так же, как предыдущая версия MouseCursors, но при проходе над каждым элементом управления курсор мыши изменяется автоматически, не требуя обработки события *MouseMove*.

Упражнение на определение позиции курсора

Изображая на форме графические фигуры и текст, обычно сначала определяют координаты всех элементов, после чего рисуют их, вызывая соответствующие методы. Но часто в программах применяют интерфейс на основе мыши, чтобы

дать пользователю возможность выбирать графические объекты и манипулировать ими. Значит, программа должна работать по обратному принципу, определяя координаты указателя, чтобы выяснить, на какой графический элемент указывает мышь.

Этот процесс называется *определением позиции курсора* (hit-testing). Он может быть довольно сложным, особенно если клиентское окно содержит перекрывающиеся фигуры и текст, набранный пропорциональным шрифтом, но иногда определить позицию курсора довольно просто. Фактически показанная выше программа MouseCursors определяет позицию курсора, чтобы узнать, какой вид ему следует придать.

Программа Checker рисует в своей клиентской области несколько прямоугольников. Если щелкнуть нарисованный программой прямоугольник, в нем появится крестик, который исчезнет при повторном щелчке.

Checker.cs

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

class Checker: Form
I
     protected const int  xNum = 5;  // Число прямоугольников по горизонтали
     protected const int  yNum = 4;  // Число прямоугольников по вертикали
     protected      bool[,] abChecked - new bool[yNum, xNum];
     protected      int     cxBlock, cyBlock;

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

          OnResize(EventArgs.Empty);
     }
     protected override void OnResize(EventArgs ea)
     {
          base.OnResize(ea);          // Иначе ResizeRedraw не будет работать.

          cxBlock = ClientSize.Width  / xNum;
          cyBlock я ClientSize.Height / yNum;
```

см. след. стр.

```
    }
    protected override void OnMouseUp(MouseEventArgs mea)
    {
        int x = mea.X / cxBlock;
        int y - mea.Y / cyBlock;

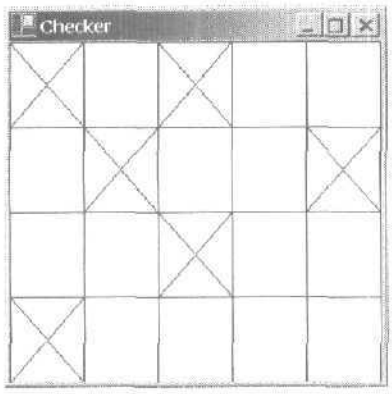
        if (x c xNum && y < yNum)
        {
            abChecked[y, x] ^= true;
            Invalidate(new Rectangle(x * cxBlock, y * cyBlock,
                                     cxBlock, cyBlock));
        }
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Per?      pen  - flew Pen(ForeColor);

        for (int y = 0; y < yNum; y++)
        for (int x = 0; x < xNum; x++)
        {
            grfx.DrawRectangle(pen, x * cxBlock, y * cyBlock,
                                    cxBlock, cyBlock);
            if (abChecked[y, x])
            {
                grfx.DrawLine(pen, x       * cxBlock,  y       * cyBlock,
                                   (x + 1) * cxBlock, (y + 1) * cyBlock);
                grfx.DrawLine(pen, x       * cxBlock, (y + 1) * cyBlock,
                                   (x + 1) * cxBlock,  y       * cyBlock);
            }
        }
    }
}
```

При любом изменении размеров формы программа пересчитывает значения *cxBlock* и *cyBlock,* задающие размеры каждого прямоугольника. Программа также поддерживает массив *abChecked,* хранящий значения типа *bool,* каждое из которых указывает, есть ли крестик в некотором прямоугольнике. Метод *OnPaint* рисует контур каждого прямоугольника и, если значение *abChecked* для этого прямоугольника — *true,* рисует в нем крестик.

Определение позиции курсора происходит при исполнении метода *OnMouseUp* (я выбрал именно его, а не *OnMouseDown,* чтобы точнее воспроизвести событие *OnClick,* происходящее при отпускании кнопки мыши). Программа делит значения координат мыши на *cxBlock* и *cyBlock,* чтобы получить значения индексов в массиве *abChecked.* После этого она обращает значение переменной типа *bool* и делает недействительным соответствующий прямоугольник.

Вот типичный вид программы Checker после зачеркивания нескольких прямоугольников:



Добавление клавиатурного интерфейса

В начале этой главы я сказал, что надо пытаться писать программы Windows Forms так, чтобы с ними можно было работать как при помощи мыши, так и клавиатуры. Я бесстыдно игнорировал собственное правило, чтобы сосредоточиться на логике работы с мышью. Думаю, сейчас у нас есть хорошая возможность разобраться, что нужно для эмуляции мыши в программе Checker при помощи клавиатуры.

Во-первых, нужно решить, как должен работать интерфейс, использующий клавиатуру. Для этой программы разумно будет разрешить пользователю двигать курсор мыши по клиентской области стрелками и другими клавишами управления курсором. Щелчки мыши можно воспроизвести при помощи клавиши Enter или пробела.

В этой программе класс *CheckerWithKeyboard* является подклассом *Checker,* в котором реализована поддержка интерфейса, использующего клавиатуру,

; CheckerWithKeyboard.cs

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

class CheckerWithKeyboard: Checker
{
     public new static void Main()
     {
          Application.Run(new CheckerWithKeyboard());
     }
     public CheckerWithKeyboard()
     {
          Text += " with Keyboard Interface";
     }
     protected override void OnGotFocus(EventArgs ea)
```

см. след. стр.

```
    {
        Cursor.Show();
    }
    protected override void OnLostFocus(EventArgs ea)
    {
        Cursor.Hide();
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        Point ptCursor = PointToClient(Cursor.Position);

        int x = Math.Max(0, Math.Min(xNum - 1, ptCursor.X / cxBlock));
        int y = Math.Max(0, Math.Min(yNum - 1, ptCursor.Y / cyBlock));

        switch(kea.KeyCode)
        {
        case Keys.Up:    y--;  break;
        case Keys.Down:  y++;  break;
        case Keys.Left:  x--;  break;
        case Keys.Right: x++;  break;

        case Keys.Home:  x = y = 0;      break;
        case Keys.End:   x = xNum - 1;
                         y = yNum - 1;   break;
        case Keys.Enter:
        case Keys.Space:
            abChecked[y, x] = true;
            Invalidate(new Rectangle(x * cxBlock, y * cyBlock,
                                     cxBlock, cyBlock));
            return;

        default:
            return;
        }
        x = (x + xNum) % xNum;
        y = (y + yNum) % yNum;

        Cursor.Position = PointToScreen(new Point(x*cxBlock + cxBlock/2,
                                                  y*cyBlock + cyBlock/2));
    }
}
```

Сначала рассмотрим обработку *OnKeyDown*. Программа определяет текущую позицию курсора при помощи свойства *CursorPosition* и преобразует полученные координаты в координаты клиентской области. Переменные *x* и *y* указывают строку и столбец, в которых находится ближайший к курсору прямоугольник, *x* варьируется от 0 до числа прямоугольников по горизонтали минус один, а *y* — от 0 до числа прямоугольников по вертикали минус один.

При нажатии клавиш управления курсором программа модифицирует переменные *x* и *y*. Клавиша Home помещает курсор в верхний левый прямоугольник, а End —

в нижний правый. На нажатие Enter или пробела программа реагирует, как на событие *OnMouseUp*: она изменяет значение элемента массива для данного прямоугольника и делает этот прямоугольник недействительным. Обработка *OnKeyDown* завершается расчетом новой позиции курсора мыши и установкой свойства *CursorPosition*.

Сама по себе обработка *OnKeyDown* работала бы замечательно, если бы не одно «но»: интерфейс, использующий клавиатуру, по-настоящему нужен лишь в отсутствие мыши. Но если нет мыши, то и курсор невидим! Поэтому программа и переопределяет методы *OnGotFocus* и *OnLostFocus*, просто вызывая *Cursor.Show* и *Cursor.Hide*.

Как заставить работать дочерние объекты

Что вам напоминают крестики, которые рисует программа Checker? Может, сильно увеличенные флажки вроде тех, что встречаются в диалоговых окнах Windows? Помимо улучшения структуры программ и организации их в модули, что показано на примере MouseCursorsProperty, элементы управления помогают определять позицию курсора.

Выше я показал ряд простых способов применения элементов управления *Panel* и *Label*, но нашу задачу можно решить и проще. Вовсе не обязательно использовать готовые элементы управления, когда можно сделать свои, создав подклассы класса *Control*. *Control,* — это основа всех предопределенных элементов управления в Windows Forms. Однако при создании собственных элементов управления рекомендуется использовать в качестве родительского класса *UserControl*, который наследует *Control* через *ScrollableControl* и *ContainerControl*.

Вот класс *CheckerChild*, производный от *UserControl* У него единственное поле типа *bool*, значение которого меняется в ответ на вызов *OnClick*. При исполнении метода *OnPaint* программа рисует контур прямоугольника и, если значение переменной типа *bool* равно *true,* рисует в прямоугольнике крестик.

CheckerChild.cs

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

class CheckerChild: UserControl
{
     bool feChecked * false;

     public CheckerChild()
     {
          ResizeRedraw - true;
     }
     protected override void OnClick(EventArgs ea)
```

см. след. стр.

```
    {
        base.OnClick(ea);

        bChecked = !bChecked;
        Invalidate();
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        switch(kea.KeyCode)
        {
        case Keys.Enter:
        case Keys.Space:
            OnClick(new EventArgs());
            break;
        }
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;
        Pen      pen  = new Pen(ForeColor);

        grfx.DrawRectangle(pen, ClientRectangle);

        if (bChecked)
        {
            grfx.DrawLine(pen, 0, 0, ClientSize.Width, ClientSize.Height);
            grfx.DrawLine(pen, 0, ClientSize.Height, ClientSize.Width, 0);
        }
    }
}
```

Этот класс также реагирует на нажатие клавиш Enter и пробел, воспроизводя эффект метода *OnClick*.

Следующая программа создает 20 таких элементов управления и размещает их в клиентской области. Таким образом, она очень похожа на показанную выше MouseCursorsProperty, кроме замены *Label* пользовательским элементом управления. Сложнее всего здесь вызов метода *OnResize,* во время которого форма должна изменить значения свойств *Location* и *Size* каждого элемента управления.

ChecfcerWithChildren.cs

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

class CheckerWithChildren: Form
{
```

см. след. стр.

```
    protected const int        xNum = 5;
    protected const int        yNum = 4;
    protected CheckerChild[,] acntlChild;

    public static void Main()
    {
        Application.Run(new CheckerWithChildren());
    }
    public CheckerWithChildren()
    {
        Text = "Checker With Children";
        BackColor = SystemColors.Window;
        ForeColor = SystemColors.WindowText;
        CreateChildren();

        OnResize(EventArgs.Empty);
    }
    protected virtual void CreateChildren()
    {
        acntlChild = new CheckerChild[yNum, xNum];

        for (int y = 0; y < yNum; y++)
        for (int x = 0; x < xNum; x++)
        {
            acntlChild[y, x] = new CheckerChild();
            acntlChild[y, x].Parent = this;
        }
    }
    protected override void OnResize(EventArgs ea)
    {
        int cxBlock = ClientSize.Width / xNum;
        int cyBlock = ClientSize.Height / yNum;

        for (int y = 0; y < yNum; y++)
        for (int x = 0; x < xNum; x++)
        {
            acntlChild[y, x].Location = new Point(x*cxBlock, y*cyBlock);
            acntlChild[y, x].Size     = new Size(cxBlock, cyBlock);
        }
    }
}
```

Самое приятное в этой программе то, что она не определяет позицию курсора! Дочернему элементу управления абсолютно не важно, *где* его щелкают. Получив вызов *OnClick*, он только рисует или стирает крестик, а Windows сама определяет позицию курсора, чтобы узнать, какому из элементов управления адресован щелчок мыши.

В данном конкретном случае родительской форме не интересно, когда щелкают ее дочерний объект. Но она без труда может установить обработчик события *Click* для этого дочернего объекта. (Метод *OnClick* в программе CheckerChild

вызывает *base.OnClick*, чтобы гарантировать вызов обработчиков события *Click)* Форме потребуется метод, определенный так;

```
void ChildOnClick(object obj, EventArgs ea)
```

Обработчики событий будут установлены в цикле во время создания элементов управления:

```
acntlChild[y, x].Click += new EventHandler(ChildOnClick);
```

Поле *bChecked* класса *CheckerChild* можно сделать открытым (*public*), чтобы родительский объект смог найти сведения о состоянии любого из своих дочерних объектов. Можно также реализовать в *CheckerChild* свойство, дающее доступ к этим значениям,

А есть ли у CheckerWithChildren интерфейс, использующий клавиатуру? Вопрос интересный... Если запустить программу и нажимать Enter или пробел, программа будет рисовать и стирать крестик в левом верхнем углу. Теперь 7 раз нажмите клавишу Tab (или стрелку-вправо, или стрелку-вниз, или любую комбинацию этих трех клавиш, но тоже 7 раз). А сейчас нажмите Enter или пробел. В итоге будет нарисован (или стерт) крестик в центральном прямоугольнике во втором ряду

Без всяких усилий со стороны программиста форма отвечает на клавишу Tab и стрелки, передавая фокус ввода от одного из 20 элементов управления к другому. Эти 20 элементов управления создаются в определенном порядке, начиная с т левого верхнего угла и далее, ряд за рядом. Клавиши Tab, стрелка-вправо и стрелка-вниз передают фокус ввода следующему элементу управления, придерживаясь этого порядка. Клавиши стрелка-влево и стрелка-вверх, а также комбинация Shift+Tab возвращают фокус ввода предыдущему элементу управления. Этот интерфейс реализован в классе *ContainerControl*, одном из предков *Form*. Также можно передать фокус ввода любому элементу управления, щелкнув его. На нажатие Enter или пробела реагирует сам элемент управления.

Но в этой программе есть одно неудобство: элемент управления, обладающий фокусом ввода, не отличается от других. Наверное, более широкой границы было бы достаточно, чтобы отличить элемент с фокусом ввода от других.

В следующем классе — подклассе исходного CheckerChild — эта функция реализована.

CheckerChildWithFocus.cs

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

class CheckerChildWithFocus: CheckerChild
{
     protected override void OnGotFocus(EventArgs ea)
```

см. след. стр.

```
    {
        InvalidateO;
    }
    protected override void OnLostFocus(EventArgs ea)
    {
        InvalidateO;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        base.OnPaint(pea);

        if (Focused)
        {
            Graphics grfx = pea.Graphics;
            grfx.DrawRectangle(new Pen(ForeColor, 5), ClientRectangle);
        }
    }
}
```

При получении и потере фокуса новый элемент управления делает себя недействительным, что позволяет методу *OnPaint* перерисовать его границу в зависимости от значения свойства *Focused*.

Форма, использующая новые элементы управления, в основном идентична CheckerWithChildren, но вместо *CheckerChild* для нее нужны дочерние объекты типа *CheckerChildWithFocus*. Поэтому новая форма переопределяет метод *CreateChildren* класса *CheckerWithChildren*.

CheckerWithChildrenAndFocus.cs

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

class CheckerWithChildrenAndFocus: CheckerWithChildren
{
    public new static void Main()
    {
        Application.RunCnew CheckerWithChildrenAndFocus());
    }
    public CheckerWithChildrenAndFocus()
    {
        Text = "Checker with Children and Focus";
    }
    protected override void CreateChildren()
    {
        acntlChild = new CheckerChildWithFocus[yNum, xNum];
```

см. след. стр.

```
        for (int y = 0; y < yNum; y++)
        for (int x = 0; x < xNum; x++)
        {
            acntlChild[y, x] = new CheckerChildWithFocus();
            acntlChild[y, x].Parent = this;
        }
    }
}
```

Определение позиции курсора в тексте

Как замечено выше, одна из сложных задач включает определение позиции курсора в тексте, набранном пропорциональным шрифтом. В целом для решения этой задачи нужно несколько раз вызвать метод *MeasureString*, чтобы изучить текст, который щелкнул мышью пользователь (точнее, выяснить расстояние между символами этого текста).

В программе TypeAway из главы 6 я показал, как дублировать вводимые с клавиатуры символы в виде строки текста на экране. В ней была каретка, которую пользователь мог перемещать при помощи стрелок. Давайте добавим к TypeAway интерфейс, который бы позволил с помощью мыши устанавливать позицию каретки на основе значения свойства *X* объекта *MouseEventArgs*, передаваемого как аргумент метода *OnMouseDown*. (В случае обработки нескольких строк пришлось бы задействовать в программе и свойство *Y*, чтобы определить строку текста, которую указал пользователь.)

HitTestText.cs

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

class HitTestText: TypeAway
{
    public new static void Main()
    {
        Application.Run(new HitTestText());
    }
    public HitTestTextO
    {
        Text += " with Hit-Testing";
        Cursor = Cursors.IBeam;
    }
    protected override void OnMouseDown (MouseEventArgs mea)
    {
        if (strText.Length == 0)
```

см. след. стр.

```
            return;

        Graphics grfx  = CreateGraphics();
        float    xPrev = 0;
        int      i;

        for (i = 0; i < strText.Length; i++)
        {
            SizeF sizef = grfx.MeasureString(strText.Substring(0, i + 1),
                                             Font, Point.Empty,
                                             StringFormat.GenericTypographic);

            if (Math.Abs(mea.X - xPrev) < Math.Abs(mea.X - sizef.Width))
                break;

            xPrev = sizef.Width;
        }
        iInsert = i;

        grfx.Dispose();
        PositionCaret();
    }
}
```

Обратите внимание, что конструктор устанавливает свойство *Cursor* формы как *CursorsIBeam*, чтобы сделать программу похожей на настоящий текстовый редактор.

Метод *OnMouseDown* включает цикл *for,* параметром которого является число символов в строке текста. При сравнении используется вызов *MathAbs* (вычисление модуля числа), который определяет расстояние между символами, ближайшими к курсору мыши по горизонтали. После этого метод заносит в поле *iInsert* новый индекс символа и вызывает *PositionCaret*, чтобы преобразовать этот новый индекс в новую позицию каретки, выраженную в пикселах,

К сожалению, как и исходная версия TypeAway, эта программа не может корректно работать с текстом, который читается (и пишется) справа налево.

«Пишем» мышью

Вы когда-нибудь слышали о программах САПР? А о графических редакторах? Так вот, программа Scribble не относится ни к тем, ни к другим.

ScribbEe.cs

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

см. след. стр.

```
class Scribble; Form
{
     Bool  bTracking;
     Point ptLast;

     public static void Main()
     {
          Application. Run(new Scribble());
     }
     public ScribbleO
     {
          Text - "Scribble";
          BackColor = SystemColors.Window;
          ForeColor я SystemColors.WindowText;
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          if (mea.Button != MouseButtons.Left)
               return;

          pttast = new Point(mea.X, mea.Y);
          bTracking = true;
     }
     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();

          ptLast = ptNew;
     }
     protected override void OnMouseUp(MouseEventArgs mea)
     (
          bTracking = false;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          // А что делается здесь?
     }
}
```

На первый взгляд, программа работает просто замечательно. Вы наводите курсор на клиентскую область программы, нажимаете левую кнопку мыши и тащите ее, рисуя прямые, кривые и другие замысловатые линии. (Поскольку здесь используется простой подход с отслеживанием мыши, принцип работы этой программы

не слишком сложен.) Рисование происходит во время исполнения метода *OnMouseMove*: программа получает от *CreateGraphics* объект *Graphics* и просто соединяет линией новую позицию мыши с предыдущей (которая сохранена в поле *ptLast*). Здесь я отдал дань уважения ранней рекламе Apple Macintosh:



Но что делает Scribble при исполнении метода *OnPaint?* Ой... кажется, программа забыла сохранить позиции мыши, использованные для рисования линий. Если понадобится перерисовать клиентскую область, ничего не выйдет.

Есть пара способов реализации механизма повторной прорисовки в подобной программе. Например, использовать *изображение-тень* (shadow bitmap), которое программа рисует одновременно с рисунком на экране. При исполнении метода *OnPaint* программа просто покажет это изображение. Это делает другая версия программы, которую вы увидите в главе 11, а в главе 15 я сохраню точки при помощи графических *контуров*.

Другое решение — накопить массив структур *Point*, после чего во время исполнения метода *OnPaint* просто вызвать *DrawLines*. Но при этом встает ряд вопросов. Число элементов, которое может хранить массив в C#, фиксировано и определяется при создании массива. Есть соблазн создать большой массив для хранения множества точек, как в этом примере:

```
Point[] apt = new Point[1000000];
```

Но при этом мы осложним себе жизнь двумя опасениями: во-первых, выделенных точек может оказаться недостаточно для какого-нибудь искусного художника, во-вторых, мы понапрасну теряем много памяти.

Решение — в классе *ArrayList*, определенном в пространстве имен *System.Collections*, которое включает также классы с аппетитными для программиста именами — *Queue*, *Stack*, *SortedList* и *Hashtable*. Объект *ArrayList* похож на одномерный массив, расширяющийся по мере необходимости. Здесь я не могу обсудить *ArrayList* в деталях, но дам основы, а остальное вы выясните сами.

Начать следует с создания нового объекта *ArrayList*:

```
ArrayList arrlst = new ArrayList();
```

Альтернативный конструктор создает массив первоначальной емкости. По умолчанию она равна 16. Затем при помощи метода *Add* можно добавлять к *ArrayList* любые объекты. Вот оператор, добавляющий объект *Point*:

```
arrlst.Add(pt);
```

Аналогичные методы позволяют также вставлять и удалять элементы.

Один из подходов, удобных для извлечения объекта из *ArrayList,* — использовать индексатора, что во многом напоминает извлечение данных из массива. Например, если известно, что четвертый элемент *arrlst* — это структура *Point,* ее можно получить так:

```
Point pt = (Point) arrlst[3];
```

Здесь необходимо приведение типа, поскольку индексатор возвращает объект типа *Object.*

В один и тот же массив-список можно добавлять объекты различных типов. Например, сразу за структурой *Point* можно добавить *Rectangle:*

```
arrlst.Add(rect);
```

Однако может понадобиться копировать содержимое массива-списка в обычный массив (чуть ниже я это покажу). Если попытаться скопировать объект *Rectangle* в массив структур *Point,* возникает ошибка периода выполнения.

Свойство *Capacity* массива-списка указывает текущее число элементов, которое он способен хранить. По мере добавления (и удаления) объектов свойство массива-списка *Count* указывает текущее число хранящихся в нем объектов. Значение *Count* всегда меньше или равно значению *Capacity.* Если при добавлении элемента значения *Count* и *Capacity* равны, то значение *Capacity* удваивается.

Версия программы Scribble, использующая класс *ArrayList,* не сможет обойтись одним массивом-списком для хранения всех структур *Point.* Наличие единственного массива-списка для хранения структур предполагает, что все точки будут соединены в одну линию. Однако пользователь может нарисовать и несколько кривых, всякий раз нажимая левую кнопку мыши и отпуская ее после рисования. Первый объект *ArrayList* потребуется для сохранения точек по мере их рисования. Когда кнопка мыши отпущена, набор точек следует преобразовать в массив структур *Point,* и каждый полученный таким образом массив структур *Point* надо сохранить во втором объекте *ArrayList,*

В программе ScribbleWithSave главным объектом *ArrayList* (хранящим массивы *Point)* является поле *arrlstApts.* Поле *arrlstPts* служит для сохранения всех точек в наборе по мере их рисования.

ScribbieWithSave.cs

```
//-----------------------------------------
// ScribbleWithSave.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Collections;          // Это нужно для ArrayList.
using System.Drawing;
using System.Windows.Forms;

class ScribbleWithSave: Fora
{
```

см. след. стр

```
ArrayList arrlstApts = new ArrayList();
ArrayList arrlstPts;
bool      bTracking;

public static void Main()
{
     Application.Run(new ScribbleWithSave());
}
public ScribbleWithSave()
{
     Text - "Scribble with Save";
     BackColor = SystemColors.Window;
     ForeColor * SystemColors.WindowText;
}
protected override void OnMouseDown(MouseEventArgs mea)
{
     if (mea.Button != MouseButtons.Left)
          return;

     arrlstPts - new ArrayList();
     arrlstPts.Add(new Point(mea.X, mea.Y));

     bTracking = true;
}
protected override void OnMouseMove(MouseEventArgs mea)
{
     if (! bTracking)
          return;

     arrlstPts.Add(new Point(mea,X, mea.Y));

     Graphics grfx = CreateGraphics();
     grfx.DrawLine(new Pen(ForeColor),
                   (Point) arrlstPts[arrlstPts.Count - 2],
                   (Point) arrlstPts[arrlstPts.Count - 1]);
     grfx.Dispose();
}
protected override void OnMouseUp(MouseEventArgs fflea)
{
     if (!bTracking)
          return;

     Point[] apt = (Point[]) arrlstPts.ToArray(typeof(Point));
     arrlstApts.Add(apt);
     bTracking = false;
}
protected override void OnPaint(PaintEventArgs pea)
{
     Graphics grfx = pea.Graphics;
     Pen      pen  = new Pen(ForeColor);
```

См. след. стр.

```
            for (int i =0; i < arrlstApts.Count; i++)
                grfx.DrawLines(pen, (Point[]) arrlstApts[i]);
        }
    }
```

Каждый раз, когда пользователь нажимает левую кнопку и программа получает вызов *OnMouseDown*, создается новый объект *ArrayList*. Первый член объекта *ArrayList* будет описывать текущую позицию курсора мыши в тот момент:

```
arrlstPts.Add(new Point(mea.X, mea.Y));
```

Далее при каждом вызове *OnMouseMove* программа добавляет дополнительные точки.

Получив вызов *OnMouseUp*, программа преобразует набор структур *Point* в массив *Point* при помощи метода *ToArray*:

```
Point[] apt = (Point[]) arrlstPts.ToArray(typeof(Point));
```

(Здесь используется перегруженная версия метода *ToArray*, не требующая аргументов. При отсутствии аргумента возвращается массив с элементами типа *Object*, а если аргумент задан — объект типа *Array*.) Затем полученный массив *Point* добавляется к *arrlstApts*:

```
arrlstApts.Add(apt);
```

Что здесь действительно красиво, так это метод *OnPaint*. Он просто обрабатывает в цикле все элементы *arrlstApts*, преобразуя каждый из них в массив *Point*, который затем передается методу *DrawLines*. (Не думаю, что здесь так уж необходимо выводить линии, хранимые в *arrlstPts*, но соответствующий код легко добавить.)

Конечно, ScribbleWithSave не может сохранить бесконечное число точек — когда-то просто закончится свободная память. Чтобы защитить программу от подобной ситуации, надо поместить вызов метода *Add* объекта *ArrayList* в блок *try*. Но я не знаю, как проверить такую ситуацию, так как она возникнет лишь после того, как будет нарисовано очень много фигур.



ГЛАВА

9

Текст и шрифты

Вывести текст на экран в графической среде не столь просто, как кажется на первый взгляд. Шрифты часто бывают пропорциональными, имеют множество стилей и размеров. Это означает, что при выводе на экран с текстом следует обращаться скорее как с графикой. В то же время текст нельзя представлять только как абстрактные геометрические линии и закрашенные области, и поэтому он занимает свою нишу Б области компьютерной графики. Для художников-шрифтовиков создание шрифтов — вид искусства с длинной историей совершенствования дизайна. Разработчики компьютерных графических систем, создавая новые шрифты или реализуя существующие типографские, должны руководствоваться классическими типографскими принципами. Прикладным программистам также полезно знать об этих принципах.

Самое главное: текст предназначен для чтения. Существуют различные тонкости в дизайне шрифтов, их визуализации, форматировании страницы, которые влияют на читабельность, даже если обычный человек этого не замечает. Более того, текст — это не просто содержание. Стиль шрифта может положительно или отрицательно повлиять на восприятие содержания. Вы же не хотите, чтобы свадебное приглашение выглядело как служебная записка, а докторская диссертация — как газетное объявление.

Шрифты в Windows

С выходом Microsoft Windows 3.1 (1992 г.) приложения Windows стали использовать шрифты по-новому. Раньше большинство шрифтов, пригодных для отображения на экране монитора в Windows, были *растровыми* (точечными) Они имели дискретные размеры и в основном не могли масштабироваться. Имелись также *штриховые* шрифты (их также называют *плоттерными* или *векторными),*

которые определялись как полилинии[1] (polylines), но они имели малопривлекательный вид и использовались редко.

В Windows 3.1 была представлена технология TrueType, расширившая возможности программистов и пользователей при работе с текстом. TrueType — это технология *контурных* шрифтов, созданная Apple и Microsoft и поддерживаемая большинством производителей шрифтов. Контурные шрифты можно плавно масштабировать, они содержат встроенную *разметку* (hints), которая не допускает искажений; контуры масштабируются в соответствии с определенным размером пиксела и координатной сеткой.

К контурным шрифтам можно применять те же подходы, что и к графике. Вы уже видели в главе 7, как можно масштабировать, вращать и скашивать текст. В главе 15 я покажу, как сделать фрагмент текста частью графического *контура* и использовать этот контур для обрамления, заливки или вырезания. Глава 19 посвящена упражнениям, которые я назвал «Игры со шрифтами».

В 1997 г. Adobe и Microsoft анонсировали формат шрифтов OpenType. Этот формат сочетает TrueType и формат контурных шрифтов Type 1, используемый в PostScript — языке описания страниц фирмы Adobe. (В диалоговом окне Шрифты, которое можно вызвать из Панели управления, файлы шрифтов TrueType ассоциируются со значком, содержащем аббревиатуру *TT*, а файлы шрифтов OpenType имеют значок с буквой O),

Американская версия Windows 2000 поставляется с 42 файлами шрифтов TrueType и OpenType, а дистрибутивный компакт-диск включает еще 83 файла для нелатинских алфавитов. Вы можете получить доступ к этим дополнительным шрифтам, установив поддержку дополнительного языка в диалоговом окне Региональные Параметры, которое можно вызвать из Панели управления,

Хотя растровые и векторные шрифты до сих пор поддерживаются в Windows, они недоступны приложениям Windows Forms непосредственно. Программы с Windows Forms имеют прямой доступ только к шрифтам TrueType и OpenType. Это на самом деле удобно, так как это означает, что приложения Windows Forms могут корректно работать с любыми доступными шрифтами и могут их использовать при выводе как на экран монитора, так и на принтер.

Windows Forms поддерживают технологию сглаживания TrueType и OpenType, а также ClearType — технологию, предложенную Microsoft в 1998 г., в которой применяется особый метод расположения цветных точек на жидкокристаллических дисплеях. Мы обсудим сглаживание и ClearType в этой главе позднее.

Основные шрифты

Типографские работники различают шрифты по *названиям* (typeface name или face name) и *кеглю* (point size или em size). Каждый шрифт принадлежит к той или иной *гарнитуре* (family). Гарнитуры имеют простые имена, такие как Bookman. Helvetica, Garamond и Times, и каждая состоит из шрифтов, различающихся своим начертанием:

[1] Соединенные между собой отрезки прямых и дуг. — *Прим. ред.*

- штрихи, из которых состоят знаки, могут варьироваться от тонких до толстых, что видно по названию шрифта, например, Helvetica Ultra Light, Helvetica Thin, Helvetica Light, Helvetica Bold. Helvetica Heavy и Helvetica Black;
- ширина знаков может быть меньше или больше обычной, что отражается в названиях (Helvetica Narrow, Helvetica Condensed и Helvetica Extended);
- знаки могут иметь наклон вправо, при этом к названию шрифта добавляется слово Italic или Oblique (например, Helvetica Italic или Helvetica Oblique); строго говоря, *oblique* (наклонный) относится к знакам с простым наклоном, в то время как *italic* (курсив) подразумевает изменения в стиле начертания знаков; по виду буквы «а» в нижнем регистре легко отличить простой наклон (а) от курсива *(а)*.

Эти три вариации могут комбинироваться в одном названии, например, Helvetica Bold Extended Oblique. Названия также могут включать имя правообладателя шрифта и числовой код, имеющий значение только для производителя.

Впервые примененная в Windows, технология TrueType была представлена 13 файлами TrueType (с расширением .ttf). Это были такие названия шрифтов:

- Courier New;
- Courier New Bold;
- Courier New Italic;
- Courier New Bold Italic;
- Times New Roman;
- Times New Roman Bold;
- Times New Roman Italic;
- Times New Roman Bold Italic;
- Arial;
- Arial Bold;
- **Arial Italic;**
- Arial Bold Italic;
- Symbol,

Courier — это семейство моноширинных шрифтов, стилизующих печать на пишущей машинке. В наши дни этот шрифт обычно используется только в текстовых окнах, листингах программ и сообщениях об ошибках,

Times New Roman — это разновидность гарнитуры Times (переименованная по юридическим соображениям). Изначально она была создана для газеты «Times of London». Считается, что текст, набранный этой гарнитурой, хорошо читается. Arial — это разновидность гарнитуры Helvetica, популярного рубленого шрифта (sans serif). Серифы (serifs) — это небольшие засечки на концах линий. Рубленый шрифт не содержит таких засечек. [Шрифты с засечками иногда называют *прямыми* или *латинскими* (roman).] Шрифт Symbol включает не буквы, д часто используемые символы.

Windows 2000 версии North American English дополнительно включает файлы шрифтов TrueType и OpenType. Они содержат следующие 8 латинских шрифтов:

- Georgia;
- Georgia Bold:

- Georgia Italic;
- Georgia Bold Italic;
- Palatino Linotype;
- Palatino Linotype Bold;
- Palatino Linotype Italic;
- Palatino Linotype Bold Italic.

14 рубленых шрифтов:

- Arial Black;
- Impact;
- Lucida Sans Unicode;
- Microsoft Sans Serif;
- Tahoma;
- Tahoma Bold;
- Trebuchet MS;
- Trebuchet MS Bold;
- Trebuchet MS Italic;
- Trebuchet MS Bold Italic;
- Verdana;
- Verdana Bold;
- Verdana Italic;
- Verdana Bold Italic.

Причудливые шрифты:

- Comic Sans MS;
- Comic Sans MS Bold,

Еще один моноширинный шрифт:

- Lucida Console.

И два дополнительных символьных шрифта;

- Wingdings;
- Webdings.

Как вы можете заметить, многие (но не все) семейства шрифтов включают обычное (regular), полужирное (bold), курсивное (italic), и полужирное курсивное (bold italic) начертания. К тому же в Windows можно применить эффект подчеркивания и зачеркивания к любому шрифту.

При работе в графической среде пользователи часто применяют слово *шрифт* к тому, что на самом деле является гарнитурой. «Давайте поменяем шрифт *с* Helvetica на Verdana,» — говорят они. К тому же пользователи считают, что курсивное *и* полужирное начертание (так же, как подчеркивание и зачеркивание) — это *атрибуты* или *стили,* которые можно применить к какому-либо шрифту. Например, обычный пользователь не скажет: «Я хочу выделить это слово курсивом, поэтому мне надо переключиться со шрифта Linotype Palatino на Linotype Palatino Italic» Нет, он скорей скажет: «Я хочу поменять стиль этого слова на курсив», — не догадываясь, что это уже другой шрифт,

Windows Forms помогают вам предоставить пользователям шрифты так, как они их себе представляют, т. е. объединив несколько шрифтов (таких как Arial. Arial Bold, Aria) Italic и Arial Bold Italic) в одну гарнитуру (Arial). Хотя в гарнитуру теоретически можно включить большое количество шрифтов с различной толщиной линий и шириной знаков, единственный допустимый вариант — это комбинация полужирного, курсивного, подчеркнутого и зачеркнутого начертаний. В Windows Forms шрифт Arial Black *не считается* частью гарнитуры Arial; Arial Black считается другой гарнитурой, которая, так уж сложилось, не имеет курсивного и полужирного начертаний (в стандартной поставке Windows 2000).

Высота шрифта и межстрочный интервал

Шрифты идентифицируются не только по названиям, но и по вертикальным размерам (кеглям), измеряемым в *пунктах*. В традиционной типографике 1 пункт (пт) равен 0,01384 дюйма. Это близко к 1/72 дюйма, поэтому в компьютерной типографике принято считать, что 1 пт в точности равен 1/72 дюйма.

Кегль обычно определяют как высоту знаков в латинском алфавите, т. е. строчных и заглавных букв от A до Z, исключая диакритические знаки, от верхнего края верхнего выносного элемента до нижнего края нижнего выносного элемента. Например, в кегль укладывается полная высота знаков «bq». Это, конечно, удобное определение, но метрически не совсем точное.

В те времена, когда тексты печатались при помощи металлических литер, кегль был равен высоте литеры, на которой отливались буквы. Сами буквы были чуть меньше кегля. Сейчас такого ограничения нет, и в некоторых случаях размер букв может превышать кегль. Таким образом, кегль — это скорее термин из области типографского дизайна, чем мегрическое понятие. Размер знаков в конкретном шрифте может быть больше или меньше, чем его кегль. Кегль — это не что иное, как приблизительная мера высоты знаков шрифта.

Знание основных применяемых кеглей поможет вам на начальном этапе работы со шрифтами. Большая часть газеты «The New York Times» печатается шрифтом, кегль которого 8 пт; «Newsweek* — 9 пт; эта книга напечатана шрифтом 8,8 пт. Шрифт, применяемый в Windows по умолчанию, имеет кегль 10 пт, в Windows Forms — 8 пт. Как говорилось в главе 7, пользователь сам устанавливает разрешение монитора, и именно оно определяет видимый размер 8- и 10-пунктных шрифтов.

Я уже упоминал, что кегль иногда называют «размер эм» (em size) Этот термин означает размер квадратной металлической литеры, использовавшейся под заглавную букву M во времена литерной печати. Сегодня em применяется в основном для обозначения горизонтальных размеров. Ширина em для какого-либо определенного шрифта численно равна его кеглю. Например, в 14-пунктном шрифте длинное тире (em dash) и широкий пробел (em space) равны 14 пт. En равно половине em. В 14-пунктном шрифте, короткое тире (en dash) и узкий (нормальный) пробел (en space) равны 7 пт.

Межстрочный интервал в тексте обычно больше кегля; как правило, он составляет не менее 115% кегля. Такая пропорция основана на необходимости дополнительного места для диакритических знаков, имеющихся во многих европейских языках. Межстрочный интервал выбирается и из эстетических соображений: текст читать легче, если между строчками есть пустое пространство.

Величину рекомендуемого межстрочного интервала вы можете получить при помощи свойства *Height* и метода *GetHeight* класса *Font.* (Мы поговорим о них при обсуждении свойств класса *Font.)* Для многих шрифтов рекомендуемый межстрочный интервал обычно больше, чем кегль, но меньше величины, которую возвращает метод *MeasureString* класса *Graphics.* Как я уже упоминал, следует избегать свойства *Height,* если вы, конечно, не уверены, что имеете дело только с выводом страницы на экран. Так как свойство *Height* не включает объект *Graphics,* оно не подходит для вывода на принтер или для произвольного преобразования страницы,

Шрифты по умолчанию

Начиная со второй главы, мы используем свойство *Font* класса *Control,* наследуемое всеми подклассами, включая *Form:*

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
Font	*Font*	Чтение/запись

Как вы вскоре увидите, можно присвоить свойство *Font* другому объекту типа *Font,* и все успешные вызовы метода *DrawString,* использующие свойство *Font,* будут применять другой шрифт. (Вы также можете просто создать новый объект *Font* и использовать его прямо в *DrawString.)* Если вы изменили значение свойства *Font,* то можете вернуть ему начальное значение, используя статическое неизменяемое свойство класса *Control:*

Статические свойства *Control* (выборочно)

Тип	Свойство	Доступ
Font	*DefaultFont*	Чтение

Это можно сделать вручную:

```
Font = DefaultFont;
```

Или вы можете просто использовать такой метод:

Методы *Control* (выборочно)

```
void ResetFont()
```

Следующее свойство также имеет отношение к шрифтам:

Свойства *Control* (выборочно)

Тип	Свойство	Доступ
Int	*FontHeight*	Чтение/запись

Вы можете использовать это свойство вместо *FontHeight.* Изменение этого свойства не приведет к изменению свойства *Font.*

Начертания шрифта

В пространстве имен *System.Drawing* определены два важных класса для работы со шрифтами:

- *FontFamily* определен как строка, например «Times New Roman»;
- *Font* — это комбинация из названия шрифта (объект *FontFamily* или символьная строка, определяющая гарнитуру), атрибутов (таких как курсив или полужирный) и кегля;

Начнем с класса *Font*. Есть три категории конструкторов *Font,* основанные на:

- существующем объекте *Font:*
- символьной строке, определяющей гарнитуру:
- объекте *FontFamily*.

Простейший конструктор *Font* создает новый шрифт на базе существующего. Новый шрифт отличается лишь начертанием:

Конструкторы *Font* (выборочно)

```
Font(Font font, FontStyle fs)
```

FontStyle — это перечисление, состоящее из серии однобитных флагов:

Перечисление *FontStyle*

Член	Значение
Regular	0
Bold	1
Italic	2
Underline	4
Strikeout	8

Допустим, мы имеем *шрифт font*, полученный из свойства *Font* формы:

```
Font font = Font;
```

Вы можете создать его курсивное начертание, скажем, *fontItalic*, оператором:

```
Font fontItalic = new Font(font, FontStyle.Italic);
```

Вы можете использовать несколько членов перечисления с оператором поразрядного ИЛИ языка C#:

```
Font fontBoldStrikeout = new Font(font, FontStyle.Bold | FontStyle.Strikeout);
```

Вот программа, которая использует свойство *Font* формы и создает полужирное и курсивное начертания для вывода на экран обычного текста, полужирного и выделенного курсивом.

BoldAndItalic.cs

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

class BoldAndItalic: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new BoldAndItalic());
     }
     public BoldAndItalic()
     {
          Text = "Bold and Italic Text";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          const string strt        = "Thisis sose  ";
          const string str2        = "bold";
          const string str3        - "  text and this is some ";
          const string str4         = "italic";
          const string str5        -  " text.";
          Brush        brush       = new SolidBrush(clr);
          Font         fontRegular = Font;
          Font         fontBold    * new Font(fontRegular, FontStyle.Bold);
          Font         fontItalic  - new Font(fontRegular, FontStyle.Italic);
          float        x           = 0;
          float        y           = 0;

          grfx.DrawString(str1, fontRegular, brush, x, y);
          x += grfx.MeasureString(str1, fontRegular).Width;

          grfx.DrawString(str2, fontBold, brush, x, y);
          x += grfx.MeasureString(str2, fontBold).Width;

          grfx.DrawString(str3, fontRegular, brush, x, y);
          x += grfx.MeasureString(str3, fontRegular).Width; .

          grfx.DrawString(str4, fontItalic, brush, x, y);
          x += grfx.MeasureString(str4, fontItalic).Width;  .

          grfx.DrawString(str5, fontRegular, brush, x, y);  :
     }
}
```

Так как один из аргументов *DrawString* — *Font,* а шрифты определены только как нормальный, полужирный, курсивный и курсивный полужирный, требуется

несколько вызовов *DrawString*, чтобы вывести на экран текст, сочетающий разные начертания. Программа использует *MeasureString* для того, чтобы определить размеры каждого фрагмента текста и расположить текст горизонтально:



Если приглядеться, можно заметить пустые места между фрагментами текста на экране. Я покажу, как этого избежать, когда дойдем до детального описания класса *StringFormat*.

И еще одно предостережение, которое мы обсудим позднее: конструктор *Font* может не работать, если гарнитура, к которой принадлежит шрифт, не поддерживает требуемое начертание. Он сгенерирует исключение, которое правильно написанная программа должна обработать. Этой проблемы нет в программе BoldAndItalic, потому что в ней используется шрифт формы по умолчанию, а он имеет все виды начертаний. Однако BoldAndItalic *не будет* работать с любой произвольной гарнитурой.

Создание шрифта по названию

Следующий набор конструкторов класса *Font* очень удобен: вы определяете шрифт, выбирая гарнитуру, кегль и, если требуется, начертание;

Конструкторы *Font* (выборочно)

```
Font(string strFamily, float fSizeInPoints)
Font(string strFamily, float fSizeInPoints, FontStyle fs)
```

Название гарнитуры, которое вы указываете как первый аргумент конструктора *Font*, может быть хорошо известным нам словосочетанием: «Times New Roman», «Arial», «Courier New», «Comic Sans MS* и т. д. Вот как можно создать 24-пунктный шрифт Times New Roman:

```
Font font = new Font("Times New Roman", 24);
```

Мне нравится такой способ создания шрифтов, думаю, и вы будете отдавать предпочтение этим конструкторам *Font*. Однако у них есть и недостатки.

Используемая гарнитура должна быть типа TrueType или OpenType и установлена на компьютере, где запускается программа. Если шрифта Times New Roman нет или вы неправильно набрали его название, конструктор заменит его на шрифт, используемый по умолчанию (скорее всего на Microsoft Sans Serif). Вы уверены, что шрифт Times New Roman доступен? Скорее всего да, если вы пишете программу для себя. Опять же да, если программа предназначена для внутреннего использования в вашей компании и вы уверены, что на всех рабочих станциях установлен Times New Roman. Но вполне может случиться так (как это ни странно может звучать), что пользователи его удалили. Так как вы стремитесь к тому, чтобы ваш код как можно меньше зависел от платформы, явное использование названий гарнитур — решение ненадежное. Возможно, в будущем программы Windows Forms будут работать в другой среде, с другим и наборами шрифтов. И, хотя скорее всего в такой

среде будет установлено какое-то соответствие между новыми и старыми шрифтами, обеспечивающее корректную работу существующих программ, риск при использовании малоизвестных гарнитур остается.

Чтобы избежать проблем в будущем, лучше всего ограничиться тремя гарнитурами: Times New Roman, Arial и Courier New. Допустимо использовать некоторые псевдонимы, например, «Times» для Times New Roman и «Helvetica» для Arial

Явное указание кегля не влечет за собой таких проблем. Вы знаете, что пользователь настраивает дисплей, исходя из того, что 10-пунктный шрифт удобен Windows Forms устанавливают свойство *Font* исходя из предположения, что 8-пунктный шрифт также читабелен. Все остальное — относительно. Например, 24-пунктный шрифт втрое больше, чем нормальный шрифт Windows Forms,

Так как в одном дюйме содержится 72 пт, 24-пунктный шрифт имеет высоту примерно 1/3 дюйма. *(Приблизительно,* потому что, как я упоминал ранее, кегль — это понятие из области шрифтового дизайна, а не точная величина.) Можно считать, что в пикселах 24-пунктный шрифт имеет размер, равный примерно 1/3 величины свойства *DpiY* объекта *Graphics.*

Название гарнитуры и кегль также могут комбинироваться с начертанием шрифта. Следующая программа создает и отображает на экране 18-пунктные шрифты Courier New, Arial и Times New Roman в нормальном начертании, полужирном, курсивном и полужирном курсивном. Они имеют размер приблизительно 1/4 дюйма.

FontNames.cs

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

class FontNames: PrintableForm
{
    public new static void Main()
    {
        Application.Run(new FontNames());
    }
    public FontNames()
    {
        Text = "Font Names";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        string[]   astrFonts = { "Courier New", "Arial",
                                 "Times New Roman" };
        FontStyle[] afs      = { FontStyle.Regular, FontStyle.Bold,
                                 FontStyle.Italic,
                                 FontStyle.Bold | FontStyle.Italic };
        Brush      brush     = new SolidBrush(clr);
```

см. след. стр.

```
        float       y         = 0;

        foreach (string strFont in astrFonts)
        {
            foreach (FontStyle fs in afs)
            {
                Font font = new Font(strFont, 18, fs);
                grfx.DrawString(strFont, font, brush, 0, y);
                y += font.GetHeight(grfx);
            }
        }
    }
```

Этот класс наследуется от *PrintableForm*, поэтому вы можете вывести шрифты на печать, щелкнув клиентскую область программы. Помните, что координаты, передаваемые методу *DrawString*, указывают на положение верхнего левого угла строки. Следовательно, координаты каждой строки должны быть заданы с учетом высоты предыдущей строки. Программа корректирует эти координаты, добавляя величину, возвращаемую методом *GetHeight* свойства *Font* после вывода строки текста с соответствующим шрифтом.

Для разных шрифтов метод *GetHeight* возвращает разные значения. Добавьте оператор *Console.WriteLine*, если вам интересно их сравнить. Вы увидите, что Times New Roman и Arial дают одинаковую величину, немного большую, чем для Courier New. Для других шрифтов все может быть иначе, поэтому не пытайтесь угадать, а используйте метод *GetHeigbt*. Вот что программа выведет на экран:



Вы можете попробовать заменить свойство *Heigbt* на метод *GetHeight* и посмотреть, что получится при выводе результата на принтер. Межстрочный интервал уменьшится на величину, которая зависит от того, насколько разрешение монитора, измеряемое в точках на дюйм (dpi), отличается от разрешения 100 dpi, установленного для принтера.

Вот листинг аналогичной программы, кагорая выводит на монитор шрифт Times New Roman с размерами от 6 до 12 пт с приращением в 1/4 пт.

FontSizes.cs

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

class FontSizes: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new FontSizes());
     }
     public FontSizes()
     {
          Text = "Font Sizes";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          string strFont = "Times New Roman";
          Brush brush = new SolidBrush(clr);
          float y = 0;
          for (float fSize = 6; fSize <= 12; fSize += 0.25f)
          {
               Font font = new Font(strFont, fSize);
               grfx.DrawString(strFont + " in " + fSize + " points",
                               font, brush, 0, y);
               y += font.GetHeight(grfx);
          }
     }
}
```

В результате работы программы мы увидим на экране:



Пожалуй, самое примечательное на этом изображении — неожиданный скачок ширины штриха знака от 1 к 2 пикселам при размере шрифта 10,5 пт. Такие переходы не так заметны при выводе на устройства с более высоким разрешением, например, на принтеры.

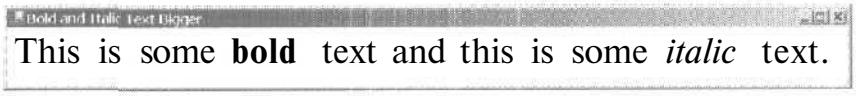
Если вы хотите применить увеличенные шрифты ко всему тексту, выводимому программой на экран в клиентской области, можете изменить свойство *Font* формы прямо в конструкторе. Слегка изменив программу BoldAndItalic, мы получим текстовую строку с 24-пунктным шрифтом.

BoldAndItalicBigger.cs

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

class BoldAndItalicBigger: BoldAndItalic
{
     public new static void Main()
     {
          Application.flun(ftewBoldAndItalicBigger());
     }
     public BoldAndItalicBigger()
     {
          Text += " Bigger";
          Font - new Font("Times New floman", 24);
     }
}
```

Вот что программа выведет на экран:



В этой программе пустые места между фрагментами текста стали еше заметнее, чем в BolclAndltalic. Чтобы их избежать, надо использовать объект *StringFormat,* как я покажу далее,

Единицы измерения размера шрифта

Вы можете не указывать размер шрифта в пунктах. Следующие два конструктора *Font* имеют в качестве одного из аргументов *GraphicsUnit:*

Конструкторы *Font* (выборочно)

```
Font(string strFamily, float fSize, GraphicsUnit gu)
Font(string strFamily, float fSize, FontStyle fs, GraphicsUnit gu)
```

Вы можете использовать любые значения перечисления *GraphicsUnit*, о которых вы узнали в главе 7 при изучении свойства *PageUnit*, кроме одного:

Перечисление *GraphicsUnit*

Член	Значение	Описание
World	0	В единицах глобальных координат
Display	1	Не работает с конструктором *Font*!
Pixel	2	В пикселах
Point	3	В 1/72 дюйма
Inch	4	В дюймах
Document	5	В 1/300 дюйма
Millimeter	6	В миллиметрах

Конструктор;

```
new Font(strFamily, float fSize)
```

идентичен конструктору:

```
new Font(strFamily, float fSize, GraphicsUnit.Point)
```

В самом деле, следующие конструкторы эквивалентны:

```
new Font(strFamily, 72)
new Font(strFamily, 72,    GraphicsUnit.Point)
new Font(strFamily, 1,     GraphicsUnit.Inch)
new Font(strFamily, 25.4f, GraphicsUnit.Millimeter)
new Font(strFamily, 300,   GraphicsUnit.Document)
```

Все эти конструкторы приводят к созданию идентичных 72-пунктных шрифтов. Здесь нет ничего сложного, надо только помнить, что один дюйм равен 72 пт, или 25,4 мм.

Недопустимыми аргументами конструктора *Font* являются *GraphicsUnit Pixel* и *GraphicsUnit.World*. Если тексты вводятся на монитор и используется преобразование страниц по умолчанию (т. е. все координаты и размеры задаются в пикселах), вы можете использовать такие конструкторы для создания 72-пунктных шрифтов:

```
new Font(strFamily, grfx.DpiY, GraphicsUnit.Pixel)
new Font(strFamily, grfx.DpiY, GraphicsUnit.World)
```

В качестве второго аргумента задается количество пикселов в одном дюйме по вертикали.

Следующая программа демонстрирует эквивалентность этих двух конструкторов. Она создает 24-пунктные шрифты семью различными способами.

Я использую здесь 24-, а не 72-пунктные шрифты просто потому, что они лучше подходят для отображения на мониторе. В каждом из конструкторов, величина, передаваемая в качестве второго аргумента — это просто 1/3 величины, применявшейся для создания 72-пунктных шрифтов.

TwentyFourPointScreenFonts.cs

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

class TwentyFourPointScreenFonts: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new TwentyFourPointScreenFonts());
     }
     public TwentyFourPointScreenFonts()
     {
          Text = "Twenty-Four Point Screen Fonts";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, iat cy)
     {
          Brush  ftr«sn  = new SolidBrush(clr);
          float  y       = 0;
          Font   font;
          string strFamily = "Times New Roman";

          font = new Font(strFamily, 24);
          grfx.DrawString("No GraphiesUnit, 24 points", font, brush, 0, y);
          y += font.GetHeight(grfx);

          font = new Font(strFamily, 24, GraphicsUnit.Point);
          grfx.DrawString("GraphicsUnit.Point, 24 units", font, brush, 0, y);
          y += font.GetHeight(grfx);

          font = new Font(strFamily, 1/ 3f, GraphicsUnit.Inch);
          grfx.DrawString("GraphicsUnit.Inch, 1/3 units", font, brush, 0, y);
          y += font.GetHeight(grfx);

          font = new Font(strFamily, 25.4f / 3, GraphicsUnit.Millimeter);
          grfx.DrawString("GraphicsUnit.Millimeter, 25.4/3 units",
                          font, brush, 0, y);
          y += font.GetHeight(grfx);

          font = new Font(strFamily, 100, GraphiesUnit.Document);
          grfx.DrawString("GraphicsUnit.Document, 100 units",
                          font, brush, 0, y);
          y += font.GetHeight(grfx);

          font = new Font(strFamily, grfx.DpiY / 3, GraphicsUnit.Pixel);
          grfx.DrawString("GraphicsUnit.Pixel, " + grfx.DpiY / 3 + " units",
```

см.. след. стр.

```
                font, brush, 0, y);
            y += font.GetHeight(grfx);

            font = new Font(strFamily, grfx.DpiY / 3, GraphicsUnit.World);
            grfx.DrawString("GraphicsUnit.World, " + grfx.DpiY / 3 + " units",
                font, brush, 0, y);
        }
    }
```

На мониторе все семь строк текста имеют одинаковую высоту. Если вы щелкнете клиентскую область программы для вывода на печать, то столкнетесь с проблемой. Первые пять строк текста будут иметь нормальный вид. Конструкторы успешно создадут 24-пункгные шрифты для принтера. Но шрифт последних двух строк будет слишком большим.

Как вы помните, свойства *DpiX* и *DpiY* объекта *Graphics* дают истинное разрешение принтера: 300, 600, 720 или выше. Для последних двух конструкторов *Font* в программе задана 1/3 от этой величины, поэтому второй аргумент будет равен 100, 200, 240 или больше. При выводе на печать страница преобразуется по умолчанию так, как если бы выходное устройство имело разрешение 100 dpi. Такое сочетание размеров шрифта и параметров преобразования страницы приводит к тому, что при печати шрифт имеет высоту 1, 2, 2,3 дюйма или еще большую.

Чтобы создать 72-пункгный шрифт для вывода на печать с преобразованием страницы по умолчанию, используя *GraphicsUniPixel* или *GraphicsUnit.World* применяются конструкторы:

```
new Font(strFamily, 100, GraphicsUnit.Pixel)
new Font(strFamily, 100, GraphicsUnit.World)
```

Чтобы создать 24-пункгный шрифт для вывода на принтер, надо использовать 1/3 от 100. Следующая программа почти не отличается от предыдущей — только последние два конструктора *Font* создают 24-пункгные шрифты, пригодные для вывода на принтер.

TwentyFourPointPrinterFonts.cs

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

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

см. след. стр.

```
    {
        Text = "Twenty-Four Point Printer Fonts";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush  brush     = new SolidBrush(clr);
        float  y         = 0;
        Font   font;
        string strFamily = "Times New Roman";

        font = new Font(strFamily, 24);
        grfx.DrawString("No GraphicsUnit, 24 points", font, brush, 0, y);
        y += font.GetHeight(grfx);

        font = new Font(strFamily, 24, GraphicsUnit.Point);
        grfx.DrawString("GraphicsUnit.Point, 24 units", font, brush, 0, y);
        y += font.GetHeight(grfx);

        font * new Font(strFamily, 1/ 3f, GraphicsUnit.Inch);
        grfx.DrawString("GraphicsUnit.Inch, 1/3 units", font, brush, 0, y);
        y += font.GetHeight(grfx);

        font = new Font(strFamily, 25.4f / 3, GraphicsUnit.Millimeter);
        grfx.DrawString("GraphicsUnit.Millimeter, 25.4/3 units",
                        font, brush, 0, y);
        y += font.GetHeight(grfx);

        font = new Font(strFamily, 100, GraphicsUnit.Document);
        grfx.DrawString("GraphicsUnit.Document, 100 units",
                        font, brush, 0, y);
        y += font.GetHeight(grfx);

        font - new Font(strFamily, 100f / 3, GraphicsUnit.Pixel);
        grfx.DrawString("GraphicsUnit.Pixel, " + 100f / 3 + " units",
                        font, brush, 0, y);
        y += font.GetHeight(grfx);

        font - new Font(strFamily, 100f / 3, GraphicsUnit.World);
        grfx.DrawString("GraphicsUnit.World, " + 100f / 3 + " units",
                        font, brush, 0, y);
    }
}
```

При печати на принтере все семь строк текста будут одинаковой высоты. На мониторе для первых пяти строк размер шрифта будут составлять 24 пт, а для двух последних — возможно, меньше в зависимости от того, насколько разрешение вашего монитора отличается от 100 dpi.

Преобразование единиц измерения

При экспериментировании с параметрами конструктора *Font* возникает вопрос: как единицы измерений шрифтов взаимодействуют с глобальным и страничным преобразованиями? И класс *Font* и класс *Graphics* используют перечисление *GraphicsUnit*. Класс *Font* использует перечисление *GraphicsUnit* в некоторых своих конструкторах, а свойство *PageUnit* класса *Graphics* принимает одно из значений этого перечисления.

Мы уже немного знаем об этом взаимодействии. Давайте проанализируем его и выведем твердые правила.

Заметьте: объекты *Font* не зависят от устройств. Когда вы создаете шрифты, не имеет значения, какое применяется преобразование — глобальное или страничное. Конструктор *Font* ничего об этом не знает. Вы можете создать объекты *Font* в любом месте вашей программы независимо от того, находится объект *Graphics* в области видимости или нет.

В основном используется только три метода, работающих с объектом *Font* (который создается с определенными размерами, измеряемыми в определенных единицах) и объектом *Graphics* (с которым ассоциируется глобальное преобразование и страничное преобразование):

- *DrawString*, метод класса *Graphics*, который имеет аргумент *Font*;
- *MeasureString*, метод класса *Graphics*, который имеет аргумент *Font*;
- *GetHeight*, метод класса *Font*, который имеет аргумент *Graphics*.

Используя эти три метода, вы должны решать проблемы, связанные с преобразованием разных графических единиц измерения. Два других метода — *DrawString Disabled* класса *ControlPaint* и *MeasureCharacterRanges* класса *Graphics* — применяют не так часто.

К этим методам можно применить только три правила. Поэкспериментируйте с программами, чтобы убедиться в их истинности.

Правило 1. *Глобальное преобразование воздействует на все одинаковым образом.*

Допустим, вам нужно отобразить какие-то графические объекты: наборы линий, закрашенные области и текст, использующий шрифты, созданные с различными аргументами *GraphicsUnit*. Затем вы решаете, что все размеры надо увеличить вдвое. Для этого перед любым вызовом графического вывода вы ставите оператор:

```
grfx.ScaleTransform(2, 2);
```

Глобальное преобразование воздействует на все одинаково. Все: каждая линия, каждая закрашенная область и каждая текстовая строка — удваивается в размерах независимо от того, как шрифт был создан. Но размеры, возвращаемые *MeasureString* и *GetHeight*, остаются неизменны,

Правило 2. *Для шрифтов, созданных с использованием метрических размеров (т. е. пунктов, дюймов или миллиметров), преобразование страницы не влияет на физические размеры текста.*

Допустим, вы создаете 72-пунктный шрифт:

```
Font font = new Font("Arial", 72, GraphicsUnit.Point);
```

и выбираете в качестве единиц измерения миллиметры:

```
grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 1;
```

Независимо от преобразования страницы физические размеры текста остаются неизменными. Так как размер шрифта равен 72 пт, это эквивалентно высоте около 25,4 единиц, где в качестве единиц измерения выступают миллиметры.

На что преобразование страницы *действительно* влияет, так это на координаты передаваемые методу *DrawString,* размеры, возвращаемые методом *MeasureString,* и высоту, возвращаемую методом *GetHeight.* Все эти координаты и размеры измеряются в миллиметрах. Пока никаких сложностей быть не должно, так как для расчета координат, передаваемых методу *DrawString,* вы используете в основном *GetHeigh* или *MeasureString.* Достаточно знать, что одни и те же преобразования имеют место, и когда получаете числа от *GetHeight* или *MeasureString,* и когда передаете числа методу *DrawString.*

Если вы вставите операторы в программу TwentyFourPointScreenFonts, чтобы изменить свойства *PageUnit* или *PageScale,* вы увидите, что первые пять строк текста не изменились. Преобразование страницы повлияли на последние строки. Вот третье правило.

Правило 3. *Для шрифтов созданных при помощи единиц* GraphicsUnit.Pixel *или* GraphicsUnit.World, *предполагается, что размеры шрифта измеряются в единицах глобальных координат,*

Иначе говоря, размер шрифта рассматривается просто как координаты и размеры, передаваемые различным методам рисования линий и закрашивания областей класса *Graphics.* Предположим, вы создаете шрифт:

```
Font font = new Font("Arial", 72, GraphicsUnit.World);
```

Преобразование страницы по умолчанию подразумевает размер шрифта 72 пиксела на мониторе и 72/100 дюйма на принтере. Если установить преобразование страницы в миллиметрах:

```
grfx.PageUnit = GraphicsUnit.Millimeter;
grfx.PageScale = 1;
```

то размер шрифта 72 будет считаться равным 72 миллиметрам, т. е. почти 3 дюймам.

Для шрифтов, созданных с помощью *GraphicsUnit.World* и *GraphicsUnit Pixel,* величины, возвращаемые методами *GetHeight* и *MeasureString,* не подвержены влиянию преобразования страницы. Физические размеры шрифта выражаются в единицах, определенных преобразованием страницы, размеры, возвращаемые этими методами, также измеряются в единицах страницы.

Наконец, я не обнаружил разницы между аргументами *GraphicsUnit Pixel* и *GrapbicsUnit.World* при создании шрифтов, хотя они и названы по-разному.

Свойства и методы класса Font

Все свойства класса *Font* доступны только для чтения. Это значит, что нельзя видоизменить шрифт, просто изменив его свойства. Вот полный список свойств класса Font:

Свойства класса *Font*

Тип	Свойство	Доступ	Описание
siring	*Name*	Чтение	Название гарнитуры
FontFamily	*FontFamily*	Чтение	Класс гарнитуры
FontStyle	*Style*	Чтение	Из конструктора
bool	*Bold*	Чтение	Истина, если полужирное начертание
bool	*Italic*	Чтение	Истина, если курсивное начертание
bool	*Underline*	Чтение	Истина, если с подчеркиванием
bool	*Strikeout*	Чтение	Истина, если с зачеркиванием
foat	*Size*	Чтение	Из конструктора
GraphicsUnit	*Unit*	Чтение	Из конструктора
float	*SizeInPoints*	Чтение	Вычисляется из *Size* •
int	*Height*	Чтение	Межстрочный интервал для монитора
byte	*GdiCharSet*	Чтение	Код набора знаков GDI
bool	*GdiVerticalFont*	Чтение	Истина, если шрифт вертикальный

Свойства *Size* и *Unit* просто возвращают величины, использованные при создании шрифта. Свойство *SizeInPoints* вычисляется на основании этих величин. При использовании *GraphicsUnit.Pixel* и *GraphicsUnit.World* вычисления основаны на разрешении монитора.

Если нет необходимости иметь дело с Win32 API, представляет интерес только один метод класса *Font,* на который я уже обращал ваше внимание. Этот метод имеет три версии:

Методы *Font* (выборочно)

```
float GetHeight()
float GetHeight(Graphics grfx)
float GetHeight(float fDpi)
```

Величина, возвращаемая методом *GetHeight,* используется для последовательного размещения строчек текста. Версия метода без аргументов применяется только для вывода на монитор и для преобразования страницы по умолчанию. Вторая, наиболее полезная версия принимает во внимание разрешение и преобразование страницы для выходного устройства. Третья позволяет получить межстрочный интервал на основе гипотетического вертикального разрешения в точках на дюйм.

Если нужно использовать Win32 API, класс *Font* предлагает три статических метода, которые позволяют создать объект *Font: FromHdc, FromHfont* и *FromLogFont. С* другой стороны, класс *Font* не может создать объект *Font* иначе, как используя один из конструкторов класса. Мы уже обсудили пять из этих конструкторов, еще о четырех речь пойдет ниже. Как я покажу в главе 16, класс *FontDialog* создает диалоговое окно, в котором пользователь может выбрать шрифт, и создает объект *Font,* который могут задействовать приложения.

Ниже приведена программа, которая показывает все свойства формы *Font,* а также результат работы трех версий *GetHeight* (третья из которых использует разрешение 100 dpi),

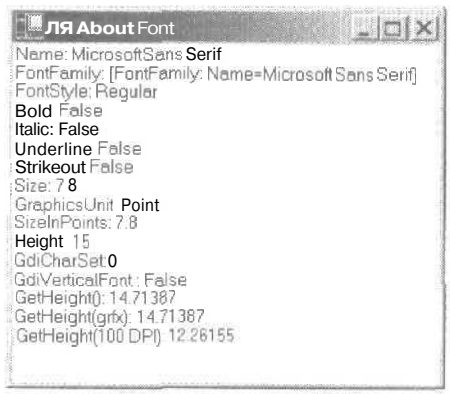
AllAboutFont.cs

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

class AHAboutFont: PrIntableForm
{
     public new static void Main()
     {
          Application.Run(new AllAboutFont());
     }
     public AllAboutFontO
     {
          Text = "All About Font";

     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.DrawString(
                    "Name: "              + Font.Name           + "\n" +
                    "FontFamily: "        + Font.FontFamily     + "\n" *
                    "FontStyle: "         + Font.Style          * "\n" +
                    "Bold: "              + Font.Bold           + "\n" +
                    "Italic: "            + Font.Italic         * "\n" +
                    "Underline: "         + Font,Underline      + "\n" +
                    "Strikeout: "         + Font.Strikeout      + "\n" +
                    "Size; "              + Font.Size           + "\n" +
                    "GraphicsUnit: "      + Font.Unit           + "\n" +
                    "SizeInPoints: "      + Font.SizeInPoints   : "\n" +
                    "Height; "            + Font.Height         + "\n" +
                    "GdiCharSet: "        + Font.GdiCharSet     * "\n" +
                    "GdiVerticalFont : "  + Font.GdiVerticalFont * "\n" +
                    "GetHeight(): "       + Font.GetHeight()    + "\n" +
                    "GetHeight(grfx): "   + Font.GetHeight(grfx) + "\n" +
                    "GetHeight(100 DPI): " + Font.GetHeight(100),
                    Font, new SolidBrush(clr), Point.Empty);
     }
}
```

Щелкните клиентскую область этой программы, чтобы вывести результат на печать. Распечатка будет выглядеть так же, как экранный вид, за исключением предпоследней величины *GetHeight*, для которой результат на распечатке будет совпадать с последней величиной. Вы можете посмотреть на свойства других шрифтов, присвоив свойству формы *Font* интересующий вас шрифт. Вот как выглядит программа на моем компьютере:



Мой монитор настроен на большие шрифты, подразумевающие разрешение 120 dpi. Межстрочный интервал в 14,71 пиксела соответствует примерно 0,123 дюйма или около 9 пт. Это подходящий межстрочный интервал для 8-пунктного шрифта. Если на компьютере установлены маленькие шрифты (подразумевается разрешение 96 dpi), величина межстрочного интервала будет равна 12,45, а свойство *Height* даст величину 13.

Конструктор класса Font, на который мы обратили внимание в первую очередь, создает новый шрифт на основе существующего, но с другим свойством стиля. Иногда необходимо создать текст в одном стиле, но с разными размерами.

Допустим, нужно создать текст, который впишется в заданный прямоугольник, Начнем с того, что выберем шрифт и введем текст. Используем метод *MeasureString*, чтобы определить размеры строки, а затем создадим новый шрифт с размерами, подогнанными к размеру прямоугольника. Следующая программа выводит на монитор текст «Howdy, world!» (вариация традиционного текста, включающая нижний выносной элемент). Текст масштабируется до максимальных размеров, вписывающихся в клиентскую область.

HowdyWorld.cs

```
//---------------------------------------
// HowdyWorld.cs © 2081 by Charles Petzold
//---------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HowdyWorld: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new HowdyWorld());
     }
     public HowdyWorld()
     {
          Text = "Howdy, world!";
```

см. след, стр.

```
            MinimumSize = SystemInformation.MinimumWindowSize + new Size(0,1);
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            Font font = new Font("Times New Roman", 10, FontStyle.Italic);
            SizeF sizef = grfx.MeasureString(Text, font);
            float fScale = Math.Min(cx / sizef.Width, cy / sizef.Height);

            font = new Font(font.Name, fScale * font.SizeInPoints, font.Style);
            sizef = grfx.MeasureString(Text, font);

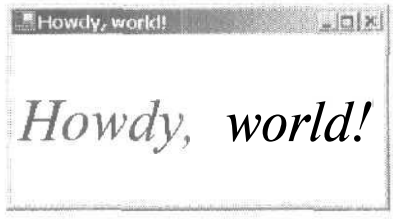
            grfx.DrawString(Text, font, new SolidBrush(clr),
                            (cx - sizef.Width) / 2, (cy - sizef.Height) / 2);
        }
    }
```

Инициализация свойства *MinimumSize* в конструкторе предотвращает «схлопывание» клиентской области, которое привело бы к нулевому размеру шрифта и сообщению об ошибке.

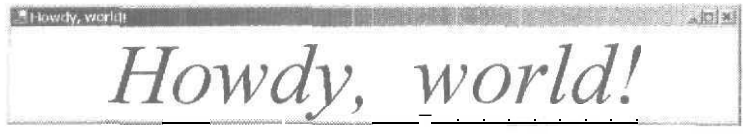
Метод *DoPage* начинает свою работу с создания объекта Font, но используемый им шрифт может быть свойством формы Font или шрифтом, созданным в другом месте программы. Смысл в том, что программе не нужно знать, какие аргументы были изначально использованы для создания шрифта.

Следующий оператор в *DoPage* использует *MeasureString*, чтобы найти длину строки (которая иногда задается как свойство формы *Text*)для шрифта, который мы только что создали. Третий оператор вычисляет коэффициент масштабирования на основе соотношения размера клиентской области (или области печати принтерной страницы) и размера текста. Заметьте: метод *Math.Min* нужен для того, чтобы найти минимальные вертикальный и горизонтальный коэффициенты масштабирования.

Далее, метод *DoPage* создает новый шрифт на основе существующего шрифта, но масштабирует кегль с коэффициентом *fScale*. Затем вновь вызывается метод *MeasureString* (или умножается *Width* и *Height* предыдущего объекта *SizeF* на *fScale*). И, наконец, строка помещается в центре клиентской области.



Если окно слишком широкое, размер шрифта подгоняется к высоте окна:



Можно ли создать шрифт, заполняющий текстом прямоугольную область с любым соотношением длины и высоты? Пусть при этом знаки будут искажены их ширина — меньше или больше обычной. Конструктор *Font* тут бессилен В конструкторах *Font* вы задаете высоту шрифта, а ширина определяется на основании высоты.

Изменить пропорцию знаков текста позволяет глобальное преобразование. Вот вариант программы HowdyWorld, который демонстрирует это.

HowdyWorldFullFit.cs

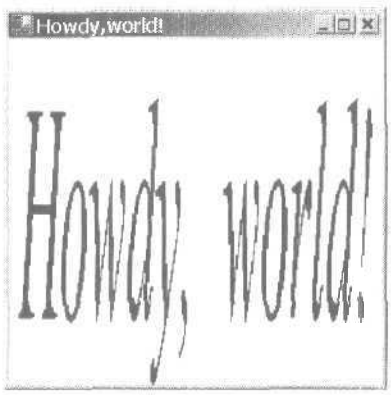
```
//-------------------------------------------
// HowdyWorldFullFit.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HowdyWorldFullFit: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new HowdyWorldFullFit());
     }
     public HowdyWorldFullFit()
     {
          Text = "Howdy, world Г.;
          MinimumSize = SystemInformation.MinimumWindowSize + new Size(0,1);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Font font = new Font("Times New Roman", 10, FontStyle.Italic);
          SizeF sizef = grfx.MeasureString(Text, font);
          float fScaleHorz = cx / sizef.Width;
          float fScaleVert * cy / sizef.Height;

        : grfx.ScaleTransform(fScaleHorz, fScaleVert);

          grfx.DrawString(Text, font, new SolidBrush(clr), 0, 0);
     }
}
```

В этом варианте коэффициенты масштабирования горизонтальных и вертикальных размеров вычисляются раздельно. Эти коэффициенты передаются методу *ScaleTransform* непосредственно. Метод *DrawString* выводит на монитор текст с началом в точке (О, О):



Как обычно, вы можете вывести результат работы программы на печать. Вы увидите знаки, растянутые почти по всей высоте страницы.

На этом рисунке (как и на предыдущем) высота текста меньше полной высоты клиентской области. Пустое место над верхними выносными элементами зарезервировано в основном для диакритических знаков. Если бы в тексте были буквы А, А, или Ä ('\x00C0', '\x00C1' и '\x00C2' в кодировке Unicode), диакритические знаки достигли бы верха клиентской области,

Если используются библиотеки Windows Forms, приложение не может определить, сколько места оставить для диакритических знаков. Не может оно определить и высоту строчного знака над линией шрифта без выносных элементов (например буквы «x»). Эти специфические, так называемые шрифтовые метрики *(font metrics)* не представлены в библиотеках Windows Forms. Как я покажу далее, другие шрифтовые метрики можно вывести из имеющихся. Если вам это действительно нужно, можете приблизительно определить x-размер, используя графические контуры (о них — в главе 15).

Новые шрифты из *FontFamily*

Четыре следующих конструктора аналогичны предыдущим за исключением того, что первым аргументом выступает объект *FontFamily*, а не текстовая строка с названием гарнитуры:

Конструкторы *Font* (выборочно)

```
Font(FontFamily ff, float fSizeInPoints)
Font(FontFamily ff, float fSizeInPoints, FontStyle fs)
Font(FontFamily ff, float fSize, GraphicsUnit gu)
Font(FontFamily ff, float fSize, FontStyle fs, GraphicsUnit gu)
```

Возникает вопрос: как получить объект *FontFamily*?

Во-первых, из имеющегося шрифта. Например, если нужно получить шрифт на основе существующего, но с другими размерами, можно сослаться на свойство *Name* имеющегося шрифта, как было показано в программе HowdyWorld:

```
Font font18 = new Font(font.Name, 18, font.Style);
```

Во-вторых, использовать свойство *FontFamily* в качестве первого аргумента:

```
Font font18 = new Font(font.FontFamily, 18, font.Style);
```

В-третьих, получить объект *FontFamily* можно, используя один из трех конструкторов *FontFamily:*

Конструкторы *FontFamily*

```
FontFamily(string strFamily)
FontFamily(GenericFontFamilies gff)
FontFamily(string strFamily, FontCollection fontcoll)
```

Первый конструктор предполагает, что объект *FontFamily* целиком определяется названием гарнитуры. Действительно, аналогичный оператор:

```
Font font = new Font(strFamily, fSizeInPoints);
```

— это просто короткая форма записи оператора:

```
Font font = new Font(new FontFamily(strFamily), fSizeInPoints);
```

Единственное нестатическое свойство *FontFamily* — это его название:

Нестатические свойства *FontFamily*

Тип	Свойство	Доступ	Описание
String	*Name*	Чтение	Название гарнитуры

Хотя можно создать *Font* не создавая *FontFamily* в явном виде, иногда полезно иметь переменную типа *FontFamily,*

```
FontFamily ff = new FontFamily(strFamily);
```

Чтобы понять, доступен ли какой-либо стиль шрифта, можно применить метод *IsStyleAvailable:*

Методы *FontFamily* (выборочно)

```
bool IsStyleAvailable(FontStyle fs)
```

Не все шрифты TrueType или OpenType имеют полужирное или курсивное начертания, и. если попытаться создать курсивное или полужирное начертания для таких шрифтов, будет сгенерировано исключение. Хуже того, не все шрифты имеют нормальное начертание! Поэтому имеет смысл писать код вот так:

```
if (ff.IsStyleAvailable(FontStyle.Italic))
    fontItalic = new Font(ff, 24, FontStyle.Italic);
else if (ff.IsStyleAvailable(FontStyle.Regular)
    fontItalic - new Font(ff, 24, FontStyle.Regular);
else
    fontItalic = new Font(ff, 24, FontStyle.Bold);
```

Этот код может не привести к созданию курсивного шрифта, но он хоть не сгенерирует исключение.

Как я уже говорил, для многих гарнитур разные начертания шрифтов содержатся в разных файлах. Иногда Windows *синтезирует* курсивное и полужирное начертания, т. е. создает курсивные, полужирные и курсивные полужирные версии, видоизменяя знаки нормального шрифта. Это происходит в случае шрифтов Symbol, Wingdings и Webdings.

Второй конструктор *FontFamily* требует участия элементов перечисления *GenericFontFamilies,* определенного в *System.Drawing.Text*:

Перечисление *GenericFontFamilies*

Член	Значение	Описание
Serif	0	Например, Times New Roman
SansSerif	1	Например, Arial
Monospace	2	Например, Courier New

Я уже предупреждал о возможных проблемах при создании шрифта на основе названия гарнитуры, если этой гарнитуры нет в системе:

```
font = new Font("Times New Roman", 24);
```

Вы будете спокойнее себя чувствовать, применяя конструкцию:

```
font = new Font(new FontFamily(GenericFontFamilies.Serif), 24);
```

Конечно, это выглядит довольно громоздко и не очень элегантно, но есть более короткая равнозначная версия, использующая одно из статических свойств *FontFamily*.

Статические свойства *FontFamily* (выборочно)

Тип	Свойство	Доступ	Описание
FontFamily	*GenericSerif*	Чтение	Например, Times New Roman
FontFamily	*GenericSansSerif*	Чтение	Например, Arial
FontFamily	*GenericMonospace*	Чтение	Например, Courier New

Можно создать шрифт, например, так:

```
font = new Font(FontFamily.GenericSerif, 24);
```

Третий конструктор *FontFamily* мы рассмотрим далее в этой главе.

Метрики дизайна

Программисты, имеющие опыт работы со шрифтами в Windows или в других графических средах, возможно, согласятся, что в Windows Forms не хватает графических средств для работы с метриками шрифтов. До сих пор мы имели дело только с тремя параметрами, относящимися к высоте знаков: кеглем (который, как я подчеркивал, является понятием типографского дизайна и имеет лишь косвенное отношение к метрическим размерам знаков шрифта), максимальным размером знаков, который можно получить из *MeasureString*, и рекомендуемым межстрочным интервалом, определяемым методом *GetHeight*, а в целочисленном виде, подходящем только для мониторов, — *Height*.

Нужна еще одна метрика шрифта, определяющая положение базовой линии (baseline) — линии, выше которой располагаются надстрочные элементы, а ниже — подстрочные элементы. Если требуется совмещать разные шрифты на одной строке, то нужно знать положение базовой линии по отношению к верхней или нижней точке знаков (то, что задается в методе *DrawString*).

На самом деле эта информация содержится в классе *Font Family*. Это не так очевидно и не будет работать для некоторых шрифтов Дальнего и Ближнего Востока, но это лучшее средство, имеющееся в нашем распоряжении.

Класс Font Family содержит четыре метода, позволяющих получить дополнительную метрическую информацию о шрифте. В каждом из этих методов в качестве аргумента выступает перечисление *FontStyle*.

Методы *FontFamily* (выборочно)

```
int GetEmHeight(FontStyle fs)
int GetCellAscent(FontStyle fs)
int GetCellDescent(FontStyle fs)
int GetLineSpacing(FontStyle fs)
```

Эти величины называются метриками дизайна, потому что задаются дизайнером шрифтов (по крайней мере для шрифтов TrueType). Они не связаны с конечным размером шрифта.

Рассмотрим пример, Если создать *FontFamily* на основе гарнитуры Times New Roman и вызвать эти четыре метода с аргументом *FontStyle Regular* (или любой другой величиной *FontStyle*), получатся такие значения:

Метрики дизайна шрифта Times New Roman

Метрика	Значение
Высота эм (Em Height)	2048
Высота с надстрочным элементом (Ascent)	1825
Высота подстрочного элемента (Descent)	443
Высота с надстрочным элементом + высота подстрочного элемента	2268
Межстрочный интервал	2355

Высота эм характеризует сетку, позволяющую дизайнеру шрифта задавать координаты линий и кривых, определяющих внешний вид знаков шрифта. Обычно используется значение 2 048, реже 1 000 или 256.

Высота с надстрочным элементом — это высота знаков шрифта над базовой линией шрифта (включая диакритические знаки), а высота подстрочного элемента — это высота знаков под базовой линией шрифта. Для гарнитуры Times New Roman сумма высоты с надстрочным элементом и высоты подстрочного элемента (которая представлена в таблице) представляет действительную высоту знаков шрифта.

Межстрочный интервал состоит из трех компонентов: высота с надстрочным элементом над базовой линией, высота подстрочного элемента под базовой линией и дополнительное место под подстрочным элементом, как показано на рисунке:



Для некоторых шрифтов значение межстрочного интервала больше, чем сумма высоты с надстрочным элементом и высоты подстрочного элемента, а для некоторых они равны,

Не пытайтесь найти высоту эм на этом рисунке! Высота эм служит для связи с другими метриками дизайна.

Давайте создадим 72-пунктный шрифт Times New Roman и присвоим свойству *PageUnit* значение *GraphicsUnit.Point*. Это значит, что *GetHeight* возвращает значение в пунктах независимо от разрешения графического устройства. Мы получим такие значения (округленные до двух знаков после запятой):

Метрики шрифта Times New Roman

Свойство или метод	**Значение** (в **пунктах)**
font.SizeInPoints	72,00
font.GetHeight(grfx)	82,79

Есть ли соответствие между этой таблицей и предыдущей, показывающей метрики дизайна? Да, потому что величина, полученная из *GetHeight,* выводится из метрик дизайна. Метрика дизайна, названная высотой эм, соответствует кеглю шрифта. Если разделить 72 на 1 048 мы получим 0,03515625. Это коэффициент масштабирования для преобразования координат знаков шрифта в кегль. Умножив его на межстрочный интервал (2355), вы получите 82,79 — величину, возвращаемую методом *GetHeight.*

Смысл в том. что можно применить один и тот же коэффициент отдельно к высоте с надстрочным элементом и к высоте подстрочного элемента метрик дизайна, чтобы получить величины, которых у нас до этого не было;

Метрики Times New Roman

Метрика	Значение метрики дизайна	Значение для 72-пунктного шрифта (в пунктах)	Свойство или **метод**
Высота эм	2048	72,00	*fontSizelnPoints*
Высота с надстрочным элементом	1825	64,16	
Высота подстрочного элемента	443	15,57	
Высота надстрочного элемента + высота подстрочного элемента	2268	79,73	
Межстрочный интервал	2355	82,79	*font.GetHeight(grfx)*

Информацию, полученную из метрик дизайна, можно использовать для позиционирования текста на базовой линии шрифта. Предположим, мы хотим поместить базовую линию горизонтально к центре клиентской области и разместить 144-пунктный шрифт Times New Roman на этой линии. Вот программа, которая делает это.

TextOnBaseline.cs

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

class TextOnBaseline: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new TextOnBaseline());
     }
     public TextOnBaseline()
     {
          Text = "Text on Baseline";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          float ySaseline = cy / 2;
          Pen   pen       = new Pen(clr);
               // Проводим базовую линию через центр клиентской области.

          grfx.DrawLine(pen, 0, yBaseline, cx, yBaseline); '.

               // Создаем 144-пунктный шрифт.

          Font font = new Font("Times New Roman", 144);

               // Получаем и вычисляем некоторые метрики.

          float cyLineSpace = font.GetHeight(grfx);
          int   iCellSpace  = font.FontFamily.GetLineSpacing(font.Style);
          int   iCellAscent = font.FontFamily.GetCellAscent(font.Style);
          float cyAscent    = cyLineSpace * iCellAscent / iCellSpace;

               // Отображаем текст на базовой линий.

          grfx.DrawString("Baseline", font, new SolidBrush(clr),
                          0, yBaseline - cyAscent);
     }
}
```

Величина *cyAscent* — это высота с надстрочным элементом для 144-пунктного шрифта Times New Roman. Если вычесть ее из вертикальных координат базовой линии, получим координаты для размещения текста на базовой линии:



Если приглядеться, можно заметить, что закругленные части некоторых букв располагаются немного ниже базовой линии. Так и должно быть.

Массивы гарнитур

FontFamily имеет еще одно статическое свойство и еще один статический метод. Они схожи и очень важны. Оба возвращают набор объектов *FontFamily,* соответствующий установленным в системе шрифтам TrueType и OpenType:

Статическое свойство *FontFamily* (выборочно)

Тип	Свойство	Доступ
FontFamily[]	*Families*	Чтение

Если выполнить оператор:

```
FontFamily[] aff = FontFamily.Families;
```

каждый элемент массива *aff* будет объектом *FontFamily*. Если вам доводилось сталкиваться с перечислением шрифтов в программах Win32 API, вы оцените простоту такого подхода.

Этот массив позволяет легко получить список доступных шрифтов и вывести на внешнее устройство образцы текста с этими шрифтами:

NaiveFamiliesList.cs

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

class flaiveFaaBlest1st: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new NaiveFamiliesList());
```

см. след. стр.

```
    }
    public NaiveFamiliesList()
    {
        Text = "Naive Font Families List";
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush       brush = new SolidBrush(clr);
        float       y     = 0;
        FontFamily[] aff  = FontFamily.Families;

        foreach (FontFamily ff in aff)
        {
            Font font = new Font(ff, 12);
            grfx.DrawString(ff.Name, font, brush, 0, y);
            y += font.GetHeight(grfx);
        }
    }
}
```

Оператор *foreach* перебирает все элементы *FontFamily*. Для каждого элемента конструктор *Font* создает 12-пунктный шрифт, *DrawString* использует его для вывода на монитор названия гарнитуры, а *GetHeight* готовит вертикальные координаты для вывода следующей строки.

Эта программа может не работать на некоторых компьютерах. На моем конструктор *Font* генерирует исключение для самой первой гарнитуры массива.

К ошибке приводит шрифтовой файл Ahronbd.ttf, который устанавливается с дистрибутивного диска Windows 2000 для поддержки гарнитуры Hebrew. Он содержит шрифт Aharoni Bold, единственный шрифт гарнитуры Aharoni (по крайней мере в Windows 2000). Это значит, что следующий оператор будет работать

```
font = new Font("Aharoni", 12, FontStyle.Bold)
```

А этот:

```
font = new Font("Aharoni", 12)
```

— не будет, так как он равносилен;

```
font = new Font("Aharoni", 12, FontStyle.Regular)
```

а шрифт Aharoni с нормальным начертанием не реализован. (Вот лентяи!)

Эту проблему можно решить двумя способами. Первый — использовать конструкции *try* и *catch*. Поместите код из предыдущего цикла *foreach* в блок *try*. В блоке *catch* выведите название гарнитуры, из-за которой возникают проблемы, со знаком «звездочка», используя свойство формы *Font:*

```
foreach (FontFamily ff in aff)
{
    try
    {
        Font font = new Font(ff, 12);
```

```
            grfx.DrawString(ff.Name, font, brush, 0, y);
            y += font.GetHeight(grfx);
        }
        catch
        {
            grfx.DrawString("* " + ff.Name, Font, brush, 0, y);
            y += Font.GetHeight(grfx);
        }
    }
}
```

Вместе с тем лучше не использовать блоки *try* и *catch,* если есть другие способы. В нашем случае альтернативный подход состоит в применении метода *IsStyleAvailable* класса *FontFamily.* Вот лучшее решение задачи, которое работает независимо от того, установлен ли шрифт Aharoni Bold.

BetterFamiliesList.cs

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

class BetterFamiliesList: PrintableForm
{
    public new static void Main()
    {
        Application. fiynfnew BetterFamiliesList());
    }
    public BetterFamiliesList()
    {
        Text - "Better Font Families List";
    }
    protected override void fioPage{Sraphics grfx, Color clr, int cx, int cy)
    {
        Brush       brush = new SolidBrush(clr);
        float       y     = 0;
        FontFamily[] aff  = FontFamily.Families;

        foreach (FontFamily ff in aff)
        {
            if(ff.IsStyleAvailable(FontStyle.Regular))
            {
                Font font = new Font(ff, 12);
                grfx.DrawString(ff.Name, font, brush, 0, y);
                y += font.GetHeight(grfx);
            }
            else
            {
                grfx.DrawString("* " + ff.Name, Font, brush, 0, y);
```

см. след. стр.

```
                y += Font.GetHeight(grfx);
            }
        }
    }
}
```

Если у вас установлено большое количество шрифтов, этот подход не позволит вам увидеть все шрифты, даже на большом мониторе или на распечатке,

Если не рассматривать полосы прокрутки, правильнее сформатировать список в колонки. Следующая программа так и работает. А если и этого не хватит для вывода всех шрифтов, попробуйте изменить поле *iPointSize* на 10, 8 или 6.

FamiiiesList.cs

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

class FamiliesList: PrintableForm
{
    const int iPointSize * 12;

    public new static void Main()
    {
        Application.fiun(8ewFamiliesList());
    }
    public FamiliesList()
    {
        Text = "Font Families List";

    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        Brush        brush = new SolidBrush(clr);
        float        x = 0, y = 0, fMaxWidth = 0;
        FontFamily[] aff = GetFontFamilyArray(grfx);

        foreach (FontFamily ff in aff)
        {
            Font   font  = CreateSampleFont(ff, iPointSize);
            SizeF sizef = grfx.MeasureString(ff.Name, font);

            fMaxWidth * Math.Max(fMaxWidth, sizef.Width);
        }
        foreach (FontFamily ff in aff)
        {
            Font  font    = CreateSampleFont(ff, iPointSize);
            float fHeight = font.GetHeight(grfx);
```

см. след. стр.

```
                if (y > 0 && y + fHeight > cy)
                {
                    x += fMaxWidth;
                    y  = 0;
                }
                grfx.DrawString(ff.Name, font, brush, x, y);

                y += fHeight;
            }
        }
        protected virtual FontFamily[] GetFontFamilyArray(Graphics grfx)
        {
            return FontFamily.Families;
        }
        Font CreateSampleFont(FontFamily ff, float fPointSize)
        {
            if (ff.IsStyleAvailable(FontStyle.Regular))
                return new Font(ff, fPointSize);

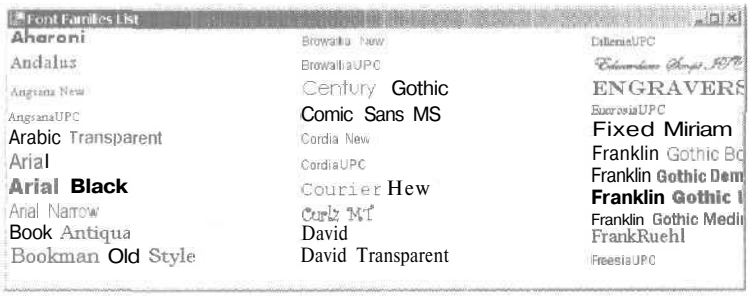
            else if (ff.IsStyleAvailable(FontStyle.Bold))
                return new Font(ff, fPointSize, FontStyle.Bold);

            else if (ff.IsStyleAvailable(FontStyle.Italic))
                return new Font(ff, fPointSize, FontStyle.Italic);

            else
                return Font;
        }
    }
```

В методе *DoPage* два цикла *foreach*. В первом определяется ширина названия каждой гарнитуры с соответствующим шрифтом и запоминается максимальная ширину, второй — использует эту величину для построения колонок.

Обратите внимание на метод *CreateSampleFont* в конце тела класса. Я применяю его в методе *DoPage* вместо конструктора *Font*. *CreateSampleFont* с помощью метода *IsStyleAvailable* определяет, какое использовать начертание — нормальное, курсивное или полужирное. Это позволяет вывести на экран образец текста со шрифтом Aharoni, тогда как предыдущая программа не могла этого сделать. Вот результат работы программы на моем компьютере:



Эти методы используются приложениями, включающими собственные специализированные шрифты. После создания объекта *PrivateFontCollection* и вызова этих методов приложение может через свойство *Families* получить массив объектов *FontFamily*, пригодный для создания объектов *Font*. Или (если программа знает, какие гарнитуры включены в коллекцию) можно применить третий конструктор *FontFamily* (см. список в разделе "Новые шрифты из *FontFamily*») для создания объектов *FontFamity* на основе шрифтовых файлов этой коллекции,

Вариации метода *DrawString*

Мы уже ознакомились с шестью вариантами метода *DrawString:*

Методы *DrawString* класса *Graphics*

```
void DrawString(string str, Font font, Brush brush, PointF ptf)
void DrawString(string str, Font font, Brush brush, float x, float y)
void DrawString(string str, Font font, Brush brush, RectangleF rectf)
void DrawString(string str, Font font, Brush brush, PointF ptf,
                StringFormat sf)
void DrawString(string str, Font font, Brush brush, float x, float y,
                StringFormat sf)
void DrawString(string str, Font font, Brush brush, RectangleF rectf,
                StringFormat sf)
```

Версии, использующие структуру *PointF,* идентичны версиям, которые применяют пару значений с плавающей запятой. Это просто два разных способа задания координат начальной точки новой строки. Все четыре перегруженных метода *DrawString,* которые применяют *PointF* или пару чисел с плавающей запятой, обычно выводят одну строку текста. Однако если текст содержит знак перевода каретки ('\n' или '\x000A'), последующий текст отображается на новой строке.

Две версии *DrawString,* в которых есть аргумент *RectangleF,* переносят слишком широкий текст на новую строку, чтобы он уместился в прямоугольнике. Если одно слово не помещается в прямоугольник, метод разместит столько знаков слова, сколько поместится, а остальные перенесет на следующую строку.

Эти версии свойства *DrawString* распознают знак Unicode '\x00A0' — неразрывный пробел. Он используется вместо нормального пробела, когда перенос тек< та в этом месте будет выглядеть неуместным, например, в строке:

```
"World War\x00A0I"
```

Б этом случае, если знак *I* не помещается на строке, метод *DrawString* перенесет строку после слова *World.*

Эти версии *DrawString* не обрабатывают корректно знак '\x00AD' (мягкий перенос). Обычно он вставляется в местах возможного переноса в длинных словах:

```
"ex\x00ADtraor\x00ADdi\x00ADnary"
```

Теоретически формирователь текста должен переносить строку в этих местах и отобразить знак переноса — дефис. Если в переносе нет нужды, дефис не отображается. Однако метод *DrawString* показывает эти дефисы всегда и даже переносит строку *перед* дефисом,

Эти версии метода *DrawString* также неправильным образом обрабатывают знак '\x2011' в кодировке Unicode (неразрывный дефис), используемый в таких словах, как, например:

```
"T\x2011shirt"
```

Буква T с дефисом в конце строчки будет выглядеть странно. В некоторых шрифтах TrueType этого знака нет. Для шрифтов, включающих его (таких как Lucida Sans Unicode), метод *DrawString* стремится избежать переноса строки на знаке «неразрывный дефис», но перед дефисом вставляет пустое место.

Хотя *DrawString* может переносить слова, он имеет свои ограничения. Так, *DrawString* способен вывести на монитор фрагмент текста в вашем приложении, но он не сможет выделить отдельное слово во фрагменте курсивом или полужирным начертанием, так как аргумент *Font* метода *DrawString* определяет шрифт для всего фрагмента. (Заметим, что *существует* способ выделить часть текста *подчеркиванием* при вызове *DrawString*. Я покажу как это сделать, в программе UnderlinedText.)

Следующее ограничение. Как вы знаете, свойство *Alignment* класса *StringFormat* позволяет управлять горизонтальным выравниванием текста в прямоугольнике. Можно выровнять абзац по центру, по левому или правому краю прямоугольника, но не по ширине. *DrawString* не способен вставить пустые места между словами так, чтобы левый край и правый край были выровнены.

Эту проблему можно решить двумя способами: либо написать собственную программу форматирования текста, либо использовать элемент управления *RichTextBox*, у которого больше возможностей форматирования, чем у *DrawString*. Мы рассмотрим *RichTextBox* в главе 18.

Сглаживание текста

В главе 5 показано, как свойства *SmoothingMode* и *PixelOffsetMod* класса *Graphics* позволяют устранять контурные неровности при рисовании линий и кривых. Windows также позволяет управлять сглаживанием при выводе текста на экран, *Эту* функциональность можно включить, установив флажок Smooth Edges Of Screen Fonts (Сглаживать неровности экранных шрифтов) на вкладке Effects (Эффекты) диалогового окна Display Properties (Свойства экрана).

Можно также переопределить пользовательскую настройку, установив свойство *TextRenderingHint* класса *Graphics*:

Свойства *Graphics* (выборочно)

Тип	Свойство	Доступ
TextRenderingHint	*TextRenderingHint*	Чтение/запись

TextRenderingHint — это перечисление, определенное в пространстве имен *System.Drawing.Text*.

ClearType — технология, аналогичная устранению контурных неровностей, но использующая специальный способ размещения цветных точек в жидкокристаллических дисплеях.[2]

[2] Об этой технологии время от времени вспоминают на протяжении последних двадцати лет. См. обсуждение этой технологии и ее истории на сайте Стива Гибсона: *http://grc.com/cleartype.htm*.

Перечисление *TextRenderingHint*

Член	Значение	Описание
SystemDefault	0	По умолчанию
SingleBitPerPixelGridFit	1	Без сглаживания, с подгонкой под координатную сетку
SingleBitPerPixel	2	Без сглаживания, без подгонки под координатную сетку
AntiAlias	3	Со сглаживанием, без подгонки под координатную сетку
AntiAliasGridFit	4	Со сглаживанием, с подгонкой под координатную сетку
ClearTypeGridFit	5	ClearType для жидкокристаллических мониторов

Следующая программа демонстрирует использование всех шести значений перечисления.

AntiAliasedText.cs

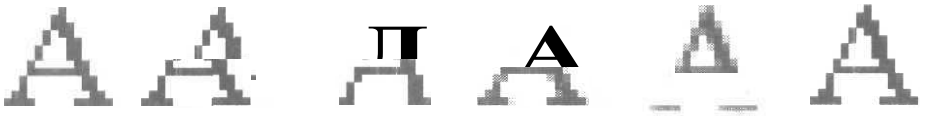
```
//-----------------------------------------
// AntiAliasedText.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;

class AntiAliasedText: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new AntiAliasedText());
     }
     public AntiAliasedText()
     {
          Text = "Anti-Aliased Text";
          Font = new Font("Times New Roman", 12);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush brush = new SolidBrush(clr);
          string str = "A";
          int cxText = (int) grfx.MeasureString(str, Font).Width;

          for (int i = 0; i < 6; i++)
          {
               grfx.TextRenderingHint = (TextRenderingHint)i;
               grfx.DrawString(str, Font, brush, 1 * cxText, 0);
          }
     }
}
```

Программа выводит прописную букву A шесть раз, используя шесть значений *TextRenderingHint*. Вы можете скопировать это изображение в буфер обмена, на-

жав Alt+Print Scrn, затем вставить изображение в графическую программу и увеличить его:



В зависимости от параметров вашего дисплея первое значение (*SystemDefault*) будет соответствовать либо второму (*SingleBitPerPixelGridFit*), либо четвертому (*AntiAlias*), либо, если вы используете шрифты ClearType на жидкокристаллическом мониторе, первое значение будет соответствовать шестому *(ClearTypeGridFit).* Ни второе, ни третье значения перечисления (*SingleBitPerPixel*) не сглаживают текста. Заметьте, что в третьей букве А несколько пустых пикселов. Преобразователь шрифтов в растровый формат применяет более строгие критерии для определения того, должен ли пиксел быть окрашен.

Четвертый и пятый образцы (*AntiAliasGridFit*) используют сглаживание. Пикселы окрашиваются в оттенки серого цвета с разной интенсивностью в зависимости от степени удаленности от теоретического контура знака.

При использовании *ClearTypeGridFit* пикселы слева и справа от штрихов окрашиваются в разные цвета в зависимости от расположения цветных точек по горизонтали на жидкокристаллических мониторах, Не применяйте ClearType на обычных мониторах. Некоторые жидкокристаллические мониторы позволяют пользователю поворачивать экран на 90° и работать с ним в портретном режиме. ClearType при этом работать не будет.

Определение размеров строки

Начиная с главы 2, *MeasureString* используется для определения размеров текстовой строки для точного позиционирования текста. Но *MeasureString* способен и на другое. Как вы, возможно, заметили, метод *DrawString* выводит текст, не изменяя фона. (Это не установлено по умолчанию в Windows GDI.) Если нужно убрать фон, вы можете задать координаты точки вывода текста в сочетании с величиной *SizeF,* которую возвращает метод *MeasureString* при вызове *FillRectangle.*

У метода *MeasureString* 7 версий:

Методы *MeasureString* класса *Graphics*

```
SizeF MeasureString(string str, Font font);
SizeF MeasureString(string str, Font font, int iWidth);
SizeF MeasureString(string str, Font font, SizeF sizef);
SizeF MeasureString(string str, Font font, int iWidth,
                    StringFormat strfmt);
SizeF MeasureString(string str, Font font, SizeF sizef,
                    StringFormat strfmt);
SizeF MeasureString(string str, Font font, PointF ptfOrigin,
                    StringFormat strfmt);
SizeF MeasureString(string str, Font font, SizeF sizef,
                    StringFormat strfmt,
                    out int iCharacters, out int iLines);
```

Мы уже давно используем первую версию *MeasureString*. Она возвращает ширину и высоту строки текста с заданным шрифтом. Свойство *Height* объекта *SizeF*, возвращаемое этим методом, иногда равно значению, полученному из метода *GetHeight* класса *Font*, а если текст включает знаки перевода каретки, *Height* будет кратно числу *GetHeight*.

Вторая версия *MeasureString* включает третий аргумент — ширину текста. Она удобна, если нужно вывести строку, используя метод *DrawString* с аргументом *RectangleF*, и допускается перенос строки. Свойство *Width* объекта *SizeF*, возвращаемое методом *MeasureString*, всегда меньше или равно аргументу *fffl'idth*. СВОЙСТРО *Height*, деленное на величину *GetHeight*, дает количество строк,

Третья версия *MeasureString* имеет аргумент *SizeF*, определяющий как ширину так и высоту. Если его свойство *Width* равно аргументу *fWidth* второй версии *Measur* • - *String* и если свойство *Height* определено для всех строчек текста, эта версия вернет такое же значение, как и вторая. В противном случае свойство *Height* возвращаемого объекта *SizeF* будет равно свойству *Height* аргумента *SizeF* и свойство *Width* возвращаемого объекта *SizeF* будет показывать максимальную ширину текста, который может поместиться в прямоугольник заданных размеров.

Четвертая, пятая и шестая версии аналогичны второй и третьей за исключением того, что они включают аргумент *StringFormat*. Если вам понадобится аргумент *StringFormat* при вызове *DrawString*, используйте один из методов *MeasureString*

Последняя версия *MeasureString* имеет два аргумента, в которых возвращается дополнительная информация. Они показывают количество знаков и строк текста, отображаемых методом *DrawString*, если ему передается структура *RectangleF* того же размера, как структура *SizeF*, и такой же объект *StringFormat*.

Метод *MeasureString* с этими аргументами очень удобен, когда для вывода на монитор одного блока текста требуется множество вызовов *DrawString*. Допустим, вы хотите использовать *DrawString*, чтобы распечатать на принтере текст, не помещающийся на одну страницу. *MeasureString* позволяет определить размер текста, который может поместиться на первой странице, и на основе этих данных формировать вторую страницу Я покажу, как использовать эту версию *MeasureString*, в программе TextColumns в конце главы.

Параметры *StringFormat*

Методы *DrawString* и *MeasureString* могут в качестве необязательного аргумента принимать объект типа *StringFormat*. Этот аргумент предлагает массу вариантов вывода текста на экран. Вы можете создать объект *StringFormat*, используя один из конструкторов:

Конструкторы *StringFormat*

```
StringFormat()
StringFormat(StringFormat strfmt)
StringFormat(StringFormatFlags sff)
StringFormat(StringFormatFlags sff, int iLanguage)
```

Вторая версия по существу клонирует существующий объект *StringFormat*. Третья и четвертая — создают объект *StringFormat* на основе комбинации значений пе-

речисления *StringFormatFlags*. Перечисление *StringFormatFlags* используется также при установке свойства *FormatFlags* класса *StringFormat*:

Свойства *StringFormat* (выборочно)

Тип	Свойство	Доступ
StringFormatFlags	*FormatFlags*	Чтение/запись

Перечисление *StringFormatFlags* — это набор битовых флагов:

Перечисление *StringFormatFlags*

Член	Значение
DirectionRightToLeft	1
DirectionVertical	2
FitBlackBox	4
DisplayFormatControl	32
NoFontFallback	1024
MeasureTrailingSpaces	2048
NoWrap	4096
LineLimit	8192
NoClip	16384

Некоторые из этих флагов мы рассмотрим позднее при обсуждении некоторых программ. *MeasureTrailingSpaces* используется в программе TypeAway в главе 6, программу BoldAndItalicTighter мы обсудим чуть ниже, а программу TrimmingTheText с флагами *NoWrap* и *NoClip* — далее в этой главе.

Если новый объект *StringFormat* создается конструктором по умолчанию, свойство *FormatFlags* равно 0. Замечу, что эти значения перечисления однобитовые, поэтому вы можете комбинировать их оператором логического ИЛИ языка C#. Например:

```
StringFormat strfrnt = new StringFormat(StringFormatFlags.DirectionVertical |
                                        StringFormatFlags.NoClip);
```

При установке свойства *FormatFlags* я бы рекомендовал применить оператор |=:

```
strfmt.FormatFlags |= StringFormatFlags.NoWrap;
```

При этом вы никогда случайно не сбросите один из других флагов, которые вы, возможно, установили ранее.

Объект *StringFormat* можно также получить, используя одно из этих статических свойств:

Статические свойства *StringFormat*

Тип	Свойство	Доступ
StringFormat	*GenericDefault*	Чтение
StringFormat	*GenericTypographic*	Чтение

Если рассмотреть свойства объектов *StringFonnat,* возвращаемых этими статическими свойствами, можно увидеть, что *GenericDefault* возвращает такой же объект *StringFormat,* который создается конструктором по умолчанию. Свойство *Generic-*

Typographic возвращает объект, который имеет флаги *FitBlackBox*, *LineLimit* и *NoClip*, а также некоторое значение свойства *Trimming*, которое мы обсудим позднее.

Однако объект *StringFormat*, возвращаемый свойством *GenericTypographic*, оказывает дополнительное воздействие на методы *DrawString* и *MeasureString*, недостижимое при использовании открытых свойств *StringFormat* и флагов. Этот эффект мы и рассмотрим.

Преобразование знаков и текста

Часть GDI+, отвечающая за работу с текстом, не зависит от устройства вывода. С практической точки зрения, это значит, что *MeasureString* возвращает размеры текста, не зависимые от устройства вывода. Если установить одинаковое преобразование страницы для экрана и принтера, *MeasureString* вернет одинаковое значение для заданной текстовой строки и заданного шрифта. Такая согласованность облегчает форматирование текста, чтобы он выглядел на экране так же, как при печати на принтере.

Однако желаемый принцип WYSIWYG трудно реализовать. Проблема в пикселах. Когда знаки шрифта растеризуются, координаты, выраженные числами с плавающей запятой, должны быть округлены с учетом дискретности пикселов. Такая *подгонка под координатную сетку* (grid fitting) требует специальных действий для сохранения читабельности шрифта. Скажем, два вертикальных штриха в заглавной букве Я должны быть одинаковой ширины. Даже для шрифтов маленьких размеров эти два штриха должны иметь ширину не менее 1 пиксела и должны быть разнесены не менее чем на 1 пиксел. (Если кегль слишком мал в сравнении с разрешением устройства вывода, такое требование теряет смысл, потому что текст не будет разборчивым в любом случае.)

Иногда, особенно для малых кеглей на устройствах с низким разрешением (таких как монитор), подгонка может привести к тому, что преобразованные шрифты будут гораздо больше, чем их теоретические размеры. Если собрать такие знаки (скажем, строчные знаки ;' гарнитуры Arial) вместе, это может привести к тому, что строка текста будет выглядеть значительно больше на мониторе, чем на принтере. (В статье на сайте *http://www.gotdotnet.com/team/windowsforms/gdiptext.aspx* эта проблема рассмотрена подробнее).

Соединяя разные фрагменты текста с помощью *DrawString* и *MeasureString* (как, например, в программе BoldAndItalic), что бы вы предпочли: иметь пустое дополнительное место между фрагментами или чтобы фрагменты текста перекрывались? Думаю, вы согласитесь, что наложение текста неприемлемо. Чтобы его предотвратить, в методы *DrawString* и *MeasureString* предусмотрительно заложена возможность включения небольшого пустого дополнительного места. Таким образом, если растеризатору потребуется больше места для прорисовки какого-либо шрифта, он его получит.

По умолчанию у объекта *SizeF*, возвращаемого *MeasureString*, есть свойство *Height* на 1/8 размера эм больше, чем теоретически нужно, и свойство *Width* на 1/3 размера эм больше, чем теоретическая ширина. (Помните: размер эм численно равен кеглю шрифта. Например, для 24-пунктного шрифта 1/3 размера эм равна 8 пт.) По умолчанию *DrawString* начинает выводить текст на монитор на 1/6 размера эм ниже заданной вертикальной координаты. Фактически *MeasureString* обозна-

чает прямоугольник, который шире, чем предполагаемая текстовая строка на 1/6 размера эм справа и слева.

Поэтому в BoldAndItalic и, более наглядно, в BoldAndItalicBigger видны лишние пустые места между соединенными фрагментами текста.

Следует помнить, что проблема подгонки к координатной сетке имеет место только для шрифтов с маленькими кеглями, которые выводятся на устройства с низким разрешением. Чтобы добиться независимости от устройства, добавление пустого места в *DrawString* и *MeasureString* должно реализовываться одинаково для устройств как с высоким, так и с низким разрешением. Дополнительное пространство должно быть также пропорциональным размеру шрифтов. Для 720-пунктного шрифта *MeasureString* должен вернуть размер текста в 100 раз больше, чем для 7,2-пунктного шрифта,

Что делать, если не желательно оставлять пустые места в тексте? Использовать объект *StringFormat*, основанный на *StringFormat.GenericTypographic*. Вот версия программы BoldAndItalic, которая имеет дело с таким объектом:

BotdAndItaltcTighter.cs

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

class BoldAndItalicTighter : PrintableForm
{
    public new static void Main()
    {
        Application.Run(new BoldAndItalicTighter());
    }
    public BoldAndItalicTighter()
    {
        Text = "Bold and Italic (Tighter)";
        Font = new Font("Times New Roman", 24);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        const string str1       = "This is some ";
        const string str2       = "bold";
        const string str3       = " text, and this is some ";
        const string str4       = "italic";
        const string strS       = " text.";
        Brush        brush      = new SolidBrush(clr);
        Font         fontRegular = Font;
        Font         fontBold   = new Font(fontRegular, FontStyle.Bold);
        Font         fontItalic = new Font(fontRegular, FontStyle.Italic);
        PointF       ptf        = new PointF(0, 0);
```

см. след. стр.

```
        StringFormat strfmt      = StringFormat.GenericTypographic;
        strfmt.FormatFlags      |= StringFormatFlags.MeasureTrailingSpaces;

        grfx.DrawString(str1, fontRegular, brush, ptf, strfmt);
        ptf.X += grfx.MeasureString(str1, fontRegular, ptf, strfmt).Width;

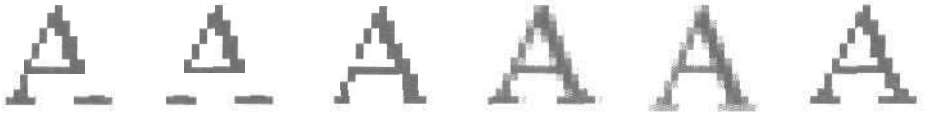
        grfx.DrawString(str2, fontBold, brush, ptf, strfmt);
        ptf.X += grfx.MeasureString(str2, fontBold, ptf, strfmt).Width;

        grfx.DrawString(str3, fontRegular, brush, ptf, strfmt);
        ptf.X += grfx.MeasureString(str3, fontRegular, ptf, strfmt).Width;

        grfx.DrawString(str4, fontItalic, brush, ptf, strfmt);
        ptf.X += grfx.MeasureString(str4, fontItalic, ptf, strfmt).Width;

        grfx.DrawString(str5, fontRegular, brush, ptf, strfmt);
    }
}
```

Заметьте: в программе установлен флаг *MeasureTrailingSpaces*.Результат выглядит неплохо:



В программе BoldAndItalicTighter можно обойтись без объекта *GenericTypographic*, потому что заранее известно, что шрифт достаточно велик и несколько пикселов не имеют значения. Если вы хотите использовать *GenericTypographic* с малыми размерами шрифта на мониторе, вы также должны применить сглаживание краевых эффектов. При сглаживании не требуется подгонка к координатной сетке, потому что каждый пиксел окрашивается в зависимости от его удаленности от теоретического контура,

Горизонтальное и вертикальное выравнивание

С классом *StringFormat* мы впервые столкнулись в главе 3, где он служил для размещения текста в центре клиентской области. Два свойства *StringFormat*, которые влияют на выравнивание текста, показаны ниже:

Свойства *StringFormat* (выборочно)

Тип	Свойство	Доступ	Описание
StringAlignment	*Alignment*	Чтение/запись	Выравнивание по горизонтали
StringAlignment	*LineAlignment*	Чтение/запись	Выравнивание по вертикали

Эти свойства имеют тип *StringAlignment* — перечисление, состоящее из трех членов,

Параметры выравнивания работают по-разному в зависимости от того, какой объект вы указываете при вызове метода *DrawString*: *PointF* или *RectangleF*. Сначала рассмотрим версию с *RectangleF,* Следующая программа применяет прямоуголь-

ную клиентскую область в девяти вызовах *DrawString,* каждый из которых использует различную комбинацию свойств *Alignment* и *LineAlignment.* Чтобы сделать ее интереснее, текст, выводимый *DrawString,* содержит знак перевода каретки.

Перечисление *StringAlignment*

Член	Значение	Описание
Near	0	Обычно слева или вверху
Center	1	Всегда по центру
Far	2	Обычно справа или внизу

StringAlignmentRectangle.cs

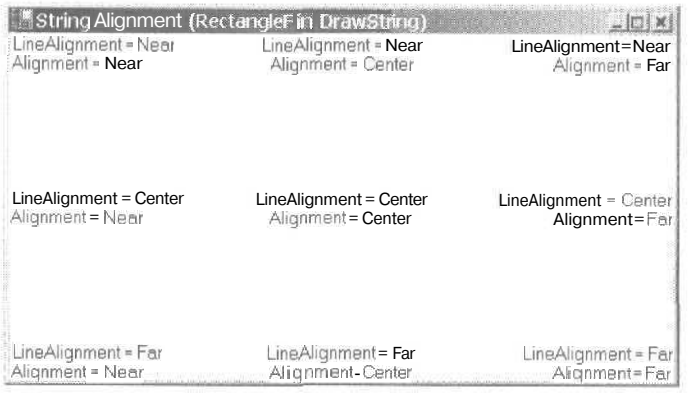
```
//-----------------------------------------------------
// StringAlignmentRectangle.cs © 2001 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class StringAlignmentRectangle : PrintableForm
{
     public new static void Main()
     {
          Application.Run(new StringAlignmentRectangle());
     }
     public StringAlignmentRectangle()
     {
          Text = "String Alignment (RectangleF in DrawString)";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush        brush  = new SolidBrush(clr);
          RectangleF   rectf  = new RectangleF(0, 0, cx, cy);
          string[]     strAlign = { "Near", "Center", "Far" };
          StringFormat strfmt = new StringFormat();

          for (int iVert = 0; iVert < 3; iVert++)
          for (int iHorz = 0; iHorz < 3; iHorz++)
          {
               strfmt.LineAlignment = (StringAlignment)iVert;
               strfmt.Alignment     = (StringAlignment)iHorz;

               grfx.DrawString(
                    String.Format("LineAlignment = {0}\nAlignment = {1}",
                                  strAlign[iVert], strAlign[iHorz]),
                    Font, brush, rectf, strfmt);
          }
     }
}
```

Три возможных значения свойства *Alignment* соответствуют выравниванию текста по левому краю, по правому или по центру прямоугольника. Три возможных значения *LineAlignment* указывают, что текст выводится сверху, по центру ил и внизу прямоугольника:



Такой аккуратный, хорошо упорядоченный вид невозможен при использовании того же объекта *PointF* в методе *DrawString*, так как некоторые фрагменты будут перекрываться с другими.

При вызове метода *DrawString* следующая программа устанавливает объект *PointF* в центре клиентской области, но использует только четыре комбинации свойств *Alignment* и *LineAlignment*.

StringAlignmentPoint.cs

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

class StringAlignmentPoint: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new StringAlignmentPoint());
     }
     public StringAlignmentPoint()
     {
          Text = "String Alignment (PointF in DrawString)";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush      brush    = new SolidBrush(clr);
          Pen        pen      = new Pen(clr);
          string[]   strAlign = { "Near", "Center", "Far" };
```

см. след. стр.

```
            StringFormat strfmt  = new StringFormat();

            grfx.DrawLine(pen, 0, cy / 2, cx, cy / 2);
            grfx.DrawLine(pen, cx / 2, 0, GX / 2, cy);

            for (int iVert = 0; iVert < 3; iVert += 2}
            for (int iHorz = 0; iHorz < 3; iHorz += 2)
            {
                strfmt.LineAlignment = (StringAlignment)iVert;
                strfmt.Alignment     = (StringAlignment)iHorz;

                grfx.DrawString(
                    String.Format("LineAlignment = {0}\nAlignment - {1}",
                                  strAlign[iVert], strAlign[iHorz]),
                    Font, brush, cx / 2, cy / 2, strfmt);
            }
        }
    }
```

Обратите внимание на два оператора *for*. Переменные *iVert* и *iHorz* принимают только значения 0 и 2, и программа использует только четыре комбинации свойств *Alignment* и *LineAlignment*, чтобы вывести на монитор такое изображение:



Объект *PointF*, передаваемый методу *DrawString*, определяет центр клиентской области. *DrawString* размещает две строчки текста относительно этой точки в зависимости от значений свойств *Alignment* и *LineAlignment,*

Если заменить первый оператор *for* в этой программе на:

```
for (int iVert = 1; iVert < 3; IVert += 2)
```

и перекомпилировать программу, то можно увидеть, как присваивание значения *StringAlignmentCenter* свойству *LineAlignment* приводит к тому, что вертикальная координата, передаваемая *DrawString*, определяет положение центра текста. В этом случае это центр двух строчек текста по вертикали:



Если же вместо этого поменять второй оператор *for* на;

```
for (int iHorz = 1; iHorz < 3; iHorz += 2)
```

две строчки текста займут центральное положение по горизонтали:



Если изменить оба цикла/or мы получим конечный вариант программы. Две строкл текста будут отцентрованы по вертикали и горизонтали относительно центра клиентской области.



Вывод «горячих» клавиш

Свойство *HotkeyPrefix* класса *StringFormat* определяет, как метод *DrawString* интерпретирует амперсанты:

Свойства *StringFormat* (выборочно)

Тип	Свойство	Доступ
HotkeyPrefix	*HotkeyPrefix*	Чтение/запись

Как *DrawString* интерпретирует амперсанты? Этот вопрос может показаться немного странным, если не знать, что амперсант имеет специальное назначение в тексте, используемом в меню, кнопках и других элементах управления: он означает, что следующий за ним знак должен быть подчеркнут и будет использоваться для быстрого доступа к элементу управления.

Свойству *HotkeyPrefix* присваивается одно из следующих значений перечисления *HotkeyPrefix*, определенных в пространстве имен *System.Drawing.Text*:

Перечисление *HotkeyPrefix*

Член	Значение	Описание
None	0	«&File» ⇒ «&File» (по умолчанию)
Show	1	«&File» ⇒ «File»
Hide	2	«&File» ⇒ «File»

По умолчанию амперсанты не имеют специального назначения и просто выводятся как знак амперсанта. Значение *Show* скрывает знак амперсанта и приводит к подчеркиванию следующего знака. Значение *Hide* скрывает знак амперсанта, но не приводит к подчеркиванию следующего знака.

Даже если вы не работаете с текстом в меню или в элементах управления, вы можете задействовать это свойство для подчеркивания отдельных букв или слов в блоке текста, передаваемом методу *DrawString*. Следующая программа демонстрирует эту возможность.

UnderlinedText.cs

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

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

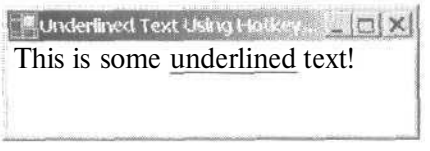
см. след. стр.

```
    public UnderlinedText()
    {
        Text = "Underlined Text Using HotkeyPrefix";
        Font = new Font("Times New Roman", 14>;
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int ey)
    {
        string       str    = "This is some &u&n&d&e&r&l&i&n&e&d text!";
        StringFormat strfmt = new StringFormat();

        strfmt.HotkeyPrefix * HotkeyPrefix.Show;

       ' grfx.DrawString(str, Font, new SolidBrush(clr), 0, 0, strfmt);
    }
}
```

Строка текста в листинге этой программы выглядит малопривлекательно, но результат работы программы смотрится неплохо:



Увы, нет средств, которые могли бы выделить отдельные слова курсивом или полужирным начертанием в блоке текста, выводимым *DrawString*.

Я использую *HotkeyPrefix* по прямому назначению в программе OwnerDrawMenu в главе 14.

Отсечение и подгонка текста

Используя версию метода *DrawString* с аргументом *RectangleF*, вы определяете не только правую границу текста, которая влияет на переносы, но и нижнюю — ограничивающую общее количество выводимого текста,

А если текст будет слишком велик и не поместится в прямоугольник?

Рассмотрим сначала стандартный случай: когда объект *StringFormat* не используется в качестве последнего аргумента *DrawString*. Если высота прямоугольника равна целому числу межстрочных интервалов, целое число строчек текста сможет поместиться в прямоугольнике. Последняя строчка текста будет содержать столько знаков, сколько сможет поместиться по ширине прямоугольника. Именно *знаков,* а не целых слов. Чтобы увидеть, как все это выглядит, модифицируем программу HuckleberryFinn из главы 3, которая ограничивает текст половиной клиентской области по ширине и по высоте,

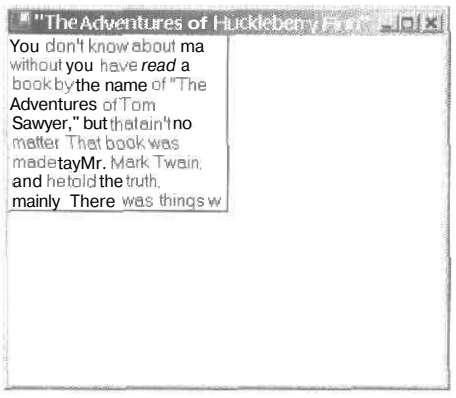
HuckleberryFinnHalfHeight.cs

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

class HuckleberryFinnHalfHeight: Form
{
     public static void Main()
     {
          Application.Run(new HuckleberryFinnHalfHeight());
     }
     public HuckleberryFinnHalfHeight()
     {
          Text = "\"TheAdventures of Huckleberry Finn\"";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          fiesizefiedraw = true;
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics  grfx = pea.Graphics;
          int       cx   = ClientSize.Width;
          int       cy   = ClientSize.Height;
          Pen       pen  = new Pen(ForeColor);
          Rectangle rect * new Rectangle(0, 0, cx / 2, cy / 2);
          grfx.DrawString(
               "You don't know about me, without you " +
               "have read a book by the name of \"The " +
               "Adventures of Tom Sawyer,\" but that " +
               "ain't no matter. That book was made by " +
               "Mr. Mark Twain, and he told the truth, " +
               "mainly. There was things which he " +
               "stretched, but mainly he told the truth. " +
               "That is nothing. I never seen anybody " +
               "but lied, one time or another, without " +
               "it was Aunt Polly, or the widow, or " +
               "maybe Mary. Aunt Polly\x2014Tom's Aunt " +
               "Polly, she is\x2014and Mary, and the Widow " +
               "Douglas, is all told about in that book" +
               "\x2014which is mostly a true book; with " +
               "some stretchers, as I said before.",
               Font, new SolidBrush(ForeColor), rect);
          grfx.DrawLine(pen, 0,       cy / 2, cx / 2, oy / 2);
          grfx.DrawLine(pen, cx / 2, 0,       cx / 2, cy / 2);
     }
}
```

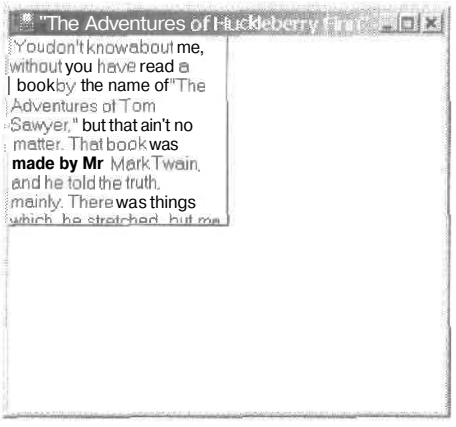
Программа также рисует линии, обозначающие границы выводимого текста.

Если области прямоугольника не хватает для целого абзаца, последняя строчка в клиентской области может заканчиваться частью слова:



При увеличении высоты прямоугольника приходит момент, когда *DrawString* решает, что места для еще одной строки текста уже достаточно. Это наступает быстрее, чем можно было бы предположить! Метод *DrawString* выводит дополнительную строку текста, когда высота прямоугольника увеличивается на 25% от размера межстрочного интервала.

Последняя строчка обрезается по краю прямоугольника.



Хотя она большей частью обрезана, можно увидеть, что эта новая строчка включает часть другого слова — первые две буквы слова *«mainly»*.

Изменить этот алгоритм по умолчанию позволяет свойство *Trimming* класса *StringFormat*:

Свойства *StringFormat* (выборочно)

Тип	Свойство	Доступ
StringTrimming	*Trimming*	Чтение/запись

Свойство *Trimming* определяет, как будет заканчиваться последняя строка, используя версию метода *DrawString* с *RectangleF*, когда размеров прямоугольника не хвата-

ет для всего текста. Это свойство принимает значение одного из членов перечисления *StringTrimming*, применяемого только при работе с этим свойством:

Перечисление *StringTrimming*

Член	Значение	Описание
None	0	Как если бы не было нижней границы,
Character	1	Заканчивается знаком.
Word	2	Заканчивается целым словом.
EllipsisCharacter	3	Заканчивается знаком с многоточием (...).
EllipsisWord	4	Закапчивается словом с многоточием (...),
EllipsisPath	5	Многоточие перед последней директорией.

Следующая программа иллюстрирует действие этих параметров,

TrimmingTheText.cs

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

class TrimmingTheText: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new TrimmingTheText());
     }
     public TrimmingTheText()
     {
          Text = "Trimming the Text";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush       brush  = new SolidBrush(clr);
          float       cyText = Font.GetHeight(grfx);
          float       cyRect = cyText;
          RectangleF  rectf  = new RectangleF(0, 0, cx, cyRect);
          string      str    = "Those who profess to favor freedom and " +
                               "yet depreciate agitation. . .want " +
                               "crops without plowing up the ground, " +
                               "they want rain without thunder and " +
                               "lightning. They want the ocean without " +
                               "the awful roar of its many waters. " +
                               "\x2014 Frederick Douglass";
          StringFormat strfmt = new StringFormat();

          strfmt.Trimming = StringTrimming.Character;
```

см. след. стр.

```
            grfx.DrawString("Character: " + str, Font, brush, rectf, strfmt);
            rectf.Offset(0, cyRect + cyText);

            strfmt.Trimming = StringTrimming.Word;
            grfx.DrawString("Word: " + str, Font, brush, rectf, strfmt);
            rectf.Offset(0, cyRect + cyText);

            strfmt.Trimming = StringTrimming.EllipsisCharacter;
            grfx.DrawString("EllipsisCharacter: " + str,
                            Font, brush, rectf, strfmt);
            rectf.Offset(0, cyRect + cyText);

            strfmt.Trimming = StringTrimming.EllipsisWord;
            grfx.DrawString("EllipsisWord: " + str,
                            Font, brush, rectf, strfmt);
            rectf.Offset(0, cyRect + cyText);

            strfmt.Trimming = StringTrimming.EllipsisPath;
            grfx.DrawString("EllipsisPath: " +
                            Environment.GetFolderPath
                                (Environment.SpecialFolder.Personal),
                            Font, brush, rectf, strfmt);
            rectf.Offset(0, cyRect * cyText);

            strfmt.Trimming = StringTrimming.None;
            grfx.DrawString("None: " + str, Font, brush, rectf, strfmt);
        }
    }
```

Эта программа определяет объект *RectangleF* с высотой, достаточной для размещения только одной строки текста. Программа выводит текст, используя шесть возможных значений *StringTrimming*. Текст представляет собой цитату из Фредерика Дугласа для всех значений, кроме *StringTrimming.EllipsisPath*, где статический метод *Environment.GetFolderPath* служит для получения пути к папке My Documents (Мои документы). Можно изменить ширину окна и посмотреть, как это влияет на текст. Вот его типичный вид:



EllipsisCharacter и *EllipsisWord* выводят многоточие (...) в конце строки, показывая, что для вывода строки не хватает места. *Character* и *EllipsisCharacter* могут вывести в конце строки только часть слова.

Член *EllipsisPath* перечисления *StringTrimming* предназначен специально для вывода названия пути к файлу. Заметьте, что многоточие содержится в середине текста для того, чтобы можно было увидеть и начало, и конец пути.

Хотя *None* и *Word* привели к одинаковому результату, между ними есть разница, и мы увидим ее позднее.

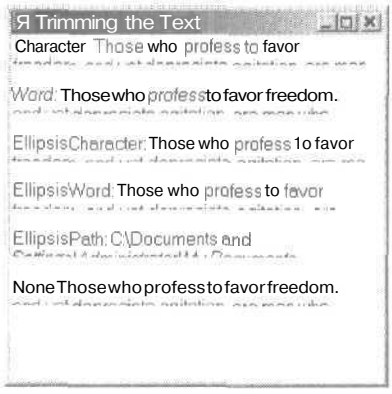
Если в программе изменить значение *cyRect* с:

```
float cyRect = cyText;
```

на:

```
float cyRect = 1.5f * cyText;
```

и перекомпилировать программу, то в результате получим такое изображение:



Метод *DrawString* выводит теперь две строки текста, и, хотя вы не можете увидеть большую часть второй строки, ее вид определяется свойством *Trimming*.

Теперь посмотрим, как флаг *NoWrap* перечисления *StringFormatFlags* влияет на вид текста. Добавьте следующий оператор после создания объекта *StringFormat*, но перед вызовом метода *DrawString*:

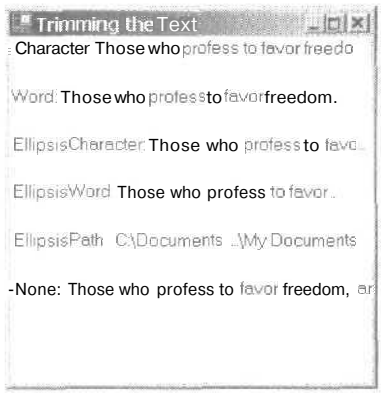
```
strfmt.FormatFlags |= StringFormatFlags.NoWrap;
```

Или можно выставить флаг в конструкторе *StringFormat*:

```
StringFormat strfmt = new StringFormat(StringFormatFlags.NoWrap);
```

Как можно понять из названия, флаг *NoWrap* запрещает автоматический переход на новую строку в *DrawString*.

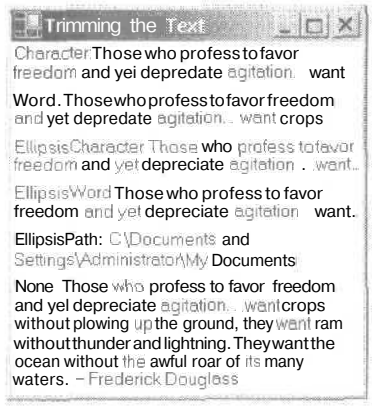
Текст все еще обрезается по правому краю прямоугольника. Но теперь ясно видно (если вы не сделали правую границу меньше клиентской области), что в случае *StringTrimming.None* последняя буква частично обрезается по правой границе. Мы столкнулись с первым случаем, когда вводится часть буквы:



А теперь уберем флаг *NoWrap*, вернее, заменим его флагом *NoClip*:

```
strfmt.FormatFlags = StringFormatFlags.NoClip;
```

который предписывает методу *DrawString* не отсекать текст, не помещающийся в прямоугольнике. В результате для любого значения перечисления, кроме *StringTrimmingNone*, выводятся две полных строки текста:



Для случая с *StringTrimmingNone* теперь выводится весь текст. Комбинация этого значения перечисления и флага *NoClip* устраняет эффект отсечения текста по нижней границе прямоугольника.

Если установить оба флага:

```
strfmt.FormatFlags |= StringFormatFlags.NoClip;
strfmt.FormatFlags |= StringFormatFlags.NoWrap;
```

то для всех значений *StringTrimming*, кроме *None*, результат будет таким же, как с флагом *No Wrap*. В случае с *StringTrimmingNone* текст не переносится и может уходить за правую границу. Ситуация такая же, как если бы вы определили *PointF*, а не *RectangleF* в методе *DrawString*,

При выводе текста в прямоугольную область, надо следить за отсечением текста. Если высота прямоугольника будет равна целому числу межстрочных интервалов, проблемы с отсечением текста не будет. Вероятно, это лучшее решение. Иначе вы должны установить флаг *NoClip* для предотвращения отсечения. Однако тогда

последняя строчка текста может частично выйти за нижнюю границу прямоугольника. В некоторых случаях (например, когда высота прямоугольника равна высоте клиентской области) последняя строчка текста будет всегда отсекаться, так как она достигает границы клиентской области. Убедитесь, что высоты прямоугольника хватает для вывода всех строк текста.

Если вы создаете объект *StringFormat*, используя конструктор по умолчанию,

```
StringFormat strfmt = new StringFormat();
```

или вы создаете его с помощью статического свойства *StringFormat:*

```
StringFormat strfmt = StringFormat.GenericDefault;
```

свойству *Trimming* изначально присвоено значение *StringTrimming.Character*. Если вы создаете объект *StringFormat*. используя статическое свойство:

```
StringFormat strfmt = StringFormat.GenericTypographic;
```

свойству *Trimming* изначально присвоено значение *StringTrimmingNone*,и выставлен флаг *NoClip*.

Позиции табуляции

Позиции табуляции указывают методу *DrawString*, как интерпретировать знаки '\t' или '\x0009' Unicode. Если *DrawString* не включает аргумента *StringFormat,* табуляция, измеряемая в пунктах, раина четырехкратному размеру шрифта (иначе говоря, четырем размерам эм). Скажем, для 9-пунктного шрифта табуляция равна 36 пт (или полдюйма), для 18-пунктного шрифта — 72 пт (1 дюйм), 36-пунктный шрифт будет иметь табуляцию в 144 пт (2 дюйма). Позиции табуляции отсчитываются от начала текста, устанавливаемого аргументами *PointF* или *RectangleF* метода *DrawString*.

Если *DrawString* включает аргумент *StringFormat,* позиций табуляции по умолчанию нет, и метод игнорирует знаки табуляции в тексте. Позиции табуляции устанавливаются методом *SetTabStops*класса *StringFormat*. Класс *StringFormat* также включает метод для получения текущих параметров настройки позиций табуляции:

Методы *StringFormat* (выборочно)

```
void SetTabStops(float fFirstTab, float[] afTabs)
float[] GetTabStops(out float fFirstTab)
```

Позиции табуляции задаются в глобальных координатах. Заметьте: они указываются как отдельным значением с плавающей запятой, так и массивом значений с плавающей запятой, из чего можно сделать вывод, что отдельное значение интерпретируется особым образом. При этом последний элемент массива также интерпретируется особым образом, хотя из синтаксиса методов этого и не видно.

Позвольте привести несколько примеров, прежде чем мы рассмотрим работу метода *SetTabStops*в общем случае. Я буду подразумевать, что единицы измерения страницы установлены в *GraphicsUnit.Point*.

Если нужна только одна позиция табуляции, скажем, 4 дюйма (288 пт), ее величину можно задать в первом аргументе метода и определить массив, состоящий из единственного 0:

```
strfmt.SetTabStops(288, new float[] { 0 });
```

Нельзя присвоить массиву значение *null*. Вы можете также использовать:

```
strfmt.SetTabStops(0, new float[] { 288, 0 });
```

Если требуются две позиции табуляции, например, через 1 дюйм (72 пт) и 3 дюйма (216 пт), можно применить:

```
strfmt.SetTabStops(0, new float[] { 72, 144, 0 });
```

Заметьте: второй элемент массива равен разнице между 216 и 72 пт. Хотя в примере массив определяется прямо в методе *SetTabStops*, его, конечно, можно определить и вне метода.

Если вам нужны табуляции с приращением в полдюйма (36 пт). используйте:

```
strfmt.SetTabStops(0, new float[] { 36 });
```

Позиции табуляции будут равны 36 пт, 72 пт, 108 пт, 144 пт и т. д.

Таким образом, *SetTabStops* может определить как отдельные позиции табуляции, так и повторяющиеся, и именно комбинация этих двух вариантов делает этот метод столь сложным. В общем случае аргументы *SetTabStops* выглядят так:

```
strfmt.SetTabStops(S, new float[] { A, B, C. . . . .N, R });
```

Я использую букву *R* для обозначения повторения *(Repeating)* и букву *S* для обозначения сдвига *(Shift)*. Обе эти величины могут быть равны 0 или отрицательным числам. Метод S устанавливает позиции табуляции в следующих положениях, измеряемых от начальной позиции текста:

```
S + A
S + A + B
S + A + B + C
S + A + B + C + ... + N
S + A + B + C + ... + N + R
```

Метод также устанавливает позиции табуляции в положениях R, 2R, 3R и т, д., но эти повторяющиеся табуляции начинаются только после самой длинной из остальных табуляций. Например:

```
strfmt.SetTabStops(100, new float[] { 50, 75, 50, 100 });
```

устанавливает табуляции в 150, 225, 275, 375, 400, 500, 600 и т. д. Размеры задаются в глобальных координатах.

Если вы хотите, чтобы все позиции табуляции были определены явно, установите последний элемент массива (который я назвал R) в 0. Вы также можете установить S в 0. Однако можно использовать этот первый аргумент метода *SetTabStops* для решения более сложных задач. Например, можно сначала определить массив, который состоит из четырех позиций табуляции, измеряемых от горизонтальной координаты 0:

```
float[] afTabs = { 100, 150, 100, 50, 0 };
```

Заметьте, что последний аргумент равен 0, поэтому повторяющихся позиций табуляции не будет.

Если хотите начать вывод текста в точке с координатой 0, вызовите *SetTabStops* с первым аргументом 0:

```
strfmt.SetTabStops(0, afTabs);
```

Этот метод установит позиции табуляции равными 100, 250, 350, 400 единицам. Если теперь понадобится выводить текст, начиная с координаты 50, но с теми же табуляциями, можно использовать -50 в качестве первого аргумента *SetTabStops*:

```
strfmt.SetTabStops(-50, afTabs);
```

Теперь позиции табуляции равны 50, 200, 300 и 350, но они отмеряются от начальной координаты 50, поэтому в действительности они равны 100, 250, 350 и 400, т. е. остались такими же, как прежде.

Давайте применим полученные знания на практике, разбив фрагмент текста на колонки. Текст взят из романа Эдит Уортон «Пора невинности» (1920). Следующий класс имеет единственное неизменяемое свойство *Text,* которое возвращает первые пять абзацев романа,

AgeOfInnocence.cs

```
//---
// AgeOfInnocence.cs © 2001 by Charles Petzold; text by Edith Wharton
//---- ------- --- ------ ------------ - ---------- -----
class AgeOfInnocence
{
    public static string Text
    {
        get
        {
            return

"On a January evening of the early seventies, Christine Nilsson was " +
"singing in Faust at the Academy of Music in New York. " +
"\n" +
"\tThough there was already talk of the erection, in remote metropolitan " +
"distances \"above the Forties, \" of a new Opera House which should " +
"compete in costliness and splendour with those of the great European " +
"capitals, the world of fashion was still content to reassemble every " +
"winter in the shabby red and gold boxes of the sociable old Academy. " +
"Conservatives cherished it for being small and inconvenient, and thus " +
"keeping out the \"new people\" whom New York was beginning to dread and " +
"yet be drawn to; and the sentimental clung to it for its historic " +
"associations, and the musical for its excellent acoustics, always so " +
"problematic a quality in halls built for the hearing of flmsio." +
"\n" +
"\tIt was Madame Nilsson's first appearance that winter, and what the " +
"daily press had already learned to describe as Van exceptionally " +
"brilliant audience \" had gathered to hear her, transported through the " +
"slippery, snowy streets in private broughams, in the spacious family " +
"landau, or in the humbler but more convenient \"Brown &c&o&u&p&й.\" To " +
```

см. след. стр.

```
"come to the Opera in a Brown &c&o&u&p&й was almost as honourable a way " +
"of arriving as in one's own carriage; and departure by the same means " +
"had the immense advantage of enabling one (with a playful allusion to " +
"democratic principles) to scramble into the first Brown conveyance in " +
"the line, Instead of waiting till the cold-and-gin congested nose of " +
"one's own coachman gleamed under the portico of the Academy, It was one " +
"of the great livery-stableman's most masterly intuitions to have " +
"discovered that Americans want to get away from amusement even more " 4.
"quickly than they want to get to it." +
"\n" 4
"\tWhen Newland Archer opened the door at the back of the club box the " +
"curtain had just gone up on the garden scene. There was no reason why " +
"the young man should not have come earlier, for he had dined at seven, " 4
"alone with his mother and sister, and had lingered afterward over a " +
"cigar in the Gothic library with glazed black-walnut bookcases and " +
"finial-topped chairs which was the only room in the house where Mrs. " +
"Archer allowed smoking. But, in the first place, New York was a " +
"metropolis, and perfectly aware that in metropolises it was \"not the " +
"thing\" to arrive early at the opera; and what was or was not \"the " +
"thing\" Played a part as important in Newland Archer's New York as the " +,
"inscrutable totem terrors that had ruled the destinies of his " +
"forefathers thousands of years ago." +
-V +
"\tThe second reason for his delay was a personal one. He had dawdled " +
"over his cigar because he was at heart a dilettante, and thinking over a " +
"pleasure to come often gave him a subtler satisfaction than its " +
"realisation. This was especially the case when the pleasure was a " +
"delicate one, as his pleasures mostly were; and on this occasion the " +
"moment he looked forward to was so rare and exquisite in quality " +
"that\x2014well, if he had timed his arrival in accord with the prima " +
"donna's stage-manager he could not have entered the Academy at a more " 4
"significant moment than just as she was singing; \"He loves me\x2014he " +
"loves me not\x2014&h&e& &l&o&v&e&s& &m&e!\" and sprinkling the falling " 4
"daisy petals with notes as clear as dew." 4
"\n";
        }
    }
}
```

Обратите внимание на знаки табуляции: они делают отступ первой строки каждого абзаца, кроме первого. Оригинальный текст содержит несколько слов, выделенных курсивом, Я использовал технику амперсанта (см. выше), чтобы выделить эти несколько слов подчеркиванием вместо курсива. Текст содержит также несколько длинных тире.

Ниже приведена программа, которая форматирует этот текст в колонки. Каждая колонка требует вызова метода *DrawString*. В начальном варианте этой программы в качестве единиц измерения я использовал *цицеро*. Цицеро равен 12 пт или (в компьютерной типографии) 1/6 дюйма. В этих единицах обычно измеряется ширина колонок в журналах и газетах. Я решил сделать колонки шириной 12 цицеро (т. е. 2 дюймам) с расстояниями между колонками в 1 цицеро. В то время

когда эта глава писалась, преобразование в цицеро вызывало проблемы в методе *DrawString*, поэтому я изменил программу так, чтобы в качестве единиц измерения использовались пункты.

TextColumns.cs

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

class TextColumns: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new TextColumns());
     }
     public TextColumns()
     {
          Text = "Edith Wharton's \"The Age of Innocence\"";
          Font = new Font("Times New flomarT, 10);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Brush        brush  = new SolidBrush(clr);
          Int          iChars, iLines;
          string       str    = AgeOfInnocence.Text;
          StringFormat strfmt = new StringFormat();

               // Единицы измерения - пункты.

          PointF[] aptf = { new PointF(cx, cy) };
          grfx.TransformPoints(CoordinateSpace.Device,
                               CoordinateSpace.Page, aptf);

          grfx.PageUnit = GraphicsUnit.Point;

          grfx.TransformPoints(CoordinateSpace.Page,
                               CoordinateSpace.Device, aptf);
          float fcx я aptf[0].X;
          float fcy * aptf[0].Y;

               // Свойства StringFormat, флаги и табуляции

          strfmt.HotkeyPrefix = HotkeyPrefix.Show;
```

см. след. стр.

```
            strfmt.Trimming     = StringTrimming.Word;
            strfmt.FormatFlags |= StringFormatFlags.NoClip;
            strfmt.SetTabStops(0, new float[] < 18 });

                // Отображение текста.

            for (int x = 0; x < fcx && str.Length > 0; x += 156}
            {
                RectangleF rectf = new RectangleF(x, 0, 144,
                                                  fcy - Font.GetHeight(grfx));

                grfx.DrawString(str, Font, brush, rectf, strfmt);
                grfx.MeasureString(str, Font, rectf.Size, strfmt,     ..
                                   out iChars, out iLines);

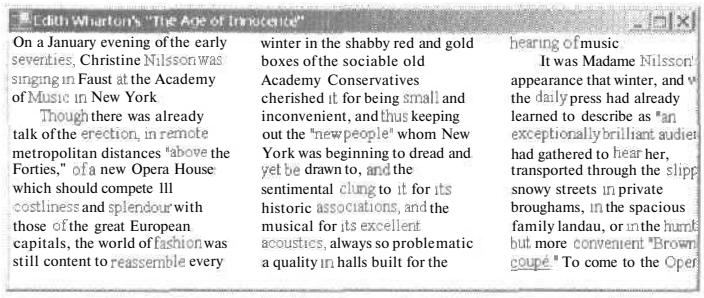
              ' str = str.Substring(iChars);
            }

        }
    }
```

Обратите внимание на свойства *StringFormat*, флаги и табуляции. *HotkeyPrefixShow* используется для подчеркивания слов, *StringTrimming.Word*— чтобы в конце колонок не обрезались слова, и *StringFormatFlagsNoClip*— чтобы строчки не обрезались внизу прямоугольника. Я установил единственную позицию табуляции (отступ первой строки) в 18 пт,

Цикл оператора/or повторяется для каждой колонки. Он продолжается до тех пор, пока не достигается правый край клиентской области или не выводится весь текст. Внутри цикла *for* высота прямоугольника вычисляется, как высота клиентской области за вычетом одной строки текста. Метод *DrawString* использует этот прямоугольник. Метод *MeasureString*определяет, сколько текста вывел метод *DrawString*. Метод *Substring* класса *String* подготавливает строку для следующей итерации цикла,

Вот что мы увидим на экране:





ГЛАВА

10

Таймер и время

Таймер — это устройство, периодически уведомляющее приложение о завершении заданного интервала времени. Ваша программа устанавливает этот интервал, как бы говоря: «Предупреждай меня каждую десятую долю секунды», — и таймер вызывает обработчик события десять раз в секунду,

В пространствах имен *System.Timers*, *System.Threading* и *System.Windows.Forms* определены три разных класса с именем *Timer*. Я буду использовать тот, что определен в пространстве *System.Windows.Forms*С этим таймером программисты для Microsoft Windows знакомы лучше всего. Он совмещен с другими событиями Windows и наиболее прост в использовании.

Хотя таймер и не столь важное устройство ввода, как клавиатура или мышь, он все же может быть очень полезен и находит применение во многих приложениях Windows Forms. Наиболее очевидным приложением для таймера является программа-часы, и в этой главе таких программ будет предостаточно. Однако есть и другие, возможно, не такие очевидные области применения таймера.

- **Многозадачность** Хотя Windows является средой с вытесняющей многозадачностью, рекомендуется, чтобы после обработки события программы возвращали управление Windows как можно быстрее. Иначе темп может существенно замедлиться. Если программа должна выполнить большой объем работы, ее можно разделить на несколько частей и выполнять каждую часть по получении события таймера.
- **Обновление данных** Таймер подходит для обновления постоянно изменяющейся информации, например, при выводе данных о свободных ресурсах или проценте выполнения какой-либо задачи.
- **Автосохранение** Таймер позволяет периодически сохранять на диске результаты работы программы.
- **Прекращение работы демо-версий программ** Некоторые демо-версии программ самостоятельно завершаются, скажем, через 30 минут после запус-

ка. В таких приложениях таймер подает сигнал на прекращение работы. (Пример такой программы, CloseInFive, см. в этой главе.)

- **Пошаговое продвижение** Обычно необходимо с заданной скоростью перемещать графические объекты в игре или переключать страницы в программах-описаниях. Таймер помогает избежать несовместимости, связанной с разными скоростями процессоров. Таймер часто применяют в анимации,

Таймер можно рассматривать и как гарант, обеспечивающий передачу управления программе. Если после выполнения конструктора или обработчика события программа отдает управление, обычно нельзя определить, когда произойдет следующее событие. В этом смысле событие таймера более определенно.

Я сказал *более*, потому что таймер не обладает ритмичностью метронома. События, вырабатываемые классом *Timer,* синхронизированы с другими событиями. Иначе говоря, событие таймера никогда не прервет процедуру обработки другого события, выполняющуюся в том же потоке. Если обработка события идет слишком долго, она задержит событие таймера.

Класс *Timer*

Класс *Timer* невелик и относительно прост. Обычно объект *Timer* создается конструктором по умолчанию:

```
Timer timer = new Timer();
```

Класс содержит одно событие:

Событие *Timer*

Событие	Метод	Делегат	Аргумент
Tick	*OnTick*	*EventHandler*	*EventArgs*

Где-то в вашем классе следует задать обработчик события для таймера:

```
void TimerOnTick(object obj, EventArgs ea)
{
    ⋮
}
```

Естественно, вы можете присвоить ему любое другое имя. Далее следует сопоставить обработчик события созданному ранее объекту *Timer.*

```
timer.Tick += new EventHandler(TimerOnTick);
```

Класс *Timer* включает в себя всего два свойства:

Свойства *Timer*

Тип	Имя	Доступ	Описание
int	*Interval*	Чтение/запись	Интервал инициирования события в миллисекундах.
bool	*Enabled*	Чтение/запись	Для включения таймера устанавливается значение *true*

Свойство *Interval* устанавливается равным количеству миллисекунд между вызовами обработчика события таймера. Например, этот оператор устанавливает период срабатывания таймера в одну секунду:

```
timer.Interval = 1000;
```

Вы можете присвоить этому свойству значение 1, однако это не гарантирует выработки события таймера каждую миллисекунду. Например, в Windows 2000, даже если установить *Interval* в единицу таймер будет вызывать обработчик события примерно раз в 10 миллисекунд.

Windows округляет присвоенное значение в большую сторону до ближайшего числа, кратного периоду внутренних часов ОС. В Windows 2000, например, присвоение *Interval* значений от 11 до 20 приведет к вызову обработчика события таймера раз в 20 миллисекунд. Но и значение *Interval,* равное 20, не означает, что каждую секунду обработчик будет вызываться ровно 50 раз. Если инициированное таймером событие задерживается более, чем на 20 миллисекунд, оно объединяется со следующим событием таймера. В очереди может находиться только одно событие таймера.

Но вызовов обработчика событий не последует вовсе, пока таймер не будет включен:

```
timer.Enabled = true;
```

Управлять работой таймера позволяют также следующие методы:

Методы *Timer* (выборочно)

```
void Start()
void Stop()
```

Вызов метода *Start* эквивалентен установке свойству *Enabled* значения *true, а* метода *Stop* — установке свойству *Enabled* значения *false.*

Эти свойства можно изменять в процедуре обработчика события таймера, тем самым сбрасывая его. Помните, что первым аргументом, передаваемым обработчику события, является объект, связанный с этим обработчиком. В данном случае это объект *Timer,* и получить доступ к нему можно так:

```
Timer timer = (Timer) obj;
```

Следующая программа использует *одноразовый* таймер, который называется так потому, что во время обработки события таймера он выключается.

CloseInFive.cs

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

class CloseInFive: Form
{
    public static void Main()
    {
```

см. след. стр.

```
        Application.Run(newCloseInFive());
    }
    public CloseInFive()
    {
        Text = "Closing in Five Minutes";

        Timer timer = new Timer();
        timer.Interval = 5 * 60 * 1000;
        timer.Tick += new EventHandler(TimerOnTick);
        timer.Enabled = true;
    }
    void TimerOnTick(object obj, EventArgs ea)
    {
        Timer timer = (Timer) obj;

        timer.Stop();
        timer.Tick -= new EventHandler(TimerOnTick);

        Close();
    }
}
```

Обработчик события таймера не только выключает сам таймер, но и закрывает приложение. Эта программа является примером реализации демо-версии, которая дает возможность ознакомиться с преимуществами программы, но не позволяет использовать ее долгое время.

При закрытии приложения остановка таймера и отключение обработчика события не необходимы, однако, если вы используете таймер для одноразовых действий, это будет полезно, поскольку в таком случае объект *Timer* будет уничтожен сборщиком мусора.

Примером другой крайности, противоположной CloseInFive, является усыпляющая программа, которая устанавливает таймер один раз и постоянно обрабатывает его события.

RandomRectangle.cs

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

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

см. след. стр.

```
    public RandomRectangle()
    {
        Text = "Random Rectangle";

        Tiffler timer    = new Timer();
        timer.Interval = t;
        timer.Tick    += new EventHandler(TimerOnTick);
        timer.Start();
    }
    void TimerOnTick(object obj, EventArgs ea)
    {
        Random rand * new Random();

        int x1 - rand.Next(ClientSize.Width);
        int x2 = rand.Next(ClientSize.Width);
        int y1 = rand.Next(ClientSize.Height);
        int y2 = rand.Next(ClientSize.Height);

        Color color = Color.FromArgb(rand.Next(256),
                                     rand.Next(256),
                                     rand.Next(256));

        Graphics grfx - CreateGraphics();
        grfx.FillRectangle(new SolidBrush(color),
                           Math.Min(x1, x2), Math.Min(y1, y2),
                           Math.Abs(x2 - x1), Math.Abs(y2 - y1));
        grfx.Dispose();
    }
}
```

Структура *DateTime*

Если вы хотите написать программу-часы, вам следует узнать кое-что о представлении даты и времени в .NET Framework,

В пространстве имен *System* определена наиболее важная структура даты и времени — *DateTime*. Создать объект типа *DateTime* можно, используя семь конструкторов, три из которых таковы:

Конструкторы *DateTime* (выборочно)

```
DateTime(int year, int month, int day)
DateTime(int year, int month, int day, int hour, int minute, int second)
DateTime(int year, int month, int day, int hour, int minute, int second, int msec)
```

Значение года может лежать в диапазоне от 1 до 9 999, месяца — от 1 до 12, дня — от 1 до числа дней в соответствующем месяце данного года. Значение количества часов лежит в пределах от 0 до 23, минут и секунд — от 0 до 59, миллисекунд — от 0 до 999. Если любой из аргументов выходит за разрешенные пределы, конструктор генерирует исключение.

Кроме того, конструктор *DateTime* генерирует исключение в том случае, если значения года, месяца и дня несовместимы. Например, значение месяца, равное 2, и дня — равное 29, допустимо только для високосного года. Конструкторы *DateTime* применяют правила високосного года для Григорианского календаря (введенного Папой Григорием XIII в 1582 г. и разошедшегося по всему миру в последующие столетия). В Григорианском календаре год является високосным, если число, обозначающее этот год, делится на 4, но не делится на 100 или если оно делится на 400. 1900 г. не является високосным; 2000 — является. (До введения Григорианского календаря високосным считался каждый четвертый год без исключений.) Конструктор *DateTime* соблюдает правила високосного года даже для годов, предшествовавших созданию Григорианского календаря.

Класс *DateTime* имеет 15 свойств, каждое из которых доступно только для чтения. Десять из них приведены ниже:

Свойства *DateTime*

Тип	Имя	Доступ	Описание
int	*Year*	Чтение	от 1 до 9999
int	*Month*	Чтение	от 1 до 12
int	*Day*	Чтение	от 1 до 31
int	*Hour*	Чтение	от 0 до 23
int	*Minute*	Чтение	от 0 до 59
int	*Second*	Чтение	от 0 до 59
int	*Millisecond*	Чтение	от 0 до 999
int	*DayOfWeek*	Чтение	от 0 (воскресенье) до 6 (суббота)
int	*DayOfYear*	Чтение	от 1 до 366
DateTime	*Date*	Чтение	Установка времени в 0 (полночь)

Первые семь представляют собой хорошо знакомые атрибуты даты и времени, и принимают значения заданные в конструкторе. Свойства *DayOJWeek* и *DayOfYear* предоставляют дополнительную информацию о текущей дате: номер дня о неделе (начиная с воскресенья) и в году. Свойство *Date* возвращает объект *DateTime*. представляющий тот же день что и текущий объект *DateTime,* но значения его свойств *Hour, Minute, Second u Millisecond* устанавливаются в 0.

Класс *DateTime* содержит три особенно полезных статических свойства;

Статические свойства *DateTime*

Тип	Имя	Доступ	Описание
DateTime	*Now*	Чтение	Текущие местные дата и время
DateTime	*Today*	Чтение	Текущая местная дата
DateTime	*UtcNow*	Чтение	Текущие дата и время всемирного скоординированного времени (UTC)

Свойство *DateTimeNow* возвращает *структуру DateTime,* заполненную значениями местного времени и даты. Например, чтобы получить текущие время и дату, вызовите;

```
DateTime dt = DateTime.Now;
```

Теперь значения даты и времени можно получить из свойств *DateTime* объекта *dt*. Свойство *DateTime.Today* возвращает аналогичную структуру *DateTime*, содержащую сегодняшнюю дату, но значения всех свойств, касающихся времени, установлены в 0.

Статическое свойство *UtcNow* возвращает структуру *DateTime*, которая содержит значения текущего времени и даты для всемирного скоординированного времени (UTC).

Структура *DateTime* содержит методы и перегруженные операторы, позволяющие производить вычисления над датами и временем. Для объектов *DateTime* можно применять все операторы сравнения (==, !=, <, >, <=, и >=). Кроме того, поддерживается сложение и вычитание, но эти операции задействуют объекты *TimeSpan*, о которых я расскажу чуть позже.

Местное и всемирное время

Во всем мире люди думают о полудне как о времени, когда солнце находится в зените, и о полуночи — как о середине ночи. Поскольку эти два события не происходят по всему миру одновременно, люди, живущие в разных частях света, устанавливают свои часы по-разному. Некогда хаотичная, эта тенденция привела к введению четких часовых поясов, устанавливаемых государством, и рассчитываемых как часовые или получасовые смещения относительно среднего времени по Гринвичу.

Английское местечко Гринвич играет важную роль во введении стандартов времени, поскольку там расположена Королевская Гринвичская Обсерватория, основанная в 1675 г. для разработки способов астрономической навигации для кораблей, находящихся в море. В 17бО-х обсерватория начала публиковать морские альманахи, в которых первый меридиан (линия долготы 0°) для удобства был принят проходящим через Гринвич. В 1884 г. эта система меридианов была принята в качестве мирового стандарта, хотя до 1911 г. французы применяли систему меридианов, согласно которой первый меридиан проходил через Париж.

В 1833 г. гринвичские астрономы начали каждый день в 13:00 запускать шар, который был виден с кораблей на Темзе. Так было основано среднее время по Гринвичу. В 1840-х это время было объявлено стандартом для всей Великобритании и заменило все различные местные значения.

Хотя люди до сих пор используют среднее время по Гринвичу в качестве мирового стандарта, применение термина *всемирное скоординированное время* (UTC) считается более корректным. [Всемирное скоординированное время обозначается аббревиатурой UTC в виде компромисса между порядком слов в английском (Coordinated Universal Time — CUT) и французском (*Temps UniverselCoordonné* — TUC) языках.] Согласно международному договору от 1972 г. время UTC одинаково для всего мира.

Местное время определяется положительным или отрицательным смещением от UTG. Временные зоны к запячу от Гринвича соответствуют более раннему времени, к востоку — более позднему. Так, Восточное поясное время, установленное на восточном побережье Соединенных Штатов, на 5 часов отстает от UTC. Но, прибавив ко времени UTC 19 часов, вы *не* получите то же значение. Такая операция даст верное время, но неправильную дату.

Существует также изящный прием, называемый переходом на летнее время. Основной его принцип весьма прост: с приближением дня летнего солнцестояния солнце встает раньше, а садится позже, так что нетрудно встать чуть раньше и получить вечером побольше солнечного света. Одни страны используют переход на летнее время, другие — нет. Первые часто переходят на летнее и зимнее время в разные дни. Есть страны (в первую очередь США), где такой переход осуществляется частично: многие штаты переходят на летнее время, но некоторые этого не делают.

Windows позволяет установить временную зону для вашего компьютера в диалоговом окне Date/Time Properties (Свойства: Дата и время), которое можно вызвать из Панели управления или дважды щелкнув часы в панели задач Windows. Таи же можно указать, переходить ли на летнее время. Очевидно, что для каждого компьютера местное время рассчитывается как смещение от UTC, основанное на часовом поясе, в котором расположен данный компьютер, и учитывающее переход на летнее время.

Сама по себе структура *DateTime* не использует UTC или местное время. При вызове одного из конструкторов *DateTime* вы указываете значения даты и времени, которые могут соответствовать местному времени, UTC, дате и времени вашего рождения или чему угодно,

Как я уже сказал, свойства *Now* и *Today* возвращают местное время и дату, а *UtcNow*— время UTC, вычисленное с учетом часового пояса, в котором расположен данный компьютер. Следующие методы позволяют преобразовать значении времени, хранящиеся в объекте *DateTime,* в местное время или UTC:

Методы *DateTime* (выборочно)

```
DateTime ToLocalTime()
DateTime ToUniversalTime()
```

Например, если переменная *dtLocal* содержит местное время и дату, преобразовать значения времени и даты для UTC можно так:

```
DateTime dtUtc = dtLocal.ToUniversalTime();
```

Если переменная *dtLocal* на самом деле содержала значения даты и времени для UTC, то вы получите неверный результат. Бы сами должны следить за тем, к какому часовому поясу относится ваш объект *DateTime.*

Информацию о часовом поясе, которому принадлежит данный компьютер, и о правилах перехода на летнее время для данного часового пояса можно получить из класса *TimeZone*пространства имен *System.* Класс *TimeZone*определен как абстрактный. Это значит, что вы можете работать с ним только через подклассы. Однако сам по себе этот класс предоставляет некоторую полезную информацию. Он содержит одно статическое свойство, возвращающее экземпляр класса. Этот экземпляр представляет часовой пояс, установленный для данного компьютера:

Статическое свойство *TimeZone*

Тип	Имя	Доступ
TimeZone	*CurrentTimeZone*	Чтение

Если, например, вы вызовете:

```
TimeZone tz = TimeZone.CurrentTimeZone;
```

то *tz* будет содержать информацию о часовом поясе, в котором находится данный компьютер. Два нестатических свойства возвращают имя данного часового пояса:

Свойства *TimeZone*

Тип	Имя	Доступ
string	*StandardName*	Чтение/запись
string	*DaylightName*	Чтение/запись

Например, если компьютер расположен на восточном побережье США, на нем правильно установлены параметры часового пояса и вы установите переменную *tz*, как показано выше, свойство *tz.StandardName* вернет строку «Eastern Standard Time», а свойство *tz.DaylightName* — строку «Eastern Daylight Time».

Ниже приведены методы класса *TimeZone:*

Методы *TimeZone*

```
TimeSpan GetUtcOffset(DateTime dt)
DateTime ToLocalTime(DateTime dt)
DateTime ToUniversalTime(DateTime dt)
DaylightTime GetDaylightChanges(int iYear)
bool IsDaylightSavingTime(DateTime dt)
bool IsDaylightSavingTime(DateTime dt, DaylightTime dlt)
```

Метод *GetUtcOffset* возвращает разницу между местным временем и UTC в виде объекта *TimeSpan*. (Вскоре я опишу его подробнее. Пока же достаточно сказать, что эта структура задана в пространстве имен *System* и служит для выражения отрезков времени в сотнях наносекунд.) *GetUtcOffset* учитывает переход на летнее время. Для того же компьютера на восточном побережье США метод:

```
tz.GetUtcOffset(new DateTime(2002, 2, 2))
```

вернет значение -5:00:00, соответствующее разнице в -5 часов. Это значение следует прибавить к UTC, чтобы получить Восточное поясное время, Вызов метода:

```
tz.GetUtcOffset(new DateTime(2002, 8, 29))
```

вернет значение -4:00:00, или -4 часа, что объясняется переходом на летнее время.

Методы *ToLocalTime* и *ToUniversalTime* аналогичны соответствующим методам структуры *DateTime,* но основаны на конкретном объекте *TimeZone,* Если б у вас был способ получить объекты *TimeZone* для других часовых поясов мира, вы могли бы использовать эти методы для преобразования местного времени во Всемирное координированное время для других часовых поясов.

Оператор:

```
DaylightTime dlt = tz.GetDaylightChanges(2002);
```

возвращает объект типа *DaylightTime*. Этот класс определен в пространстве имен *System.Globalization*. Он содержит три свойства:

Свойства *DaylightTime*

Тип	Имя	Доступ
DateTime	*Start*	Чтение
DateTime	*End*	Чтение
TimeSpan	*Delta*	Чтение

Свойства *Start* и *End* показывают, что переход на летнее время имеет место 7 апреля 2002, в 2:00, а на зимнее — 27 октября 2002, в 2:00. (Оба осуществляются в воскресное утро.) Значение *Delta* указывает разницу во времени, равную 1 часу

Для часовых поясов северного полушария дата, возвращаемая свойством *Start*, расположена в году ранее, чем дата, возвращаемая свойством *End*. В южном полушарии сезоны меняются местами: дата начала перехода расположена в году позднее, а конца — раньше,

Число отсчетов

Еще один способ представления даты и времени — указать количество временных отсчетов длительностью 100 наносекунд. Внутри структуры *DateTime* дата и время хранятся в виде числа таких отсчетов, прошедших с полуночи 1 января 1 года нашей эры. Два оставшихся свойства *DateTime* возвращают это значение 1 число таких отсчетов, прошедших с полуночи:

Свойства *DateTime* (выборочно)

Тип	Имя	Доступ	Описание
long	*Ticks*	Чтение	Число отсчетов по 100 наносекунд, прошедших с 1.1.0001.
TimeSpan	*TimeOfDay*	Чтение	Число отсчетов по [00 наносекунд, прошедших с полуночи,

Помните: значение типа *long* имеет размер 64 бита и поэтому может хранить очень большие значения, что, безусловно, требуется в данном случае. Заметьте, что свойство *TimeOfDay* возвращает объект типа *TimeSpan*, выражающий отрезки времени в сотнях наносекунд.

Для 1 января 2001 г. свойство *Ticks* возвращает значение 631 139 040 000 000 000. В миллисекунде содержится 10 000 таких отсчетов, в секунде — 10 000 000, в минуте — 600 000 000, в часе — 36 000 000 000, а за один день их проходит 864 000 000 000. (Кстати, все эти константы доступны как поля структуры *TimeSpan*.) Это значит, что за эти 2 000 лет прошло 730 485 дней, что в среднем составляет 365,2425 дней в году.

Для Григорианского календаря 365,2425 дней в году — верное значение: большинство годов длится 365 дней. Дополнительный день каждые четыре года прибавляет 0,25 к среднему значению дней в году. Вычитание дополнительного дня каждые 100 лет уменьшает это значение на 0,01. Дополнительный день каждые 400 лет увеличивает его на 0,0025. Другими словами:

```
365 + 1/4 - 1/100 + 1/400 = 365.2425
```

Создать объект *DateTime* можно, используя значение типа *long,* содержащее количество временных отсчетов по 100 наносекунд, прошедших с 1 января первого года нашей эры:

Конструкторы *DateTime* (выборочно)

```
Datefime(long ticks)
```

Объекты *TimeSpan* можно создавать подобным образом или явно указывая дни и время. Вот полный список конструкторов *TimeSpan:*

Конструкторы *TimeSpan*

```
TimeSpan(long ticks)
TimeSpan(int hours, int minutes, int seconds)
TimeSpan(int days, int hours, int minutes, int seconds)
TimeSpan(int days, int hours, int minutes, int seconds, int milliseconds)
```

Конструкторы *DateTime* и *TimeSpan* очень похожи, поэтому важно понимать разницу между ними. Объект *DateTime* соответствует конкретному значению даты и времени; свойство *Ticks* объекта *DateTime* возвращает количество временных отсчетов по 100 наносекунд, прошедших с 1 января 1 года нашей эры.

Объект *TimeSpan* соответствует длительности, т. е. периоду прошедшего времени. Заметьте: аргументы конструкторов *TimeSpan* не указывают какой либо конкретный день, час, минуту, секунду и миллисекунду. Они соответствуют определенному *числу* дней и *числу* часов и т. д. В конструкторах *TimeSpan* не используются месяцы и годы, поскольку в разных месяцах и годах разное количество дней.

На значения, которые могут быть переданы в качестве аргументов конструкторам *TimeSpan* в отличие от конструкторов *DateTime* ограничения не накладываются. Так, оператор:

```
TimeSpan ts = new TimeSpan(1000, 1000, 1000, 1000, 1000);
```

абсолютно корректен.

Вот полный список свойств объектов *TimeSpan.*

Свойства *TimeSpan*

Тип	Имя	Доступ	Описание
long	*Ticks*	Чтение	Число интервалов по 100 наносекунд
int	*Days*	Чтение	Целое число дней
int	*Hours*	Чтение	от 0 до 23
int	*Minutes*	Чтение	от 0 до 59
int	*Seconds*	Чтение	от 0 до 59
int	*Milliseconds*	Чтение	от 0 до 999
double	*TotalDays*	Чтение	= *(double) Ticks / TicksPerDay*
double	*TotalHours*	Чтение	= *(double) Ticks / TicksPerHour*
double	*TotalMinutes*	Чтение	= *(double) Ticks / TicksPerMinute*
double	*TotalSeconds*	Чтение	= *(double) Ticks / TicksPerSecond*
double	*TotalMilliseconds*	Чтение	= *(double) Ticks j TicksPerMillisecond*

Для последних пяти свойств я указал формулы, при помощи которых они вычисляются из значений соответствующих полей структуры *TimeSpan*.

Ранее я говорил о том, что структура *DateTime* содержит перегруженный оператор сложения. Однако к объекту *DateTime* можно прибавить только объект *TimeSpan*. Если *dt*, *dt1* и *dt2* — объекты *DateTime*, а *ts* — объект *TimeSpan*, вы вправе провести такое сложение:

```
dt2 = dt1 + ts;
```

или такое:

```
dt += ts;
```

Например, вы хотите увеличить значение времени и даты на 45 минут:

```
dt += new TimeSpan(0, 45, 0);
```

или на одну неделю:

```
dt += new TimeSpan(7, 0, 0, 0);
```

Это безопасный способ таких вычислений, (Другой безопасный способ — использовать методы *Add* класса *DateTime*.) Оператор вычитания определяется двумя способами. Вы можете вычесть одну дату из другой и получить объект *TimeSpan*:

```
ts = dt2 - dt1;
```

Или вычесть объект *TimeSpan* из объекта *DateTime*, получив как результат объект *DateTime*:

```
dt2 = dt1 - ts;
```

или:

```
dt -= ts;
```

Объекты *TimeSpan* можно складывать, вычитать или сравнивать друг с другом любым способом.

Мировые системы летосчисления

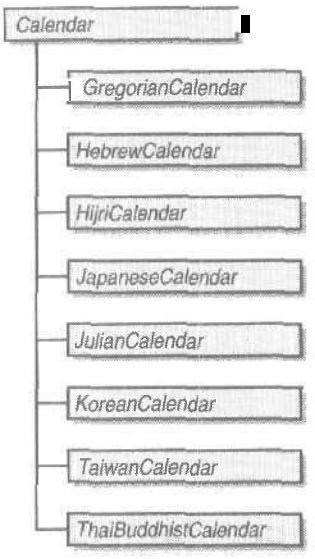
Ниже перечислены три последних конструктора *DateTime*:

Конструкторы *DateTime* (выборочно)

```
DateTime(int year, int month, int day, Calendar cal)
DateTime(int year, int month, int day, int hour, int minute, int sec, Calendar cal)
DateTime(int year, int month, int day, int hour, int minute, int sec, int msec,
Calendar cal)
```

В качестве последнего аргумента передается объект типа *Calendar*. Он показывает, как интерпретировать аргументы года, месяца и дня. Я уже говорил, что конструкторы без аргумента *Calendar* интерпретируют даты в стиле Григорианского календаря.

Calendar — это абстрактный класс в пространстве имен *System.Globalization*, которое включает в себя и восемь производных классов класса *Calendar*.



Hijri — это одно из названий исламского календаря.

Если в качестве последнего аргумента конструктора *DateTime* указать объект типа *Calendar,* то становится возможным применение других правил соответствия дней, месяцев и годов. Например, оператор:

```
DateTime dt = new DateTime(1900, 2, 29);
```

вызовет исключение, поскольку согласно Григорианскому календарю 1900 г. не являлся високосным. Однако оператор:

```
DateTime dt = new DateTime(1900, 2, 29, new JulianCalendar());
```

исключения не сгенерирует, поскольку в Юлианском календаре високосным считается каждый год, кратный 4.

Если вы создадите такой объект, передав ему аргумент *JulianCaiendar,* а затем просмотрите значения свойств, то увидите, что значение свойства *Month* ~ 3 (март), а свойства *Day* — 13. Свойства *Year, Month и Day* структуры *DateTime* всегда представляют дату в Григорианском стиле. Конструктор преобразует дату, указанную в его аргументах, в число 100-наносекундных порций, а свойства структуры *DateTime* переводят его в дату по Григорианскому календарю.

Принятие Григорианского календаря привело к тому, что сразу за днем 4 октября 1582 г следовало 15 октября 1582, т, е. было пропущено 10 дней.[1] Если вы создадите такой объект:

```
dt = new DateTime(1582, 10, 5, new JulianCalendar());
```

то свойство *Month* будет содержать значение 10, но свойство *Day* — значение 15.

[1] За несколько лет до празднования нового тысячелетия было издано несколько книг, рассказывающих об истории Юлианского и Григорианского календарей. Возможно самая короткая и яркая из них — «Загадки тысячелетия: Рациональное руководство по точному отсчету» Стивена Джей Гоулда [Stephen Jay Gould, *Questioning the Millennium: A Rationalist'sGuide to a Precisely Arbitrary Countdown* (New York: Harmony Books, 1997)].

Становится интереснее. Предположим, вы сделали так:

```
dt = new DateTime(5762, 5, 20, new HebrewCalendar());
```

Да, это на самом деле год еврейского календаря — двадцатый день месяца шват 5762 г. Получившаяся в результате этого вызова структура *DateTime* содержит *в* свойстве *Year* значение 2002, а в свойстве *Month* и *Day — 2,* Таким образом, мы осуществили преобразование между еврейским и Григорианским календарем. Если последним аргументом, передаваемым конструктору *DateTime,* стоит объект *HebrewCalendar*, то для некоторых годов аргумент *Month* может принимать значение 13.

Подобным образом можно определить дату исламского календаря:

```
dt = new DateTIme(1422, 11, 20, new HijriCalendar());
```

Аргументы конструктора выставлены на двадцатый день месяца дуль-каеда 1422 г. И снова в результате структура *DateTime* содержит в свойстве *Year* значение 2002, а в свойстве *Month* и *Day — 2.*

Для преобразования даты из Григорианской в какую-либо другую систему летосчисления следует создать экземпляр требуемого календаря, например.

```
HebrewCalendar hebrewcal = new HebrewCalendarO;
HijriCalendar hijrical = new HijriCalendar();
```

Кроме того, вам понадобится объект *DateTime*:

```
DateTime dt = new DateTime(2002, 2, 2);
```

Наконец, чтобы преобразовать дату из Григорианской в еврейскую или исламскую систему, потребуются следующие три метода:

Методы *Calendar* (выборочно)

```
int GetYear(DateTime dt)
int GetMonth(DateTime dt)
int GetDayOfMonth(DateTime dt)
```

Например, оператор:

```
hijrical.GetYear(dt)
```

возвращает значение 1422.

Удобная интерпретация

Одними из важнейших являются методы *DateTime,* представляющие дату и время в форме, удобной для восприятия человеком. Такое преобразование может казаться довольно простым, если забыть о том, что люди во всем мире записывают дату и время по-разному, В одних странах используется 24-часовая система измерения времени; в других — 12-часовая. Где-то пишут число до месяца, где-то — наоборот. Если в дате указывается название месяца или дня недели, хорошо бы указать их на местном языке. И даже в пределах одной страны пользователи могут настраивать отображение даты и времени по-своему.

Эти параметры, соответствующие государственным стандартам или личным предпочтениям пользователя, можно просмотреть и изменить в диалоговом окне

Regional Options (Язык и стандарты) доступном из Панели управления. На вкладке General (Общие) можно изменить местоположение (и это отразится на форме отображения даты и времени), установив их согласно стандартам выбранной страны. В полях Time (Время) и Date (Дата) можно изменить стандартный формат отображения на собственный. Форматы отображения даты и времени, зависящие от конкретной страны или предпочтений пользователя, называются *локализованными* (culture-specific).

Выводить дату и время в формате, установленном пользователем, считается хорошим тоном, однако иногда это нежелательно. Иногда дата и время должны быть введены в документ, с которым будут работать в странах с другим форматом отображения подобной информации, или он будет объединен с другими документами. В таком случае в программе лучше использовать какой-то совместимый формат, соответствующий некоторым международным стандартам. В .NET Framework, для такого формата применяется термин *регионально-независимый* (culture-invariant).

В последующих примерах я буду использовать дату 1 июня 2002 г. и местное время 3:05:01 р. т. Значения месяца, дня, часа, минуты и секунды специально выбраны состоящими из одной цифры, чтобы определить, происходит ли отбрасывание незначащих нулей при форматировании.

Допустим, *dt* — объект типа *DateTime*. Если вы просто передадите этот объект методу *Console.WriteLine*:

```
Console.WriteLine(dt);
```

или методу *String.Format*:

```
str = String.Format("{0}", dt);
```

он вызовет метод *ToString* объекта *DateTime*. Предыдущий оператор эквивалентен такому:

```
str = dt.ToString();
```

Метод *ToString* преобразует дату и время в строку символов локализованного формата. Для установок «U.S. English», строка, возвращенная методом *ToString*, будет выглядеть так:

```
6/1/2002 3:05:01 PM
```

Структура *DateTime* также предоставляет несколько дополнительных версий метода *ToString*, которым передается один или два аргумента. Эти версии позволяют форматировать дату и время в локализованном или регионально-независимом формате:

Методы *ToString* структуры *DateTime*

```
string ToString()
string ToString(string strFormat)
string ToString(IFormatProvider ifp)
string ToString(string strFormat, IFormatProvider ifp)
```

Аргумент *string* обычно представляет собой букву, определяющую стиль форматирования. Скоро я расскажу об этих буквах подробнее. Также аргумент *string* может содержать набор букв, задающих собственный формат.

Аргумент *IFormatProvider* определяет интерфейс. Для этого аргумента вам потребуется экземпляр класса, реализуемый *IFormatProvider*. Один из таких классов – *DateTimeFormatInfo* из пространства имен *System.Globalization*. (Подробнее о форматировании см. документацию класса *DateTimeFormatInfo*.) Класс *DateTimeFormatInfo* содержит два статических свойства, возвращающих экземпляры класса.

Статические свойства *DateTimeFormatInfo*

Тип	Имя	Доступ
DateTimeFormatInfo	*CurrentInfo*	Чтение
DateTimeFormatInfo	*InvariantInfo*	Чтение

Применить локально-независимое форматирование можно так:

```
strDT = dt.ToString(DateTimeFormatInfo.InvariantInfo);
```

или так:

```
strDT = dt.ToString(strFormat, DateTimeFormatInfo.InvariantInfo);
```

Применить форматирование, совместимое с локальными параметрами можно так:

```
strDT = dt.ToString(DateTimeFormatInfo.CurrentInfo);
```

или так:

```
strDT = dt.ToString(strFormat, DateTimeFormatInfo.CurrentInfo);
```

Кроме того, локализованное форматирование можно получить, указав в качестве второго аргумента метода *ToString* параметр *null*.

```
strDT = dt.ToString(strFormat, null);
```

или используя *strFormat* в качестве единственного аргумента:

```
strDT = dt.ToString(strFormat);
```

Чтобы применить стандартные форматы даты и времени, следует передать методу *ToString* в качестве первого аргумента один из символов, перечисленных в таблице. Во второй колонке показан формат, в котором будет представлена информация, если в качестве местоположения компьютера в Панели управления выбрать параметр «United States», а вторым аргументом метода *ToString* указать значение *null* или *DateTimeFormatInfo.CurrentInfo* либо использовать версию метода *ToString* с единственным аргументом. В правой колонке показан формат, который будет использован, если в качестве второго аргумента указать *DateTimeFormatInf.InvariantInfo*. Этот формат не зависит от параметров, выставленных в Панели управления.

Форматы даты и времени метода *ToString*

Символ, обозначающий формат	Аргумент, определяющий форматирование	
	CurrentInfo для США	*InvariantInfo*
D	6/1/2002	06/01/2002
D	Saturday June 01, 2002	Saturday, 01 June 2002
f	Saturday, June 01, 2002 3:05 PM	Saturday 01 June 2002 15:05
F	Saturday, June 01, 2002 3:05:01 PM	Saturday 01 June 2002 15:05:01
g	6/1/2002 3:05 PM	06/01/2002 15:05
G или *null*	6/1/2002 3:05:01 PM	06/01/2002 15:05:01
m или *M*	June 01	June 01
r или *R*	Sat, 01 Jun 2002 15:05:01 GMT	Так же, как для *CurrentInfo*
s	2002-06-01T15:05:01	Так же, как для *CurrentInfo*
t	3:05 PM	15:05
T	3:05:01 PM	15:05:01
u	2002-06-01 15:05:01Z	Так же. как для *CurrentInfo*
U	Saturday June 01, 2002 7:05:01 PM	Saturday 01 June 2002 19:05:01
V или *Y*	June. 2002	2002 June

Символы, обозначающие формат, представляют собой первые буквы слов:

Мнемоника символов форматирования *DateTime*

Символ	Мнемоника
d	Date (Дата)
f	Full (Полный)
g	General (Общий)
m	month/day (месяц/день)
r	RFC
s	sortable (упорядоченный)
t	time (время)
u	universal (мировой)
y	year/month (год/месяц)

Если строчная и прописная буквы обозначают разный формат (как, например, *d* и *D*), прописная буква обозначает более длинную строку. Для букв *r*, *R*, *s* и *u* формат возвращаемой строки не зависит от второго аргумента метода *ToString*.

Если не указать аргумент форматирования или указать значение *null*, метод возвращает строку, соответствующую локализованным правилам форматирования и указанию аргумента *G* (Общий). Таким образом, оператор:

```
dt.ToString()
```

эквивалентен:

```
dt.ToString((string)null)
```

или:

```
dt.ToString((IFormatProvider)null)
```

или такому:

```
dt.ToString(null, null);
```

Все они возвращают строку, отформатированную в соответствии с локализованными правилами аргумента *G*. Такую же строку возвращают методы *ToString*с единственным аргументом *IFormatProvider*.

При использовании аргумента *r* или *R* метод *ToString* возвращает строку в формате RFC 1123[2]. Строка, получаемая при использовании аргумента 5, является строкой формата ISO 8601[3] и считается универсальной и упорядоченной. Буква *T* в центре этой строки называется *указателем времени* и разделяет дату и время. Даты, начинающиеся с обозначения месяца или числа, труднее сортировать в таком формате. Формат, обозначаемый аргументом *u,* похож на формат *s,* однако в нем нет указателя времени и строка заканчивается буквой Z. В кругах военных и радистов Всемирное скоординированное время иногда называют *время Зулу* (Zulu time), *буква Z* обозначает время Зулу и нулевую долготу (от слова «zero» — ноль).

Аргумент *U* приводит к преобразованию времени во всемирное скоординированное. Оно подразумевает, что в объекте *DateTime* указано местное время.

Структура *DateTime* содержит четыре других довольно удобных метода дли форматирования данных. Все они являются локализованными методами:

Методы *DateTime* (выборочно)

Метод	Возвращаемая строка (для параметров U.S. English)
string ToShortDateString()	6/1/2001
string ToLongDateString()	Saturday, June 01, 2002
string ToShortTimeString()	3:05 PM
string ToLongTimeString()	3:05:01 PM

Строки, возвращаемые этими методами, идентичны результатам применения аргументов *d, D, t* и *T,* соответственно.

Давайте ради эксперимента в диалоговом окне Regional Options (Язык и стандарты) выберем в качестве местонахождения параметр Germany. Теперь сведения о дате и времени будут форматироваться, как показано на следующей странице

Правая колонка не изменилась. В колонке с локализованными форматами используются немецкие названия месяцев и дней недели и внесены некоторые другие изменения,

Иногда метод *ToString* не просто форматирует строку Если изменить параметр местоположения компьютера на одну из арабских стран, то в диалоговом окне

[2] RFC расшифровывается как «request for comment" (запрос на комментарии) и означает способ распространения стандартов Интернета. Спецификация на отображение времени и даты в RFC 1123 — это несколько измененная версия спецификации RFC 822. Стандарты RFC доступны на многих Web-узлах, например *http://www.ietf.org*.

[3] Стандарт ISO 8601 («Date elements and interchange formats—Information interchange—Representation of dates and times») можно получить на Web-узле ISO по адресу *http://www.iso.cb*. Стандарт ISO 8601 определяет набор форматов отображения даты и времени. Формат, используемый методом *ToString* объекта *DateTime,* является первым из трех расширенных форматов, описанных в параграфе 5.4.1.a.

Regional Options (Язык и стандарты) можно будет выбрать Григорианский или исламский календарь. Задав параметру расположения значение «Hebrew», вы сможете выбрать либо Григорианский, либо еврейский календарь. Если выбран еврейский или исламский календарь, то при выводе данных в локализованном формате дата будет преобразована автоматически.

Форматы даты и времени метода *ToString* (для Германии)

Символ, обозначающий формат	Аргумент, определяющий форматирование *CurrentInfo* для Германии	*InvariantInfo*
d	01.06.2002	06/01/2002
D	Samstag, 1. Juni 2002	Saturday, 01 June 2002
f	Samstag, 1. Juni 2002 15:05	Saturday, 01 June 2002 15:05
F	Samstag, 1. Juni 2002 15:05:01	Saturday, 01 June 2002 15:05:01
g	01.06.2002 15:05	06/01/2002 15:05
G или *null*	01.06.2002 15:05:01	06/01/2002 15:05:01
m или *M*	01 Juni	June 01
r или *R*	Sat, 01 Jun 2002 15:05:01 GMT	Так же, как для *CurrentInfo*
s	2002-06-01T15:05:01	Так же, как для *CurrentInfo*
t	15:05	15:05
T	15:05:01	15:05:01
u	2002-06-01 15:05:01Z	Так же, как для *CurrentInfo*
U	Samstag, 1. Juni 2002 19:05:01	Saturday, 01 June 2002 19:05:01
y или *Y*	Juni 2002	2002 June

Простые локализованные часы

Следующая программа использует аргумент форматирования *F* для вывода даты и времени в клиентской части формы.

SimpleClock.cs

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

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

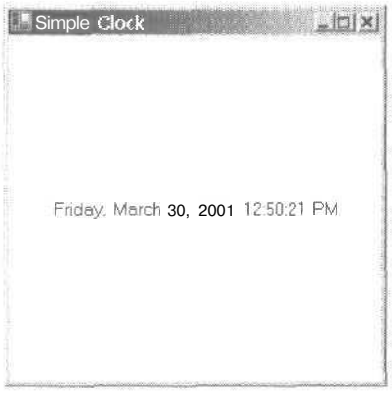
См. след. стр.

```
        Text - "Simple Clock";
        BackColor * SystemColors.Window;
        ForeColor - SystemColors.WindowText;

        Timer timer    = new Timer();
        timer.Tick    += new EventHandler(TimerOnTick);
        timer.Interval = 1000;
        timer.Start();
    }
    private void TimerOnTick(object sender, EventArgs ea>
    {
        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        StringFormat strfmt  = new StringFormat();
        strfmt.Alignment     - StringAlignment.Center;
        strfmt.LineAlignment = StringAlignment.Center;

        pea.Graphics.DrawString(DateTime.Now.ToString("F"),
                                Font, new SolidBrush(ForeColor),
                                ClientRectangle, strfmt);
    }
}
```

Программа инициализирует таймер на срабатывание каждую секунду и при обработке события *OnTick* просто делает клиентскую часть формы недействительной. Вот как это выглядит в случае стандартного формата для США:



Но позвольте спросить: мы что, потратили столько времени на изучение работы со шрифтами только для того, чтобы создать такие убогие часы? Вряд ли.

Давайте ограничимся лишь отображением времени и максимально увеличим шрифт. Для этого потребуется лишь более творчески взглянуть на метод *OnPaint*.

DigitalClock.cs

```
//-------------------------------------------
// DigitalGlock.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;
class DigitalClock: Form
{
     public static void Main()
     {
          Application.Run(new DigitalClock());
     }
     public DigitalClock()
     {
          Text = "Digital Clock";
          BackColor - SystemColors.Window;
          ForeColor = SystemColors.WindowText;
          ResizeRedraw - true;
          MinimumSize = SystemInformation.MinimumWindowSize + new Size(0, 1);

          Timer timer    = new Timer();
          timer.Tick   += new EventHandler(TimerOnTick);
          timer.Interval= 1000;
          timer.Start();
     }
     private void TimerOnTick( object obj, EventArgs ea)
     {
          Invalidate();
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx    = pea.Graphics;
          string   strTime = DateTime.Now.ToString("T");
          SizeF    sizef   - grfx.MeasureString(strTime, Font);
          float    fScale  = Math.Min(ClientSize.Width  / sizef.Width,
                                      ClientSize.Height / sizef.Height);
          Font     font    = new Font(Font.FontFamily,
                                      fScale * Font.SizeInPoints);

          sizef = grfx.MeasureString(strTime, font);

          grfx.DrawString(strTime, font, new SolidBrush(ForeColor),
                          (ClientSize.Width  - sizef.Width ) / 2,
                          (ClientSize.Height - sizef.Height) / 2);
     }
}
```

В методе *OnPaint* строка, содержащая сведения о времени в нужном формате, сохраняется в переменной *strTime* и затем, используя прием о котором я говорил

в главе 9, устанавливает максимально возможный для данных размеров клиентской области размер шрифта. Вот как это выглядит (для местных параметров США):



Увы, для того чтобы время можно было увидеть из любой точки комнаты, мы пожертвовали отображением даты. Возможно ли отображать и то и другое, сохраняя при этом большой размер текста? Конечно! Весь фокус в том, что для этого не стоит использовать форматы, в которых дата и время выводятся в одной строке, а разделить их, отображая на экране друг под другом. Для этого между датой и временем поместим символ перевода строки.

DigitalClockWithDate.cs

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

class DigitalClockWithDate : DigitalClock
{
     public new static void Main()
     {
          Application.Run(new DigitalClockWithDate());
     }
     public DigitalClockWithDate()
     {
          Text += " with Date";
     }
     protected override void OnPaint(PaintEventArgs pea)
     {
          Graphics grfx    = pea.Graphics;
          DateTime dt      = DateTime.Now;
          string   strTime = dt.ToString("d") + "\n" + dt.ToString("T");
          SizeF    sizef   = grfx.MeasureString(strTime, Font);
          float    fScale  = Math.Min(ClientSize.Width  / sizef.Width,
                                      ClientSize.Height / sizef.Height);
```

см. след. стр.

```
            Font font = new Font(Font.FontFamily,
                                 fScale * Font.SizeInPoints);
            StringFormat strfmt = new StringFormat();
            strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

            grfx.DrawString(strTime, font, new SolidBrush(ForeColor),
                            ClientRectangle, strfmt);
        }
    }
```

Метод *Measure-String* возвращает высоту двух строк текста и ширину самой длинной из них. Чтобы расположить строки в центре клиентской области, в методе *DrawString* я применил объект *StringFormat*. Для местных параметров США это выглядит так:



Вызов метода *OnPaint* каждую секунду может заставить вас беспокоиться об эффективности участка кода, отвечающего за прорисовку клиентской области. Не лучше ли, например, создавать требуемый шрифт в методе *OnResize*? Лучше. Но чтобы все это работало правильно, надо потрудиться. Размер шрифта зависит от размера клиентской области и высоты и ширины строки текста. Конечно, обычно ширина текстовой строки не меняется ежесекундно. Но иногда она все-таки меняется. Ширина строки, указывающей время, зависит от того, одной или двумя цифрами отображается текущий час. А если время отображается в 24-часовом формате, строка с датой получается шире, и эта ширина зависит от названия месяца и числа.

Если для создания подходящего шрифта в программе предусмотрен отдельный метод, он должен содержать в своих полях сам шрифт и ширину текста, использующуюся для вычисления нужного размера шрифта. Метод вычисления размера шрифта, очевидно, должен вызываться из метода *OnResize*. Но он должен вызываться также и из метода *OnPaint,* если с момента последнего изменения размера шрифта ширина строки изменилась.

Часы в стиле ретро

Для оформления программы с часами вы вправе использовать любой шрифт TrueType, установленный на вашем компьютере. Просто введите в конструктор программы оператор, изменяющий свойство *Font* формы:

```
Font = new Font("Comic Sans MS", 12);
```

А затем позвольте методу *OnPaint* определить нужный размер шрифта,

Чтобы придать часам вид «ретро», можно выбрать шрифт, выглядящий как 7-сегментный жидкокристаллический дисплей. Вместо такого шрифта можно использовать класс *SevenSegmentDisplay*.

SevenSegmentDisplay.cs

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

namespace Petzold.ProgrammingWindowsWithCSharp
{
class SevenSegmentDisplay
{
    Graphics grfx;

        // Определяем высвечивающиеся сегменты для всех десяти цифр.

    static byte[,] bySegment = {{1, 1, 1, 0, 1, 1, 1},      // 0
                               {0, 0, 1, 0, 0, 1, 0},       // 1
                               {1, 0, 1, 1, 1, 0, 1},       // 2
                               {1, 0, 1, 1, 0, 1, 1},       // 3
                               {0, 1, 1, 1, 0, 1, 0},       // 4
                               {1, 1, 0, 1, 0, 1, 1},       // 5
                               {1, 1, 0, 1, 1, 1, 1},       // 6
                               {1, 0, 1, 0, 0, 1, 0},       // 7
                               {1, 1, 1, 1, 1, 1, 1},       // 8
                               {1, 1, 1, 1, 0, 1, 1}};      // 9

        // Координаты, определяющие положение каждого из сени сегментов.

    readonly Point[][] apt = new Point[7][];

    public SevenSegmentDisplay(Graphics grfx)
    {
        this.grfx = grfx;
                // Инициализация невыровненного массива Point.

        apt[0] = new Point[] {new Point( 3,  2), new Point(39,  2),
                              new Point(31, 10), new Point(11, 10)};

        apt[1] = new Point[] {new Point< 2,  3), new Point(10, 11),
                              new Point(10, 31), new Point( 2, 35)};

        apt[2] = new Point[] {new Point(40,  3), new Point(40, 35),
```

см. след. стр.

```
                              new Point(32, 31), new Point(32, 11)};

        apt[3] = new Point[] {new Point( 3, 36), new Point(11, 32),
                              new Point(31, 32), new Point(39, 36),
                              new Point(31, 40), new Point(11, 40)};

        apt[4] = new Point[] {new Point( 2, 37), new Point(10, 41),
                              new Point(10, 61), new Point( 2, 69)};

        apt[5] = new Point[] {new Point(40, 37), new Point(40, 69),
                              new Point(32, 61), new Point(32, 41)};

        apt[6] = new Point[] {new Point(11, 62), new Point(31, 62),
                              new Point(39, 70), new Point( 3, 70)};
    }
    public SizeF MeasureString(string str, Font font)
    {
        SizeF sizef = new SizeF(0, grfx.DpiX * font.SizeInPoints / 72);

        for (int i = 0; i < str.Length; i++)
        {
            if (Char.IsDigit(str[i]))
                sizef.Width += 42 * grfx.DpiX * font.SizeInPoints
                                                        / 72 / 72;
            else if (str[i] == ':')
                sizef.Width += 12 * grfx.DpiX * font.SizeInPoints
                                                        / 72 / 72;
        }
        return sizef;
    }
    public void DrawString(string str, Font font,
                           Brush brush, float x, float y)
    {
        for (int i = 0; i < str.Length; i++)
        {
            if (Char.IsDigit(str[i]))
                x = Number(str[i] - '0', font, brush, x, y);

            else if (str[i] == ':')
                x = Colon(font, brush, x, y);
        }
    }
    float Number(int num, Font font,
                 Brush brush, float x, float y)
    {
        for (int i = 0; i < apt.Length; i++)
            if (bySegment[num, i] == 1)
                Fill(apt[i], font, brush, x, y);

        return x + 42 * grfx.DpiX * font.SizeInPoints / 72 / 72;
```

см. след. стр.

```
    }
    float Colon(Font font, Brush brush, float x, float y)
    {
        Point[][] apt = new Point[2][];

        apt[0] = new Point[] {new Point( 2, 21), new Point( 6, 17),
                              new Point(10, 21), new Point( 6, 25)};

      ; apt[1] = new Point[] {new Point( 2, 51), new Point( 6, 47),
                              new Point(10, 51), new Point( 6, 55)};

        for <int i = 0; 1 < apt.Length; i++)
            Fill(apt[i], font, brush, x, y);

        return x + 12 * grfx.DpiX * font.SizeInPoints / 72 / 72;
    }
    void Fill(Point[] apt, Font font, Brush brush, float x, float y)
    {
        PointF[] aptf = new PointF[apt.Length];

        for (int i = 0; i < apt.Length; i++)
        {
            aptf[i].X = x + apt[i].X * grfx.DpiX *
                                       font.SizeInPoints / 72 / 72;
            aptf[i].Y в y + apt[i].Y * grfx.DpiY *
                                       font.SizeInPoints / 72 / 72;
        }
        grfx.FillPolygon(brush, aptf);
    }
}
}
```

Класс *SevenSegmentDisplay* содержит один открытый конструктор, которому передается аргумент типа *Graphics*, и два открытых метода *MeasureString* и *DrawString*, которым передаются такие же аргументы, что и наиболее популярным версиям этих методов из класса *Graphics*. Идея в том, чтобы создать объект *SevenSegmentDisplay*, которому требуются такие же аргументы, что и объекту *Graphics*, и использовать эти методы вместо методов класса *Graphics*.

Метод *DrawString* класса *SevenSegmentDisplay* может обрабатывать только 11 кодов символов: коды для 10 цифр и двоеточия. Для этих двух случаев в нем предусмотрены вызовы закрытых методов *Number* и *Colon*. *Number* использует статический массив *bySegment*, который содержит данные о том, какие из семи сегмент<в «зажигать» для каждой из десяти цифр. (Вероятно, в этом массиве вместо типа *byte* следовало бы использовать элементы типа *bool*, но я подумал, что список значений *true* и *false* был бы трудночитаемым. Кроме того, это вряд ли сильно оптимизировало бы машинный код.) Невыровненный, *доступный только для чтения* массив типа *Point* с именем *apt* задает координаты для каждого из семи сегментов индикатора. Эти координаты рассчитываются для символа шириной 42 и высотой 7 2.

Закрытый метод *Fill* масштабирует эти координаты с учетом размера шрифта и с помощью метода *FillPolygon* закрашивает внутреннюю часть красным цветом.

Использующая этот класс программа-часы почти идентична DigitalClock за исключением того, что метод *OnPaint* начинается с создания объекта *SevenSegmentDisplay*, который затем служит для вызовов *MeasureString* и *DrawString* вместо объекта *Graphics*.

SevenSegmentClock.cs

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

Glass SevenSegmentClock: Form
{
     DateTime dt;  :

     public static void Main()
     {
          Application.Run(new SevenSegmentClock());
     }
     public SevenSegmentClock()
     {
          Text = "Seven-Segment Clock";
          BackColor = Color.White;
          ResizeRedraw * true;
          MinimumSize = SystemInformation.MinimumWindowSize + new Size(0, 1);

          dt = DateTime.Now;

          Timer timer    = new Timer();
        , timer.Tick    += new EventHandler(TimerOnTick);
          timer.Interval = 100;
          timer.Enabled  ~ true;
     }
     void TimerOnTick(object obj, EventArgs ea)
     {
          DateTime dtNow = DateTime.Now;
          dtNow = new DateTime(dtNow.Year, dtNow.Month, dtNow.Day, .
                               dtNow.Hour, dtNow.Minute, dtNow.Second);
          if (dtNow != dt)
        "..{
               dt = dtNow;
               Invalidate();
          }
```

см. след. стр.

```
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        SevenSegmentDisplay ssd = new SevenSegmentDisplay(pea.Graphics);

        string strTime = dt.ToString("T",
                                     DateTimeFormatInfo.InvariantInfo);
        SizeF  sizef   = ssd.MeasureString(strTime, Font);
        float  fScale  = Math.Min(ClientSize.Width  / sizef.Width,
                                  ClientSize.Height / sizef.Height);
        Font   font    = new Font(Font.FontFamily,
                                  fScale * Font.SizeInPoints);

        sizef = ssd.MeasureString(strTime, font);

        ssd.DrawString(strTime, font, Brushes.Red,
                       (ClientSize.Width  - sizef.Width) / 2,
                       (ClientSize.Height - sizef.Height) / 2);
    }
}
```

Заметьте: я использовал регионально-независимый метод *ToString* класса *DateTime*. Эта программа — отличный пример того случая, в котором лучше работать с регионально-независимой строкой, поскольку требуется точно знать, с какими символами она будет работать, и не сталкиваться с обозначениями а. т. или р. т.:



Аналоговые часы

Популярность цифровых часов была обусловлена их новизной, но, если можно так выразиться, маятник качнулся назад к стрелочным часам. При создании аналоговых часов нет необходимости разбираться с различными форматами даты и времени, но сложность графики перевешивает это преимущество. Пользователь вправе ожидать от таких часов динамического изменения размера вместе с изменением размера окна.

Для большей универсальности я решил реализовать логику отображения часов в качестве дочернего элемента управления окна подобно классу *CheckerChild* в программе CheckerWithChildren в главе 8. Это облегчит введение часов в другое приложение или написание приложения, отображающего множество циферблатов. Вот листинг класса *ClockControl*.

ClockControl.cs

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

namespace Petzold.ProgrammingWindowsWithCSharp
{
class ClockControl: UserControl
{
     DateTime dt;

     public ClockControl()
     {
          ResizeRedraw = true;
          Enabled - false;
     }
     public DateTime Time
     I
          get
          {
               return dt;
          }
          set
          {
               Graphics grfx = CreateGraphics();
               InitializeCoordinates(grfx);

               Pen pen = new Pen(BackColor);

               if (dt.Hour != value.Hour)
               {
                    DrawHourHand(grfx, pen);
               }
               if (dt.Minute != value.Minute)
               {
                    DrawHourHand(grfx, pen);
                    DrawMinuteHand(grfx, pen);
               }
               if (dt.Second != value.Second)
```

см. след. стр.

```
            {
                DrawMinuteHand(grfx, pen);
                DrawSecondHand(grfx, pen);
            }
            if (dt.Millisecond != value.Millisecond)
            {
                DrawSecondHand(grfx, pen);
            }
            dt = value;
            pen = new Pen(ForeColor);

            DrawHourHand(grfx, pen);
            DrawMinuteHand(grfx, pen);
            DrawSecondHand(grfx, pen);

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

        InitializeCoordinates(grfx);
        DrawDots(grfx, brush);
        DrawHourHand(grfx, pen);
        DrawMinuteHand(grfx, pen);
        DrawSecondHand(grfx, pen);
    }
    void InitializeCoordinates(Graphics grfx)
    {
        if (Width == 0 || Height == 0)
            return;

        grfx.TranslateTransform(Width / 2, Height / 2);

        float finches = Math.Min(Width / grfx.DpiX, Height / grfx.DpiY);

        grfx.ScaleTransform(fInches * grfx.DpiX / 2000,
                            finches * grfx.DpiY / 2000);
    }
    void DrawDots(Graphics grfx, Brush brush)
    {
        for (int i = 0; i < 60; i++)
        {
            int iSize = i % 5 == 0 ? 100 : 30;

            grfx.FillEllipse(brush, 0 - iSize / 2, -900 - iSize / 2,
                                    iSize, iSize);
```

оп. след. стр.

```
            grfx.RotateTransform(6);
        }
    }
    protected virtual void DrawHourHand(Graphics grfx, Pen pen)
    {
        GraphicsState gs = grfx.Save();
        grfx.RotateTransform(360f * Time.Hour / 12 *
                             30f * Time.Minute / 60);

        grfx.DrawPolygon(pen, new Point[]
                         {
                             new Point(0,  150), new Point( 100, 0),
                             new Point(0, -600), new Point(-100, 0)
                         });
        grfx.Restore(gs);
    }
    protected virtual void DrawMinuteHand(Graphics grfx, Pen pen)
    {
        GraphicsState gs * grfx.Save();
        grfx- RotateTransform(360f * Time.Minute / 60 +
                              6f * Time.Second / 60);

        grfx.DrawPolygon(pen, new Point[]
                         {
                             new Point(0,  200), new Point( 50, 0),
                             new Point(0, -800), new Point(-50, 0)
                         });
        grfx.Restore(gs);
    >
    protected virtual void DrawSecondHand(Graphics grfx, Pen pen)
    ▮
        GraphicsState gs = grfx.Save();
        grfx.RotateTransform(360f * Time.Second / 60 +
                             6f * Time.Millisecond / 1000);

        grfx.DrawLine(pen, 0, 0, 0, -800);
        grfx.Restore(gs);
    }
}
}
```

Класс *ClockControl* унаследован от класса *UserControl* и переопределяет метод *OnPaint*. В конструкторе *ClockControl* элементу управления *ResizeRedraw* присваивается значение *true*, а элементу *Enabled — false*. Для работы классу *ClockControl* не нужен ввод с клавиатуры или мыши, так что этот ввод будет контролироваться родительскими объектами.

Обратите внимание на закрытое поле типа *DateTime*, которому я присвоил имя *dt*, и на доступное для чтения и записи, открытое свойство с именем *Time*, предоставляющее другим объектам доступ к этому полю. Элемент управления не использует собственный таймер и не устанавливает это свойство самостоятельно; он лишь

отображает время, соответствующее значению свойства *Time*. Обновление значения свойства *Time* входит в задачу класса, создающего экземпляр объекта *ClockControl*.

Код аксессора *set* свойства *Time*, выглядит чрезмерно длинным. Есть искушение сделать этот аксессор коротким — примерно так:

```
dt = value;
Invalidate();
```

Вызов *Invalidate* приведет к вызову метода *OnPaint*, а тот в свою очередь перерисует часы. Однако такое упрощение накличет беду. Вызов *Invalidate* приведет к тому, что фон элемента управления будет стерт и придется перерисовывать все часы целиком, что даст в результате раздражающее мерцание изображения. Я использовал более элегантный подход. Позвольте мне вернуться к аксессору *set* свойства *Time* после обсуждения метода *OnPaint*.

OnPaint создает перо и кисть, используя основной цвет элемента управления, и вызывает пять других методов. Прежде всего метод *InitializeCoordinates* устанавливает систему координат с началом в центре элемента управления и изотропными координатами, содержащими \ 000 точек на четырех направлениях.

Далее метод *DrawDots* рисует точки, отмечающие минуты и часы. Он использует методы класса *Graphics*: метод *FillEllipse* для вывода точки в позиции 12:00 и *RotateTransform* — для поворота на 6° к следующей точке. Методы *DrawHourHand*, *DrawMinuteHand* и *DrawSecondHand* также используют метод *RotateTransform*. Я сделал три этих метода виртуальными, чтобы иметь возможность переопределить их в дальнейшем (точнее, в программе из главы 13).

При самом рисовании (часовой и минутной стрелок методом *DrawPolygon* и секундной — методом *DrawLine*) предполагается, что стрелки направлены строго вверх. Вызов метода *RotateTransform* перед рисованием стрелок поворачивает их на требуемый угол. Каждая процедура рисования стрелок содержит вывод метода *Save* класса *Graphics* для сохранения текущего состояния перед вызовом *RotateTransform*. По завершении рисования вызывается метод *Restore*.

На положение часовой стрелки влияют два свойства структуры *DateTime*: *Hour* и *Minute*. Положение минутной стрелки зависит от свойств *Minute* и *Second*, а положение секундной — от свойств *Second* и *Millisecond*. Это позволяет стрелкам не перескакивать с позиции на позицию, а плавно двигаться по кругу.

Теперь можно вернуться к аксессору *set* свойства *DateTime*. После вызова *CreateGraphics* для получения объекта *Graphics* следует вызов метода *InitializeCoordinates*, инициализирующего систему координат. Затем создается перо фонового цвета. Это нужно для оптимального стирания изменяющей свое положение стрелки. Тут правда, есть небольшая проблема: стирание какой-либо стрелки может повредить две другие. Поэтому придется перерисовывать все три стрелки. Несмотря на большой объем рисования, такой подход значительно снижает мерцание изображения,

Теперь, имея такой элемент управления, создать форму, использующую его, будет довольно просто,

В конструкторе программа создает объект класса *ClockControl*, устанавливает свойство *Parent* элемента управления на форму и инициализирует свойство *Time* текущим значением даты и времени.

AnalogClock.cs

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

class AnalogClock: Form
{
     ClockControl clkctl;

     public static void Main()
     {
          Application.Run(new AnalogClock());
     }
     public AnalogClock()
     {
          Text = "Analog Clock";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

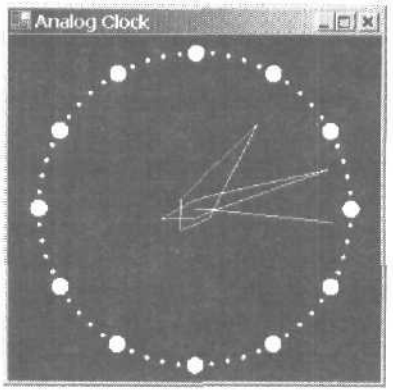
          clkctl = new ClockControl();
          clkctl.Parent    = this;
          clkctl.Time      = DateTime.Now;
          clkctl.Dock      = DockStyle.Fill;
          clkctl.BackColor = Color.Black;
          clkctl.ForeColor = Color.White;

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

Далее устанавливается свойство элемента управления, о котором я еще не упоминал, — *Dock*. Оно содержится в классе *Control, и* в главе 12 я расскажу о нем подробнее. Пока скажу лишь о том, что присвоение ему значения *DockStyle.Fill* растягивает элемент управления на всю поверхность предка. Таким образом, часы будут автоматически изменять свой размер, чтобы соответствовать размерам формы.

В последнюю очередь свойству *BackColor* сопоставляется черный цвет, а свойству *ForeColor* — белый. Это делается лишь для того, чтобы показать, что элемент

управления не управляет своим цветом. Это делает предок. Ну и, кроме того, белые часы на черном фоне выглядят довольно здорово:



Обработка конструктора завершается установкой таймера на интервал в 100 миллисекунд (1/10 секунды). Обычно часы требуется обновлять лишь раз в секунду, но если сделать так в данном случае, секундная стрелка будет перемещаться рывками. Обработчик события *TimerOnTick* просто устанавливает свойство *Time* элемента управления на текущую дату и время.

Если программе не требуется плавного перемещения секундной стрелки, интервал срабатывания таймера можно установить в 1 000 миллисекунд и присваивать свойству *Time* объект *DateTime*, содержащий в своем свойстве *Millisecond* значение 0. Поскольку свойство *Millisecond* доступно только для чтения, его обнуление потребует повторного создания объекта *DateTime*. Тогда обработчик *TimerOnTick* будет выглядеть так:

```
DateTime dt = DateTime.Now;
dt = new DateTime(dt.Year, dt.Month, dt.Day, dt.Hour, dt.Minute, dt.Second);
clkctl.Time = dt;
```

Есть и другие способы показать, что отображаемое элементом управления значение времени полностью контролируется предком. Попробуйте заменить код обработчика события таймера *TimerOnTick* на;

```
clkctl.Time += new TimeSpan(10000000);
```

Часы устанавливаются на правильное время, но идут в 10 раз быстрее. А если попробовать:

```
clkctl.Time -= new TimeSpan(1000000);
```

то часы будут идти с нормальной скоростью, но в обратном направлении.

Пятнашки

Пришла пора написать игру. Ну, что-то вроде головоломки. Она была изобретена в 1870-х, вероятно, знаменитым американским создателем головоломок Сэмом Ллойдом (1841–1911). В течение некоторого времени она была весьма популярна, особенно в Европе, и была известна под разными именами, например, «пятнашки» в России и *Jeu de Taquin* («задира») во Франции,

В своем классическом виде игра состоит из пятнадцати квадратных блоков, пронумерованных от 1 до 15. Квадраты находятся в рамке 4X4, и одно место остается пустым. Квадраты можно передвигать внутри рамки в горизонтальном или вертикальном направлении на пустую позицию, изменяя таким образом позицию и самого пустого места,

Сэм Ллойд располагал все квадраты в правильном порядке, за исключением двух последних (с номерами 14 и 15), которые были переставлены местами. Он предлагал 1 000 долларов тому, кто сумеет упорядочить квадраты. Награды никто так и не получил, поскольку в такой позиции головоломка не имеет решения[4].

Эта головоломка была воплощена в одной из первых компьютерных игр, написанных для Apple Macintosh, которая называлась PUZZLE. Также она входила в состав ранних версий SDK Microsoft Windows под именем MUZZLE и представляла собой единственный пример программы в SDK, написанной не на C, а на Microsoft Pascal. Обе эти программы отображали 15 квадратов в упорядоченном виде и меню для их перемешивания. Вам следовало восстановить исходное состояние или расположить квадраты в другом порядке, например, не построчно, а колонками. Поскольку мы еще не работали с меню, моя программа будет перемешивать квадраты при запуске. (Тут нам понадобится таймер.)

В качестве фишек мы используем дочерние окна, свойству *Enabled* которых будет присвоено значение *false*, что позволит предку контролировать ввод с клавиатуры и мыши. Как правило, текст на недоступных элементах управления отображается серым шрифтом, но это не обязательно и в нашем случае не понадобится. Для отображения граней, придающих трехмерность, метод *OnPaint* использует обычные цвета.

JeuDeTaqu1nTile.cs

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

class JeuDeTaquinTile: UserControl
{
    int iNum;

    public JeuDeTaquinTile(int iNum)
```

см. след. стр.

[4] Впервые математический анализ был применен к пятнашкам в 1879 г. в одной из статей *American Journal of Mathematics*. Лежащие в основе этой игры математические принципы были обобщены в книге Джеймса Ньюмена «Мир математики» [James R. Newman. *The World of Mathematics* (New York: Simon and Schuster, 1956), 4: 2429-2432]. Этот четырехтомник был переиздан в 1988 г. издательством Tempus Books и в 2000 г. — издательством Dover Books. О пятнашках рассказывается на страницах 2405 — 2408 издания Tempus.

```
    {
        this.iNum = iNum;
        Enabled = false;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

        grfx.Clear(SystemColors.Control);

        int cx = Size.Width;
        int cy = Size.Height;
        int wx = SystemInformation.FrameBorderSize.Width;
        int wy = SystemInformation.FrameBorderSize.Height;

        grfx.FillPolygon(SystemBrushes.ControlLightLight,
            new Point[] {new Point( 0, cy), new Point(0, 0),
                         new Point(cx, 0), new Point(cx - wx, wy),
                         new Point(wx, wy), new Point(wx, cy - wy)});

        grfx.FillPolygon(SystemBrushes.ControlDark,
            new Point[] { new Point(cx, 0), new Point(cx, cy),
                          new Point(0, cy), new PointCwx, cy - wy),
                          new Point(cx - wx, cy - wy),
                          new Point(cx - wx, wy)});

        Font font = new Font("Arial", 24);
        StringFormat strfat = new StringFormat();
        strfmt.Alignment = strfmt.LineAlignment = StringAlignment.Center;

        grfx.DrawString(iNum.ToString(), font, SystemBrushes.ControlText,
                        ClientRectangle, strfmt);
    }
}
```

Программа, создающая фишки и передвигающая их, немного сложнее. В качестве фишки она создает элемент управления (и соответствующим образом устанавливает размеры клиентской области) переопределяя метод *OnLoad* класса *Form*. Метод *OnLoad* вызывается вскоре после первого отображения формы на экране; исходя из собственного опыта, отмечу, что создавать объект *Graphics* и устанавливать размер клиентской области лучше именно в методе *OnLoad*, а не в конструкторе. Обработка *OnLoad* завершается вызовом защищенного метода *Randomize* — он перемешивает фишки при помощи таймера.

JeuDeTaquin.cs

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

class JeuDeTaquin: Form
{
     const int           nRows = 4;
     const int           nCols = 4;
     Size                sizeTile;
     JeuDeTaquinTile[,] atile = new JeuDeTaquinTile[nRows, nCols];
     Random              rand;
     Point               ptBlank;
     int                 iTimerCountdown;

     public static void Main()
     {
          Application.Run(new JeuDeTaquin() );
     }
     public JeuDeTaquin()
     {
          Text = "Jeu De Taquin";
          BorderStyle = FormBorderStyle.Fixed3D;
     }
     protected override void OnLoad(EventArgs ea)
     {
               // Вычисление размеров фишек на форме.

          Graphics grfx = CreateGraphics();

          sizeTile  = new Size((int)(2 * grfx.DpiX / 3),
                               (intX2 * grfx.DpiY / 3));
          ClientSize = new Size(nCols * sizeTile.Width,
                                nRows * sizeTile.Height);
          grfx.Dispose();

               // Создание фишек.

          for (int iRow = 0; iRow < nRows; iRow++)
          for (int iCol = 0; iCol < nCols; iCol++)
          {
               int iNum = iRow  * nCols + iCol + 1;

               if (iNum == nRows * nCols)
                    continue;
```

см. след. стр.

```
            JeuDeTaquinTile tile = new JeuDeTaquinTile(iNum);
            tile.Parent = this;
            tile.Location = new Point(iCol * sizeTile.Width,
                                      iRow * sizeTile.Height);
            tile.Size = sizeTile;

            atile[iRow, iCol] = tile;
        >
        ptBlank = new Point(nCols - 1, nRows - 1);

        Randomize();
    }
    protected void fiandoroizs0
    {
        rand=newflftftdom0;
        iTimerCountdown = 64 * nRows * nCols;

        Timer timer    = new Timer();
        timer.Tick     += new EventHandler(TimerOnTick);
      , timer.Interval = 1;
        timer.Enabled  = true;
    }
    void TimerOnTick(object obj, EventArgs ea)
    {
        int x = ptBlank.X;
        int y = ptBlank.Y;

        switch(rand.Next(4))
        {
        case 0:  x++;  break;
        case 1;  x-;   break;
        case 2:  y++;  break;
        case 3:  y-;   break;
        }
        if (x >= 0 44 x < nCols && y >= 0 && y < nRows)
            MoveTile(x, y);

        if (-iTimerCountdown == 0)
        {
            ((Timer)obj).Stop();
            ((Timer)obj).Tick -= new EventHandler(TimerOnTick);
        }
    }
    protected override void OnKeyDown(KeyEventArgs kea)
    {
        if (kea.KeyCode == Keys.Left && ptBlank.X < nCols - 1)
            MoveTile(ptBlank.X * 1, ptBlank.Y);

        else if (kea.KeyCode == Keys.Right && ptBlank.X > 0)
```

см. след. стр.

```
                MoveTile(ptBlank.X - 1, ptBlank.Y);

            else if (kea.KeyCode == Keys.Up && ptBlank.Y < nRows - 1)
                MoveTile(ptBlank.X, ptBlank.Y + 1);

            else if (kea.KeyCode == Keys.Down && ptBlank.Y > 0)
                MoveTile(ptBlank.X, ptBlank.Y - 1);

            kea.Handled = true;
        }
        protected override void OnMouseDown(MouseEventArgs mea)
        {
            int x = mea.X / sizeTile.Width;
            int y = mea.Y / sizeTile.Height;

            : if (x == ptBlank.X)
              {
                if (y < ptBlank.Y)
                    for (int y2 = ptBlank.Y - 1; y2 >= y; y2--)
                        MoveTile(x, y2);

                else if (y > ptBlank.Y)
                    for (int y2 = ptBlank.Y + 1; y2 <= y; y2++)
                        MoveTile(x, y2);
            }
            else if <y == ptBlank.Y)
            {
                if (x < ptBlank.X)
                    for (int x2 = ptBlank.X - 1; x2 >= x; x2--)
                        MoveTile(x2, y);

                else if (x > ptBlank.X)
                    fsr (int x2 = ptBlank.X + 1; x2 <= x; x2++)
                        MoveTile(x2, y);
            }
        }
        void MoveTile(int x, int y)
        {
            atile[y, x].Location = new Point(ptBlank.X * sizeTile.Width,
                                             ptBlank.Y * sizeTile.Height);

            atile[ptBlank.Y, ptBlank.X] * atile[y, x];
            atile[y, x] = null;
            ptBlank = new Point(x, y);
        }
}
```

Остальная часть программы просто обрабатывает ввод с клавиатуры и мыши, приводя к вызову метода *MoveTile*, представленного в конце листинга,

Двухмерный массив *atile* содержит объекты-фишки. Например, объект хранящийся в элементе *atile[3,1]*, представляет собой фишку из четвертого ряда второй колонки. Один из элементов массива *atile* всегда хранит значение *null*. Этот элемент соответствует пустой клетке на доске. Поле *ptBlank* также хранит координату пустой клетки. *Пустышка* — как я ее называю — управляет интерфейсом пользователя; так что *ptBlank* играет главную роль в коде пользовательского интерфейса. Любая перемещаемая программой фишка должна соседствовать с пустой клеткой и перемещаться на ее место.

Однако при использовании мыши не обязательно щелкать фишку, расположенную по соседству с пустой клеткой. Если вы щелкните фишку из ряда или колонки, в которой есть пустышка, программа разом переместит несколько фишек, произведя несколько вызовов метода *MoveTile*. Этот метод физически перемещает фишку (устанавливая ее свойство *Location* согласно положению пустышки) и соответствующим образом изменяет массив *atile* и поле *ptBlank*.

С клавиатуры программа управляется клавишами со стрелками. Если подумать, нажатие любой из этих клавиш имеет вполне ясное значение. Так, нажатие клавиши «вниз» перемещает фишку, расположенную над пустым местом (если она там есть), на это место.

Вот как выглядела эта программа, когда я наполовину собрал головоломку:



Создание собственного элемента управления оказалось полезным уже для третьей программы в книге. Как вам. несомненно, известно, Windows и Windows Forms .NET Framework содержат массу готовых элементов управления: кнопки, надписи, текстовые поля, списки, полосы прокрутки и многие другие. Мы подробно исследуем их в главе 12.



ГЛАВА

11

Изображения и битовые карты

Компьютерную графику обычно подразделяют на *векторную* и *растровую*. В математике вектор — это комбинация длины и направления, он может быть представлен линией в координатном пространстве. Векторная графика — это построение линий, кривых и заполненных областей с применением аналитической геометрии. При использовании контурных шрифтов текст также можно считать векторной графикой.

Термин *растр* (raster) связан с технологией производства видеодисплеев и относится к построению составного изображения на основе нескольких строк развертки. В растровой графике изображения описываются прямоугольными массивами пикселов, которые называют *битовыми картами* (bitmap),

И векторная, и растровая графика связана с устройствами графического отображения информации, Сегодня используются в основном растровые устройства, В лазерных и струйных принтерах печатаемое изображение представляет собой прямоугольный массив пикселов, закрашенных точками. Изображение на мониторах с электронно-лучевой трубкой (ЭЛТ) — это набор горизонтальных строк развертки, состоящих из последовательностей пикселов. Биты, определяющие ЭЛТ-изображение, хранятся в памяти видеоплаты.

Сейчас растровые устройства вывода широко распространены, однако на заре компьютерной графики (1950-е) память была слишком дорогой для создания растровых дисплеев. Подсоединенный к компьютеру видеодисплей работал как осциллограф: электронный луч не выводил горизонтальные строки развертки, а отклонялся под управлением компьютера, формируя прямые и кривые линии на экране. Векторные печатающие устройства также были довольно широко распространены и дошли до наших дней в виде плоттеров.

И векторы, и растры занимают свое место в мире компьютерной графики. Очевидно, что архитектурный чертеж следует выполнять, применяя векторную графику, а строить реалистичное изображение готового здания — растровую. Векторную и растровую графику обычно используют в разных приложениях: программы работы с *графикой* (drawing) основаны на векторной, а программы для *живописи* (painting) — на растровой графике. Программы обработки *фотоизображений* — это разновидность «живописных» программ, обрабатывающих реалистичные изображения с цифровых камер или отсканированные фотографии.

Как уже говорилось, размер и ориентацию векторных изображений можно изменять без потери разрешения. Так, благодаря масштабированию векторных контуров, шрифт размером 10 пт при десятикратном увеличении не будет иметь неровностей. Однако растровые изображения обычно аппаратно-зависимы, и это нельзя игнорировать. В битовых картах определены конкретные размеры пикселов. При увеличении битовой карты могут возникать неровности, а при уменьшении — потеря информации (в GDI+ для решения этих проблем применяются алгоритмы сглаживания). Растровые изображения также содержат специфическую информацию о цвете, которую не всегда можно представить на конкретном устройстве вывода.

Преобразовать векторное изображение в растровое очень легко: нужно лишь нарисовать на поверхности битовой карты разные линии, кривые, заполненные области и текст (мы проделаем это в этой главе). Однако обратное преобразование осуществляется гораздо труднее и возможно лишь для простых изображений.

Большинство старых, традиционных книг по компьютерной графике практически полностью посвящено векторной графике. Преобладание растровой графики сегодня — недавний феномен, обусловленный недорогой памятью, сканерами и цифровыми камерами. Его возникновению также способствовали технологии сжатия битовых карт, например JPEG, позволившие уменьшить объем памяти для хранения изображений,

В World Wide Web практически вся графика хранится в виде битовых карт; для большинства пользователей Web синонимами термина «компьютерная графика» являются JPEG- и GIF-файлы. Нельзя сказать, что это хорошо во всех отношениях. Гораздо эффективнее было бы хранить и передавать основную массу Web-графики в виде векторных изображений, учитывая, что зачастую исходное представление рисунков в программах подготовки изображений — это линии, кривые и заполненные области. И все же последние попытки распространения стандартов векторной графики для Web не слишком удались.[1]

[1] Подробнее о предлагавшемся стандарте Scalable Vector Graphics (SVG, масштабируемая векторная графика) см. на Web-узле World Wide Web Consortium по адресу *www.w3.org/Graphics/SVG*О языке Vector Markup Language (VML, язык векторной разметки), поддерживаемом последними версиями Internet Explorer — на Web-узле Microsoft по адресу *msdn.microsoft.com/workshop/author/vm*Обе технологии основаны на языке XML. Найти Web-узлы, представляющие графику с использованием SVG или VML, довольно трудно.

Поддержка битовых карт

Пространство имен *System.Drawing* включает два класса, *Image* и *Bitmap,* реализующих в .NET большую часть функций поддержки растровой графики. Классы *Bitmap* и *Metafile* (подробнее — в главе 23) являются производными от класса *Image*:

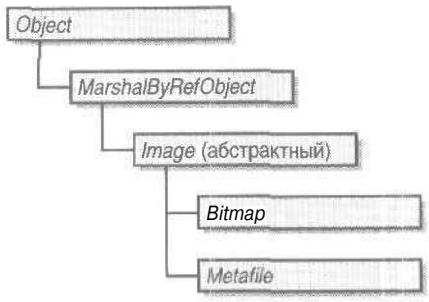


Image — абстрактный класс, экземпляр которого нельзя создать конструктором. И все же *Image* имеет два статических метода (если считать перегруженные — четыре), которые возвращают объекты типа *Image.* Эти весьма мощные методы позволяют загружать битовую карту или метафайл из файла или потока. Сразу можно предположить, что они работают с BMP-файлами — «родным» форматом битовых карт Windows. Что может показаться неожиданным, эти методы могут загружать файлы и в некоторых других растровых форматах, включая GIF, JPEG, PNG и TIFF. Класс *Image* имеет еще один статический метод, позволяющий создавать объект *Bitmap* на основе дескриптора битовой карты Win32.

Полученный объект *Image* можно вывести на экран или принтер одним из методов *DrawImage* класса *Graphics,* Кроме того, используя статический метод *FromImage* класса *Graphics,* можно вернуть объект *Graphics,* соответствующий изображению. Это позволит приложению рисовать на битовой карте. Дополнительные методы класса *Image* позволяют сохранить объект *Image* в одном из поддерживаемых форматов. Таким образом, класс *Image* имеет встроенную функцию преобразования форматов.

Если вам требуется лишь загружать и выводить растровые изображения, класс *Image* — это, вероятно, все, что вам нужно. Класс *Bitmap* расширяет класс *Image.* предоставляя конструкторы для создания новых битовых карт определенного размера и цветового формата. Кроме того. *Bitmap* позволяет напрямую считывать и записывать отдельные пикселы, а также обращаться к данным битовой карты как к блоку памяти.

В классе *Bitmap* также имеется конструктор, позволяющий загружать битовую карту, встроенную в .exe-файл в виде *ресурса.* Его можно использовать для загрузки значков и нестандартных курсоров. Мы еще обсудим двоичные ресурсы подробнее.

Иногда программисту трудно решить, какой класс использовать для конкретной задачи — *Image* или *Bitmap. Если* все нужные действия можно выполнить через класс *Image,* выбирайте его. Преимущество такого решения в том, что код будет работать и с метафайлами (за некоторыми исключениями).

Форматы файлов битовых карт

Битовая карта — это прямоугольный массив битов, соответствующих пикселам на графическом устройстве вывода. Битовая карта имеет определенную высоту и ширину, измеряемую в пикселах. Кроме того, она обладает определенной *насыщенностью цвета* (color depth), измеряемой числом битов на пиксел (bits per pixel, обычно сокращается как *bpp*). Все пикселы битовой карты имеют одинаковое число бит, которое определяет количество уникальных цветов в изображении;

$$\text{Число цветов} = 2^{\text{Число бит на пиксел}}$$

Число бит на пиксел может составлять от 1 до 32 (и более), при этом одни форматы распространены больше, другие — меньше.

В таких графических средах, как Windows, для представления цветов используются коды RGB (red-green-blue, «красный-зеленый-синий»), где каждый код основного цвета равен 1 байту, а полный код цвета RGB — 3 байтам, т. е. 24 битам. Такая схема довольно точно отражает способность человеческого глаза различать цвета. Кроме того, она примерно соответствует возможностям современных мониторов отображать разные цвета. Дополнительный байт может определять уровни прозрачности, начиная полной непроницаемостью и кончая полной прозрачностью. И все же 32 бита на пиксел — это не верхний предел. Некоторые приложения, например, программы для работы с изображениями в медицине, требуют большего числа бит на пиксел для повышенного разрешения.

Битовая карта с разрядностью 1 бит на пиксел содержит *двухуровневое* (bilevel), или *монохромное* (monochrome) изображение. Возможны лишь два цвета; обычно, но не всегда это черный и белый. Такая битовая карта, как правило, включает небольшую таблицу (или палитру) цветов, указывающую два цвета, связанных с двумя значениями битов.

На заре эпохи Windows были распространены изображения с разрядностью 4 бита на пиксел; их можно найти и сейчас. Например, значки зачастую являются 16-цветными изображениями. Эти 16 цветов представляют собой комбинацию красного, зеленого и синего основных цветов в стандартной и темной версиях. Такая битовая карта включает таблицу цветов, указывающую точные цвета, соответствующие 16 возможным комбинациям пикселов.

Разрядность одного из самых распространенных форматов битовых карт — 8 бит на пиксел. Зачастую изображение представляется в *градациях серого* (gray scale), и 8 бит соответствуют 256 или меньшему числу градаций серого, начиная черным и кончая белым. Цветные изображения также можно хранить в битовых картах с 8 битами на пиксел, тогда для изображения выбираются 256 или менее конкретных цветов. Отобранные цвета иногда называют *оптимизированной палитрой* (optimized palette) изображения.

В битовой карте с разрядностью 16 бит на пиксел обычно используется 5 бит для каждого из уровней красного, зеленого и синего; 1 бит остается незадействованным. Таким образом, код основного цвета может иметь 32 значения, что в целом дает 32 768 уникальных цветов. Иногда зеленому цвету выделяют дополнительный бит, так как именно к этому цвету наиболее чувствителен глаз человека. Такие битовые карты иногда называют «5-6-5», указывая число бит для каждого из основных цветов. Разрядность в 15 или 16 бит на пиксел обычно называется *сред-*

ним качеством цветопередачи (high color) и недостаточна для представления градаций цвета на реалистичных изображениях.

Разрядность *полноцветной* (full-color, true-color) битовой карты — 24 бита на пиксел. Каждый пиксел — это 24-битовый код цвета RGB. При использовании в битовой карте точно 3 байтов на пиксел возможны проблемы производительности: обычно наиболее эффективная работа 32-разрядных процессоров обеспечивается при доступе к 32-разрядным значениям на 32-разрядных границах памяти.

Можно использовать битовые карты с разрядностью 32 бита на пиксел, в которых реально используются 24 бита, а 1 байт добавлен только для оптимизации производительности. Этот дополнительный байт может также содержать сведения о прозрачности, называемые *альфа-каналом* (alpha channel). Альфа-значение определяет уровень прозрачности каждого конкретного пиксела,

Битовые карты могут быть очень большими. Так, если разрешение монитора — 1 600 X 1 200 пикселов, битовая карта с разрядностью 24 bpp и размером, равным размеру экрана, занимает около 5 Мб. Именно поэтому значительные усилия затрачивались на разработку методов *сжатия* изображений.

Один из простейших методов, известный практически всем, кто знает о сжатии изображений, — это *кодирование длин серий* (run-length encoding, RLE). Например, при наличии 12 последовательных голубых пикселов имеет смысл сохранить не повторяющиеся пикселы, а их число. RLE обычно подходит для изображений с ограниченным количеством цветов, например мультипликационных.

Программа, использующая более сложные алгоритмы, должна проанализировать данные на наличие повторяющихся комбинаций. Значительный прогресс в сжатии изображений имел место в конце 1970-х, когда Джейкоб Зиф (Jacob Zif) и Абрахам Лемпель (Abraham Lempel) опубликовали методы сжатия LZ77 и LZ78. Эти алгоритмы на лету выявляют совпадающие последовательности данных и эффективно их используют. В 1984 г. Терри Уэлч (Terry Welch) из Sperry Research Center (в настоящее время — часть Unisys), основываясь на алгоритме LZ78, описал принципы алгоритма LZW. На LZW основаны несколько популярных форматов сжатия изображений.[2] Однако недавно Unisys запретила применять алгоритм LZW без лицензии. Учитывая, что LZW уже использовался в нескольких укоренившихся стандартах (включая GIF), большая часть программистов пренебрегла требованиями Unisys; кроме того, этот алгоритм намеренно перестали использовать в новых форматах сжатия.

Алгоритм RLE и все производные алгоритма LZ — это методы сжатия *без потерь* (loseless), позволяющие полностью восстановить исходную информацию из сжатых данных (доказать, что конкретный алгоритм сжатия без потерь работает не для всех файлов, очень просто: после сжатия размер некоторых файлов вырастает!). Сжатие без потерь полезно при работе с электронными таблицами и текстовыми документами. Гораздо реже его применяют для реалистичных изображений, например оцифрованных фотографий,

---

[2] Хороший источник сведений об истории, технологии и программировании методов сжатия данных — книга *The Data Compression Book*. 2nd ed. (New York: M&T Books, 1985). Mark Nelson, Jean-Loup Gailly.

В связи с этим в последнее время, когда потребовалось сжатие фотографических изображений, стали популярны методы сжатия *с потерями* (lossy). Сжатие с потерями достигает наилучших результатов, опуская незаметные или малозаметные глазу человека данные. Однако при сильном сжатии с потерями возможно заметное ухудшение качества изображения.

Форматы файлов битовых карт, поддерживаемые классом *Image,* указываются в статических свойствах класса *ImageFormat,* определенного в пространстве имен *System.Drawing.Imaging:*

Статические свойства ImageFormat

Тип	Свойство	Доступ	Описание
ImageFonnat	*Bmp*	Чтение	Аппаратно-независимый растр Windows (device-independent bitmap, DIB)
ImageFormat	*MemoryBmp*	Чтение	DIB в памяти (без заголовка файла)
ImageFormat	*Icon*	Чтение	Формат значков Windows
ImageFormat	*Gij*	Чтение	Формат обмена графическими данными (CompuServe Graphics Interchange Format)
ImageFormat	*Jpeg*	Чтение	Формат Объединенной группы экспертов по обработке фотоизображений (Joint Photographic Experts Group)
ImageFormat	*Png*	Чтение	Переносимая сетевая графика (Portable Network Graphics)
ImageFormat	*"It*	Чтение	Теговый формат файлов изображений (Tag Image File Format)
ImageFormat	*Exif*	Чтение	Формат обмениваемых изображений (Exchangeable image format)
ImageFormat	*Wmf*	Чтение	Метафайл Windows (Windows metafile)
ImageFonnat	*Emf*	Чтение	Расширенный метафайл Windows (Windows enhanced metafile)

Возможно, вам знакомо большинство этих форматов. На всякий случай ниже дано краткое описание каждого из них.

- Bmp Собственный формаг файлов битовых карт Windows, или аппаратно-независимый растр (device-independent bitmap, DIB). Формат DIB был создан на основе формата битовых карт OS/2 1.1 и появился в Windows 3.0. До этого формат битовых карт в Windows основывался на требованиях конкретных устройств вывода. И хотя некоторые приложения использовали старый формат для обмена файлами, он не предназначался для этой цели.

  Основная часть DIB определена в документации определенных структур, используемых в Win32 API, в частности, *BITMAPFILEHEADERBITMAPINFO,BITMAPINFOHEADER* и их производных. Подробнее о формате DIB см. главу 15 моей книги «Programming Windows», 5th ed. (Microsoft Press, 1999). DIB-файлы обычно хранятся без сжатия. Для некоторых цветовых форматов применяется малоиспользуемая схема сжатия RLE.
- **MemoryBmp** DIB в памяти — DIB без заголовка *BITMAPFILEHEADER.*
- **Icon** Формат файлов значков Windows, расширение формата Windows DIB.

- Gif Произносится «джиф». Был разработан в конце 1980-х для сети CompuServe (одной из первых интерактивных информационных служб) и остается одним из двух самых популярных форматов графики в World Wide Web. В формате GIF применяется LZW-сжатие. Документы Gif87a.txt и Gif98a.txt, описывающие данный формат файлов, широко распространены в Интернете. Спецификация GIF включает элементарную (но популярную) функцию анимации.
- **Jpeg** Произносится «джейпег» и расшифровывается как Joint Photographic Experts Group (Объединенная группа экспертов по фотоизображениям) — собрание представителей крупных промышленных предприятий, разработавшее набор методов сжатия (как с потерями, так и без потерь) полутоновых фотоизображений. Официальный Web-узел JPEG находится по адресу *http://www.jpeg.org*. Действующая спецификация JPEG доступна в виде стандарта ISO, Ее печатную версию, а также другую полезную информацию см. в книге *JPEG: Still Image Data Compression Standard* by William B. Pennebaker and Joan L. Mitchell (New York: Van Nostrand Reinhold, 1993).

  Согласно стандарту JPEG *не является* форматом файлов. Правильное название того, что обычно называют форматом файлов JPEG — JPEG File Interchange Format (JFIF, формат обмена файлами JPEG). Подробнее о JFIF см. документ *http://www.jpeg.org/public/jfif.pdf*. JFIF включает метод сжатия JPEG с потерями и наряду с GIF стал одним из двух самых популярных форматов графики в Web (в спецификации HTML нет специального упоминания ни GIF, ни JPEG, и все же они стали стандартами Web-графики де-факто).
- **Png** Произносится «пинг». Portable Network Graphics — это формат сжатия без потерь, разработанный под протекцией World Wide Web Consortium (W3C) в качестве бесплатной и не требующей лицензирования альтернативы GIF. Большинство современных Web-браузеров поддерживают PNG, а также GIF и JPEG. Подробнее о формате PNG — по адресу *http://www.w3c.org/Graphics/PNG*. Если нужно сжимать нефотографические изображения и вы не хотите использовать GIF, лучшее решение — задействовать PNG.
- **Tiff** Формат Tag Image File Format разработан совместно компаниями Aldus (создателем популярной программы PageMaker) и Microsoft. Сейчас спецификация TIFF принадлежит Adobe, вы найдете ее по адресу *http://partners.adobe.com/asn/developer/pdfs/tn/TIFF6.pdf*.
- **Exif** Формат Exchangeable image format разработан организацией, которая сейчас называется Japan Electronic Industry Development Association (JEIDA, Японская ассоциация развития электронной промышленности), для цифровых камер. Спецификацию EXIF см. на Web-узле International Imaging Industry Association [I3A; ранее — Photographic and Imaging Association, Inc. (PIMA)] по адресу *http://www.pima.net/standards/it10/PIMA15740/Exif_2-1.PDF*.
- **Wmf** Не является форматом файлов битовых карт! Это формат метафайлов, которые представляют собой набор функций построения графики (обычно *векторной*), хранящихся в двоичной форме. WMF относится к *старому* формату метафайлов Windows, применявшемуся до 32-разрядных версий этой ОС.
- **Emf** Расширенный формат метафайлов Windows, появившийся в 32-разрядных версиях Windows. С метафайлами можно работать, как и с любыми дру-

гими изображениями, однако они относятся к отдельному классу. Подробнее о метафайлах см. главу 23 этой книги.

Загрузка и вывод изображений

Класс *Image* является абстрактным, т. е. его экземпляр нельзя создать конструктором. Однако, как уже упоминалось, он содержит четыре статических метода, позволяющих получить объект *Image,* и два метода для получения объекта *Bitmap*.

Статические методы класса *Image*

```
Image Image.FromFile(string strFilename)
Image Image.FromFile(string strFilename, bool bUseImageColorManagement)
Image Image.FromStream(Stream stream)
Image Image.FromStream(Stream stream, bool bUseImageColorManagement)
Bitmap Image.FromHbitmap(IntPtr hBitmap)
Bitmap Image.FromHbitmap(IntPtr hBitmap, IntPtr hPalette)
```

Два последних метода обычно применяют только при взаимодействии с Win32-кодом. Однако первые два очень мощны и просты в использовании:

```
Image image = Image.FromFile("CuteCat.jpg");
```

Одна из замечательных возможностей данного метода — то, что он определяет формат файла по его содержимому, а не по расширению. Например, если бы файл CuteCat.jpg был PNG-файлом, которому вы ошибочно присвоили неверное расширение, метод *FromFile* все равно бы выполнился. При невозможности найти или открыть файл, а также в случае проблем с его содержимым метод *FromFile* генерирует исключение.

Два других метода используют вместо имени файла объект типа *Stream*, однако файлы и потоки тесно связаны. *Stream* — это абстрактный класс пространства имен *System.IO*, реализующий такие методы, как *Read*, *Write* и *Seek*. Во многих случаях поток — это просто открытый файл. И все же объект *Stream* может также предоставлять последовательные данные, хранимые в блоке памяти или передаваемые по сети. Подробнее о файлах и потоках см. приложение А.

Логические аргументы методов *FromFile* и *FromStream* отвечают за управление цветом, но эта тема выходит за рамки данной книги.

Согласно документации первые четыре метода возвращают объект типа *Image,* однако, вызвав для возвращаемого значения метод *GetType*, вы обнаружите, что его тип — *System.Drawing.Bitmap* или *System.Drawing.Imaging.Metafile* в зависимости от типа загруженного файла (или потока).

В большинстве программ этой главы используется файл Apollo11FullColor.jpg, хранящийся на прилагаемом компакт-диске в каталоге Images And Bitmaps, содержащем проекты этой главы. Это знаменитый портрет космонавта Базза Олдрина, снятый Нилом Армстронгом при помощи камеры Hasselblad на поверхности Луны, Как следует из имени файла, он содержит полноцветную битовую карту с разрядностью 24 бита на пиксел. Для экспериментов в этой же папке имеются файлы Apollo11GrayScale.jpg и Apollo11Palette.png с разрядностью 8 бит на пиксел. Ширина изображения во всех трех файлах — 220, а высота — 240 пикселов. Разрешение составляет 72 dpi,

Для начала — небольшая программа, которая получает объект *Image* с помощью метода *ImageFromFile* и затем выводит его методом *DrawImage* объекта *Graphics*.

ImageFromFile.cs

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

class ImageFromFile: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new ImageFromFile());
     }
     public ImageFromFile()
     {
          Text = "Image From File";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Image image = Image.FromFile(". .\\..\\..\\Apollo11FullColor.jpg");
          grfx.DrawImage(image, 0, 0);
     }
}
```

Аргумент метода *FromFile* указывает расположение JPEG-файла относительно расположения файла ImageFromFile.exe. Если вы переместили один из файлов в другую папку, метод *FromFile* не найдет JPEG-файл и сгенерирует исключение. Аргументы метода *DrawImage* определяют положение верхнего левого угла изображения относительно клиентской области. Если в качестве аргументов передана точка (0,0), изображение будет размещено в левом верхнем углу области.



Кроме того, можно щелкнуть клиентскую область и распечатать изображение (заметьте: данный класс создан на основе класса *PrintableForm*, обсуждавшегося в главе 5). Думаю, вам будет приятно увидеть, что при печати картинка не уменьшается до размеров почтовой марки, как часто происходит при печати растровых изображений.

Для сравнения загрузите этот же JPEG-файл в другое приложение (скажем, в программу Imaging, поставляемую с Windows 2000). Вероятно, вы заметите, что в ImageFromFile изображение чуть больше по размеру, чем в других приложениях. Скоро я объясню, почему это так,

Сейчас я хочу привлечь ваше внимание к двум недостаткам программы ImageFromFile, которые сознательно проигнорировал раньше, чтобы продемонстрировать простоту загрузки и вывода растрового изображения. На самом деле это не так просто, как показывает ImageFromFite! О первой проблеме я уже говорил: если метод *FromFile* не находит файл по указанному адресу, он генерирует исключение. Программа должна уметь его перехватывать. Вторая проблема — место вызова *FromFile*. В подобной программе вызов должен выполняться только раз, лучше всего — в ее конструкторе. Программа может сохранить объект *Image* в поле и обратиться к нему из метода *OnPaint*. Вот усовершенствованная версия ImageFromFile,

BetterImageFromFile.cs

```
//-----------------------------------------------
// BetterImageFromFile.cs © 2001 Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
«sing System.Windows.Forms;

class BetterImageFromFile: PrintableForm
{
     Image image;

     public new static void Main()
     {
          Application.Run(new BetterImageFromFile());
     }
     public BetterImageFromFile()
     {
          Text = "Better Image From File";

          string strFileName = "..\\..\\..\\Apollo11FullColor.jpg";

          try
          {
               image = Image.FromFile(strFileName);
          }
          catch
          {
```

см. след. стр.

```
                MessageBox.Show("Cannot find file " + strFileName + "!",
                                Text, MessageBoxButtons.OK, MessageBoxIcon.Hand);
            }
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            if (image == null)
                return;

            grfx.DrawImage(image, 0, 0);
        }
}
```

Тем не менее для простоты в большинстве следующих программ этой главы существование файла не проверяется.

Показанный ниже статический метод *Image FromStream* полезен при наличии доступа к открытому файлу, а также при получении потока от источника, отличного от файловой системы. Например, метод *FromStream* позволяет загрузить изображение из Интернета. Следующая программа обращается к Web-узлу NASA и загружает файл, который я обрезал и внес еще некоторые изменения для создания демонстрируемого в этой главе файла Apollo11.

ImageFromWeb.cs

```
//-------------------------------------------
// ImageFromWeb.cs © 2001 Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.IO;
using System.Net;
using System.Windows.Forms;

class ImageFromWeb: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new ImageFromWeb());
    }
    public ImageFromWeb()
    {
        Text = "Image From Web";

        string strUrl =
            "http://images.jsc.nasa.gov/images/pao/AS11/10075267.jpg";
        WebRequest  webreq = WebRequest.Create(strUrl);
        WebResponse webres = webreq.GetResponse();
```

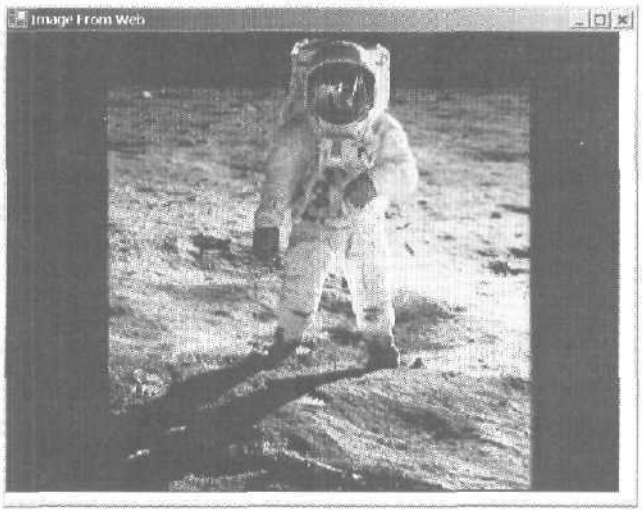
ст. след. стр.

```
            Stream      stream = webres.GetResponseStream();
            image = Image.FromStream(stream);
            stream.Close();
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            grfx.DrawImage(image, 0, 0);
        }
}
```

Операторы, использующие классы *WebRequest* и *WebResponse*, демонстрируют стандартный подход к загрузке файлов из Web. В этой программе метод *GetResponseStream* класса *WebResponse* получает доступный для чтения поток JPEG-файла. На данном этапе поток можно просто передать методу *ImageFromStream*:

```
image = Image.FromStream(stream);
```

В программе ImageFromWeb нет некоторых функций, стандартных для приложений, загружающих файлы из Интернета. Возможно, вам стоит реализовать индикатор выполнения (что легко сделать через объект типа *ProgressBar)* и перенести код чтения потока во второй подпроцесс,



Сведения об изображении

Класс *Image* включает несколько свойств, предоставляющих сведения об объекте, Первыми идут три свойства, указывающих размер изображения в пикселах,

Свойства *Image* (выборочно)

Тип	Свойство	Доступ
Size	*Size*	Чтение
int	*Width*	Чтение
int	*Height*	Чтение

Свойства *Width* и *Height* согласуются со свойством *Size*. Используйте то, что больше вам подходит.

Большинство современных форматов битовых карт также включает сведения о разрешении изображения в точках на дюйм или аналогичных метриках. Для некоторых изображений, в том числе и моего, эти сведения о разрешении могут не иметь смысла. Вы можете увеличивать или уменьшать размер изображения, а оно останется таким же. Однако для некоторых битовых карт (тех, в которых изображение должно соответствовать размеру отображаемого объекта) указание физического размера картинки может быть полезным.

Чем определяется разрешение? Обычно исходной программой, создавшей битовую карту. Например, если изображение было отсканировано с разрешением 300 dpi, обрабатывающее ПО обычно задает ему это же разрешение. При создании картинки в приложении для живописи ей обычно задается экранное разрешение (предполагается, что вы создаете изображение для текущего размера экрана).

Дополнительные свойства класса *Image* позволяют получить горизонтальное и вертикальное разрешение объекта *Image*, а также итоговый метрический размер,

Свойства *Image* (выборочно)

Тип	Свойство	Доступ	Описание
float	*HorizontalResolution*	Чтение	В точках на дюйм
float	*VerticalResolution*	Чтение	В точках на дюйм
SizeF	*PhysicalDimension*	Чтение	В сотых долях миллиметра

Если изображение не содержит сведений о разрешении, свойства *HorizontalResolution* и *VerticalResolution* возвращают разрешение дисплея. Значение свойства *PhysicalDimension* (в частности поточу, что в ранних версиях Windows Forms оно не работает) можно проигнорировать и вычислить метрический размер картинки самостоятельно. Так, следующие операторы вычисляют размер в дюймах.

```
float cxInches = image.Width / image.HorizontalResolution;
float cyInches = image.Height / image.VerticalResolution;
```

В предыдущих программах этой главы для вывода изображения использовалась такая версия метода *DrawImage*:

```
grfx.DrawImage(image, x, y)
```

Она масштабирует изображение на основе его метрического размера! Именно поэтому в предыдущих программах картинка имела несколько иной размер, чем в приложениях Windows. Методы визуализации изображений мы обсудим ниже,

Еще одно свойства класса [mage указывает формат пикселов. Этот формат определяет глубину цвета и то. как пикселы соответствуют цветам.

Свойства класса *Image* (выборочно)

Тип	Свойство	Доступ
PixelFormat	*PixelFormat*	Чтение

Данное свойство возвращает один из следующих элементов перечисления *PixelFormat*, определенных в пространстве имен *System.Drawing.Imaging*.

Перечисление *PixelFormat* (выборочно)

Член	Значение
Undefined или *DontCare*	0x00000000
Format16bppRgb555	0x00021005
Format16bppRgb565	0x00021006
Format24bppRgb	0x00021808
Format32bppRgb	0x00022009
Format1bppIndexed	0x00030101
Format4bppIndexed	0x00030402
Format8bppIndexed	0x00030803
Format16bppArgb1555	0x00061007
Format32bppPArgb	0xOOOE20()B
Format16bppGrayScale	0x00101004
Format48bppRgb	0x0010300C
Format64bppPArgb	Ox()01C400E
Format32bppArgb	0x0026200A
Format64bppArgb	0x0034400D

Число после слова *Format* указывает количество бит на пиксел: 1, 4, 8, 16, 32, 48 или 64. Форматы с разрядностью 1, 4 и 8 bpp являются индексированными, т. е. значения пикселов являются индексами палитры цветов- Форматы, имеющие в названии *Rgb*, хранят для каждого пиксела значение красного, зеленого и синего цветов. Форматы *Argb* также включают альфа-канал, определяющий прозрачность. Форматы *PArgb* содержат значения красного, зеленого и синего цветов с учетом альфа-значения. Может показаться, что численные значения элементов перечисления произвольны, однако, изучив их внимательнее, вы заметите определенный порядок. Возьмем, например, две крайних правых шестнадцатеричных цифры. Каждое значение уникально, начиная 0x00 (зловещее значение *Undefined* или *DontCare)* и кончая 0xOE,

Две следующих цифры справа указывают число бит на пиксел: 0x01, 0x04, 0x08, 0x10, 0x18, 0x20, 0x30 или 0x40. Другие цифры — это флаги. Ниже приведены значения перечисления *PixelFormat,* включая *Max,* которое указывает число форматов (вместе с *Undefined* — 15), а также значения, описывающие флаги.

Перечисление *PixelFormat* (выборочно)

Элемент	Значение	Описание
Max	0x0000000F	Число форматов,
Indexed	0x00010000	Биты пикселов являются индексами в палитре.
Gdi	0x00020000	Формат Windows GDI,
Alpha	0x00040000	Содержит бит или байт прозрачности.
PAlpha	0x00080000	В значениях для цветов учтена прозрачность.
Extended	0x00100000	Использует больше 1 байта для основного цвета или градации серого.
Canonical	0x00200000	Стандартный формат,

Класс *Image* также имеет статические методы, позволяющие извлечь большую часть этих сведений, не углубляясь в значения разрядов.

Статические методы класса *Image* (выборочно)

```
int GetPixelFormatSize(PixelFormat pf)
bool IsAlphaPixelFormat(PixelFormat pf)
bool IsCanonicalPixelFormat(PixelFormat pf)
bool IsExtendedPixelFormat(PixelFormat pf)
```

Первый метод возвращает число бит на пиксел.

Если изображение индексированное (что можно определить, выполнив поразрядное логическое умножение *PixelFormat* и *FixelFormat.Indexed*) у него имеется палитра цветов. Получить палитру позволяет свойство *Palette*.

Свойства класса *Image* (выборочно)

Тип	Свойство	Доступ
ColorPalette	*Palette*	Чтение/запись

Довольно странное свойство. Как видно, оно доступно для чтения и записи. Однако сам класс *ColorPalette* (определенный в пространстве имен *System.Drawing.Image*) является *изолированным* (sealed). Это значит, что нельзя создать его подклассы и у него нет общедоступных конструкторов, т. е. нельзя создать его экземпляры. Таким образом, получить в .NET объект *ColorPalette* позволяет только данное свойство класса *Image,*

Сам класс *ColorPalette* обладает двумя неизменяемыми свойствами.

Свойства класса *ColorPalette* (выборочно)

Тип	Свойство	Доступ
Color[]	*Entries*	Чтение
int	*Flags*	Чтение

Свойство *Entries* возвращает массив цветов из цветовой палитры изображения,

Еще одно свойство класса *Image* указывает формат файла,

Свойства класса *Image* (выборочно)

Тип	Свойство	Доступ
ImageFormat	*RawFormat*	Чтение

Я уже упоминал класс *ImageFormat.* Он включает статические свойства для всех поддерживаемых форматов битовых карт.

Свойство *RawFormat* класса *Image* довольно трудно применять. Его можно использовать только со свойством экземпляра (т. е. только с нестатическим свойством) класса *ImageFormat,* возвращающим глобально уникальный идентификатор (globally unique identifier. GUID) объекта *ImageFormat,*

Свойство экземпляра *ImageFormat*

Тип	Свойство	Доступ
Guid	*Guid*	Чтение

Программа может только проверить, имеет ли конкретный объект *Image* определенный тип *ImageFormat*. Например, при наличии объекта *Image* с именем *image*, загруженного из JPEG-файла, выражение:

```
image.RawFormat.Guid == ImageFormat.Png.Guid
```

возвращает *false*, а:

```
image.RawFormat.Guid == ImageFormat.Jpeg.Guid
```

возвращает *true*. Выражение:

```
image.RawFormat.ToString()
```

возвращает строку:

```
[ImageFormat: b96b3cae-0728-11d3-9d7b-0000f81ef32e]
```

а:

```
ImageFormat.Jpeg.ToString()
```

— строку:

```
Jpeg
```

Вот и решайте, что именно вывести пользователю!

Подробнее о других свойствах и методах класса *Image* см. далее,

Представление изображений

Как показано в предыдущих программах, метод класса *Graphics*, выводящий изображение, называется *DrawImage*. Наличие 30 его версий обеспечивает большую гибкость при программировании. Есть и еще один метод вывода изображений, *DrawImageUnscaled*, но его функциональность не шире, чем у *DrawImage*.

Как и в случае с выводом текста, вывод изображения включает обработку объекта, уже имеющего определенный размер. Строка текста, связанная с определенным шрифтом, имеет размер, и растровое изображение также обладает размером, вернее, *двумя*: в пикселах и метрическим. Вывод в метрическом размере (осуществляемый простейшими версиями метода *DrawImage*) полезен для аппаратно-независимой обработки изображений. И все же, если единицы измерения при выводе — пикселы, для оценки пиксельного размера представленного таким образом изображения потребуется немного посчитать (подробнее об этом — чуть позже). Иногда, особенно при их интеграции с элементами управления, изображения требуется выводить в пиксельном размере. Метод *DrawImage* не использует пиксельный размер автоматически, однако его легко настроить для этого.

Обычно битовые карты устойчивы к разного рода преобразованиям, например вращению. В Windows Forms и GDI+ глобальное преобразование влияет на вывод изображений так же, как и на вывод текста.

Первый аргумент всех версий метода *DrawImage* — объект типа *Image*. Самый последний — точка начала координат. Она может быть представлена двумя значениями типа *integer* или *float*, значением типа *Point*, *PointF*, *Rectangle* или *RectangleF* и определяет глобальную координату левого верхнего угла картинки.

Следующие четыре метода *DrawImage* масштабируют изображение, используя его метрический размер,

Методы *DrawImage* класса *Graphics* (выборочно)

```
void DrawImage(Image image, int x, int y)
void DrawImage(Image image, float x, float y)
void DrawImage(Image image, Point pt)
void DrawImage(Image image, PointF ptf)
```

Размер изображения не зависит от преобразований страницы, но зависит от глобального преобразования. Эти четыре метода *DrawImage* используются аналогично вызову метода *DrawString*, которому передан объект *Font*, созданный с применением метрического размера. Итоговое изображение имеет одинаковый метрический размер как на дисплее, так и на принтере.

Например, размер изображений Apollo 11 — 220x240 пикселов, а разрешение — 72 dpi. Таким образом, при выводе с помощью вышеупомянутых версий метода DrawImage их размер составит приблизительно 3x3 $^{1}/_{3}$ дюйма.

JPEG-файл с Web-узла NASA, загружаемый программой ImageFromWeb, имеет размер 640X480 пикселов, но не содержит информации о разрешении. В этом случае предполагается, что разрешение изображения равно экранному (обычно 96 или 120 dpi).

Иногда требуется предварительно оценить размер выводимого изображения, например, чтобы разместить его посередине прямоугольника. Поскольку четыре версии метода *DrawImage* выводят его в метрическом размере, для его центровки потребуется выполнить дополнительные действия,

Centerimage.cs

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

class CenterImage: PrintableForm
{
    Image image;

    public new static void Main()
    {
        Application.Run(new CenterImage());
    }
    public CenterImage()
    {
        Text = "Center Image";

        image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
```

см. след. стр.

```
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int ey)
    {
        grfx.PageUnit = GraphicsUnit.Pixel;
        grfx.PageScale = 1;

        RectangleF rectf = grfx.VisibleClipBounds;

        float cxImage = grfx.DpiX * image.Width /
                                    image.HorizontalResolution;
        float cyImage = grfx.DpiY * image.Height /
                                    image.VerticalResolution;

        grfx.DrawImage(image, (rectf.Width  - cxImage) / 2,
                              (rectf.Height - cyImage) / 2);
    }
}
```

Значения параметров *cxImage* и *cyImage* измеряются в пикселах. Разделив ширину и высоту изображения в пикселах на горизонтальное и вертикальное разрешение, вы получите размеры картинки в дюймах. Умножив полученный результат на значения свойств *DpiX* и *DpiY*, вы получите размер выведенного изображения в пикселах устройства.

Если изображение выводится только на дисплей, можно вычесть значения *cxImage* и *cyImage* из ширины и высоты клиентской области и разделить результат на 2. Однако такой расчет не подходит для принтера. Поэтому в начале метода *DoPage*, я перехожу к координатам в пикселах и методом *VisibleClipBounds* получаю размер устройства вывода в пикселах. Затем из ширины и высоты устройства вычитаются значения *cxImage* и *cyImage,* и результат делится на 2.

Вписывание изображения в прямоугольник

Следующие четыре метода *DrawImage* указывают прямоугольник, в который нужно поместить изображение. Размеры прямоугольника даются в глобальных координатах.

Методы *DrawImage* класса *Graphics* (выборочно)

```
void DrawImage(Image image, int x, int y, int cx, int cy)
void DrawImage(Image image, float x, float y, float cx, float cy)
void DrawImage(Image image, Rectangle rect)
void DrawImage(Image image, RectangleF rectf)
```

Эти методы подгоняют изображение под размеры прямоугольника, растягивая или сжимая его. Одно из распространенных применений данных методов – вывод изображения в пиксельном, а не метрическом размере. Если единицы измерения страницы — пикселы, просто вызовите;

```
grfx.DrawImage(image, x, y, image.Width, image.Height);
```

Приведенная ниже программа выводит изображение в пиксельном размере с выравниванием по центру клиентской области (или страницы принтера).

CenterPixelSizeImage.cs

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

class CenterPixelSizeImage: PrintableForm
{
     Image image;

     public new static void Main()
     {
          Application.Run(new CenterPixelSizeImage());
     }
     public CenterPixelSizeImage()
     {
          Text = "Center Pixel-Size Image";

          image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.DrawImage(image, (cx - image.Width)  / 2, ,
                                (cy - image.Height) / 2,
                                image.Width, image.Height);
     }
}
```

Разрешение вашего монитора скорее всего превышает 72 dpi, и поэтому картинка будет иметь меньший размер, чем при выводе методом *DrawImage*.



Если вывести битовую карту размером 220x240 пикселов с помощью этой версии *DrawImage* на принтер с разрешением 100 dpi, размер изображения составит примерно 2,2X2,4 дюйма. Если в методе *DoPage* сделать единицей измерения страницы пиксел, печатное изображение будет гораздо меньше — скорее всего вы столкнетесь с эффектом почтовой марки, распространенным в простых средах графического программирования.

Следующая программа загружает изображение и выводит его на всю клиентскую область (или печатную область страницы принтера).

ImageScaleToRectangle.cs

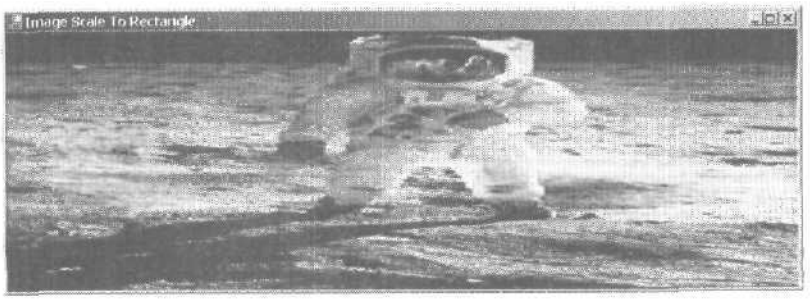
```
//-----------------------------------------------
// ImageScaleToRectangle.cs © 2001 Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ImageScaleToRectangle: PrintableForm
{
     Image image;

     public new static void Main()
     {
          Application.Run(new ImageScaleToRectangle());
     }
     public ImageScaleToRectangle()
     {
          Text = "Image Scale To Rectangle";

          image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.DrawImage(image, 0, 0, cx, cy);
     }
}
```

При значительном изменении ширины или высоты клиентской области изображение будет соответствующим образом искажаться.



Если вам нужно подогнать картинку под размер прямоугольника, такой эффект — явно не то, что требовалось. Вы, вероятно, хотите масштабировать изображение *изотропно*, т. е. одинаково в обоих направлениях. Вот программа, масштабирующая картинку более правильно,

ImageScaleIsotropic.cs

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

class ImageScaleIsotropic: PrintableForm
{
     Image image;

     public new static void Main()
     {
          Application.flun(newImageScaleIsotropic());
     }
     public ImageScaleIsotropic()
     {
          Text = "Image Scale Isotropic";

          image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
     }
     protected override void DoPage(Graphics grfx, Color clr, iRt cx, int cy)
     {
          ScaleImageIsotropically(grfx, image, new Rectangle(0, 0, cx, cy)); :
     }
     void ScaleImageIsotropically(Graphics grfx, Image image, Rectangle rect)
     {
          SizeF sizef = new SizeF(image.Width / image.HorizontalResolution,
                                  image.Height / image.VerticalResolution);

          float fScale = Math.Min(rect.Width  / sizef.Width,
                                  rect.Height / sizef.Height);

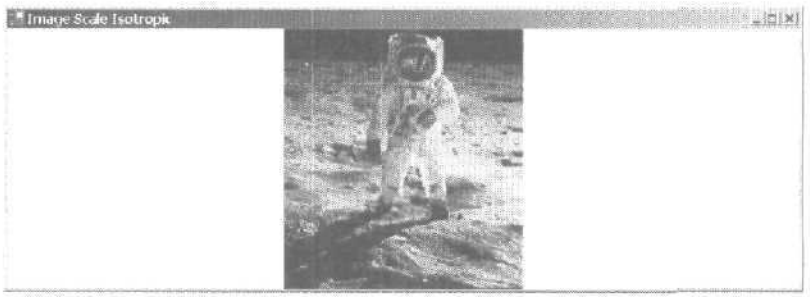
          sizef.Width  *= fScale;
          sizef.Height *= fScale;

          grfx.DrawImage(image, rect.X + (rect.Width  - sizef.Width ) / 2,
                                rect.Y + (rect.Height - sizef.Height) / 2,
                                sizef.Width, sizef.Height);
     }
}
```

Метод *ScaleImageIsotropically* работает практически всегда, кроме случаев, когда горизонтальное и вертикальное разрешения устройства различны (это ветре-

чается у некоторых принтеров) и когда параметр *PageUnit* имеет значение *GraphicsUnit.Pixel* (в принтерах по умолчанию используются *другие* единицы измерения),

Для начала метод *ScaleImageIsotropically* рассчитывает структуру *SizeF,* определяющую размер объекта *Image* в дюймах (если горизонтальное и вертикальное разрешения изображения одинаковы, данный этап не нужен). Затем вычисляется некоторый коэффициент, равный меньшему из двух значений: частному от деления высоты прямоугольника на высоту изображения и ширины прямоугольника на ширину изображения. Этот коэффициент *fScale* используется для изотропного масштабирования изображения в прямоугольнике. Затем метод определяет начало координат прямоугольника и передает полученную информацию методу *DrawImage.* Вот что получается:



Версии *DrawImage,* ориентированные на прямоугольник, могут не только масштабировать изображение, но и кое-что другое. Если указать отрицательную ширину, картинка будет повернута по вертикальной оси, и вы получите ее зеркальную копию. При отрицательной высоте метод поворачивает ее по горизонтальной оси и отображает вверх ногами. В любом случае верхний левый утол исходного, неперевернутого изображения позиционируется согласно координатам *Point* и *PointF* прямоугольника, указанными в *DrawImage,*

Следующая программа выводит четыре изображения, и некоторые из них имеют отрицательную ширину или высоту. Во всех четырех случаях второй и третий аргументы метода *DrawImage* указывают центр клиентской области.

ImageReflection.cs

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

class ImageReflection: PrintableForm
{
    Image image;

    public new static void Main()
    {
```

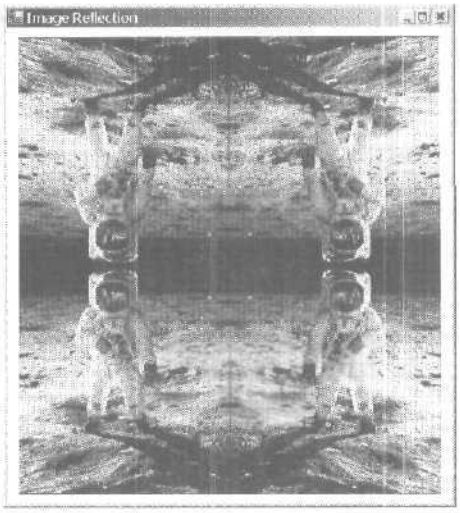
см. след. стр.

```
        Application.Run(new ImageReflection());
    }
    public ImageReflection()
    {
        Text = "Image Reflection";

        image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        int cxImage = image.Width;
        int cyImage = image.Height;

        grfx.DrawImage(image, cx / 2, cy / 2,  cxImage,  cyImage);
        grfx.DrawImage(image, cx / 2, cy / 2, -cxImage,  cyImage);
        grfx.DrawImage(image, cx / 2, cy / 2,  cxImage, -cyImage);
        grfx.DrawImage(image, cx / 2, cy / 2, -cxImage, -cyImage);
    }
}
```

Вот эти четыре изображения.



Заметьте: при масштабировании программа использует пиксельный размер изображения,

Вращение и наклон

Следующие два метода позволяют еще сильнее деформировать картинку. Они сдвигают, масштабируют, перекаптивают или вращают ее, подгоняя под параллелограмм,

```
new Rectangle(95, 0, 50, 55)
```

Рассмотрим программу, которая выводит именно эту часть изображения.

PartialImage.cs

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

class PartialImage: PrintableForm
{
     Image image;

     public new static void Main()
     {
          Application.Run(new PartialImage());
     }
     public PartialImageO
     {
          Text = "Partial Image";

          image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg"); '
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Rectangle rect = new Rectangle(95, 5, 50, 55);

          grfx.DrawImage(image, 0, 0, rect, GraphicsUnit.Pixel);
     }
}
```

Каков будет размер итогового изображения? Поскольку разрешение файла - *72* dpi, он составит 50/72 X 55/72 дюйма,



Кроме того, имеется четыре метода, позволяющих указать исходный и конечный прямоугольники.

Методы *DrawImage* класса *Graphics* (выборочно)

```
void DrawImage(Image image, Rectangle rectDst,
               int xSrc, int ySrc, int cxSrc, int cySrc,
               GraphicsUnit gu, *)
void DrawImage(Image image, Rectangle rectDst,
               Rectangle rectSrc,
               GraphicsUnit gu)
void DrawImage(Image image, Rectangle rectDst,
               float x, float y, float cx, float cy,
               GraphicsUnit gu, *)
void DrawImage(Image image, RectangleF rectfDst,
               RectangleF rectfSrc,
               GraphicsUnit gu)
```

Здесь исходный прямоугольник указывается в пикселах, а конечный— в глобальных координатах. Аргумент *GraphicsUnit* должен иметь значение *GraphicsUnit.Pixel*.

Заметьте: в конце списка аргументов двух методов я поставил звездочку. Она означает, что эти методы могут также принимать три дополнительных аргумента: объект *ImageAttribute*, функцию обратного вызова для отмены вывода изображения и данные, передаваемые этой функции. Таким образом, существует три дополнительных версии каждого из методов со звездочкой (к сожалению, они не обсуждаются в этой книге),

Следующая программа выводит шлем Олдрина (исходный прямоугольник) на всю клиентскую область или страницу принтера (конечный прямоугольник).

PartialImageStretch.cs

```
//-----------------------------------------
// PartialImageStretch.cs © Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class PartialImageStretch: PrintableForm
{
     Image image;

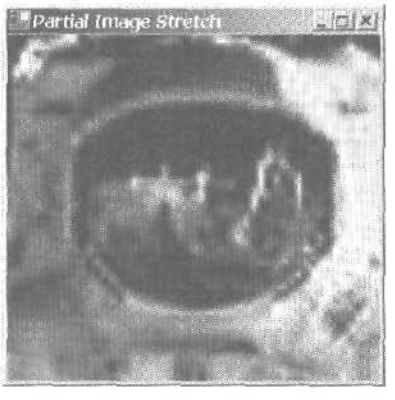
     public new static void Main()
     {
          Application.Run(new PartialImageStretch());
     }
     public PartialImageStretch()
     {
          Text = "Partial Image Stretch";
```

см. след. стр.

```
            image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            Rectangle rectSrc = new fcectangle(95, 5, 50, 55);
            Rectangle rectDst = new Rectangle( 0, 0, ex, cy);

            grfx.DrawImage(image, rectDst, rectSrc, GraphicsUnit.Pixel);
        }
}
```

Вот итоговое изображение, растянутое до размеров клиентской области.



Если соотношение размеров конечного прямоугольника отличается от исходного, картинка искажается. Однако для вывода частичных изображений можно легко использовать обсуждавшийся ранее метод *ScaleImageIsotropically*.Чтобы вывести картинку в пискельном размере, задайте ширину и высоту конечного прямоугольника равными размерам исходного прямоугольника.

При увеличении подобным способом небольших изображений возможна ступенчатость. И все же GDI+ производит интерполяцию и пытается сгладить изображения,

Завершить описание набора *DrawImage* можно методами, позволяющими выводить частичное изображение по трем точкам. Звездочка означает, что существует три дополнительных версии каждого метода,

Методы *DrawImage* класса *Graphics* (выборочно)

```
void DrawImage(Image image, Point[] aptDst,
               Rectangle rectSrc, GraphicsUnit gu, *)
void DrawImage(Image image, PointF[] aptfDst
               RectangleF rectfSrc, GraphicsUnit gu, *)
```

Исходный прямоугольник указанв пикселах, конечные точки даны в глобальных координатах, а аргумент *GraphicsUnit* должен быть *GraphicsUnitPixel*.Вот пример программы.

PartialImageRotate.cs

```
//-----------------------------------------
// PartialImageRotate.cs © 2001 Charles Petzold
//-----------------------------------------
«sing System;
«sing System.Drawing;
using System.Windows.Forms;

class PartialImageRotate: PrintableForm
{
     image image;

     public new static void Main()
     {
          Application.Run(new PartialImageRotate());
     }
     public PartialImageRotate()
     {
          Text = "Partial Image Rotate";

          image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
     }
     protected override void DoPage(Graphics grfx, Color clr, int ex, int cy)
     {
          Point[] aptDst * { new Point(0, cy / 2),
                             new Point(cx / 2, 0),
                             new Point(cx / 2, cy) };

          Rectangle rectSrc = new Rectangle(95, 5, 50, 55);

          grfx.DrawImage(image, aptDst, rectSrc, GraphicsUnit.Pixel);
     }
}
```

Вот теперь, если вы по-прежнему внимательно следите за повествованием, вы могли насчитать 30 методов *DrawImage*.

Рисование на изображении

Пока что мы выводили растровое изображение на дисплей и принтер. Но на изображении можно и рисовать. Если подумать, вывод изображения на дисплей в Windows — это фактически рисование на большой битовой карте, хранящейся в памяти видеоадаптера. У многих принтеров вывод также основан на содержимом памяти, организованном в виде битовой карты. Таким образом, имеет смысл реализовать в Windows-программе рисование на *любых* битовых картах при помощи тех же функций вывода графики, что используются для дисплея и принтера.

Для рисования на изображении нужно получить ссылающийся на него объект *Graphics*. Это позволяет сделать статический метод класса *Graphics*,

Статические методы *Graphics* (выборочно)

```
Graphics Graphics.FromImage(Image image)
```

Например, следующий оператор получает объект *Graphics* с именем *grfxImage,* основанный на объекте *Image* с именем *image:*

```
Graphics grfxImage = Graphics.FromImage(image);
```

Закончив работу с таким объектом *Graphics,* вызовите метод *Dispose* и уничтожьте его.

Метод *Graphics.FromImage* работает не со всеми форматами. Если значение свойства *PixelFormat* изображения — *Format1bppIndexed, Format4bppIndexed, Format8bppIndexed, Format16bppGrayScale* или *Format16Argb1555,* он генерирует исключение. Это ограничение может пригодиться. Например, вы получили объект *Graphics,* ссылающийся на индексированное изображение, и пытаетесь нарисовать что-либо с использованием цвета, отсутствующего в палитре изображения. Аналогичная проблема возникнет, если вы получите объект *Graphics* для картинки в градациях серого или картинки с 1 битом прозрачности.

Получить объект Graphics для изображения, загруженного из старого метафайла Windows (WMF), или расширенного метафайла Windows (EMF) нельзя. Кое-чтс о рисовании на метафайлах я расскажу в главе 23.

Объект *Graphics,* полученный методом *Graphics.FromImage* можно использовать так же, как и объект *Graphics* для дисплея или принтера. Просмотрев значения свойств *DpiX* и *DpiY,* вы обнаружите, что они равны значениям свойств *HorizontalResolution* и *VerticalResolution* объекта *Image.* По умолчанию при преобразовании страницы параметр *PageUnit* имеет значение *GraphicsUnit.Display,* а параметр *PageScale* — 1, что для изображений аналогично значению параметра *PageUnit,* равному *GraphicsUnit.Pixels.* Значение свойства *VisibleClipBounds* данного объекта *Graphics* по умолчанию равно высоте и ширине изображения в пикселах.

Для объекта *Graphics* можно задать другие параметры преобразования страницы. При изменении значений свойств *PageUnit* и *PageScale* свойство *VisibleClipBounds* будет указывать размеры изображения в единицах измерения страницы. Так, если переопределить значения свойств *PageUnit* и *PageScale.*

```
grfxImage.PageUnit = GraphicsUnit.Inch;
grfxImage.PageScale = 1;
```

свойство *VisibleClipBounds* будет указывать размер в дюймах. Этот размер также можно рассчитать, разделив пиксельный размер изображения на значения свойств *HorizontalResolution* и *VerticalResolution.*

Чуть позже мы будем выводить текст на поверхности битовой карты. Возникает вопрос: какой шрифт использовать? Удовлетворяет ли нашим требованиям значение свойства *Font* формы по умолчанию?

Значение свойства *Font* формы по умолчанию — это шрифт размером 8 пт. Если вывести текст на битовой карте этим шрифтом и затем отобразить изображение в метрическом (а не пиксельном) размере, текст будет такого же размера, как при выводе в самой клиентской области. Но если разрешение картинки меньше разрешения дисплея, текст будет выглядеть иначе. Контуры текста на изображении будут более грубыми, чем контуры текста, выводимого в окне,

Рассмотрим пример. Разрешение используемых в этой главе битовых карт — 72 dpi, т. е. один пиксел на пункт. Это значит, что высота шрифта размером 8 пт при таком разрешении составит около 8 пикселов. Если применить свойство *Font* формы по умолчанию и вызвать:

```
Font.GetHeight(72)
```

или воспользоваться объектом *Graphics,* созданным на основе одного из прилагающихся к главе растровых изображений, и вызвать:

```
Font.GetHeight(grfxImage)
```

вы получите межстрочный интервал в 8,83 пиксела, а значит, высота символа составляет около 8 пикселов,

8 пикселов явно мало для вывода символов с нормальными очертаниями. И все же при использовании свойства *Font* формы по умолчанию на изображении с разрешением 72 dpi метод *DrawString* не сможет выводить текст высотой более 8 пикселов. Если разрешение дисплея превышает 72 dpi и изображение выводится в пиксельном размере, текст на нем будет очень мелким. При выводе картинки в метрическом размере текст будет крупнее. На дисплее с разрешением 96 dpi изображение (а значит, и текст на нем) увеличатся приблизительно на треть, а на дисплее с разрешением 120 dpi — на две трети. Таким образом, эти 8 пикселов растягиваются до обычного размера шрифта формы по умолчанию.

Рассмотрим программу.

DrawOnImage.cs

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

class DrawOnImage: PrintableForm
{
     Image  image;
     string str = "Apollo11";

     public new static void Main()
     {
          Application.Run(new DrawOnImage());
     }
     public DrawOnImage()
     {
          Text = "Draw on Image";
          image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");

          Graphics grfxImage = Graphics.FromImage(image);

          grfxImage.PageUnit = GraphicsUnit.Inch;
```

см. след. стр.

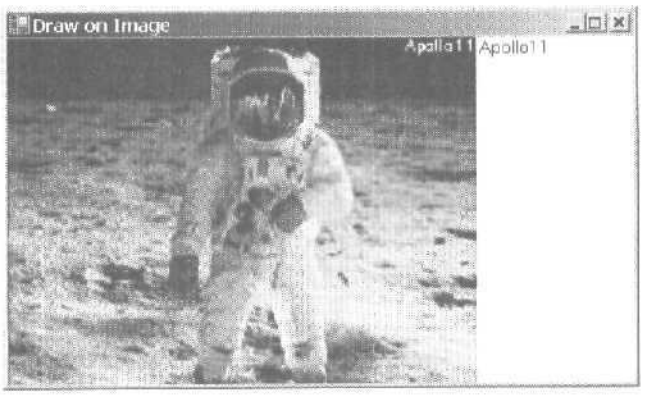
```
            grfxImage.PageScale = 1;

            SizeF sizef = grfxImage.MeasureString(str, Font);

            grfxImage.DrawString(str, Font, Brushes.White,
                        grfxImage.VisibleClipBounds.Width - sizef.Width, 0);

            grfxImage.Dispose();
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
        {
            grfx.PageUnit = GraphicsUnit.Pixel;
            grfx.DrawImage(image, 0, 9);
            grfx.DrawString(str, Font, new SolidBrush(clr),
                      grfx.DpiX * image.Width / image.HorizontalResolution, 0);
        }
    }
```

Ради интереса я изменил параметры преобразования страницы для объекта *Graphics,* связанного с изображением, и теперь единицей измерения при выводе является дюйм. Я определяю размеры строки методом *MeasureString* и затем задаю координаты в методе *DrawString* так, что строка выводится в верхнем правом углу изображения. Метод *DoPage* вызывает для вывода картинки метод *DrawImage* и затем, для сравнения, также вызывает метод *DrawString* для вывода той же строки текста справа от изображения (изменение единиц измерения страницы на дюймы позволяет правильно позиционировать текст при выводе на принтер). Вы видите, что текст на изображении более грубый и несколько искажен:



Можно ли устранить искажение? Оно вызвано применением шрифта, имеющего фиксированную высоту в 9 пикселов. Решить проблему позволяет шрифт с большим размером в пикселах. Для этого потребуется использовать изображе! ия с разрешением больше экранного или увеличить размер шрифта хотя бы до 12 пикселов. Во втором случае текст будет крупнее, но хоть читаемым.

Если картинка выводится в пиксельном размере, следует использовать другой подход. Чтобы размер текста па изображении соответствовал нормальному раз-

меру текста в клиентской области, масштабируйте шрифт, используя коэффициент отношения экранного разрешения к разрешению изображения. Вот пример.

DrawOnPixelSizeImage.cs

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

class DrawOnPixelSizeImage: PrintableForm
{
     Image image;
     string str = "Apollo11";

     public new static void Main()
     {
          Application.Run(new DrawOnPixelSizeImage());
     }
     public DrawOnPixelSizeImage()
     {
          Text = "Draw on Pixel-Size Image";
          image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");

          Graphics grfxImage = Graphics.FromImage(image);
          Graphics grfxScreen = CreateGraphics();

          Font font = new Font(Font.FontFamily,
                    grfxScreen.DpiY / grfxImage.DpiY * Font.SizeInPoints);

          SizeF sizef = grfxImage.MeasureString(str, font);

          grfxImage.DrawString(str, font, Brushes.White,
                               image.Width - sizef.Width, 0);
          grfxImage.Dispose();
          grfxScreen.Dispose();
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.DrawImage(image, 0, 0, image.Width, image.Height);
          grfx.DrawString(str, Font, new SolidBrush(clr), image.Width, 0);
     }
}
```

Конструктор создает новый объект *Font* (с именем *font),* идентичный шрифту по умолчанию, но с размерами в пунктах, масштабированный с применением коэффициента отношения экранного разрешения к разрешению изображения. Этот шрифт используется в вызовах методов *MeasureString* и *DrawString.* Так как я

не задавал нестандартных параметров преобразования страницы, для позиционирования строки в вызове метода *DrawString* я могу использовать ширину картинки в пикселах. Выводимый на изображении шрифт визуально имеет такой же размер, что и выводимый в клиентской области.



Теперь два шрифта выглядят одинаково, хотя их размер в пунктах различен. Они кажутся идентичными по размеру, поскольку выводятся на двух разных поверхностях (картинка и клиентская область) с разными разрешениями. Так как в случае с изображением я задавал размер шрифта, основываясь на разрешении изображения, при печати на принтере шрифты будут отличаться друг от друга па столько, на сколько виртуальное разрешение принтера в 100 dpi отличается от экранного.

Я не упомянул о том, что в вызове *DrawString* для рисования на изображении применяется метод *Brushes.White*. Это то, что нужно (конечно, с учетом черного фона), однако для рисования вы должны заранее знать, что представляет собой изображение!

Можно также рисовать на специально создаваемых для этого пустых битовых картах. Подробнее об этом — чуть позже.

Подробнее о классе *Image*

Класс *Image* включает несколько дополнительных методов, предоставляющих ограниченные возможности сохранения и манипуляции изображениями.

Методы *Save* класса *Image* (выборочно)

```
void Save(string strFilename, ImageFormat if)
void Save(Stream stream, ImageFormat if)
```

Метод *Save* не работает с объектами *Image*, загруженными из метафайла или хранящимися в битовой карте в памяти. Кроме того, нельзя сохранить изображение в формате метафайла или битовой карты в памяти,

Следующие два метода предоставляют возможности изменения размера, а также вращения и переворота изображения соответственно.

Методы *Image* (выборочно)

```
Image GetThumbnailImage(int cx, int cy,
                        Image.GetThumbnailImageAbort gtia,
                        IntPtr pData
void RotateFlip(RotateFliptype rft);
```

Метод *GetThumbnailImage* создает пиктограммы — уменьшенные изображения, позволяющие приложениям представлять их содержимое пользователю, экономя время и место. Тем не менее метод *GetTbumbnaillmage* фактически является универсальной функцией изменения размера изображения. Картинку можно как увеличить, так и уменьшить. Последние два аргумента указывают функцию обратного вызова, но им можно задать значения *null* и 0, что не скажется на работе метода. Следующая программа создает пиктограмму, размещаемую в квадрате со стороной 64 пиксела.

Thumbnail.cs

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

class Thumbnail: PrintableForm
{
     const int iSquare = 64;
     Image imageThumbnail;

     public new static void Main()
     {
          Application.fi«n(new Thumbnail ());
     }
     public Thumbnail()
     {
          Text = "Thumbnail";

        . Image image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");

          int cxThumbnail, cyThumbnail;

          if (image.Width > image.Height)
          {
               cxThumbnail = ISquare;
               cyThumbnail = iSquare * image.Height / image.Width;
           }
          else
          {
               cyThumbnail = iSquare;
```

см. след, стр.

```
            cxThumbnail = iSquare - image.Width / image.Height;
        }
        imageThumbnail = image.GetThumbnailImage(cxThumbnail, cyThumbnail,
                                                 null, (IntPtr) 0);
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {
        for (int y = 0; y < cy; y += iSquare)
        for (int x = 0; x < cx; x += iSquare)
            grfx.DrawImage(imageThumbnail,
                           x + (iSquare - imageThumbnail.Width) / 2,
                           y + (iSquare - imageThumbnail.Height) / 2,
                           imageThumbnail.Width, imageThumbnail.Height);
    }
}
```

Методом *DoPage* программа заполняет всю клиентскую область (или страницу принтера) пиктограммами.



Метод *GetThumbnailImage* создает новое изображение, а *RotateFlip* — изменяет существующее. Он принимает единственный аргумент — член перечисления *RotateFlipType*.

Перечисление *RotateFlipType*

Член	Значение	Результат
RotateNoneFlipNone ***Rotate*** *180FlipXY*	*0*	Hello
Rotate90FlipNone *Rotate270FlipXY*	*1*	Hello
Rotate *180FlipNone* *RotateNoneFlipXY*	*2*	Hello
Rotate2 70FlipNone *Rotate90FlipXY*	*3*	Hello
RotateNoneFlipX *Rotate180FlipY*	*4*	Hello

Перечисление *RotateFlipType* *(продолжение)*

Член	Значение	Результат
Rotate90FlipX *Rotate270FlipY*	*5*	
Rotate 180FlipX *RotateNoneFlipY*	*6*	
Rotate270FlipX *Rotate90FlipY*	*7*	

Перечисление включает 16 элементов, но только 8 уникально влияют на изображение. Четыре из этих восьми элементов меняют местами значения свойств *Width* и *Height* объекта *Image.*

Если вам нужно повернуть или перевернуть изображение и одновременно сохранить исходную копию оригинального объекта *Image,* применяйте метод *Clone.*

```
Image imageCopy = (Image) image.Clone();
```

Класс *Bitmap*

Выше мы обсуждали работу с объектами типа *Image.* Как я уже говорил в начале этой главы, пространство имен *System.Drawing* также включает класс *Bitmap,* наследуемый от *Image.* Все свойства *Image* применимы и к *Bitmap.* Любые операции, допустимые в отношении *Image,* допустимы и в отношении *Bitmap.* Но и это еще не все: класс *Bitmap* позволяет работать прямо с битами битовой карты,

У класса *Image* нет конструкторов, а у *Bitmap* их 12. Следующие конструкторы загружают объект *Bitmap* из файла, потока или ресурса.

Конструкторы *Bitmap* (выборочно)

```
Bitmap(string strFilename)
Bitmap(string strFilename, bool bUseImageColorManagement)
Bitmap(Stream stream)
Bitmap(Stream stream, bool bUseImageColorManagement)
Bitmap(Type type, string strResource)
```

Первые четыре по сути дублируют статические методы *FromFile* и *FromStream* класса *Image.* Пятый конструктор загружает объект *Bitmap* из ресурсов, обычно встраиваемых в .exe-файлы приложений. Подробнее о ресурсах — далее в этой главе.

Далее приведен набор конструкторов, которые создают новые объекты *Bitmap* на основе существующих объектов *Image,*

Конструкторы *Bitmap* (выборочно)

```
Bitmap(Image image)
Bitmap(Image image, Size size)
Bitmap(Image image, int cx, int cy)
```

Хотя первый аргумент этих конструкторов определен как *Image,* он может быть и другим объектом *Bitmap.* Первый конструктор аналогичен методу *Clone* класса

Image — он создает идентичный исходному новый объект *Bitmap*. Второй и третий похожи на метод *GetThumbnailImage* — они изменяют размер изображения. Во всех случаях новая битовая карта наследует формат пикселов исходной карты, а разрешение новой карты устанавливается равным разрешению дисплея.

Последние четыре конструктора не имеют аналогов в классе *Image*. Они создают новые объекты *Bitmap* с пустыми изображениями.

Конструкторы *Bitmap* (выборочно)

```
Bitmap(int cx, int cy)
Bitmap(int cx, int cy, PixelFormat pf)
Bitmap(int cx, int cy, Graphics grfx)
Bitmap(int cx, int cy, int cxRowBytes, PixelFormat pf, IntPtr pBytes)
```

Первые три инициализируют пикселы значением 0, которое в различных форматах битовых карт имеет разный смысл. В битовых RGB-картах О соответствует черному цвету. В ARGB-картах О обозначает прозрачность. Четвертый также может принимать указатель на массив байтов, инициализирующий растровое изображение.

Первый конструктор создает объект *Bitmap* заданного размера с форматом пикселов *PixelFormat.Format32bppArgb*. Это 32 бита на пиксел с альфа-каналом, определяющим прозрачность. Горизонтальное и вертикальное разрешения задаются равными разрешению дисплея. Второй конструктор позволяет указать элемент перечисления *PixelFormat,* если вас не устраивает формат *PixelFormat.Format32bppArgb*.

Третий конструктор позволяет указать объект *Graphics*. Независимо от того, связан указанный объект с дисплеем или принтером, и независимо от того, является ли принтер цветным, этот конструктор всегда создает объект *Bitmap* с форматом пикселов *PixelFormat.Format32bppPArgb*. Заметьте: данный формат пиксела предполагает, что значение альфа-канала уже *учтено* (premultiplied). Самое важное следствие создания битовой карты на основе объекта *Graphics* — то, что свойствам *HorizontalResolution* и *VerticalResolution* объекта *Bitmap* задаются значения свойств *DpiX* и *DpiY* объекта *Graphics*. Для принтера это означает не 100, а 300, 600, 720 и больше dpi.

Например, разрешение вашего принтера — 600 dpi. Вам нужна битовая карта, созданная на основе разрешения принтера. И зачем вам это? Затем, что при печати битовой карты все на ней (включая текст) должно выглядеть настолько красивым и плавным, насколько позволяет разрешение принтера. Квадратная битовая карта со стороной 2 дюйма и разрешением 600 dpi будет иметь пиксельный размер 1 200x1 200 пикселов и займет 5 Мб памяти. Помните, что при выводе такой битовой карты на экран или принтер надо использовать ее метрический размер! (Не волнуйтесь — скоро мы рассмотрим это на примере.)

Если вы создаете битовые карты, разрешения которых не соответствуют ни экранному разрешению, ни разрешению принтера, можно изменить разрешение загруженной или созданной битовой карты с помощью метода *SetResolution* класса *Bitmap*.

Методы ***Bitmap*** **(выборочно)**

```
void SetResolution(float xDpi, float yDpi)
```

Как нарисовать что-либо на поверхности битовой карты? Есть три подхода.

- Создать объект *Graphics* для данной карты и рисовать на ней, как на обычном графическом устройстве. Об этом мы уже говорили в предыдущих разделах. Помните, что создать объект *Graphics* можно не для всех допустимых форматов пикселов!
- Воспользоваться методами *SetPixel* и *GetPixel* класса *Bitmap* и задать (или получить) цвет определенных пикселов битовой карты.
- Воспользоваться методами *LockBits* и *UnlockBits* класса *Bitmap* и обратиться прямо к данным битовой карты.

К сожалению, я не смогу продемонстрировать второй и третий подходы в этой книге.

Программа «Hello World» с битовой картой

Следующая программа создает битовую карту и рисует на ней вечную мантру программиста высотой в 72 пункта. Затем битовая карта выводится в клиентской области и (не обязательно) на странице принтера.

HelloWorldBitmap.cs

```
//-----------------------------------------
// HelloWorldBitmap.cs © 2061 Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class HelloWorldBitmap: PrintableForm
{
    const float fResolution = 300;
    Bitmap bitmap;

    public new static void Main()
    {
        Application.Run(new HelloWorldBitmap());
    }
    public HelloWorldBitmap()
    {
        Text = "Hello, World!";

        bitmap = new Bitmap(1, 1);
        bitmap.SetResolution(fResolution, fResolution);

        Graphics grfx = Graphics.FromImage(bitmap);
```

см. след. стр.

```
            Font      font = new Font("Times New Roman", 72);
            Size      size = grfx.MeasureString(Text, font).ToSize();

          • bitmap * new Bitmap(bitmap, size);
            bitmap.SetResolution(fResolution, fResolution);

            grfx = Graphics.FromImage(bitmap);
            grfx.Clear(Color.White);
            grfx.DrawString(Text, font, Brushes.Black, 0, 0);
            grfx.Dispose();
        }
        protected override void DoPage(Graphics grfx, Color clr, int cx, int ey)
        {
            grfx.DrawImage(bitmap, 0, 0);
        }
}
```

Битовая карта создается в конструкторе программы, и, как видите, код довольно запутан. Проблема такова: я хотел, чтобы битовая карта по размеру точно соответствовала выводимой строке, но не хотел использовать разрешение битовой карты, связанное с реальным устройством вывода. Проблема ясна? Чтобы получить размеры строки текста, нужен метод *MeasureString* объекта *Graphics*. Однако чтобы объект *Graphics* основывался на битовой карте с произвольным разрешением устройства, для его получения нужен объект *Bitmap* или *Image*!

В связи с этим конструктор создает две карты. Первая очень маленькая — квадрат со стороной 1 пиксел, но этого вполне хватает. Программа с помощью константы *fResolution* назначает этой битовой карте разрешение 300 dpi. Затем получает объект *Graphics*, создает шрифт Times New Roman высотой 72 пункта и вызывает *MeasureString*.

Для создания новой битовой карты используются размеры, возвращаемые методом *MeasureString,* Эта карта также должна иметь разрешение 300 dpi. После этого программа получает объект *Graphics* для новой карты, изменяет цвет фона на белый и выводит текст черным шрифтом.

Программа выводит битовую карту на дисплей и принтер методом *DrawImage*. Итоговое изображение выглядит как обычный текст высотой 72 пункта. Однако настоящее доказательство того, что программа делает что-то интересное, — вывод на принтер: символы шрифта отображаются такими же ровными и неискаженными, как и в случае с разрешением 300 dpi. Попробуйте уменьшить константу *fResolution* (скажем, до 30 dpi) и посмотрите, что получится на дисплее и принтере.

Изображение-тень

Иногда реализация метода *OnPaint* требует много процессорного времени и памяти. Скажем, клиентская область может содержать сложное изображение, строившееся долгое время. В этом случае отличным решением является создание *изображения-тени* (shadow bitmap) — карты, на которой программа также рисует при рисовании на клиентской области вне *OnPaint*. Затем *OnPaint* сводится к простому вызову *DrawImage*.

В главе 8 приведена программа Scribble, позволяющая рисовать мышью на клиентской области. Но тогда я мог показать вам единственное решение, сохраняющее изображение для обновления при выполнении метода *OnPaint*. Это было реализовано в программе ScribbleWithSave, сохранявшей все координаты в объектах *ArrayList*. Ничего некорректного в таком подходе нет. Фактически, чтобы позволить пользователю редактировать рисунок путем изменения отдельных линий, нужно сохранять все точки координат. ScribbleWithSave — это первый шаг к созданию программы для работы с графикой, сохраняющей рисунок в одном из форматов метафайлов,

Вот новая версия программы Scribble — ScribbleWithBitmap, которая хранит все изображение в большой битовой карте. Данная программа может стать первым шагом к созданию программы для живописи,

ScribbleWithBitmap.cs

```
//---------------------------------------------
// SeribbleWith8itraap.cs © 2001 Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class ScribbleWithBitmap: Form
{
     bool    bTracking;
     Point   ptLast;
     Bitmap  bitmap;
     Graphics grfxBm;

     public static void Main()
     {
          Application.Run(new ScribbleWithBitmap());
     }
     public ScribbleWithBitmap()
     {
          Text = "Scribble with Bitmap";
          BackColor - SystemColors.Window;
          ForeColor - SystemColors.WindowText;

               // Создаем битовую карту.

          Size size = SystemInformation.PrimaryMonitorMaximizedWindowSize;
          bitmap = new Bitmap(size.Width, size.Height);

               // Создаем объект Graphics на основе битовой карты.

          grfxBm - Graphics.FromImage(bitmap);
          grfxBm.Clear(BackColor);
```

см. след. стр.

```
    }
    protected override void OnMouseDown(MouseEventArgs mea)
    {
        if (mea.Button != MouseButtons.Left)
            return;

        ptLast = new Point(mea.X,mea.Y);
        bTracking * tree;
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        if (!bTracking)
            return;

        Point ptNew = new Point(mea.X, mea.Y);

        Pen pen = new Pen(ForeColor);
        Graphics grfx - CreateGraphics();
        grfx.DrawLine(pen, ptLast, ptNew);
        grfx.Dispose();

            // Рисуем на битовой карте.

        grfxBm.DrawLine(pen, ptLast, ptNew);

        ptLast = ptNew;
    }
    protected override void OnMouseUp(MouseEventArgs mea)
    {
        bTracking * false;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics grfx = pea.Graphics;

            // Выводим битовую карту.

        grfx.DrawImage(bitmap, 0, 0, bitmap.Width, bitmap.Height);
    }
}
```

Операторы, добавленные в базовую программу Scribble, выделены комментариями — их немного. В конструкторе я создаю битовую карту, размер которой определяется возвращаемым значением *SystemInformationPrimaryMonitorMaximizedWindowSize*. Метод *FromImage* класса *Graphics* получает объект *Graphics*, и затем вызовом метода *Clear* инициализируется битовая карта. При выполнении метода *OnMouseMove* метод *DrawLine* рисует на битовой карте и в клиентской области. При выполнении метода *OnPaint* осуществляется отображение битовой карты вызовом метода *DrawImage*.

Версия ScribbleWitbBitmap гораздо короче и приятнее версии ScribbleWithSave. Однако за эту простоту надо платить: в прямом смысле теряются координаты ломанных линий, Извлечь их снова из битовой карты не так-то легко.

Вот еще одно отличие: эффективность *OnPaint* в ScribbleWithSave зависит от степени сложности рисунка. Чем больше на нем кривых, тем больше времени требуется методу *OnPaint* на их перерисовку. Скорость работы *OnPaint* в ScribbleWithBitmap не зависит от сложности изображения.

Как я уже говорил, ScribbleWithBitmap создает в конструкторе битовую карту, имеющую размер развернутого окна. Это хорошо показывает, какой размер должен быть у изображения-тени. Однако если при работе с ScribbleWithBitmap пользователь изменит размер экрана, битовая карта может перестать соответствовать действительности. На такой случай установите обработчик события *DisplaySettingsChanged* класса *SystemEvents*, определенного в пространстве имен *Microsoft.Win32*. Но как же реагировать на само событие? Если размер дисплея увеличился, можно создать новую битовую карту соответствующего размера и скопировать в нее содержимое старой карты. А если размер дисплея уменьшился? Следует ли просто создать битовую карту меньшего размера и отсечь часть имеющегося изображения? Непростая задача!

Двоичные ресурсы

Выше мы обсудили много способов загрузки битовой карты из файла для случаев, когда вам требуется вывести конкретное изображение.

Но, как вы знаете, в случае с приложениями, предназначенными пользователям, хранение битовых карт в отдельных файлах — не всегда хорошее решение, Что, если исполнимый файл программы и битовая карта будут отделены друг от друга? Чрезмерно усердный пользователь может освобождать пространство на диске, удаляя «ненужные» файлы, и вдруг — бац! — нет больше битовой карты.

Поэтому зачастую полезно хранить небольшие двоичные файлы (в частности, битовые карты, значки и нестандартные курсоры) прямо в .exe-файле приложения. Таким образом, они не смогут потеряться. Файлы, хранимые в исполнимом файле, называются ресурсами.

В Visual C# .NET для создания двоичных ресурсов имеется интерактивный редактор изображений. Чтобы добавить в программу двоичный ресурс, выберите в меню Project команду Add New Resource. В диалоговом окне Add New Item выберите из расположенного слева списка Categories пункт Resources, а справа — Bitmap File, Cursor File или Icon File. Затем задайте имя файла.

В окне Properties растровых файлов можно указать их размеры и цветовой формат. Формат файлов курсора по умолчанию — 32x32 пиксела, 16 цветов. При рисовании можно также использовать цвета Screen и Reverse Screen, Screen является прозрачным. Он позволяет придать курсору непрямоугольную форму (такую форму имеет большинство курсоров). Reverse Screen обращает цвет фона под курсором и используется довольно редко. Вам также следует указать «горячую точку* курсора.

Цветовой формат файлов значков по умолчанию — 16 цветов плюс Screen и Reverse Screen. Screen часто применяется для придания значкам формы, отличной

от квадрата. Reverse Screen был очень популярен на заре эры Windows, однако сейчас используется гораздо реже.

Обычно формат создаваемого значка — 32x32 пиксела, 16 цветов. Однако значки зачастую отображаются в уменьшенном размере, как квадрат со стороной 16 пикселов. Windows отображает 32-пиксельный значок как 16-пиксельный, удаляя каждый второй столбец и строку пикселов. Если после удаления 75% содержимого ваш значок выглядит не совсем хорошо, создайте его 16-пиксельную версию, которая будет храниться в этом же файле значка. Для переключения между форматами значков в редакторе изображений служит команда New Image Type меню Image.

Теперь о самом важном правиле создания файлов битовых карт, курсоров или значков, которые станут ресурсами. Сосредоточьтесь.

Если в окне Solution Explorer среды Visual C#. NET выбрать растровый фа йл значка или курсора, являющийся частью проекта, становится доступно окно свойств данного файла. Задайте в нем свойству Build Action значение Embedded Resource. Это свойство заставляет Visual C# .NET встроить файл ресурса в .exe-файл проєкта. Для загрузки такого ресурса в программе служит конструктор *Bitmap*, *Cursor* или *Icon*.

Рассмотрим пример с ресурсом-значком. Название проекта — ProgramWithIcon — означает, что имя файла программы — ProgramWithIcon.cs и программа содержит класс *ProgramWithIcon*Для рифмы я решил назвать файл значка ProgramWithIcon.ico. Значок изображает картотечный шкаф. Фактически программа толь со загружает значок и задает значение свойства *Icon* формы,

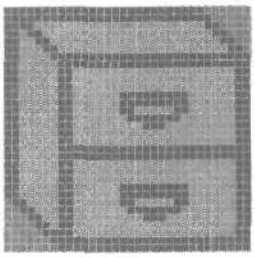
ProgramWithIcon.cs

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

class ProgramWithIcon: Form
{
    public static void Main()
    {
        Application.Run(new ProgramWithIcon());
    }
    ; public ProgramWithIcon()
    {
        Text = "Program with Icon";

        Icon * new Icon(typeof(ProgramWithIcon),
                        "РгодramWithIcon.ProgramWithIcon.ico");
    }
}
```

ProgramWithIcon.ico



При самостоятельной работе с этой программой в Visual C#. NET не забудьте задать свойству Build Action файла значка значение Embedded Resource,

Для загрузки значка используется конструктор класса *Icon.*

Конструкторы *Icon* (выборочно)

```
Icon(Type type, string strResource)
```

Другие конструкторы класса *Icon* позволяют загружать значки из файлов и потоков. Если файл значка или ресурс содержит несколько изображений, можно попробовать получить значок, указывая нужный размер.

Конструкторы *Icon* (выборочно)

```
Icon(Icon icon, Size size)
Icon(Icon icon, int cx, int cy)
```

Эти конструкторы не растягивают и не сжимают значки, а пытаются найти доступные значки нужного вам размера. Определить размер значка позволяют свойства класса Icon,

Свойства *Icon* (выборочно)

Тип	Свойство	Доступ
Size	*Size*	Чтение
int	*Width*	Чтение
int	*Height*	Чтение

Приведенные ниже методы класса *Graphics* позволяют вывести значок в клиентской области.

Методы *Graphics* (выборочно)

```
void DrawIcon(Icon icon, int x, int y)
void DrawIcon(Icon icon, Rectangle rect)
void DrawIconUnstretched(Icon icon, Rectangle rect)
```

Программа ProgramWithIcon просто задает свойству *Icon* формы значение, возвращенное конструктором *Icon,* и все. Обратите внимание на значок в верхнем левом углу формы.



Теперь рассмотрим используемый в программе конструктор *Icon:*

```
new Icon(typeof(ProgramWithIcon), "ProgramWithIcon.ProgramWithIcon.ico")
```

ProgramWithIcon настолько часто появляется в проекте и в этом операторе, что трудно сказать, что есть что. В связи с этим мы подробно проанализируем конструктор.

Первый аргумент конструктора — это класс *ProgramWithIcon.* В операторе *typeof* можно применить имя любого класса, определенного в программе. Кроме того, можно использовать имя любой определенной в программе структуры, перечисления, интерфейса или делегата.

В любой части кода класса *ProgramWithIcon* выражение:

```
typeof(ProgramWithIcon)
```

аналогично:

```
this.GetType()
```

или, если проще:

```
GetType()
```

Такая эквивалентность позволяет сократить конструктор до:

```
new Icon(GetType(), "ProgramWithIcon.ProgramWithIcon.ico")
```

с сохранением полной работоспособности программы,

Второй аргумент конструктора *Icon* — имя файла. Например, если имя файла значка MyIcon.ico, конструктор *Icon* будет выглядеть так:

```
new Icon(GetType(), "ProgramWithIcon.MyIcon.ico")
```

Первая часть имени в кавычках называется *пространством имен* (namespace), однако это пространство имен ресурса. Не путайте его с пространством имен .NET Framework. По умолчанию Visual C# .NET называет это пространство имен ресурса так же, как проект. Но ему можно задать и другое название, изменив значение свойства Default Namespace в диалоговом окне Property проекта. Заданное имя должно соответствовать первой части заключенного в кавычки имени в конструкторе *Icon.* Свойству Default Namespace можно даже задать пустое значение; при этом вторым аргументом конструктора *Icon* станет только имя файла:

```
new Icon(GetType(), "ProgramWithIcon.ico");
```

или MyIcon.ico, или любое другое.

При запуске компилятора C# из командной строки для каждого ресурса применяется ключ */res*. Например, при использовании ключа компилятора:

```
/res:ProgramWithIcon.ico
```

значок загружается так

```
new Icon(GetType(), "ProgramWithIcon.ico")
```

Кроме того, значку можно задать расширенное имя, за которым через запятую будет идти имя файла:

```
/res:ProgramWithIcon.ico,ProgramWithIcon.ProgramWithIcon.ico
```

Загрузить такой значок позволяет конструктор:

```
new Icon(GetType(), "ProgramWithIcon.ProgramWithIcon.ico")
```

При использовании пространства имен ресурса по умолчанию, которое среда Visual C# .NET назначает вашему проекту, возможна проблема. Например, вы создали проект ProgramWithIconPlus, в котором класс *ProgramWithIconPlus* наследует параметры класса *ProgramWithIcon*. В проекте вы также создали новый файл ProgramWithIconPlus.cs и добавили ссылку на существующий файл ProgramWithIcon.cs. Однако создавать новый значок для новой программы вы не стали, а сделали в проекте ProgramWithIconPlus ссылку на файл ProgramWithIcon.ico. Конструктор в классе *ProgramWithIcon* по-прежнему будет загружать значок. А если значок попытается загрузить программа ProgramWithIconPlus? Она аварийно завершит работу, сгенерировав исключение. Что же происходит? Оператор конструктора класса *ProgramWithIcon*, загружающий значок, выглядит так:

```
Icon = new Icon(typeof(ProgramWithIcon),
                "ProgramWithIcon.ProgramWithIcon.ico");
```

Однако пространство имен ресурса по умолчанию для программы ProgramWithIconPlus — это ProgramWithIconPlus. а не ProgramWithIcon. Каково простейшее решение проблемы? Изменить значение свойства Default Namespace проекта ProgramWithIconPlus на ProgramWithIcon. Или задать всем свойствам Default Namespace пустые значения и использовать в конструкторе только имя файла,

Курсор загружается так же, как и значок. Конструктор *Cursor* выглядит так:

Конструкторы *Cursor* (выборочно)

```
Cursor(Type type, string strResource)
```

О конструкторе класса *Bitmap* я уже упоминал.

Конструкторы *Bitmap* (выборочно)

```
Bitmap(Type type, string strResource)
```

Давайте разбираться с конструктором *Bitmap,*

Анимация

В Windows Forms и GDI+ не хватает нескольких важных для анимации функций. В главе 8 я говорил, что GDI+ не поддерживает XOR-рисование, позволяющее стереть то, что уже нарисовано повторным рисованием. Другая проблема в том, что Windows Forms не позволяет считывать пикселы с экрана. При анимации зачастую полезно считать блок пикселов с экрана как битовую карту, нарисовать что-либо на ней и вывести результат на экран.

И все же элементарная анимация в программах Windows Forms возможна. Один из способов — *покадровая анимация* (frame animation) — заключается в последовательном выводе одинаковых по размеру битовых карт, наподобие фильма. Следующая программа загружает четыре битовых карты как ресурсы и затем с помощью события Timer выводит мигающий глаз.

Wink.cs

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

class Wink: Form
{
     protected Image[] aimage  = new Image[4];
     protected int     iImage = 0, iIncr = 1;

     public static void Main()
     {
          Application.Run(new Wink());
     }
     public Wink()
     {
          Text = "Wink";
          ResizeRedraw = true;
          BackColor = Color.White;

          for (int i = 0; i < 4; i++)
               aimage[i] = new Bitmap(GetType(),
                                      "Wink.Eye" + (i + 1) + ".png");
          Timer timer = new Timer();
          timer.Interval = 100;
          timer.Tick += new EventHandler(TimerOnTick);
          timer.Enabled = true;
     }
     protected virtual void TimerOnTick(object obj, EventArgs ea)
     {
          Graphics grfx = CreateGraphics();
```

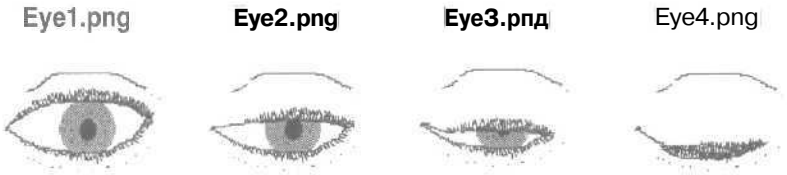
см. след. стр.

```
            grfx.DrawImage(aimage[iImage],
                           (ClientSize.Width  - aimage[iImage].Width)  / 2,
                           (ClientSize.Height - aimage[iImage].Height) / 2,
                           aimage[iImage].Width, aimage[iImage].Height);
            grfx.Dispose();

            iImage += iIncr;

            if (iImage == 0)
                iIncr = -1;

            else if (iImage == 0)
                iIner = 1;
        }
    }
```



Заметьте: при загрузке ресурсов конструктор использует имена Wink.Eye1.png, Wink.Eye2.png и т. д. *Wink* — это название пространства имен. Метод *TimerOnTick* с помощью *DrawImage* последовательно выводит каждое изображение по центру клиентской области. Вот программа в действии:



При анимации надо выводить изображения в пиксельном размере. Это позволяет предотвратить уменьшение и увеличение изображения, а также экономит время процессора.

Шутки ради я создал в следующей программе дочерний класс *Wink* и методом *RotateFlip* вывел рядом с левым глазом правый.

DualWink.cs

```
//-----------------------------------
// flual^lnk.Gs © 2001 Charles Petzold
//-- ---------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class DualWink: Wink
C
     Image[] aimageRev = new Image[4];

     public new static void Main()
     {
          Application.Run(new DualWink());
     )
     public DualWink()
     {
          Text = "Dual " + Text;

          for(int i - 0; i < 4; i++)
          {
               aimageRev[i] = (Image) aimage[i].Clone();
               aimageRev[i].RotateFlip(RotateFlipType.RotateNoneFlipX);
          }
     }
     protected override void TimerOnTick(object obj, EventArgs ea)
     {
          Graphics grfx * CreateGraphics();

          grfx.DrawImage(aimage[iImage],
                    ClientSize.Width / 2,
                     (ClientSize.Height - aimage[iImage].Height) / 2,
                     aimage[iImage].Width, aimage[iImage].Height);

          grfx.DrawImage(aimageRev[3 - iImage],
                    ClientSize.Width / 2 - aimageRev[3 - iImage].Width,
                     (ClientSize.Height - aimageRev[3 - iImage].Height) / 2,
                     aimageRev[3 - iImage].Width,
                     aimageRev[3 - iImage].Height);

          grfx.Dispose();

          iImage += iIncr;

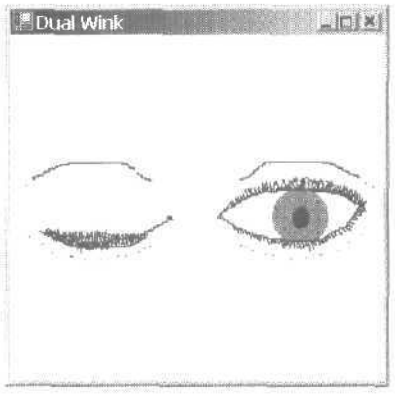
          if (iImage == 3)
               iIncr - -1;

          else if (iImage == 0)
```

см. след. стр.

```
                iIncr * 1;
        }
    }
```

В этом проекте нужны ссылки на четыре PNG-файла, связанных с программой Wink. И помните мое предупреждение о наследовании классов, загружающих ресурсы! В проекте DualWink мне пришлось изменить название пространства имен ресурса с DualWink на Wink.



А сейчас то, чего вы все ждали. — прыгающий мяч! Программа Bounce создает квадратную битовую карту, рисует на ней красный мяч и затем выводит итоговую карту в разных частях клиентской области, имитируя отскок мяча от стен.

Bounce.cs

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

class Bounce: Form .
i
    const int iTimerInterval = 25;    // S миллисекундах
    const int iBallSize - 16;         // Часть клиентской облает»
    const int iMoveSize * 4;          // Часть размера мяча

    Bitmap bitmap;
    int    xCenter, yCenter;
    int    cxRadius, cyRadius, cxMove, cyMove, cxTotal, cyTotal;

    public static void Main()

    *
        Application.Run(new Bounce());
    }
```

см. след. стр.

```
public Bounce()
{
    Text = "Bounce";
    ResizeRedraw = true;
    BackColor = Color.White;

    Timer timer = new Timer();
    timer.Interval = iTimerInterval;
    timer.Tick += new EventHandler(TimerOnTick);
    timer.Start();

    OnResize(EventArgs.Empty);
}
protected override void OnResize(EventArgs sa)
{
    Graphics grfx = CreateGraphics();
    grfx.Clear(BackColor);

    float fRadius = Math.Min(ClientSize.Width  / grfx.DpiX,
                             ClientSize.Height / grfx.DpiY)
                                 / iBallSize;

    cxRadius = (int) (fRadius * grfx.DpiX);
    cyRadius = (int) (fRadius * grfx.DpiY);

    grfx.Dispose();

    cxMove = Math.Max(1, cxRadius / iMoveSize);
    cyMove = Math.Max(1, cyRadius / iMoveSize);

    cxTotal = 2 * (cxRadius + cxMove);
    cyTotal = 2 * (cyRadius + cyMove);

    bitmap = пек Bitmap(cxTotal, cyTotal);

    grfx = Graphics.FromImage(bitmap);
    grfx.Clear(BackColor);

    DrawBall(grfx, new Rectangle(cxMove, cyMove,
                                 2 * cxRadias, 2 * cyRadius»;
    grfx.Dispose();

    xCenter = ClientSize.Width / 2;
    yCenter = ClientSize.Height / 2;
}
protected virtual void DrawBall(Graphics grfx, Rectangle reet)
{
    grfx.FillEllipse(Brushes.Red, rect);
}
```

см. след.стр.

```
    void TimerOnTick(object obj, EventArgs ea)
    {
        Graphics grfx = CreateGraphics();

        grfx.DrawImage(bitmap, xCenter - cxTotal / 2,
                               . yCenter - cyTotal / 2,
                               cxTotal, cyTotal);
        grfx.Dispose();

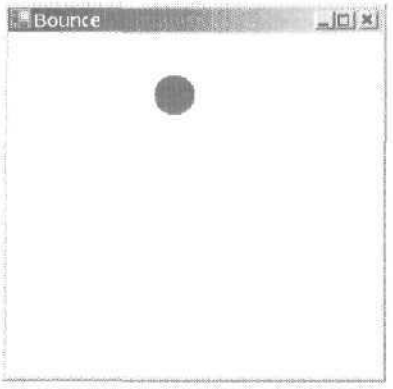
        xCenter += cxMove;
        yCenter += cyMove;

        if ((xCenter + cxRadius >= ClientSize.Width) ||
            (xCenter - cxRadius <= 0»
                cxMove " -cxMove;

        if ((yCenter + cyRadius >= ClientSize.Height) If
            (yCenter - cyRadius <= 0))
                cyMove = -cyMove;
    }
}
```

Конечно, главный вопрос — не то, как мяч выводится в клиентской области, а то, как стирается его предыдущее изображение и может ли программа реализовать это без мерцания. Решение данной проблемы в том, что размер битовой карты больше размера мяча, и границы вокруг мяча стирают предыдущее его изображение,

Программа Bounce заново строит битовую карту при каждом вызове метода *OnResize* формы. Радиус мяча равен 1/16 ширины или высоты клиентской области в зависимости от того, что из них меньше. Однако программа создает битовую карту большего размера, оставляя с каждой стороны по 1/4 радиуса мяча (изменить оба коэффициента очень легко). Сначала битовая карта закрашивается белым цветом, а затем на ней рисуется мяч (я поместил код рисования мяча в защищенный виртуальный метод, чтобы в одной из следующих глав нарисовать мяч, который будет выглядеть получше).



Размеры полей вокруг мяча хранятся в параметрах *cxMove* и *cyMove*. Совершенно не случайно, что значения этих параметров точно соответствуют *шагу* перемещения по битовой карте при вызове метода TimerOnTrick.

В общем случае такой простой подход к анимации может и не сработать. Сделайте фоном клиентской области что-либо, отличное от сплошного цвета, и весь вышеописанный способ распадется.

Список изображений

В главе 12 мы начнем работать с элементами управления, в частности, с кнопьами, метками и полосами прокрутки, а затем — с меню, списками, редактируемыми полями и т. д. Вы увидите, что на поверхности элемента управления в дополнение к тексту или вместо него можно размещать битовые карты. И высший пилотаж: панели инструментов под меню приложений обычно состоят из строки небольших битовых карт.

В пространстве имен *System.Windows.Forms* есть класс *ImageList,* упрощающий работу с наборами изображений. Список изображений — это гибкий массив объектов *Image* одинакового размера и цветового формата. Добавление изображений в объект *ImageList* (при помощи процесса, о котором я вкратце расскажу ниже) и доступ к ним аналогичны работе с массивом. При добавлении в объект *ImageList* изображениям не обязательно иметь одинаковый размер, и все же при извлечении из этого объекта они будут приведены к одному размеру.

Вот самые важные свойства класса *ImageList.*

Свойства *ImageList* (выборочно)

Тип	Свойство	Доступ
Size	*ImageSize*	Чтение/запись
ColorDepth	*ColorDeptb*	Чтение/запись
Color	*TransparentColor*	Чтение/запись
ImageList.ImageCollection	*Images*	Чтение

Значение свойства *ImageSize* по умолчанию — квадрат со стороной 16 пикселов; при добавлении изображений в список оно не изменяется автоматически. Возможно, вам потребуется задать его самостоятельно на основе размера добавляемых объектов *Image* и, вероятно, разрешения дисплея, с использованием которого выполняется программа,

Свойство *ColorDeptb* — это член перечисления *ColorDeptb.*

Перечисление *ColorDepth*

Член	Значение
Depth4Bit	4
Depth8Bit	S
Depth16Bit	16
Depth24Bit	2!
Depth32Bit	32

Значение свойства *ColorDepth* по умолчанию — *Depth8Bit*, и его требуется изменять вручную в зависимости от используемых изображений. К счастью, перечисление *ColorDepth* определено довольно рационально. Благодаря этому, если у вас есть объект *Image* (например, с именем *image*), который нужно поместить в список изображений, можно методом *Image.GetPixelFormatSize* получить формат его пикселов и число бит на пиксел и затем преобразовать результат в значение типа *ColorDepth*.

```
imglst.ColorDepth = (ColorDepth) Image.GetPixelFormatSize(image.PixelFormat);
```

Знаю, что четвертое свойство с именем *Images* выглядит несколько устрашающе, так как его тип определен как *ImageList.ImageCollection*. Длинное имя означает лишь то, что это класс *ImageCollection*, определенный в классе *ImageList*. Обращаться к классу *ImageCollection* в приложении не нужно: чтобы задействовать его свойства и методы, достаточно сослаться на свойство *Images* — именно в нем хранятся все изображения объекта *ImageList*.

Функциональность свойства *Images* проявляется в ряде других классов пространства имен *System.Windows.Forms*. В главе 12 обсуждается свойство *Controls* класса *Control*, тип которого — *Control.ControlCollection*. В главе 14 — свойство *MenuItems* класса *Menu* с типом *Menu.MenuItemCollection*. Все эти свойства работают примерно одинаково. Типы всех этих свойств реализуют интерфейсы *IList*, *ICollection* и *IEnumerable* (определенные в пространстве имен *System.Collections*), которые позволяют данным свойствам работать как расширяемым массивам.

Для создания объекта типа *ImageList* вызывается конструктор по умолчанию.

```
ImageList imglst = new ImageList();
```

Затем задаются значения свойств *ImageSize* и *ColorDepth*. Добавить объекты *Image* в список изображений позволяет один из следующих методов.

Методы *ImageList.ImageCollection* (выборочно)

```
void Add(Image image)
void Add(Image image, Color clrTransparent)
void Add(Icon icon)
void AddStrip(Image image)
```

Поскольку эти методы определены в классе *ImageList.ImageCollection*, вызвать их можно через свойство *Images* объекта *ImageList*. На самом деле все гораздо проще, чем кажется:

```
imglist.Images.Add(image);
```

Каждому добавляемому изображению назначается индекс, начиная с 0. Метод *AddStrip* добавляет несколько изображений, число которых зависит от ширины передаваемого методу изображения и ширины, определяемой свойством *ImageSize*.

Следующее свойство указывает число изображений в объекте *ImageList*.

Свойства *ImageList.ImageCollection* (выборочно)

Тип	Свойство	Доступ
int	*Count*	Чтение

Свойство *Count* можно использовать в выражениях так:

```
imglst.Images.Count
```

Гораздо важнее то, что к свойству *Images* можно обращаться, как при работе с массивом. Выражение:

```
imglst.Images[2]
```

возвращает третий объект *Image* из списка изображений. Если список включает менее трех изображений, выражение вызывает исключение. Кроме того, индексы позволяют заменить изображение в списке:

```
imglst.Images[3] = image;
```

Если же в списке меньше четырех изображений, это выражение вызовет исключение.

Можно также определить, присутствует ли конкретное изображение в списке, и получить его индекс,

Методы *ImageList.ImageCollection* (выборочно)

```
bool Contains(Image image)
int  IndexOf(Image  image)
```

Например, выражение:

```
imglst.Images.IndexOf(image)
```

возвращает 2, если объект *image* — третий в списке, и -1, если его в списке нет. Из списка можно удалять как отдельные, так и все изображения,

Методы *ImageList.ImageCollection* (выборочно)

```
void  RemoveAt(int index)
void  Clear()
```

Как рассказывается в следующих главах, объект *ImageList* можно использовать совместно с различными элементами управления, в частности, с элементом управления *ToolBar*. Сам по себе *ImageList* — это удобный способ хранения нескольких изображений одинакового размера. Для вывода этих изображений класс *ImageList* предоставляет метод *Draw*.

Методы *Draw* класса *ImageList*

```
void  Draw(Graphics grfx,  Point  pt,  int  index)
void  Draw(Graphics grfx,  int x,  int  y,  int  index)
void  Draw(Graphics grfx,  int x,  int  y,  int  cx,  int  cy,  int  index)
```

Заметьте: индекс изображения в списке — последний аргумент. Например, выражение:

```
imglst.Draw(grfx, x, y, 1)
```

выводит второе изображение из списка.

Обращайте особое внимание на координаты, передаваемые методам *Draw*. Первые два метода принимают точки координат в единицах измерения устройства (пикселах). Размер изображения основывается на свойстве *ImageSize* объекта *ImageList* и измеряется в пикселах. На эти два метода не влияют ни преобразование страницы, ни глобальное преобразование! Координаты устройства предназначены для максимизации производительности, но на принтерах это приводит к эффекту почтовой марки. Третий метод *Draw* принимает координату, а также ширину и высоту в глобальных координатах.

Класс *PictureBox*

PictureBox ~ еще один класс элементов управления, связанный с изображениями. Он происходит от класса *Control* (и, следовательно, может обрабатывать ввод с клавиатуры и мыши), но редко когда выполняет что-либо, кроме вывода изображения. Вот его самые важные свойства:

Свойства *PictureBox* (выборочно)

Тип	Свойство	Доступ
Image	*Image*	Чтение/запись
BorderStyle	*BorderStyle*	Чтение/запись
PictureBoxSizeMode	*SizeMode*	Чтение/запись

Рамкой вокруг изображения управляют элементы перечисления *BorderStyle:*

Перечисление *BorderStyle*

Член	Значение
None	0
FixedSingle	1
Fixed3D	2

Значение по умолчанию — *None*. Перечисление *PictureBoxSizeMode* определяет способы отображения изображений на элементах управления:

Перечисление *PictureBoxSizeMode*

Член	Значение
Normal	0
StretchImage	1
AutoSize	2
CenterImage	3

Значение по умолчанию — *Normal*. Как и в случае с другими элементами управления, указать расположение *PictureBox* относительно родительского элемента позволяет свойство *Location*, а задать ширину и высоту — свойство *Size*. Если задать свойству *SizeMode* значение *PictureBoxSizeMode.Normal* или *PictureBoxSizeMode.CenterImage*, изображение будет выводиться в графическом окне в пиксельном (не метрическом) размере.

При значении *PictureBoxSizeMode.Normal* картинка выравнивается по верхнему левому краю элемента управления. Если размер элемента управления больше писельного размера изображения, справа и снизу от последнего вы увидите фоновый цвет элемента (*BackColor*)управления. Если элемент управления меньше изображения, правая и нижняя части изображения будут скрыты.

При значении *PictureBoxSizeMode.CenterImage* изображение размещается по центру элемента управления. И все же оно по-прежнему выводится в пиксельном размере и в зависимости от размеров картинки может отображаться частично или с рамкой цвета фона элемента управления.

Если задать свойству *ClientSize* элемента управления *PictureBox* значение свойства *Size* объекта *Image*, размеры элемента управления будут точно соответствовать размеру изображения (свойство *ClientSize* элемента управления определяет размер в пределах рамки). Кроме того, можно задать свойству *SizeMode* значение *PictureBoxSizeMode.AutoSize* и привязать размер элемента управления к размеру объекта *Image*.

При значении *PictureBoxSizeMode.StretchImage* картинка увеличивается до размеров элемента управления. Однако в случае несовпадения форматов изображения и элемента управления изображение искажается,

А где же элемент перечисления *PictureBoxSizeMode*, масштабирующий изображение изотропно? Увы, такого нет. Я устранил этот недостаток при помощи элемента управления *PictureBoxPlus*, который переопределяет элемент управления *PictureBox* и имеет дополнительное свойство *NoDistort*.

PictureBoxPlus.cs

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

namespace Petzold.ProgrammingWindowsWithCSharp
{
    class PictureBoxPlus : PictureBox
    {
        bool bNoDistort = false;

        public bool NoDistort
        {
            get { return bNoDistort; }
            set
            {
                bNoDistort = value;
                Invalidate();
            }
        }
        protected override void OnPaint(PaintEventArgs pea)
```

см. след. стр.

```
        {
            if ((Image != null) && NoDistort &&
                (SizeMode == PictureBoxSizeMode.StretchImage))
                ScaleImageIsotropically(pea.Graphics, Image,
                ClientRectangle);
            else
                base.OnPaint(pea);
        }
        void ScaleImageIsotropically(Graphics grfx, Image image,
                                     Rectangle reet)
        {
            SizeF sizef =
                new SizeF(image.Width / image.HorizontalResolution,
                          image.Height / image.VerticalResolution);

            float fScale = Math.Min(rect.Width  / sizef.Width,
                                    rect.Height / sizef.Height);

            sizef.Width  *= fScale;
            sizef.Height *= fScale;

            grfx.DrawImage(image,
                           rect.X + (rect.Width  - sizef.Width ) / 2,
                           rect.Y * (rect.Height - sizef.Height) / 2,
                           sizef.Width, sizef.Height);
        }
    }
}
```

Данный элемент управления выводит изображение методом *ScaleImageIsotropically*, только если значение свойства *SizeMode* — *PictureBoxSizeMode.StretchImage* и значение свойства *NoDistort* — *true*. В противном случае элемент управления вызывает метод *OnPaint* базового класса.

Следующая программа тестирует элемент управления *PictureBoxPlus* при помощи его свойства *Dock*, растягивая элемент управления на всю клиентскую область формы. По функциональности эта программа аналогична ImageScaleIsotropic.

PictureBoxPlusDemo.cs

```
//-------------------------------------------,----: "
// PictureBoxPlusDemo.cs © 2001 Charles Petzold
//-----_____-------_____________________-----_____---
using Petzold.ProgrammingWindowsWithCSharp;
using System;
using System.Drawing;
using System.Windows.Forms;

class PictureBoxPlusDemo: Form
{
```

см. след. стр.

```
    public static void Main()
    {
        Application.Run(new PictureBoxPlusDemo());
    }
    public PictureBoxPlusDemo()
    {
        Text = "PictureBoxPlus Demo";

        PictureBoxPlus picbox = new PictureBoxPlus();
        picbox.Parent = this;
        picbox.Dock = DockStyle.Fill;
        picbox.Image = Image.FromFile("..\\..\\..\\Apollo11FullColor.jpg");
        picbox.SizeMode = PictureBoxSizeMode.StretchImage;
        picbox.NoDistort = true;
    }
}
```



ПРИЛОЖЕНИЕ

А

Файлы и потоки

Большинство операций ввода-вывода в .NET Framework реализованы в пространстве имен *SystemJO.* При первом знакомстве (да и не только при первом) *SytemJO* может устрашить кого угодно. То, что .NET Framework предлагает богатый набор классов файлового ввода-вывода, не слишком успокаивает. Для С-программиста, привыкшего для операций файлового ввода-вывода использовать арсенал библиотечных функций типа *fopen, fread, fwrite* и *fprintf*, файловый ввод-вывод в .NET может выглядеть довольно сложным и запутанным.

Это приложение призвано помочь вам в знакомстве с *SystemJO.* Мне бы хотелось обратить ваше внимание на действительно важные моменты и, кроме того, объяснить, к чему это обилие классов.

В .NET Framework различаются файлы и потоки. *Файл* представляет собой поименованные данные, хранящиеся на диске; часто, кроме имени, указан и каталог При открытии для чтения/записи файл становится *потоком.* Поток — это сущность, допускающая операции чтения и записи. Но потоки — это не только открытые файлы. Данные, получаемые по сети, являются потоком, кроме того, поток можно создать в памяти. В консольных приложениях, клавиатурный ввод и текстовый вывод — тоже потоки.

Важнейший класс файлового ввода-вывода

Если вы собираетесь изучить только один класс из пространства имен *SystemJO,* это должен быть класс *FileStream.* Этот базовый класс используется для открытия, чтения, записи и закрытия файлов. *FileStream* наследуется от абстрактного класса *Stream,* и множество его свойств и методов являются производными из этого класса.

Для открытия существующего файла или создания нового нужно создать объект класса *FileStream.* который имеет пять конструкторов;

Конструкторы *FileStream* (выборочно)

```
FileStream(string strFileName, FileMode fm)
FileStream(string strFileName, FileMode fm, FileAccess fa)
FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs)
FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs,
           int iBufferSize)
FileStream(string strFileName, FileMode fm, FileAccess fa, FileShare fs,
           int iBufferSize, bool bAsync)
```

Кроме этого, есть еще четыре дополнительных конструктора, основанных на системном описателе файла. Их удобно использовать для взаимодействия с существующим кодом. *FileMode*, *FileAccess*, и *FileShare* — это перечисления, определенные в пространстве имен *System.IO*.

FileMode определяет, открывается ли существующий файл или создается новый, а также что делать, если открываемого файла не существует или, наоборот, создаваемый файл уже есть.

Перечисление *FileMode*

Член	Значение	Описание
CreateNew	1	Ошибка, если файл существует,
Create	2	Если файл существует, его содержимое удаляется.
Open	3	Ошибка, если файл не существует.
OpenOrCreate	4	Если файл не существует, он создается.
Truncate	5	Ошибка, если файл не существует; содержимое файла удаляется.
Append	6	Ошибка, если фэйл открывается на чтение; если файл не существует, он создается; указатель устанавливается на конец файла,

Под ошибкой я подразумеваю то, что конструктор *FileStream* вызывает исключение типа *IOException* или *FileNotFoundException*. Почти всегда конструктор *FileStream* следует вызывать в блоке *try,* чтобы аккуратно обработать все ошибки, связанные с возможным наличием или отсутствием файла.

Если вы не укажете аргумент *FileAccess,* файл открывается как для чтения, так и для записи. Аргумент *FileAccess* указывает режим доступа к файлу: чтение, запись или обе операции;

Перечисление *FileAccess*

Член	Значение	Описание
Read	1	Ошибка при использовании *FileMode.CreateNew*, *FileMode.Create* *FileMode.Truncate*, или *FileMode.Append*.
Write	2	Ошибка, если файл имеет атрибут «только для чтения».
ReadWrite	3	Ошибка при использовании *FileMode Append* или файл имеет атрибут «только для чтения».

Аргумент *FileAccess* нельзя опустить только в одном случае: когда вы открываете файл с указанием *FileMode.Append*, конструктор не выполнится, если файл открывается для чтения. Так как по умолчанию файлы открываются для чтения и записи, такой конструктор никогда не выполнится:

```
new FileStream(strFileName, FileMode.Append)
```

Если вы хотите использовать *FileMode.Append,* включите аргумент *FileAccess.Write*:

```
new FileStream(strFileName, FileMode.Append, FileAccess.Write)
```

Если вы не укажете аргумент *FileShare*, файл открывается для монопольного использования вашим процессом. Ни один другой процесс (включая ваш) не сможет открыть этот файл. Более того, если другой процесс уже открыл этот файл и вы не указываете аргумент *FileShare*, конструктор *FileStream* не сработает. Указать режим совместного доступа к файлу позволяет аргумент *FileShare*.

Перечисление *FileShare* (выборочно)

Член	Значение	Описание
None	0	Другие процессы не будут иметь доступа к файлу; значение по умолчанию.
Read	1	Разрешает другим процессам чтение из файла.
Write	2	Разрешает другим процессам запись в файл.
ReadWrite	3	Дает другим процессам полный доступ к файлу.

Обычно, если вы собираетесь только читать файл, другим процессам также разрешается читать его; иначе говоря, *FileAccess.Read* следует использовать совместно с *FileShare.Read*. Это правило работает в обе стороны: если другой процесс открывает файл с аргументами *FileAccess.Read* и *FileShare.Read*, ваш процесс не сможет открыть его с другими аргументами.

Свойства и методы *FileStream*

После того, как вы открыли файл, создав объект *FileStream,* можете обращаться к следующим пяти свойствам, реализованным в классе *Stream* и переопределенным в классе *FileStream*.

Свойства *Stream*

Тип	Свойства	Доступ
bool	*CanRead*	Чтение
bool	*CanWrite*	Чтение
bool	*CanSeek*	Чтение
long	*Length*	Чтение
long	*Position*	Чтение/запись

Первые два зависят от значения *FileAccess,* указанного при создании объекта *FileStream*. Свойство *CanSeek* всегда равно *true* для открытых файлов. Оно может быть *false* для других типов потоков (к примеру, сетевых).

Свойства *Length* и *Position* применимы только к потокам, в которых возможно позиционирование. Заметьте, что *Length u Position* — целые числа, имеющие тип *long,* и теоретически обеспечивают размер файла до 9 X 10^{12}, или 9 терабайт, По крайней мере на ближайшую пару лет этого должно хватить.

Простой способ перемещаться по файлу — применение свойства *Position*. (Чуть позже я рассмотрю более удобный метод *Seek*.) Например, если *fs* — объект типа *FileStream,* следующее выражение позволяет перейти на сотый байт в файле:

```
fs.Position = 100;
```

Перейти на конец файла (например, чтобы дописывать данные) можно так:

```
fs.Position = fs.Length;
```

Все следующие методы класса *Stream* переопределены в классе *FileStream.*

Методы *Stream* (выборочно)

```
int  ReadByte()
int  Read(byte[] abyBuffer, int iBufferOffset, int iCount)
void WriteByte(byte byValue)
void Write(byte[] abyBuffer, int iBufferOffset, int iCount)
long Seek(long lOffset, SeekOrigin so)
void SetLength(long lSize);
void Flush()
void Close()
```

Для чтения одного байта вы можете применить *ReadByte,* для чтения нескольких — *Read.* Оба метода возвращают значение *int,* но интерпретируется оно для них по-разному. *ReadByte* в случае нормального исполнения возвращает следующий байт из файла, приведенный к типу *int* без знака. Например, байт OxFF приводится к целому 0x000000FF, или 255. Возвращаемое значение -1 указывает на попытку чтения за пределами файла.

Read возвращает количество прочитанных в буфер байтов (не более *iCount*). Для файлов *Read* возвращает значение, равное *iCount,* если только *iCount* не превышает количества оставшихся в файле байтов. Возвращаемое значение 0 указывает ня то, что в файле больше не осталось байтов. В потоках других типов (например, сетевых) *Read* может вернуть значение, меньшее *iCount,* но большее либо равное 1, если не был прочитан весь поток. Второй аргумент методов *Read* и *Write* является смещением в буфере, а не в потоке!

Метод *Seek* похож на функции перемещения по файлу в языке С. Перечисление *SeekOrigin* определяет точку отсчета для аргумента *lOffset* метода *Seek:*

Перечисление *SeekOrigin*

Член	Значение
Begin	O
Current	1
End	2

Если поток допускает запись и позиционирование, метод *SetLength* устанавливает новую длину файла, возможно, обрезая часть содержимого, если новое значение длины файла меньше старого. Метод *Flush* записывает все данные из буферов в памяти в файл,

Независимо от того, что может произойти или не произойти после сбора мусора для объекта *FileStream,* всегда явно вызывайте метод *Close* для всех открытых вами файлов.

Если не обращать внимания на обработку исключений, в большинстве случаев вы можете прочесть весь файл в память четырьмя операторами, включая выделение буфера памяти в соответствии с размером файла:

```
FileStream fs = new FileStream("MyFile", FileMode.Open,
                               FileAccess.Read, FileShare.Read);
Byte[] abyBuffer = new Byte[fs.Length];
fs.Read(abyBuffer, 0, (int) fs.Length);
fs.Close();
```

Я сказал «в большинстве случаев», так как в этом коде предполагается, что длина файла меньше, чем 2^{31} (или 2 Гб). Точнее, это предположение принимается во внимание при приведении последнего аргумента метода *Read* от 64-битного *long* к 32-битному *int*. Если длина файла более 2 Гб, для его чтения потребуется несколько вызовов *Read*. (Но скорее всего не стоит даже и пытаться прочесть файл длиной в несколько гигабайт в память!)

FileStream прекрасно работает в традиционной программе, выводящей шестнадцатеричные коды символов.

HexDump.cs

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

class HexDump
{
    public static int Main(string[] astrArgs)
    {
        if (astrArgs.Length == 0)
        {
            Console.WriteLine("Syntax: HexDump file1 file2 ...");
            return 1;
        }
        foreach (string strFileName in astrArgs)
            DumpFile(strFileName);

        return 0;
    }
    protected static void DumpFile(string strFileName)
    {
        FileStream fs;

        try
        {
            fs = new FileStream(strFileName, FileMode.Open,
                                FileAccess.Read, FileShare.Read);
        }
        catch (Exception exc)
        {
            Console.WriteLine("HexDump: {0}", exc.Message);
            return;
```

см. след. стр.

```
        }
        Console.WriteLine(strFileName);
        DumpStream(fs);
        fs.Close();
    }
    protected static void DumpStream(Stream stream)
    {
        byte[] abyBuffer = new byte[16];
        long   lAddress  = 0;
        int    iCount;

        while ((iCount = stream.Read(abyBuffer, 0, 16» > 0)
        {
            Console.WriteLine(ComposeLine(lAddress, abyBuffer, iCount));
            lAddress += 16;
        }
    }
    public static string ComposeLine(long lAddress, fcyteH abyBuffer,
                                     int iCount)
    I
        string str * String.Format("{0:X4}-{1:X4)  ",
                           (uint) lAddress / 65536, (ushort) lAddress);

        for (int i = 0; i < 16; i++)
        {
            str += Ci < iCount) ?
                           String.Format("{0:X2}", abyBuffer[i]) : "  ";
            Str += (i == ? && iCount > 7) ? "-" : " ";
        >
        str += " ";

        for (int i = 0; i < 16; i++)
        {
            char ch = (i < iCount) ? Convert.ToChar(abyBuffer[i]) : ' ';
            str += Char.IsControl(ch) ? "." ; ch.ToString();
        }
        return str;
    }
}
```

В этой программе используется версия *Main c* одним аргументом — массивом строк, каждая из которых является одним параметром командной строки. В отличие от функции *main* в C метод *Main* в C# не содержит количества аргументов, кроме того, имя программы не входит в число аргументов. Если БЫ запустите эту программу, например так:

```
HexDump file1.cs file2.exe
```

то аргументом *Mam* будет массив строк из двух элементов. Групповые символы в аргументах автоматически *не* раскрываются. (Я еще вернусь к раскрытию групповых символов в этом приложении.)

После того, как HexDump успешно открывает файл, в программе используется метод *Read* для чтения 16-байтовых фрагментов из файла, и затем метод *ComposeLine* выводит их, Я использовал метод *ComposeLine* в главе 16.

Хочу вкратце указать еще на две особенности *FileStream*. При совместном доступе к файлу вы можете блокировать/разблокировать секции файла для монопольного использования:

Методы *FileStream* (выборочно)

```
void Lock(long lPosition, long lLength)
void Unlock(long lPosition, long lLength)
```

Если файловая система поддерживает асинхронные чтение/запись и к тому же вы использовали последний конструктор из приведенной ранее таблицы и указали для последнего аргумента значение *true,* для асинхронного чтения/записи в файл можно применить методы *BeginRead*, *EndRead*, *BeginWrite* и *EndWrite*.

Проблема, связанная с *FileStream*

Ранее я утверждал, что *FileStream* — важнейший класс в *SystemIO*, так как он позволяет открывать, читать и записывать файлы. Казалось бы, основные и наиболее часто используемые операции!

Трудность, однако, в том, что в отношении приведения типов C# далеко не так гибок, как C. К примеру, C-программист, чтобы считать из файла значение типа *int* может получить адрес целочисленной переменной и после приведения его к типу *указатель на байт* использовать в функции *fread*. Но в C# приведение чего-либо к байтовому массиву не сработает. Методы *FileStream Read* и *Write* работают только с байтовыми массивами, и ничто другое им не подходит.

Конечно, так как байт — минимальная единица данных, всегда можно считывать байты и объединять их в другие базовые типы данных (например, *char* или *int),* а базовые типы можно раскладывать на байты в процессе подготовки к записи, Но не думаю, что вам бы хотелось этим заниматься самостоятельно

Итак, если чтение и запись байтовых массивов — не предел ваших мечтаний, скорее всего недостаточно будет ограничиться знанием класса *FileStream*. Как я покажу чуть позже, для чтения/записи текстовых файлов используются *StreamReader* и *StreamWriter*, а для чтения/записи двоичных — *BinaryReader* и *BinaryWriter*.

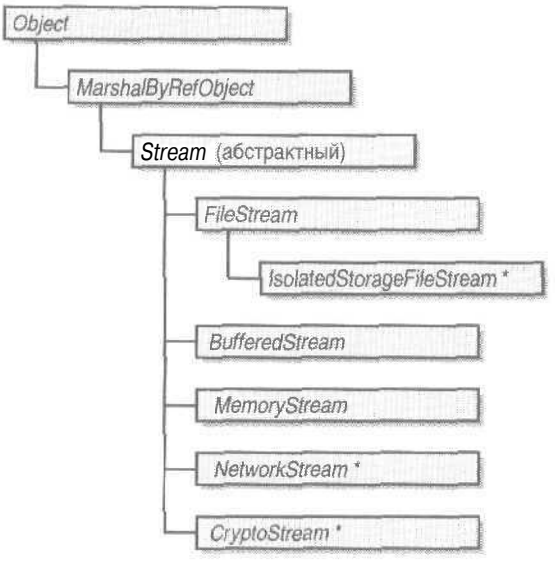
Другие потоковые классы

FileStream является одним из нескольких классов — наследников абстрактного класса *Stream*. Для класса, объект которого не может быть создан, *Stream* играет очень важную роль в .NET Framework. На иерархической диаграмме показаны шесть классов — наследников *Stream*:

Потоковые классы со звездочкой определены не в *SystemJO,* а в другом пространстве имен.

Кроме того, некоторые методы других классов из .NET Framework возвращают объекты типа *Stream*. Например, ниже я покажу, что программа .NET, читающая файлы

из Web, использует объект *Stream*. А пример программы из главы 11 показывает, что картинки (например, JPG-файлы) тоже могут загружаться из потоков,



С целью повышения производительности класс *FileStream* создает буферизированный поток. Определенный участок в памяти используется так, что не каждый вызов *ReadByte*, *Read*, *WriteByte* и *Write* приводит к обращению к функциям ОС чтения или записи файлов.

Если ваш объект *Stream* — небуферизированный поток, вы можете превратить его в такой поток, используя класс *BufferedStream*.

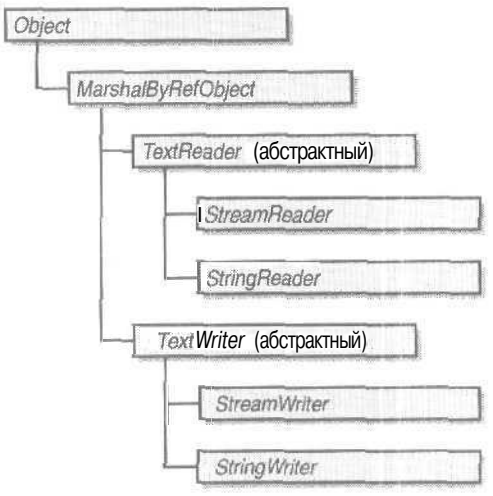
Класс *MemoryStream* позволяет создать расширяемый участок в памяти, доступ к которому можно осуществить методами класса *Stream*. Я использую класс *MemoryStream* в программе CreateMetafileMemory из главы 23 и, кроме того, в нескольких программах из главы 24.

Чтение и запись текста

Текстовый файл, состоящий только из строк текста, разделенных маркерами конца строки, — очень распространенный тип файла. В пространстве имен *System.IO* определены специальные классы для чтения и записи текстовых файлов. Иерархия этих классов показана на следующей странице.

Хотя эти классы не наследники *Stream*, они наверняка используют этот класс.

Задержим внимание на классах *StreamReader* и *StreamWriter*, предназначенных для чтения текстовых файлов или потоков. Два других неабстрактных класса, *StringReader* и *StringWriter*, строго говоря, не являются классами файлового ввода-вывода. Они используют аналогичные методы для чтения/записи строк. Их я вкратце рассмотрю в приложении В; кроме того, я использую класс *StringWriter* в программе EnumMetafile из главы 23.



Текстовые файлы могут показаться весьма простой формой хранения данных, однако в последние годы обработка текста изрядно усложнилась в связи с распространением Unicode.

Тип данных *System.Char* в .NET (и псевдоним *char* в C#) — это 16-битное значение, представляющее символ в наборе символов Unicode. Тип *System.String* .NET (и псевдоним *string* в C#) представляют строку символов Unicode. Но что происходит в C#-программе при записи строк в файл? Должны ли они быть записаны в Unicode? Это имеет смысл, только если каждое приложение, читающее данные из этого файла ожидает, что строки будут в Unicode! Вы скорее всего откажетесь от Unicode, если будете знать, что другие приложения, читающие данные из этого файла, ожидают 8-битных символов ASCII,

Первые 25б символов Unicode представляют собой 128 символов ASCII и 128 символов расширения ASCII ISO Latin Alphabet No. 1. (Сочетание этих двух наборов символов часто называется в документации по Windows API набором символов *ANSI.)* Например, заглавная буква A имеет код 0x41 в ASCII и 0x0041 в Unicode. Строки Unicode, состоящие только (или преимущественно) из символов ASCII, содержат массу нулей. Эти нули — большая проблема для огромного количества традиционных программ на C и программ для Unix, воспринимающих нулевой байт как символ конца строки.

Для облегчения решения этих проблем в классе *StreamWriter* предусмотрено управление тем, как строки Unicode в C#-программе преобразуются при записи в файл. Вы можете управлять этим преобразованием, используя классы, определенные в пространстве имен *System.Text.* Точно так же *StreamReader* позволяет читать текстовые файлы разных форматов и преобразовывать строки из этих файлов в Unicode.

Сначала взглянем на класс *StreamWriter.* Этот класс применяется для записи текста в новые или существующие файлы,

Четыре конструктора *StreamWriter* позволяют создавать объекты типа *StreamWriter* с указанием имени файла:

Конструкторы *StreamWriter* (выборочно)

```
StreamWriter(string strFileName)
StreamWriter(string strFileName, bool bAppend)
StreamWriter(string strFileName, bool bAppend, Encoding enc)
StreamWriter(string strFileName, bool bAppend, Encoding enc, int iBufferSize)
```

Эти конструкторы открывают файл для записи, вероятно, вызывая внутри конструктор *FileStream*. По умолчанию, если файл существует, его содержимое будет уничтожено. Параметр *bAppend* позволяет изменить это поведение. Остальные конструкторы создают объект типа *StreamWriter* на основе существующего объек га *Stream*.

Конструкторы *StreamWriter* (выборочно)

```
StreamWriter(Stream stream)
StreamWriter(Stream stream, Encoding enc)
StreamWriter(Stream stream, Encoding enc, int iBufferSize)
```

При использовании конструктора без параметра *Encoding,* объект *StreamWriter не* будет сохранять строки в формате Unicode, выделяя 2 байта на каждый символ. Но и в ASCII строки также преобразованы не будут. Вместо этого объект *StreamWriter* будет сохранять строки в формате UTF-8, который я рассмотрю позже.

Для указания параметра *Encoding* любого конструктора *StreamWriter* вам требуется объект типа *Encoding* — класса, определенного в пространстве имен *System.Text*. Простейший способ (и обычно этого достаточно) получения такого объекта — использовать статические свойства класса *Encoding*.

Статические свойства *Encoding*

Тип	Свойство	Доступ
Encoding	*Default*	Чтение
Encoding	*Unicode*	Чтение
Encoding	*BigEndianUnicode*	Чтение
Encoding	*UTF8*	Чтение
Encoding	*UTF7*	Чтение
Encoding	*ASCII*	Чтение

Кроме того, параметр *Encoding* конструктора *StreamWriter* может быть экземпляром одного из классов пространства имен *System.Text*, являющихся наследниками *Encoding*. К этим классам относятся *ASCIIEncoding*, *UnicodeEncoding*, *UTF7Encoding* и *UTF8Encoding*. Конструкторам этих классов часто нужны определенные аргументы, и, возможно, стоит их изучить, если вам не хватает статических свойств.

Когда указывается кодировка *Encoding.Unicode*, для каждого записываемого в файл символа нужно 2 байта, причем младший байт идет первым в соответствии с архитектурой «прямого порядка» (little-endian architecture), применяемой в процессорах Intel. Файл или поток начинается с байтов OxFF и OxFE, соответствующих символу Unicode OxFEFF, определенному в стандарте Unicode как байтовый указатель границы (byte order mark, BOM).

При кодировке *Encoding.BigEndianUnicode* данные сохраняются так, что первым идет старший байт, Файл или поток начинается с байтов OxFE и OxFF, что также соответствует символу Unicode OxFEFF. Символ Unicode 0xFFFE умышленно не определен, чтобы приложения могли определять порядок следования байтов в файле по его первым двум байтам,

Если вы хотите хранить строки в Unicode, но при этом не хотите, чтобы использовались байтовые указатели границы, то в качестве параметра *Encoding* передайте конструктору *StreamWriter* объект типа *UnicodeEncoding*:

```
new UnicodeEncoding(bBigEndian, bIncludeByteOrderMark)
```

Установите два булевских параметра соответственно.

UTF-8 представляет собой схему кодирования символов, предназначенную для представления символов Unicode без нулевых байтов (и, таким образом, пригодную для языка C и ОС Unix), UTF расшифровывается как *UCS Transformation Format*, UCS в свою очередь означает *Universal Character Set,* другое название стандарта кодирования символов ISO 10646, совместимого с Unicode.

В UTF-8 каждый символ Unicode переводится в последовательность из 1-6 ненулевых байтов. Символы Unicode из диапазона ASCII (от 0x0000 до Ox007F) для своего хранения требуют 1 байта. Таким образом, строки Unicode, содержащие только символы ASCII, переводятся в файлы ASCII. UTF-8 определен в RFC 2279. (RFC, Request For Comment — документация Интернет-стандартов. RFC можно получить из разных источников, в том числе с Web-сайта Internet Engineering Task Force, *http://www.ietf.org*.)

При использовании *Encoding.UTF8* класс *StreamWriter* переводит текстовые строки Unicode в UTF-8. Кроме того, в начало файла или потока записываются 3 байта: OxEF, OxBB и OxBF. Эти байты — не что иное, как маркер начала последовательности Unicode, переведенный в UTF-8.

Если вы хотите хранить строки в кодировке UTF-8, но не хотите, чтобы применялся маркер начала последовательности, не используйте *Encoding.UTF8*. Вместо этого возьмите *Encoding.Default* или один из конструкторов, не имеющих аргумента *Encoding*. В таком случае также будет применяться кодировка UTF-8, но 3 идентификационных байта появляться не будут.

Кроме того, вы можете создать объект типа *UTF8Encoding* и передать его как параметр *StreamWriter*. Используйте:

```
new UTF8Encoding()
```

ИЛИ:

```
new UTF8Encoding(false)
```

чтобы BOM не использовался, и:

```
new UTF8Encoding(true)
```

чтобы использовать BOM.

UTF-7 определен в RFC 2152. Символы Unicode переводятся в последовательность байтов, у которых старший бит установлен в О, UTF-7 применяется в средах, в которых допускаются только 7-битные значения, например, в электронной

почте. Для использования UTF-7 передайте конструктору *StreamWriter* параметр *Encoding.UTF7*. В UTF-7 не применяются идентификационные байты.

При указании кодировки *Encoding.ASCII* результирующий файл или поток содержит только символы ASCII, т. е, символы в диапазоне OxOO-Ox7E Каждый символ Unicode, не попадающий в этот диапазон, заменяется вопросительным знаком (код ASCII Ox3F). Это единственная кодировка, при применении которой данные могут быть потеряны.

Класс *StreamWriter* имеет несколько полезных свойств:

Свойства *StreamWriter* (выборочно)

Тип	Свойство	Доступ
Stream	*BaseStream*	Чтение
Encoding	*Encoding*	Чтение
bool	*AutoFlush*	Чтение/запись
string	*NewLine*	Чтение/запись

Свойство *BaseStream* возвращает либо объект *Stream*, использованный для создания объекта *StreamWriter,* либо объект *Stream,* созданный классом *StreamWriter* на основе указанного вами имени файла. Если базовый поток можно позиционировать, вы можете использовать этот объект для того, чтобы выполнять операции позиционирования над потоком.

Свойство *Encoding* возвращает кодировку, указанную вами в конструкторе, либо *UTF8Encoding* если вы ее не указали. Если вы установите свойство *AutoFlush* в *true.* то после каждой записи будет выполняться сброс буфера.

Свойство *NeivLine* унаследовано из *TextWriter.* По умолчанию оно равно строке «\r\n» (возврат каретки и новая строка), но его можно изменить на «\n» (новая строка). Если вы измените его на что-либо другое, эти файлы нельзя будет прочесть с помощью объектов *StreamReader.*

Многосторонность класса *StreamWriter* требует двух методов *Write* и *WriteLine,* унаследованных от *TextWriter.*

Методы *TextWriter* (выборочно)

```
void Write(...)
void WriteLine(...)
void Flush()
void Close()
```

TextWriter поддерживает 17 версий *Write* и 18 версий *WriteLine,* позволяющих указать в качестве параметра любой объект (все эти методы наследуются *StreamWriter*). Объект-аргумент преобразуется в строку с помощью его собственного метола *ToString. WriteLine* выводит эту строку до символа конца строки. Версия *WriteLine* без аргументов просто выводит символ конца строки. Кроме того, существуют версии методов *Write* и *WriteLine* форматирующие строки аналогично методам *Console.Write* и *Console .WriteLine.*

Далее представлена маленькая программа, дописывающая текст к одному и тому же файлу каждый раз при запуске,

StreamWriterDemo.cs

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

class StreamWriterDemo
{
     public static void Main()
     {
          StreamWriter sw = new StreamWriter("StreamWriterDemo.txt", true);

          sw.WriteLine("You ran the StreamWriterDemo program on {0}",
                       DateTime.Now);

          sw.Close();
     }
}
```

Обратите внимание на параметр *true,* переданный конструктору. Именно он позволяет произвести добавление к файлу. Строки Unicode преобразуются в UTF-8, хотя в данном случае они будут ASCII-строками.

Я уже упоминал класс *Console,* Устройства ввода-вывода в нем определены как объекты типа *TextWriter.* Попробуйте вставить эти строки в начало метода *Main* в программе HexDump:

```
StreamWriter sw = new StreamWriter("prn", false, Encoding.ASCII);
Console.SetOut(sw);
```

Кроме того, вам придется добавить выражение *using:*

```
using System.Text;
```

Теперь весь вывод программы будет отправлен на принтер.

Класс *StreamReader* предназначен для чтения текстовых файлов или потоков. Существует пять конструкторов, открывающих файл для чтения:

Конструкторы *StreamReader* (выборочно)

```
StreamReader(string strFileName)
StreamReader(string strFileName, Encoding enc)
StreamReader(string strFileName, bool bDetect)
StreamReader(string strFileName, Encoding enc, bool bDetect)
StreamReader(string strFileName, Encoding enc, bool bDetect, int iBufferSize)
```

Кроме этого, пять конструкторов предназначено для создания объекта *StreamReader* на основе существующего потока:

Конструкторы *StreamReader* (выборочно)

```
StreamReader(Stream stream)
StreamReader(Stream stream, Encoding enc)
StreamReader(Stream stream, bool bDetect)
StreamReader(Stream stream, Encoding enc, bool bDetect)
StreamReader(Stream stream, Encoding enc, bool bDetect, int iBufferSize)
```

Если вы установите *bDetect* в *true,* конструктор попытается определить кодировку файла по первым двум или трем байтам. С другой стороны, вы можете явно указать кодировку. Если вы установите *bDetect* в *true* и при этом укажете кодировку, конструктор использует ее, только если не сможет самостоятельно определить кодировку файла. (Например, конструктор не может отличить ASCII от UTF-7, так как файлы в этих кодировках не начинаются с маркера начала последовательности и содержат символы в диапазоне от 0x00 до Ox7F.)

Класс *StreamReader* содержит два неизменяемых свойства:

Свойства *StreamReader*

Тип	Свойство	Доступ
Stream	*BaseStream*	Чтение
Encoding	*CurrentEncoding*	Чтение

Свойство *CurrentEncoding* может изменяться в интервале от создания объекта до первой операции чтения из файла или потока, так как объект может определить идентификационные байты только после первого чтения.

Следующие методы служат для считывания одного или нескольких элементов данных, а также закрытия текстовых файлов:

Методы *StreamReader*

```
int Peek()
int Read()
int Read(char[] achBuffer, int iBufferOffset, int iCount)
string ReadLine()
string ReadToEnd()
void Close()
```

И метод *Peek,* и первый метод *Read* — оба возвращают следующий символ в потоке или −1, если был достигнут конец потока. Следует явно приводить возвращаемое значение к типу *char,* если оно не равно −1. Второй метод *Read* возвращает количество прочитанных символов или 0, если был достигнут конец потока.

Метод *ReadLine* читает следующую строку вплоть до следующего маркера конца строки и удаляет символы конца строки из результирующей строки. Метод возвращает строку нулевой длины, если строка текста содержит только маркер конца строки, и *null,* если был достигнут конец потока.

Метод *ReadToEnd* возвращает все от текущей позиции до конца файла, а если был достигнут конец потока, возвращает *null.*

Следующая программа предполагает, что аргументом командной строки является URI (Universal Resource Identifier) HTML-файла (или другого текстового файла) в Web. Объект *Stream* для этого файла создается стандартным способом — через классы *WebRequest* и *WebResponse*. Затем на основе этого потока создается объект *StreamReader*, и используется метод *ReadLine* для чтения каждой строки, которая затем выводится с помощью *Console.WriteLine* с указанием ее номера.

HtmlDump.cs

```
// ----------- ' -------- -----'-- -------
// HtmlDump.cs © 2001 by Charles Petzold
//-------------------------------------
using System;
using System.IO;
using System.Net;

class HtmlDump
{
    public static int Main(string[] astrArgs)
    {
        if (astrArgs.Length == 0)
        {
            Console.WriteLine("Syntax: HtmlDump URI");
            return 1;
        }

        WebRequest webreq;
        WebResponse webres;

        try
        {
            webreq = WebRequest.Create(astrArgs[0]);
            webres = webreq.GetResponse();
        }
        catch (Exception exc)
        {
            Console.WriteLine("HtmlDump: {0}", exc.Message);
            return 1;
        }

        if (webres.ContentType.Substring(0, 4) != "text")
        {
            Console.WriteLine("HtmlDump: URI must be a text type.");
            return 1;
        }

        Stream       stream = webres.GetResponseStream();
        StreamReader strrdr = new StreamReader(stream);
```

см. след. стр.

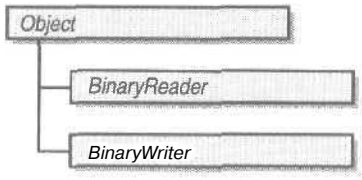
```
            string      strLine;
            int         iLine = 1;

            while ((strLine = strrdr.ReadLine()) != null)
                Console.WriteLine("{0:D5}: {1}", iLine++, strLine);

            stream.Close();
            return 6;
        }
    }
```

Ввод-вывод двоичных файлов

По определению любой файл, не являющийся текстовым, — двоичный. Я уже рассматривал класс *FileStream*, позволяющий читать и записывать байты. Но большинство двоичных файлов состоит из таких данных, типы которых требуют для своего хранения несколько байтов. Если у вас нет желания писать код для составления и разделения целых чисел и других типов данных на составляющие их байты, задействуйте классы *BinaryReader* и *BinaryWriter*, производные от класса *Object*:



Конструкторы этих классов требуют объект *Stream*. Если вам нужно использовать с этими классами файл, сперва создайте новый объект типа *FileStream* (или получите его иначе). Для класса *BinaryWriter* параметр *Encoding* определяет способ хранения текста в потоке:

Конструкторы *BinaryWriter*

```
BinaryWriter(Stream stream)
BinaryWriter(Stream stream, Encoding enc)
```

Конструкторы для *BinaryReader* выглядят точно так же:

Конструкторы *BinaryReader*

```
BinaryReader(Stream stream)
BinaryReader(Stream stream, Encoding enc)
```

Оба класса содержат единственное неизменяемое свойство — *BaseStream*, представляющее собой объект *Stream*, переданный конструктору.

Метод *Write* класса *BinaryWriter* определен для всех базовых типов, а кроме того, для массивов байтов и символов.

Открытые методы *BinaryWriter*

```
void Write(...)
void Write(byte[] abyBuffer, int iBufferOffset, int iBytesToWrite)
void Write(char[] achBuffer, int iBufferOffset, int iBytesToWrite)
long Seek(int iOffset, SeekOrigin so)
void Flush()
void Close()
```

В качестве параметра для *Write* может быть использован объект любого базового типа *(pool, byte, sbyte, byte[], char, char[], string, short, ushort, int, uint, long, ulong, float, double* или *decimal*).

Эти методы не сохраняют информацию о типах данных. Для каждого типа используется необходимое количество байтов. Например, для типа *float* требуется 4 байта, для *bool* — 1 байт. Размеры массивов не сохраняются. Массив байтов из 256 элементов требует для сохранения 256 байтов.

Записываемые в файл строки предваряются длиной в байтах, записываемой 7-битными кодированными целыми числами. (7-битное кодирование целого числа требует столько байт, сколько нужно для сохранения целого числа в фрагментах по 7 битов. Первый байт — это младшие 7 битов целого числа и т. д. Старший бит каждого байта установлен в 1, если далее следует байт кодированного целого числа. Класс *BinaryWriter* содержит метод *Write7BitEncodedInt* выполняющий эту кодировку.)

Метод *Close* закрывает поток, на котором основан объект *BinaryWriter*.

Класс *BinaryReader* содержит методы для чтения различных типов данных.

Методы *BinaryReader* (выборочно)

```
bool ReadBoolean()
byte ReadByte()
byte[] ReadBytes(int iCount)
sbyte ReadSByte()
char ReadChar()
char[] ReadChars(int iCount)
short ReadInt16()
int ReadInt32()
long ReadInt64()
ushort ReadUInt16()
uint ReadUInt32()
ulong ReadUInt64()
float ReadSingle()
double ReadDouble()
decimal ReadDecimal()
```

Эти методы генерируют исключение типа *EndOfStreamException*, если достигнут конец потока. В большинстве случаев ваша программа знает формат двоичного файла, который она читает, и может избежать выхода за границы потока. Однако для максимальной защиты, следует помещать операции чтения внутри блоков *try* на случай повреждений в файлах.

Также возможно чтение отдельных символов или массивов байтов и символов:

Методы *BinaryReader* (выборочно)

```
int PeekChar()
int Read()
void Read(byte[] abyBuffer, int iBufferOffset, int iBytesToRead)
void Read(char[] achBuffer, int iBufferOffset, int iBytesToRead)
void Close()
```

Методы *PeekChar* и *Read* оперируют с символами, а не байтами и предполагают, что кодировка файла — UTF-8, если в конструкторе не была явно указана кодировка. Эти методы возвращают -1, если был достигнут конец файла,

Если у вас есть опыт программирования файлового ввода-вывода на языке С. вам скорее всего знакомы обычные методы чтения/записи двоичных структур, Например, вы можете определить структуру:

```
typedef struct
{
    int   i;
    float f;
    char  ch[10];
    int   j;
    float g;
}
STRUCTDEF;
```

и переменную типа *STRUCTDEF* таким образом:

```
STRUCTDEF mystruct;
```

Если вы открываете файл функцией *fopen* и даете указателю на *FILE* имя *file*, вы можете записать содержимое этой структуры в файл функцией *fwrite*:

```
fwrite(&mystruct, sizeof(STRUCTDEF), 1, file);
```

точно так же ее можно прочесть из файла:

```
fread(&mystruct, sizeof(STRUCTDEF), 1, file);
```

Это очень легко в силу того, что в С содержимое структуры хранится в одном блоке памяти. Первый аргумент функций *fwrite* и *fread* определен как указатель на *void*, и вместо него можно передавать указатель на что угодно.

В C# вы лишены этой свободы приведения типов. Для чтения/записи двоичных данных вам придется использовать совершенно другой (и более структурированный) подход. Вместо того чтобы определять структуры типа *STRUCTDEF*, вы определяете классы. Сохраняя экземпляр класса в файле, вы должны также сохранить информацию, необходимую для воссоздания этого объекта из файла. В хорошо спроектированном классе C# сохраняются все свойства класса, необходимые для воссоздания объекта,

Допустим, класс *SampleClass* содержит три свойства для воссоздания объекта: *Value* типа *float*, *Text* типа *string* и объект типа *Fish*, хранящийся в свойстве *BasicFish*.

(*Fish* — это другой созданный вами класс.) Для *SampleClass* определен конструктор, создающий новый объект с этими тремя свойствами:

```
public SampleClass(float fValue, string strText, Fish fish)
```

Пусть для хранения множества различных объектов, включая объекты типа *SampleClass,* используется двоичный файл, Каждый созданный вами класс реализует как обычный метод *Write,* так и статический метод *Read.* Вот пример метода *Write* класса *SampleClass.* Обратите внимание на параметр *BinaryWriter.*

```
public void Write(BinaryWriter bw)
{
    bw.Write(Value);
    bw.Write(Text);
    BasicFish.Write(bw);
}
```

Так как свойства *Value* и *Text* имеют базовые типы данных, в этом методе просто вызывается метод *Write* класса *BinaryWriter.* Однако для свойства *BasicFish* следует вызвать аналогичный метод *Write,* реализованный в классе *Fish,* передав параметр *BinaryWriter.*

Метод *Read* должен быть статическим, так как он должен создавать экземпляр класса *SampleClass* после чтения двоичных данных из файла:

```
public static SampleClass Read(BinaryReader br)
{
    float fValue = br.ReadSingle();
    string strText = br.ReadString();
    Fish fish = Fish.Read(br);
    return new SampleClass(fValue, strText, fish);
}
```

Заметьте: класс *Fish* должен содержать аналогичный статический метод *Read.*

Класс *Environment*

Отвлечемся на время от пространства имен *SystemJO* и обратим внимание на класс *Environment,* определенный в пространстве имен *System. Environment* содержит набор свойств и методов, весьма удобных для получения информации о компьютере, на котором выполняется программа, и текущем пользователе. Как и следует из его имени, *Environment* позволяет получать значения переменных среды. (Я использую последнюю особенность класса *Environment* в программе EnvironmentVars в главе 18.)

Два метода *Environment* возвращают информацию о файловой системе:

Статические методы *Environment* (выборочно)

```
string[] GetLogicalDrives()
string GetFolderPath(Environment.SpecialFolder sf)
```

У меня обычная система, в которой установлен привод CD-ROM и Zip-привод Iomega, и на моем компьютере *GetLogicalDrives* возвращает четыре строки в указанном порядке:

```
A: \
C \
D:\
I- \
```

Параметр функции *GetFolderPath* является членом перечисления *EnvironmentSpecialFolder*. Правая колонка следующей таблицы содержит строки, возвращаемые *GetFolderPath* на компьютере с обычной установкой Windows 2000, многоточие указывает, что возвращаемая строка содержит имя пользователя (совпадающее со значением, возвращаемым статическим свойством *EnvironmentUserName*).

Перечисление *Environment.SpecialFolder*

Член	Значение	Обычно возвращаемые значения
Programs	2	C:\Documents and Settings\...\Start Menu\ Programs
Personal	5	C:\Documents and Settings\...\My Documents
Favorites	6	C:\Documents and Settings\...\Favorites
Startup	1	C:\Documents and Settings\...\Start Menu\ Programs\Startup
Recent	8	C:\Documents and Settings\...\Recent
SendTo	9	C:\Documents and Scttings\...\SendTo
StartMenu	11	C:\Documents and Settings\...\Start Menu
DesktopDirectory	16	C:\Documents and Settings\...\Desktop
Templates	21	C:\Documents and Settings\...\Templates
ApplicationData	26	C:\Documents and Settings\...\Application Data
LocalApplicationData	28	C:\Documents and Settings\...\Local Settings\ ApplicationData
InternetCache	32	C:\Documents and Settings\...\Local Settings\ Temporary Internet Files
Cookies	33	C:\Documents and Settings\...\Cookies
History	34	C:\Documents and Settings\...\Local Settings\ History
CommonApplicationData	35	C:\Documents and Settings\All Users\ Application Data
System	37	C:\WINNT\System32
ProgramFiles	38	C:\Program Files
CommonProgramFiles	43	C:\Program Files\Common Files

Как ни странно, перечисление *SpecialFolder* определено в классе *Environment*. Вместо того, чтобы вызывать *GetFolderPath* вот так:

```
Environment.GetFolderPath(SpecialFolder.Personal)   // Не выполнится!
```

перед *SpecialFolder* надо указать класс, в котором определено перечисление:

```
Environment.GetFolderPath(Environment.SpecialFolder.Personal)
```

Кроме того, класс *Environment* содержит несколько свойств, относящихся к файловой системе и файловому вводу-выводу:

Статические свойства *Environment* (выборочно)

Тип	Свойство	Доступ
String	*SystemDirectory*	Чтение
String	*CurrentDirectory*	Чтение/запись

Свойство *SystemDirectory* возвращает ту же строку, что и метод *GetFolderPath*, с параметром *Environment.SpecialFolder.System*.

Свойство *CurrentDirectory* позволяет программе выяснить или установить текущий диск и каталог для приложения. При определении текущего каталога можно задавать относительный путь, включая строку «..», указывающую на родительский каталог. Чтобы перейти в корневой каталог на другом диске, используйте букву диска:

```
Environment.CurrentDirectory = "D:\\";
```

Если текущим диском не является диск C, вызовите:

```
Environment.CurrentDirectory = "C:";
```

и текущим каталогом станет каталог, бывший текущим на диске C перед сменой диска. С другими дисками эта конструкция работает не так, т. е. вызов:

```
Environment.CurrentDirectory = "D:";
```

всегда устанавливает в качестве текущего каталога корневой каталог на диске D.

Как вы вскоре увидите, в классах, определенных в пространстве имен *SystemJO* существует эквиваленты *GetLogicalDrives* и *CurrentDirectory*.

Разбор имен файлов и каталогов

Периодически возникает задача разбора имен файлов и каталогов. Например, бывает нужно получить имя каталога или диска по полному имени файла. Класс *Path,* определенный в пространстве имен *SystemJO*, состоит исключительно из статических методов и свойств только для чтения, облегчающих решение таких задач. В следующей таблице в двух правых колонках приводится примерные возвращаемые значения для тех случаев, когда аргумент *strFileName* равен строке, указанной в заголовке таблицы. В этих примерах предполагается, что текущий каталог — C:\Docs.

Статические методы Path (примеры)

Метод	\DirA\MyFile	DirA\MyFile.txt
`bool IsPathRooted(string strFileName)`	True	false
`bool HasExtension(string strFileName)`	False	true
`string GetFileName(string strFileName)`	MyFile	MyFile.txt
`String GetFileNameWithoutExtension (string strFileName)`	MyFile	MyFile
`string GetExtension(string strFileName)`		.txt
`string GetDirectoryName (string strFileName)`	\DirA	DirA
`string GetFullPath(string strFileName)`	C:\DirA\MyFile	C:\Docs\DirA\ MyFile.txt
`string GetPathRoot(string strFileName)`	\	

Что интересно: как DirA, так и MyFile вполне могут не существовать, однако методы прекрасно выполнятся. В этих методах производятся манипуляции над строками, иногда с применением текущего каталога.

Следующие два метода возвращают новый путь и имя файла:

Статические методы *Path* (выборочно)

```
string Combine(string strLeftPart, string strRightPart)
string ChangeExtension(string strFileName, string strNewExtension)
```

Метод *Combine* объединяет путь (слева) с путем и/или каталогом (справа). Для этой цели следует применить именно *Combine,* а не простую конкатенацию строк. Де то в том, что в случае *Combine* вам не придется беспокоиться о наличии или отсутствии обратных слэшей в конце левой части или начале правой. Метод *ChangeExtension* просто меняет расширение с одной строки на другую. В новое расширение надо включить точку. Чтобы удалить расширение, установите аргумент *strNewExtension* в *null*.

Следующие методы позволяют получить каталог, в котором программа может хранить временные файлы, а также полное уникальное имя для временного файла:

Статические методы *Path* (выборочно)

```
string GetTempPath()
string GetTempFileName()
```

Если вам нужно использовать собственный разбор имени файла и пути, не прописывайте жестко в программе символы, которые ожидаете встретить в строках. Примените следующие статические неизменяемые поля класса *Path*:

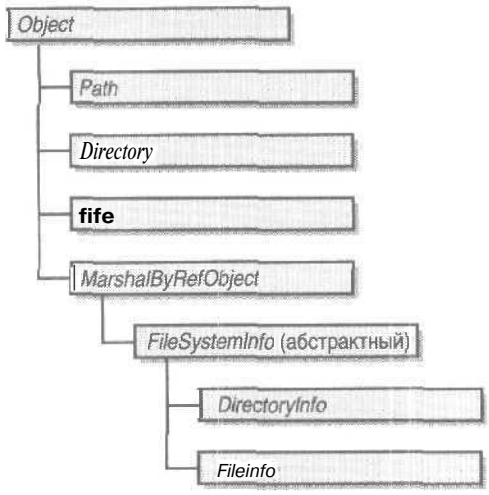
Статические поля *Path*

Тип	Поле	Доступ	По умолчанию в среде Windows
char	*PathSeparator*	Только чтение	;
char	*VolumeSeparatorChar*	Только чтение	:
char	*DirectorySeparatorChar*	Только чтение	\
char	*AltDirectorySeparatorChar*	Только чтение	/
char[]	*InvalidPathChars*	Только чтение	«<>|

Параллельные классы

Еще одна распространенная задача, связанная с файловым вводом-выводом, - получение списка всех файлов и подкаталогов в каталоге. Исторически решения этой задачи отличались громоздкостью. Стандартные библиотеки языка C не содержали встроенных возможностей такого типа, возможно, потому, что списки каталогов в Unix были текстовыми файлами, которые можно было напрямую читать и разбирать.

Четыре класса предоставляют информацию о файлах и каталогах: *Directory*, *File*, *DirectoryInfo* и *FileInfo*. Все они (так же, как и ранее рассмотренный класс *Path*) запечатаны и не могут быть предками. Ниже приведена иерархия классов:



Объекты классов *Directory* и *File* не могут быть созданы, эти два класса состоят только из статических методов.

DirectoryInfo и *Fileinfo не* содержат статических методов и свойств, для их использования необходим объект типа *Directorylnfo* или *Fileinfo*. Оба эти класса наследуют абстрактному классу *FileSystemInfo*; таким образом, у них общие свойства и методы.

Как и следует ожидать из их имени, классы *Directory* and *Directorylnfo* предлагают одинаковые методы за исключением того, что методы *Directory* статические и требуют указания имени каталога в качестве аргумента. Свойства и методы *Directorylnfo* нестатические, и на имя каталога, к которому будут применяться эти методы и свойства, указывает аргумент конструктора.

Точно так же *File* и *Fileinfo* предлагают аналогичные методы за исключением того, что в статических методах *File* указывается конкретное имя файла и экземпляр *File* создается с указанием имени файла в конструкторе.

Возможно, вы недоумеваете, когда же использовать *File*, а когда *Fileinfo* (или то же самое для каталогов: использовать *Directory* или *Directorylnfo"). Если вам нужен только один элемент возвращаемой информации, скорее всего проще использовать статические методы *File* или *Directory*. Если же вам требуется несколько элементов, скорее всего разумнее создать объект типа *Fileinfo* или *Directorylnfo* и затем задействовать его методы и свойства. Но в принципе можете смело использовать любой класс в любой ситуации.

Работа с каталогами

Начнем с классов *Directory* и *Directorylnfo*. Следующие три статических метода класса *Directory* не имеют эквивалентов в классе *Directorylnfo:*

Статические методы *Directory* (выборочно)

```
string[] GetLogicalDrives()
string GetCurrentDirectory()
void SetCurrentDirectory(string strPath)
```

Эти методы по сути — «двойники» статического метода *GetLogicalDrives* и статического свойства *CurrentDirectory* класса *Environment.*

Чтобы использовать свойства и методы класса *Directory Info,* нужно создать объект типа *DirectoryInfo,* например конструктором *DirectoryInfo:*

Конструктор *DirectoryInfo*

```
DirectoryInfo(string strPath)
```

Каталог не обязательно должен существовать. Более того, первым шагом при создании нового каталога является создание объекта типа *DirectoryInfo.*

Создав объект типа *DirectoryInfo,* вы можете выяснить, существует ли каталог. Даже если нет, вы можете получить определенную информацию о нем, как если бы он существовал. В двух правых столбцах следующей таблицы приведены примеры. Заголовки столбцов — строки, переданные *конструктору DirectoryInfo,* Предполагается, что текущий каталог C:\Docs,

Свойства *DirectoryInfo* (выборочно)

Тип	Свойство	Доступ	DirA	DirA\DirB.txt
bool	*Exists*	Чтение		
string	*Name*	Чтение	DirA	DirB.txt
string	*FullName*	Чтение	C:\Docs\DirA	C:\Docs\DirA\ DirB.txt
string	*Extension*	Чтение		.txt
DirectoryInfo	*Parent*	Чтение	C:\Docs	C:\Docs\DirA
DirectoryInfo	*Root*	Чтение	C:\	C:\

FullName и *Extension* унаследованы из класса *FileSystemInfo.*

Некоторые из этих свойств имеют «двойников» — статические методы класса *Directory.* Так как эти методы статические, им нужен аргумент — имя интересующего вас каталога:

Статические методы *Directory* (выборочно)

```
bool Exists(string strPath)
DirectoryInfo GetParent(string strPath)
string GetDirectoryRoot(string strPath)
```

Ранее я упоминал, что можно создать объект *DirectoryInfo* для несуществующего каталога. Затем вы можете создать каталог на диске, вызвав метод *Create,* кроме того, можно создать подкаталог в этом каталоге;

Методы *DirectoryInfo* (выборочно)

```
void Create()
DirectoryInfo CreateSubdirectory(string strPath)
void flefresh()
```

Заметьте: метод *CreateSubdirectory* возвращает другой объект *DirectoryInfo,* содержащий информацию о новом каталоге. Если указанный каталог уже есть, исклю-

чение *не* генерируется. Каталог, используемый для создания объекта *DirectoryInfo* или переданный в качестве аргумента *CreateSubdirectory*, может содержать несколько уровней имен каталогов.

Если при создании объекта *DirectoryInfo* каталог не существует, а затем вы вызываете *Create*, свойство *Exists* автоматически не станет равным *true*. Для обновления информации, содержащейся в *DirectoryInfo*, следует вызвать метод *Refresh* (унаследованный от *FileSystemInfo*).

Класс *Directory* также содержит статический метод для создания нового каталога:

Статические методы *Directory* (выборочно)

```
DirectoryInfo CreateDirectory(string strPath)
```

Метод *Delete* класса *DirectoryInfo* позволяет удалить каталог:

Методы *Delete* класса *DirectoryInfo*

```
void Delete()
void Delete(bool bRecursive)
```

В классе *Directory* существуют соответствующие статические методы:

Статические методы *Delete* класса *Directory*

```
void Delete(string strPath)
void Delete(string strPath, bool bRecursive)
```

При использовании второго варианта *Delete* из этих таблиц с установленным в *true* параметром *bRecursive* методы также удаляют все файлы и подкаталоги в указанном каталоге. В противном случае каталог должен быть пуст, иначе генерируется исключение.

Хотя такая информация более полезна для файлов, следующие четыре свойства завершают обзор свойств *DirectoryInfo:*

Свойства *DirectoryInfo* (выборочно)

Тип	Свойство	Доступ
FileAttributes	*Attributes*	Чтение/запись
DateTime	*CreationTime*	Чтение/запись
DateTime	*LastAccessTime*	Чтение/запись
DateTime	*LastWriteTime*	Чтение/запись

Эти свойства унаследованы от класса *FileSystemInfo* и все они, *кроме Attributes,* имеют «двойников» в виде статических методов класса *Directory*:

Статические методы *Directory* (выборочно)

```
DateTime GetCreationTime(string strPath)
DateTime GetLastAccessTime(string strPath)
DateTime GetLastWriteTime(string strPath)
void SetCreationTime(string strPath. DateTime dt)
```

```
void SetLastAccessTime(string strPath, DateTime dt)
void SetLastWriteTime(string strPath, DateTime dt)
```

Структура *DateTime* определена в пространстве имен *System*. *FileAttributes* является набором битовых флагов, определенных как перечисление:

Перечисление *FileAttributes*

Член	Значение
ReadOnly	0x00000001
Hidden	0x00000002
System	0x00000004
Directory	0x00000010
Archive	0x00000020
Device	0x00000040
Normal	0x00000080
Temporary	0x00000100
SparseFile	0x00000200
ReparsePoint	0x00000400
Compressed	0x00000800
Offline	0x00001000
NotContentIndexed	0x00002000
Encrypted	0x00004000

Для каталогов бит *Directory* (0x10) всегда установлен.

Перенести каталог и все его содержимое в другое место на том же диске можно методом *MoveTo*:

Методы *DirectoryInfo* (выборочно)

```
void MoveTo(string strPathDestination)
```

С другой стороны, можно использовать статический метод *Move* класса *Directory*:

Статические методы *Directory* (выборочно)

```
void Move(string strPathSource, string strPathDestination)
```

При использовании обоих методов каталог назначения не должен существовать.

Остальные методы возвращают либо список всех файлов и подкаталогов для каталога, либо только тех, что удовлетворяют указанному с помощью групповых символов (вопросительных знаков и звездочек) шаблону. Далее приведены 6 методов *DirectoryInfo:*

Методы *DirectoryInfo* (выборочно)

```
DirectoryInfo[] GetDirectories()
DirectoryInfo[] GetDirectories(string strPattern)
FileInfo[] GetFiles()
```

```
FileInfo[] GetFiles(string strPattern)
FileSystemInfo[] GetFileSystemInfos()
FileSystemInfo[] GetFileSystemInfos(string strPattern)
```

Метод *GetDirectories* возвращает набор каталогов в виде массива объектов типа *Directory Info*. Аналогично метод *GetFiles* возвращает набор файлов в виде массива объектов типа *FileInfo*. Метод *GetFileSystemInfo* возвращает как каталоги, так и файлы в виде массива объектов типа *FileSystemInfo*. Помните: *FileSystemInfo* — родительский класс для классов *DirectoryInfo* и *FileInfo*,

Класс *Directory* содержит набор аналогичных методов, но все они возвращают массив строк:

Статические методы *Directory* (выборочно)

```
string[] GetDirectories(string strPath)
string[] GetDirectories(string strPath, string strPattern)
string[] GetFiles(string strPath)
string[] GetFiles(string strPath, string strPattern)
string[] GetFileSystemEntries(string strPath)
string[] GetFileSystemEntries(string strPath, string strPattern)
```

Теперь мы полностью готовы к серьезному усовершенствованию приведенной ранее программы HexDump, с тем чтобы она понимала групповые символы в командной строке. Итак, вот программа WildCardHexDump.

WildCardHexDump.cs

```
//-----------------------------------------
// WildCardHexDump.cs © 2001 by Charles Petzold
//--- ------ ------- -- -- ---------- --- -- .
using System;
«sing System.IO;

class WildCardHexDump: HexDump
{
    public new static int Main(string[] astrArgs)
    {
        if (astrArgs.Length == 0}
        {
            Console.WriteLine("Syntax: WildCardHexDump file1 file2 ...");
            return 1;
        }
        foreach (string str in astrArgs)
            ExpandWildCard(str);

        return 0;
    >
    static void ExpandWildCard(string strWildCard)
    {
```

см. след. стр.

```
        string[] astrFiles;

        try
        {
            astrFiles = Directory.GetFiles(strWildCard);
        }
        catch
        {
            try
            {
                string strDir  = Path.GetDirectoryName(strWildCard);
                string strFile = Path.GetFileName(strWildCard);

                if (strDir == null  fj  strDir.Length == 0)
                    strDir -  ".";

                astrFiles = Directory.GetFiles(strDir, strFile);
            }
            catch
            {
                Console.WriteLine(strWildCard + ": No Files found!");
                return;
            }
        }
        if (astrFiles.Length == 0)
            Console.WriteLine(strWildCard + ": No files found!");

        foreach(string strFile in astrFiles)
            DumpFile(strFile);
    }
}
```

Кроме обычных групповых символов, мне бы хотелось иметь возможность указывать в качестве аргумента просто имя каталога. То есть я хотел, чтобы:

```
WildCardHexDump c:\
```

было эквивалентно:

```
WildCardHexDump c:\*.*
```

Для начала метод *ExpandWildCard* пробует получить список всех файлов в каталоге, указанном в командной строке:

```
astrFiles = Directory.GetFiles(strWildCard);
```

Этот вызов выполнится, если *strWildCard* содержит только имя каталога (например, «c:\»). В противном случае будет сгенерировано исключение. Вот почему я заключил его в блок *try*. Б блоке *catch* предполагается, что аргумент командной строки представляет собой сочетание пути и имени файла и разделяет эти компоненты, используя статические методы *GetDirectoryName* и *GetFileName* класса *Path*. Однако метод *GetFiles* класса *Directory* не принимает *null* или пустую строку к ак

первый аргумент. Эта проблема решается установлением имени каталога в «.» (означает текущий каталог) перед вызовом *GetFiles*.

Получение информации о файлах и работа с ними

Так же. как классы *Directory* и *DirectoryInfo*, классы *File* и *FileInfo* очень похожи и содержат множество методов и свойств-«близнецов». Как и в классе *Directory*, все методы класса *File* статические, и первым аргументом любого метода является имя файла. Класс *FileInfo* наследует *FileSystemInfo*. Объект типа *FileInfo* создается на основе имени файла, которое может включать полный или относительный путь.

Конструктор *FileInfo*

```
FileInfo(string strFileName)
```

Файл не обязательно должен существовать. Вы можете определить, существует ли файл, и получить еще некоторую информацию о нем с помощью следующих неизменяемых СВОЙСТВ:

Свойства *FileInfo* (выборочно)

Тип	Свойство	Доступ
bool	*Exists*	Чтение
string	*Name*	Чтение
string	*FullName*	Чтение
string	*Extension*	Чтение
string	*DirectoryName*	Чтение
DirectoryInfo	*Directory*	Чтение
long	*Length*	Чтение

Только одно из этих свойств имеет «близнеца» в классе *File*:

Методы *File* (выборочно)

```
bool Exists(string strFileName)
```

FileInfo содержит еще четыре свойства, предоставляющие доступ к атрибутам файла и времени создания, последнего доступа и последней записи:

Свойства *FileInfo* (выборочно)

Тип	Свойство	Доступ
FileAttributes	*Attributes*	Чтение/запись
DateTime	*CreationTime*	Чтение/запись
DateTime	*LastAccessTime*	Чтение/запись
DateTime	*LastWriteTime*	Чтение/запись

Все эти свойства, унаследованные от *FileSystemInfo*, имеют «близнецов» в виде статических методов класса *File*:

Статические методы *File* (выборочно)

```
FileAttributes GetAttributes(string strFileName)
DateTime GetCreationTime(string strFileName)
DateTime GetLastAccessTime(string strFileName)
DateTime GetLastWriteTime(string strFileName)
void SetAttributes(string strFileName, FileAttributes fa)
void SetCreationTime(string strFileName, DateTime dt)
void SetLastAccessTime(string strFileName, DateTime dt)
void SetLastWriteTime(string strFileName, DateTime dt)
```

Следующие методы позволяют копировать, перемещать и удалять файлы. Сюда же я включил метод *Refresh*, который обновляет свойства объекта после изменения файла:

Методы *FileInfo* (выборочно)

```
FileInfo CopyTo(string strFileName)
FiieInfo CopyTo(string strFileName, bool bOverwrite)
void MoveTo(string strFileName)
void Delete()
void Refresh()
```

Операции копирования, перемещения и удаления можно выполнять и с помощью класса *File*:

Статические методы *File* (выборочно)

```
void Copy(string strFileNameSrc, string strFileNameDst)
void Copy(string strFileNameSrc, string strFileNameDst, bool bOverwrite)
void Move(string strFileNameSrc, string strFileNameDst)
void Delete(string strFileName)
```

И наконец, классы *File* и *FileInfo* содержат несколько методов для открытия файлов:

Методы *FileInfo* (выборочно)

```
FileStream Create()
FileStream Open(FileMode fm)
FileStream Open(FileMode fm, FileAccess fa)
FileStream Open(FileMode fm, FileAccess fa, FileShare fs)
FileStream OpenReadC)
FileStream OpenWrite()
StreamReader OpenText()
StreamWriter CreateText()
StreamWriter AppendText()
```

Эти методы могут оказаться полезными, например, если вы получили набор объектов *FileInfo* (скажем, от метода *GetFiles*, вызванного для объекта *DirectoryInfo*) и хотите посмотреть, что интересного в каждом из них,

Ну и, конечно, вы можете использовать соответствующие статические методы, реализованные в классе *File:*

Статические методы *File* (выборочно)

```
FileStream Create(string strFileName)
FileStream Open(string strFileName, FileMode fm)
FileStream Open(string strFileName, FileMode fm, FileAccess fa)
FileStream Open(string strFileName, FileMode fm, FileAccess fa, FileShare fs)
FileStream OpenRead(string strFileName)
FileStream OpenWrite(string strFileName)
StreamReader OpenText(string strFileName)
StreamWriter CreateText(string strFileName)
StreamWriter AppendText(string strFileName)
```

Однако эти методы не предоставляют серьезных преимуществ по сравнению с соответствующими конструкторами классов *FileStream, StreamReader* или *StreamWriter.* Более того, их наличие в классе *File* было для меня одним из самых сбивающих с толку моментов, связанных с пространством имен *System.IO.* Нет никакого смысла использовать класс типа *File,* чтобы просто получить объект типа *FileStream.* Гораздо проще использовать единственный класс, если, конечно, этого достаточно для ваших целей,

ПРИЛОЖЕНИЕ

Б



Математические функции

Работа с числами — одна из основных задач программирования. Для ветеранов языка C некоторые возможности Microsoft .NET Framework и C# могут оказаться новыми. В этом приложении я рассмотрю их и, кроме того, важнейший класс *Math*, содержащий методы, эквивалентные функциям из заголовочного файла языка C *Math.h*.

Числовые типы данных

Язык C# поддерживает 11 числовых типов данных, разбитых на три категории: целочисленные, с плавающей точкой и десятичные:

Числовые типы данных C#

	Целые числа			
Разрядность	**Со знаком**	**Без знака**	**С плавающей точкой**	**Десятичный**
8	*sbyte*	***byte***		
16	*short*	*ushort*		
32	*int*	*uint*	***float***	
64	*long*	*ulong*	***double***	
128				***decimal***

В программе на C# целочисленный литерал (т. е. число без десятичной точки) считается принадлежащим типу *int,* если только оно не больше максимального значения для типа *int,* если же число больше максимального для *int* значения, для определения типа число анализируется. Число рассматривается как *uint, long* или *ulong* (именно в таком порядке) в зависимости от его значения. Литерал с десятичной точкой (или с экспонентой, обозначаемой *E* или *e* с последующим числом) предполагается принадлежащим типу *double.* Следующие суффиксы можно использовать совместно с литералами для обозначения их типов.

Суффиксы, используемые совместно с числовыми литералами

Тип	Суффикс
uint	u или *U*
long	*l* или *L*
ulong	*ul, uL, Ul, UL, lu, lU, Lu* или *LU*
float	*f* или *F*
double	*d* или *D*
decimal	*m* или *M*

Названия типов в C# представляют собой псевдонимы для структур, определенных в классе *System* .NET Framework. Все эти структуры являются производными от *ValueType*, который в свою очередь порожден *Object:*

Числовые типы данных .NET

	Целые числа			
Разрядность	**Со знаком**	**Без знака**	**С плавающей точкой**	**Десятичные**
8	*Sbyte*	***Byte***		
16	*Int16*	*UInt16*		
32	*Int32*	*UInt32*	***Single***	
64	*Int64*	*UInt64*	*Double*	
128				***Decimal***

Типы данных *SByte, UInt16, UInt32* и *UInt64* не определены в спецификации CLS (Common Language Specification). Это значит, что язык программирования может быть совместимым с CLS и при этом не поддерживать эти типы данных. Если ваш код (в виде DLL) должен поддерживаться любым языком, совместимым с CLS, не используйте как байты со знаком, так и 16-, 32- и 64-разрядные целые числа без знака.

Целочисленное переполнение

Взгляните на такой фрагмент:

```
short s = 32767;
s += 1;
```

или на такой:

```
ushort us = 0;
us -= 1;
```

Эти примеры демонстрируют переполнение и потерю значимости целых чисел, но при этом оба фрагмента абсолютно корректны в языках C и C# (по крайней мере при параметрах по умолчанию).

В первом случае целое со знаком превышает свое максимальное значение. В связи с особенностями хранения целых чисел в памяти результатом будет -32 768. Во втором случае целое число без знака становится меньше 0, и результатом будет число 65 535.

Некоторые программисты умеют пользоваться переполнением и потерей значимости целых чисел и извлекают из этого выгоду, другие становятся жертвами

ошибок. В зависимости от настройки C# может рассматривать переполнение целых чисел либо как изящный «обходной маневр», либо как досадную ошибку.

Чтобы во время работы программы производилась проверка на переполнение, применяйте ключ компилятора:

```
/checked+
```

Поведение по умолчанию обеспечивается применением ключа компилятора:

```
/checked-
```

В Visual C# .NET этот ключ компилятора можно установить в диалоговом окне Property Pages проекта. В левой части окна выберите Build в списке Configuration Properties, в правой — установите Check For Arithmetic Overflow/Underflow в True.

Если включить проверку переполнения при выполнении программы, операции увеличения или уменьшения значения переменных вроде приведенных выше будут генерировать исключение *OverflowException*.

Внутри C#-программы можно переопределить параметры компилятора, используя ключевые слова *checked* и *unchecked*. За ключевым словом должно следовать либо выражение в скобках, либо оператор или группа операторов в фигурных скобках. К примеру, при исполнении кода:

```
short s = 32767;
checked
{
    s += 1;
}
```

возникнет исключение независимо от ключей компилятора. Скорее всего вам придется заключать блоки *checked* в блоки *try*.

До этого момента я имел в виду только проверку переполнения *времени выполнения*. По умолчанию переполнения, известные на этапе компиляции, рассматриваются компилятором как ошибка независимо от ключей. Например, оператор:

```
short s = 32767 + 1;
```

будет вызывать ошибку компиляции, так как результат сложения вычисляется на этапе компиляции. Но можно использовать ключевое слово *unchecked* для отключения проверок переполнения времени компиляции. Например, две целых константы определены таю

```
const int i1 = 65536;
const int i2 = 65536;
```

Выражение:

```
int i3 = i1 * i2;
```

в обычных условиях вызовет ошибку компиляции. Так как *i1* и *i2* являются константами, компилятор пытается вычислить значение выражения и генерирует ошибку. Ключ компилятора не изменит поведения компилятора, зато его изменит ключевое слово *unchecked:*

```
int i3 = unchecked (i1 * i2);
```

Это выражение будет скомпилировано и выполнено, не генерируя исключения

Вообще следует писать программы так, чтобы они корректно компилировались и запускались с любыми ключами компилятора. Всегда, когда есть вероятность возникновения переполнения, которое вы хотели бы контролировать, заключайте выражение в блок *checked* внутри блока *try*. Если вас не волнуют возможные ошибки переполнения или вы хотите их использовать, заключайте операторы в блок *unchecked*.

Деление на 0 всегда генерирует исключение *DivideByZeroException* независимо от ключевых слов *checked* и *unchecked* или ключей компилятора.

Десятичный тип данных

Абсолютно нов для С-программиста тип данных C# *decimal,* требующий для хранения данных 16 байтов (128 бит). 128 бит разбиты на целую часть длиной 96 бит, 1 бит для знака и масштабирующий множитель, который может принимать значение от 0 до 28. Математически масштабирующий множитель является отрицательной степенью 10 и указывает количество десятичных разрядов,

Не путайте тип *decimal* с *двоично-десятичным* (binary-coded decimal, BCD). В типе BCD каждая десятичная цифра для хранения требует 4 бита. В типе *decimal* числа хранятся в виде двоичного целого числа.

Например, число типа *decimal,* равное 12,34, будет храниться как целое число Ox4D2 (1234) и множитель 2, То же число в кодировке BCD будет представлено как 0x1234.

Если десятичное число содержит 28 или меньше значащих цифр и 28 или меньше десятичных разрядов, то данные в формате *decimal* будут точно соответствовать этому числу. Для чисел с плавающей точкой это не так! При определении числа типа *float,* равного 12,34, оно будет храниться в памяти как OxC570A4 (12 939 428), деленное на 0x100000 (1 048 576), что только *приблизительно* соответствует 12,34. Даже имея тип *double,* оно будет сохранено как Ox18AE147AE147AE (б 946 802 425 218 990), деленное на 0x2000000000000 (562 949 953 421 312), что опять-таки только приблизительно соответствует 12,34.

Вот почему в задачах, где непрогнозируемые изменения младших разрядов были бы нежелательны, следует использовать тип *decimal.* Словом, тип данных с плавающей точкой отлично подходит для научных или инженерных приложений, но часто не годится для финансовых,

Чтобы изучить внутреннюю структуру типа *decimal,* рассмотрим конструктор:

Конструкторы *Decimal* (выборочно)

```
Decimal(int iLow, int iMiddle, int iHigh, bool bNegative, byte byScale)
```

Хотя первые три аргумента имеют тип *int,* они рассматриваются как целые числа без знака, и из них формируется 96-битное целое число без знака. Аргумент *byScale,* значение которого может изменяться от 0 до 28, является количеством цифр после запятой. Например, выражение:

```
new Decimal(123456789, O, O, false, 5)
```

создает число типа *decimal:*

```
1234.56789
```

Наибольшее положительно число типа *decimal*:

```
new Decimal(-1, -1, -1, false, 0)
```

или:

```
79 228 162 514 264 337 593 543 950 335
```

Оно же равно значению поля *MaxValue* структуры *Decimal*.

```
Decimal.MaxValue
```

Наименьшее положительное число типа *decimal*.

```
new Decimal(1, 0, 0, false, 28)
```

и оно равно:

```
0.0000000000000000000000000001
```

ИЛИ:

1×10^{-28}

Это число, будучи деленным на 2 в C#-программе, даст в результате 0.

Можно получить биты, использованные для хранения числа типа *decimal*,

Статические методы *Decimal* (выборочно)

```
tnt[] GetBits(decimal mValue)
```

Этот метод возвращает массив из четырех целых чисел. Первый, второй и третий элемент массива являются соответственно младшей, средней и старшей компонентой 96-битного беззнакового целого числа. Четвертый элемент содержит знак и множитель: биты с 0 по 15 установлены в 0. с 16 по 23 — содержат множитель в диапазоне от 0 до 28, с 24 по 30 установлены в 0, бит 31 равен 0, если число положительное, и 1 — если отрицательное.

Допустим, переменная *mValue* имеет тип *decimal*, и выполняется оператор:

```
int[] ai = Decimal.GetBits(mValue);
```

Если *ai[3]* отрицательно, *mValue* отрицательна. Множитель равен:

```
Cai[3] >> 16) & 0xFF
```

Я уже говорил, что представление в виде чисел с плавающей точкой часто только приблизительно соответствует представляемому значению. При выполнении арифметических операций над числами с плавающей точкой точность может уменьшиться. Почти каждый, кто использовал числа с плавающей точкой, сталкивался с ситуацией, когда число, которое должно было быть равно 4,55 (например), хранилось как 4,549999 или 4,550001.

Представление в виде *decimal* гораздо более предсказуемо. К примеру, *ml* определена так:

```
decimal m1 = 12.34;
```

Внутреннее представление *ml* состоит из целой части 1234 и множителя 2. Кроме того, допустим, что *m2* определена так:

```
decimal m2 = 56.789;
```

Целая часть равна 56789, множитель — 3. Сложим эти два числа:

```
decimal m3 = m1 + m2;
```

Внутренние механизмы сложения выглядят так: целая часть *ml* умножается на 10 (в результате получается 12 340), а множитель устанавливается в 3. Теперь целые части можно складывать напрямую: 12 340 + 56 789 = 69 129, причем множитель равен 3. Результат — 69,129. Точно, как в аптеке.

Теперь перемножим два числа:

```
decimal m4 = m1 * m2;
```

Внутренний механизм таков: целые части перемножаются (1 234 * 56 789 = 70 077 626), а множители складываются (2 + 3 = 5). Результат — 700,77626. И снова результат точный.

Что касается деления... вообще-то с делением связаны определенные трудности независимо от того, как вы его выполняете. Но в целом при использовании *decimal* вы можете: гораздо лучше контролировать точность результатов,

Бесконечность с плавающей точкой и NaN

Два типа данных с плавающей точкой — *float* и *double* — определены в соответствии со стандартом ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*

Число типа *float* состоит из 24-битной мантиссы со знаком и 8-битной экспоненты со знаком. Точность составляет приблизительно семь десятичных цифр. Значения находятся в диапазоне от:

-3.402823 X 10^38

ДО:

3.402823×10^{38}

Минимальное положительное число *muna float* равно:

1.401298×10^{-45}

Эти три значения содержатся в полях структуры *Single*:

Поля структуры *Single* (выборочно)

Тип	Поле
float	*MinValue*
float	*MaxValue*
float	*Epsilon*

Число типа *double* состоит ил 53-битной мантиссы со знаком и 11-битной экспоненты со знаком. Точность составляет примерно 15-16 десятичных цифр. Значения попадают в диапазон от:

-1.79769313486232 X 10^{308}

ДО:

1.79769313486232 X 10^{308}

Наименьшее положительное число **типа** *double* равно:

$4.94065645841247 \times 10^{-324}$

Поля *MinValue*, *MaxValue* и *Epsilon* определены также в структуре *Double*.

Вот пример кода, выполняющего деление числа с плавающей точкой на 0:

```
float f1 = 1;
float f2 = 0;
float f3 = f1 / f2;
```

Если б эти числа были целыми, было бы сгенерировано исключение *DivideByZero-Exception*. Но это числа с плавающей точкой стандарта IEEE. Исключение *не* генерируется. Более того, операции над числами с плавающей точкой *никогда* не генерируют исключений. Вместо этого в нашем случае в *f3* будет записано специальное значение. Если вы вызовете *Console.WriteLine* и выведете *f3*, будет выведено слово «бесконечность»:

```
Infinity
```

Если же вы измените значение *f1* на -1, *Console.WriteLine* выведет:

```
-Infinity
```

В стандарте IEEE положительная и отрицательная бесконечность — нормальные значения чисел с плавающей точкой. Над бесконечностями можно даже выполнять арифметические операции. Например, выражение:

```
1 / f3
```

равно 0.

Если в приведенном выше коде изменить начальное значение *f1* на 0, то *f3* будет равно специальному значению, известному как *NaN* (не является числом, *Not a Number)*. Вот как *Console .WriteLine* выводит NaN:

```
NaN
```

Можно получить NaN, прибавив положительную бесконечность к отрицательной и еще множеством других вычислений.

Структуры *Single* и *Double* содержат статические методы, позволяющие определить, является ли число типа *float* или *double* бесконечностью или NaN. Методы структуры *Single* таковы:

Статические методы структуры *Single* (выборочно)

```
bool IsInfinity(float fValue)
bool IsPositiveInfinity(float fValue)
bool IsNegativeInfinity(float fValue)
bool IsNaN(float fValue)
```

Например, выражение:

```
Single.IsInfinity(fVal)
```

возвращает *true,* если *fVal* — положительная или отрицательная бесконечность

Кроме того, структура *Single* содержит поля, представляющие такие значения:

Константные поля структуры *Single* (выборочно)

Тип	Поле
float	*PositiveInfinity*
float	*NegativeInfinity*
float	*NaN*

Такие же поля определены и в структуре *Double*. Они соответствуют конкретным битовым шаблонам, определенным в стандарте IEEE,

Класс *Math*

Класс *Math*, определенный в пространстве имен *System*, содержит исключительно статические методы и два константных поля:

Константные поля *Math*

Тип	Поле	Значение
double	*PI*	3.14159265358979
double	*E*	2.71828182845905

Разумеется, *Math.PI* представляет собой отношение длины окружности к ее диаметру, а *Math.E* является пределом:

$$\left(1+\frac{1}{n}\right)^n$$

при *n*, стремящемся к бесконечности.

Большинство методов класса *Math* определены только для значений типа *double*. Однако некоторые определены для целых чисел и чисел типа *decimal*. Следующие два метода определены для всех числовых типов данных;

Статические методы *Math* (выборочно)

```
type Max(numeric-type n1, numeric-type n2)
type Min(numeric-type n1, numeric-type n2)
```

Два значения, переданные методу, должны быть одного типа.

Следующие два метода определены для чисел типа *float*, *double*, *decimal* и всех целочисленных типов данных со знаком:

Статические методы *Math* (выборочно)

```
int Sign(signed-type s)
type Abs(signed-type s)
```

Метод *Sign* возвращает 1, если аргумент положительный, -1 если отрицательный и О — если он равен 0. Метод *Abs* возвращает аргумент, если аргумент положителен или равен 0, и аргумент с противоположным знаком, если аргумент отрицателен.

Метод *Abs*, единственный в *классе Math*, способный генерировать исключения, причем только для целых аргументов и только для конкретных значений целых

аргументов. Если аргумент равен значению *MinValue* для этого типа данных (например, -32 768 для типа данных *short*), то генерируется исключение *OverflowException* так как 32 678 не может быть представлено переменной типа *short*.

Следующие методы выполняют разные варианты округлений чисел типа *double* и *decimal*.

Статические методы *Math* **(выборочно)**

```
double Floor(double dValue)
double Ceiling(double dValue)
double Round(double dValue)
double Round(double dValue, int iDecimals)
decimal Round(decimal mValue)
decimal Round(decimal mValue, int iDecimals)
```

Floor возвращает наибольшее целое число, меньшее или равное аргументу, *Ceiling* возвращает наименьшее целое число, большее или равное аргументу. Вызов:

```
Math.Floor(3.5)
```

возвращает 3, и:

```
Math.Ceiling(3.5)
```

возвращает 4. Те же правила применяются к отрицательным числам. Вызов:

```
Math.Floor(-3.5)
```

возвращает -4. и:

```
Math.Ceiling(-3.5)
```

возвращает -3.

Метод *Floor* возвращает ближайшее целое число в сторону отрицательной бесконечности, и поэтому эту операцию иногда называют *округлением в сторону отрицательной бесконечности*. *Ceiling* возвращает ближайшее целое число в сторону положительной бесконечности, и эту операцию иногда называют *округлением в сторону положительной бесконечности*. Кроме того, можно округлять в сторону 0, в результате чего получается ближайшее к округляемому числу и 0 целое число. Округление в сторону 0 выполняется приведением числа к целому типу. Выражение:

```
(int) 3.5
```

возвращает 3, и:

```
(int) -3.5
```

возвращает -3. Округление в сторону 0 еще называют *усечением*.

Метод *Round* с одним аргументом возвращает ближайшее к аргументу целое число. Если параметр *Round* находится посредине между двумя целыми числам; т. он округляется до ближайшего четного числа. К примеру, вызов:

```
Math.Round(4.5)
```

возвращает 4, и:

```
Math.Round(5.5)
```

возвращает 6. При желании вы можете задать целое число, указывающее на количество цифр после запятой в возвращаемом значении. Например:

```
Math.Round(5.285, 2)
```

возвращает 5,28.

Остаток с плавающей точкой

С функциями, вычисляющими математические остатки с плавающей точкой, связано много путаницы, В C# операция взятия остатка, еще называемая операцией деления по модулю (%), определена для всех числовых типов данных. (В языке С оператор деления по модулю не определен для типов данных *float* и *double,* вместо этого надо вызвать функцию *fmod.*) Пример выражения C#, в котором над двумя переменными типа *float* выполняется операция деления по модулю:

```
fResult = fDividend X fDivisor;
```

Знак *fResult* совпадает со знаком */Dividend,* причем *fResult* может быть рассчитан по формуле:

```
fResult = fDividend - n * fDivisor
```

где *n* — наибольшее целое число, меньшее или равное *fDividend/ fDivisor.* Например:

```
4.5 X 1.25
```

равно 0,75. Давайте посчитаем... 4,5 / 1,25 = 3,6, следовательно, *n* равно 3. 4,5 - (3 * 1,25) = 0,75.

Стандарт ШЕЕ определяет остаток иначе: там *n* — *ближайшее к fDividena/ fDivisor* целое число. Сосчитать остаток, соответствующий стандарту IEEE, позволяет метод:

Статические методы *Math* (выборочно)

```
double IEEERemainder(double dDividend, double dDivisor)
```

Выражение:

```
Math.IEEERemainder(4.5, 1.25)
```

возвращает -0.5. Так получатся в силу того, что 4,5 / 1,25 = 3,6, а ближайшее к 3,6 целое число равно 4. Когда *n* равно 4, получается 4,5 - (4 * 1,25) = -0,5.

Степени и логарифмы

Три метода класса *Math* связаны с возведением в степень:

Статические методы *Math* (выборочно)

```
double Pow(double dBase, double dPower)
double Exp(double dPower)
double Sqrt(double dValue)
```

Pow вычисляет:

$dBase^{dPower}$

Выражение:

```
Math.Exp(dPower)
```

эквивалентно:

```
Math.Pow(Math.E, dPower)
```

А функция извлечения квадратного корня:

```
Math.Sqrt(dValue)
```

эквивалентна:

```
Math.Pow(dValue, 0.5)
```

Метод *Sqrt* возвращает NaN, если его параметр отрицателен.

Класс *Math* содержит три метода для вычисления логарифмов:

Статические методы *Math* (выборочно)

```
double Log10(double dValue)
double Log(double dValue)
double Log(double dValue, double dBase)
```

Выражение:

```
Math.Log10(dValue)
```

эквивалентно:

```
Math.Log(dValue, 10)
```

и:

```
Math.Log(dValue)
```

эквивалентно:

```
Math.Log(dValue, Math.E)
```

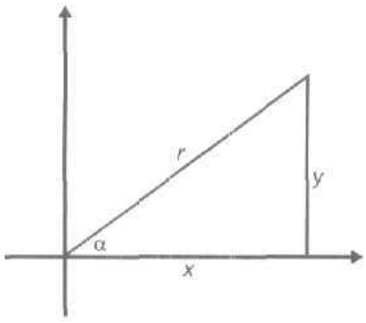
Методы вычисления логарифмов возвращают *PositiveInfinity*, если параметр равен О, и *NaN,* если параметр меньше 0.

Тригонометрические функции

Тригонометрические функции описывают соотношения между сторонами и углами треугольников. Тригонометрические функции определены для прямоугольных треугольников:

Для утла а в прямоугольном треугольнике, где x — прилежащий катет, y — противолежащий, а r — гипотенуза, основные тригонометрические соотношения выглядяттак:

$$\sin(\alpha) = y / r$$
$$\cos(a) = x / r$$
$$\tan(\alpha) = y / x$$

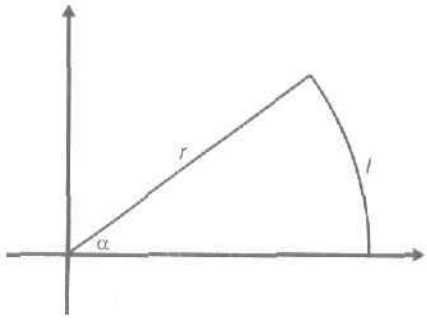


Тригонометрические функции можно использовать для определения окружностей и эллипсов. Для постоянного r и α, изменяющегося от 0° до 360°, набор координат (x, y), где:

```
x = r • sin(α)
y = r • cos(α)
```

определяет окружность с центром в начале координат и радиусом r. В главе 5 показано, как использовать тригонометрические функции для рисования кругов и эллипсов. Кроме того, тригонометрические функции используются в ряде графических упражнений из глав 13, 15, 17 и 19.

Тригонометрические функции из класса *Math* требуют углов, заданных в радианах, а не в градусах. 360° равны 2π радианам, Причину использования радианов демонстрирует рисунок, на котором угол а стягивает дугу /:



Какова длина дуги *l*? Так как длина всей окружности равна $2\pi r$, длина дуги l равна $(\alpha/360)2\pi r$, где а измеряется в градусах. Однако, если а измеряется в радианах, длина дуги l просто равна аг. Для единичной окружности (чей радиус равен 1) длина дуги / равна углу а в радианах. Определение радиана: угол, стягивающий в единичной окружности дугу длиной /, в радианах равен l.

Например, угол в 90° в единичной окружности стягивает дугу длиной $\pi/2$. Таким образом, 90° равны Л/2 радиан. Угол в 180° равен углу в π радиан. 360" соответствуют 2π радиан.

В классе *Math* определены три базовые тригонометрические функции:

Статические методы *Math* (выборочно)

```
double Sin(double dAngle)
double Cos(double dAngle)
double Tan(double dAngle)
```

Если угол задан в градусах, для перевода его в радианы умножьте значение в градусах на *K* и разделите на 180;

```
Math.Sin(Math.PI * dAngleInDegrees / 180)
```

Методы *Sin* и *Cos* возвращают значения в диапазоне от -1 до 1. Теоретически метод *Tan* должен возвращать бесконечность для аргументов Я/2 (90°) и $3\pi/2$ (270°), но вместо этого он просто возвращает очень большие числа.

Следующие методы являются обратными тригонометрическими функциями. Они возвращают угол в радианах;

Статические методы *Math* (выборочно)

Метод	Аргумент	Возвращаемое значение
`double Asin(double dValue)`	От –1 до 1	От -ТС/2 до $\pi/2$
`double Acos(double dValue)`	От -1 до 1	От Л до 0
`double Atan(double dValue)`	от $-\infty$ до со	От -ТС/2 до ТС/2
`double Atan2(double y, double x)`	От $-\infty$ до ∞	От -ТС до Я

Чтобы перевести возвращаемое значение в градусы, его надо умножить на 180 и разделить на Я.

Методы *Asm* и *Acos* возвращают NaN, если аргумент не попадает в разрешенный диапазон. Метод *Atan2* анализирует знаки обоих аргументов и определяет четверть, в которой находится угол:

Значения, возвращаемые *Atan2*

Аргумент *y*	Аргумент *x*	Возвращаемоезначение
Положительный	Положительный	От 0 до Я/2
Положительный	Отрицательный	От Я/2 до ТС
Отрицательный	Отрицательный	От Я до $3\pi/2$
Отрицательный	Положительный	От 3Я/2 до 2Я

Реже используются гиперболические тригонометрические функции. Тогда как обычные тригонометрические функции определяют окружности и эллипсы, гиперболические тригонометрические функции определяют гиперболы:

Статические методы *Math* (выборочно)

```
double Sinh(double dAngle)
double Cosh(double dAngle)
double Tanh(double dAngle)
```

Угол выражается в гиперболических радианах.



ПРИЛОЖЕНИЕ

В

Работа со строками

Практически в каждом языке своя концепция представления текстовых строк, К счастью (или к сожалению), в отличие от чисел с плавающей точкой на представления и обработку строк стандарта нет. В языке С даже отсутствует специальный тип данных для строк; строки — это просто массивы символов, заканчивающиеся нулевым байтом. Для управления ими используют указатель на первый символ массива. С-программисты очень ценят легкость, с которой строками можно манипулировать в памяти. Что они не очень ценят, так это то, что такая легкость очень просто может стать причиной серьезной ошибки.

В C# текстовые строки имеют собственный тип — *string,* являющийся псевдонимом для класса *System.String.*Тип данных *string,* конечно же, связан с типом данных *char:* объект *string* можно создать из массива символов или превратить в массив символов. Но тип данных *string* и массив *char* — это разные типы данных,

В C# строки не заканчиваются 0. Каждая строка имеет длину, и длина созданной строки изменяться не может, Кроме того, не могут меняться символы, составляющие строю/. Вот почему строки в C# называют *постоянными* (immutable). Если вам нужно изменить строку, вам придется создать новую строку. Множество методов класса *String* создает новую строку на основе существующей. Методы и свойства, поддерживаемые .NET Framework, также создают и возвращают строки.

Вот пример распространенной ловушки: вы можете предположить, что существует метод класса *String ToUpper,*переводящий все символы строки в верхний регистр, и это на самом деле так. Но вы не можете просто вызвать его для экземпляра *str* класса *string:*

```
str.ToUpper();   // Ничего не произойдет!
```

Синтаксически это выражение верно, но переменную *str* не изменяет. Строки постоянны, и символы, составляющие *str,* не могут измениться. Метод *ToUpper* создает новую строку. Нужно присвоить значение, возвращенное *ToUpper* другой строковой переменной:

```
string strUpper = str.ToUpper();
```

Или можно присвоить возвращаемое значение той же переменной:

```
str = str.ToUpper();
```

Тогда исходная строка (содержащая буквы в нижнем регистре) останется, но, так как далее нигде в программе на нее не будет ссылок, она станет жертвой процесса сборки мусора.

Еще пример. Пусть строка определена так:

```
string str = "abcdifg";
```

Вы можете обратиться к отдельному символу строки через индекс строковой переменной:

```
char ch = str[4];
```

В этом случае *ch* содержит символ T. Однако вы не можете устанавливать конкретный символ строки:

```
str[4] = 'e';  // Не выполнится!
```

Свойство индекса класса *String* предназначено только для чтения.

Но как-то *можно* заменить символы в строках C#? Можно. Вызов метода:

```
str = str.Replace('i', 'e');
```

заменит *все* вхождения 'i' на 'e'. С другой стороны, вы можете сначала вызвать метод *Remove* и создать новую строку, из которой удалена подстрока определенной длины. Например, вызов метода:

```
str = str.Remove(4, 1);
```

удалит один символ на четвертой позиции ('i'). А затем вы можете вызвать метод *Insert,* чтобы вставить новую строку, в нашем случае представляющую один символ:

```
str = str.Insert(4, "e");
```

Ту же операцию можно выполнить в одно выражение:

```
str = str.Remove(4, 1).Insert(4, "e");
```

Хотя используется только одна строковая переменная *str,* оба вызова методов в последнем выражении создают новые строки, причем «e» — это еще одна строка.

Существует другой подход. Строку можно превратить в массив символов, установить соответствующие элементы этого массива и создать новую строку на основе массива символов:

```
char[] ach = str.ToCharArray();
ach[4] = 'e';
str = new String(ach);
```

А можно соединить подстроки:

```
str = str.Substring(0, 4) + "e" + str.Substring(5);
```

Далее в этом приложении я детально рассмотрю методы класса *String.*

Тип *char*

Каждый элемент строки имеет тип *char,* являющийся псевдонимом структуры *System.Char* .NET. Одиночный символ-литерал указывается в апострофах:

```
char ch = 'A';
```

Хотя *Char* порожден *ValueType,* переменную типа *char* нельзя напрямую использовать как число. Преобразование переменной *ch* типа *char, к* примеру, в целое число требует приведения типов:

```
int i = (int) ch;
```

Символьные переменные имеют числовые значения от 0x0000 до 0xFFFF и связаны с символами из набора символов Unicode. Хороший справочник по Unicode — книга «The Unicode Standard Version 3.0» (Addison-Wesley, 2000).

Так же как и в C, обратная косая черта (\) является специальным *символом перехода.* Следующий за ней символ интерпретируется особым образом:

Управляющие символы C#

Символ	Чему соответствует	Код
\0	Null	0x0000
\a	Звуковой сигнал	0x0007
\b	Забой	0x0008
\t	Табуляция	0x0009
\n	Новая строка	0x000A
\v	Вертикальная табуляция	0x000B
\f	Подача страницы	0x000C
\r	Возврат каретки	0x000D
\"	Двойные кавычки	0x0022
\'	Апостроф	0x0027
\\	Обратный слеш	0x005C

Кроме того, одиночный символ Unicode можно указать, поставив перед четырехзначным шестнадцатеричным кодом префикс *\x* или *\u.* Например, символы '\x03A9' и '\u03A9' оба представляют собой заглавную греческую букву омега (Ω).

Как вы знаете, узнать в C, чем является символ: буквой, цифрой, управляющим символом или чем-то еще, — позволяют функции, определенные в заголовочном файле ctype.h. В C# можно использовать статические методы, определенные в структуре *Char.* Аргументом этих методов может быть или символ, или строка с индексом,

Статические методы *Char* (выборочно)

```
bool IsControl(char ch)
bool IsControl(string str, int iIndex)
bool IsSeparator(char ch)
bool IsSeparator(string str, int iIndex)
bool IsWhiteSpace(char ch)
bool IsWhiteSpace(string str, int iIndex)
bool IsPunctuation(char ch)
```

```
bool IsPunctuation(string str, int iIndex)
bool IsSymbol(char ch)
bool IsSymbol(string str, int iIndex)
bool IsDigit(char ch)
bool IsDigit(string str, int iIndex)
bool IsNumber(char ch)
bool IsNumber(string str, int iIndex)
bool IsLetter(char ch)
bool IsLetter(string str, int iIndex)
bool IsUpper(char ch)
bool IsUpper(string str, int iIndex)
bool IsLower(char ch)
bool IsLower(string str, int iIndex)
bool IsLetterOrDigit(char ch)
bool IsLetterOrDigit(string str, int iIndex)
bool IsSurrogate(char ch)
bool IsSurrogate(string str, int iIndex)
```

Вызов:

```
Char.IsControl(str[iIndex])
```

эквивалентен:

```
Char.IsControl(str, iIndex)
```

Можно обойтись без этих методов для символов ASCII (символов с кодами от 0x0000 до Ox007F), однако эти методы применимы ко всем символам Unicode. Метод *IsSurrogate* связан с диапазоном символов Unicode с кодами от 0xD800 до 0xDFFF, зарезервированным для последующего расширения,

В структуре *Char* определена еще пара полезных методов: один возвращает член перечисления *UnicodeCategory* (определенного в *System.Globalization*), другой — числовое значение символа, преобразованное в *double*:

Статические методы *Char* (выборочно)

```
UnicodeCategory GetUnicodeCategory(char ch)
UnicodeCategory GetUnicodeCategory(string str, int ilndex)
double GetNumericValue(char ch)
double GetNumericValue(string str, int ilndex)
```

Конструкторы и свойства строк

Строковую переменную позволяет определить литерал:

```
string str = "Hello, world!";
```

или литерал, вставленный прямо в вызов функции:

```
Console.WriteLine("Hello, world!");
```

или как возвращаемое значение одного из методов, возвращающих строковые переменные. Один из них — распространенный метод *ToString*, преобразующий объект в строку. Например, выражение:

```
55.ToString();
```

возвращает строку «55».

Если перед строковым литералом поставить символ @, то обратная косая черта не будет рассматриваться как символ перехода. Это удобно при указании путей:

```
string str = @"c:\temp\my file";
```

Чтобы включить в такую строку двойные кавычки, их следует использовать дважды.

Менее распространен способ создания строк одним из восьми конструкторов *String*. Пять из них работают с указателями и несовместимы со спецификацией CLS (Common Language Specification). Три остальных создают объект *String* либо повторением одиночного символа, либо преобразуя массив символов:

Конструкторы *String* (выборочно)

```
String(char ch, int iCount)
String(char[] ach)
String(char[] ach, int iStartIndex, int iCount)
```

В третьем конструкторе *iStartIndex* представляет собой индекс в массиве символов, а *iCount* определяет количество символов, начиная с указанного индекса. Длина результирующей строки совпадает с *iCount*.

Класс *String* содержит всего два неизменяемых свойства:

Свойства *String*

Тип	Свойство	Доступ
int	*Length*	Чтение
char	*[]*	Чтение

Первое возвращает количество символов в строке, второе позволяет обращаться к отдельным символам строки.

Можно определить строковую переменную, не инициализируя ее:

```
string str1;
```

Любая попытка использовать эту строку приведет к тому, что компилятор сообщит о том, что строка не инициализирована. Так как *String* — ссылочный тип, можно присваивать строковой переменной значение *null:*

```
string str2 = null;
```

которое означает, что для строки не выделяется память. Это совсем не то же самое, что создать пустую строку:

```
string str3 = "";
```

Под пустую строку память выделяется, но свойство *str3.Length* равно 0. Попытка определить длину строки, равной *null,* — например, вызовом *str2.Length* — приводит к генерации исключения.

Можно инициализировать строковую переменную пустой строкой, используя единственное доступное поле класса *String:*

Статическое поле *String*

Тип	Свойство	Доступ
string	*Empty*	Только чтение

Например:

```
string str = string.Empty;
```

Массив строк можно определить так:

```
string[] astr = new string[5];
```

Создается массив строк, каждая из которых равна *null*. Можно создать массив инициализированных строк:

```
string[] astr = { "abc", "defghi", "jkl" };
```

В этом выражении создается строковый массив из трех элементов, таким образом, *astr.Length* возвращает 3. У каждой строки своя длина, например, *astr[1].Length* возвращает 6.

Класс *String* реализует интерфейсы *IComparable*, *ICloneable*, *IConvertible* и *IEnumerable*, и значит, в классе *String* содержатся определенные в них методы. Так как класс *String* реализует интерфейс *IEnumerable*, его можно использовать в выражении *foreach*, чтобы обращаться к символам строки по одному. Выражение:

```
foreach (char ch in str)
{
    ⋮
}
```

эквивалентно (но короче, чем):

```
for (int i = 0; i < str.Length; i++)
{
    char ch = str[i];
    ⋮
}
```

Внутри блока *foreach* для *ch* разрешено только чтение. В блоке *for* для *ch* разрешены и чтение, и запись, но (как обычно) символы, составляющие строку, изменить нельзя.

Возможно, следующим после *IEnumerable* по значимости интерфейсом, реализуемым *String*, является *IComparable*, это означает, что класс *String* реализует метод *CompareTo*, позволяющий использовать массивы строк в методах *класса Array* *BinarySearch* и *Sort*. Я рассмотрю эти методы далее.

Копирование строк

Копировать строки можно по-разному. Проще всего, видимо, — использовать знак равенства:

```
string strCopy = str;
```

Как и любой класс .NET Framework, класс *String* наследует метод *ToString* от *Object*. Так как класс *String* реализует *ICloneable*, он реализует и метод *Clone*. Эти методы обеспечивают дополнительные (возможно, излишние) способы копирования строк:

Методы *String* (выборочно)

```
string ToString()
object CloneO
```

При использовании метода *Clone* результат должен быть приведен к типу *string*.

```
string strCopy = (string) str.Clone();
```

Кроме того, класс *String* реализует статический метод, копирующий строки:

Статический метод *Copy* класса *String*

```
string Copy(string str)
```

Так, *string* — псевдоним *System.String*, этот метод можно предварять *string* в нижнем регистре:

```
string strCopy = string.Copy(str);
```

или полностью определенным именем класса:

```
string strCopy = System.String.Copy(str);
```

Если в программе есть оператор *using System*, можно предварять метод именем класса *String* с прописной буквы:

```
string strCopy = String.Copy(str);
```

Два конструктора *String* преобразуют массив символов в строку. Возможно обратное преобразование строки в массив символов:

Методы *String* (выборочно)

```
char[] ToCharArray()
chart] ToCharArray(int iStartIndex, int iCount)
void CopyTo(int iStartIndexSrc, char[] achDst, int iStartIndexDst,
            int iCount)
```

Методы *ToCharArray* создают массив символов. Аргумент *iStartIndex* определяет начальный индекс в строке. Чтобы использовать метод *CopyTo*, массив *achDst* должен существовать. Первый аргумент определяет начальный индекс в строке, третий — начальный индекс в массиве. Метод *CopyTo* эквивалентен:

```
for (int i = 0; i < iCount; i++)
     achDst[iStartIndexDst + i] = str[iStartIndexSrc + i];
```

Методы *Substring* создают новую строку, представляющую собой фрагмент исходной:

Метод *Substring* класса *String*

```
string Substring(int iStartIndex)
string Substring(int iStartIndex, int iCount)
```

Первая версия возвращает подстроку, начинающуюся с указанного индекса и до конца исходной строки.

Преобразования строк

Два метода, каждый из которых существует в двух версиях, преобразуют строки в верхний или нижний регистр:

Методы *String* (выборочно)

```
string ToUpper()
string ToUpper(CultureInfo ci)
string ToLower()
string ToLower(CultureInfo ci)
```

Класс *CultureInfo* определен в *System.Globalization* и в данном случае определяет конкретный язык, используемый в конкретной стране.

Конкатенация строк

Процесс, связанный с объединением двух или более строк в одну, называется *конкатенацией*. Операция весьма распространенная. В языке C для этого используются библиотечные функции *strcat* и *strncat*. В C# оператор сложения перегружен и может осуществлять конкатенацию строк:

```
string str = str1 + str2;
```

Оператор конкатенации удобен при определении строкового литерала, слишком длинного, чтобы его можно было разместить в одной строке:

```
string str = "Those who profess to favor freedom and yet depreciate " +
             "agitation. . .want crops without plowing up the ground, they " +
             "want rain without thunder and lightning. They want the ocean " +
             "without the awful roar of its many waters. \x2014 Frederick " +
             "Douglass";
```

Можно использовать оператор +=, чтобы добавить строку в конец существующей:

```
str += "\r\n";
```

Кроме того, в классе *String* определен метод *Concat*:

Статический метод *Concat* класса *String* (выборочно)

```
string Concat(string str1, string str2)
string Concat(string str1, string str2, string str3)
string Concat(string str1, string str2, string str3, string str4)
string Concat(params string[] astr)
```

Обратите внимание на ключевое слово *params* в последней версии *Concat*. Здесь его наличие означает, что вы можете указать либо массив строк, либо любое количество строк. Например, если массив строк определить так:

```
string[] astr = { "abc", "def", "ghi", "jkl", "mno", "pqr" };
```

и передать методу *Concat:*

```
string str = string.Concat(astr);
```

результатом будет строка «abcdefghijklmnopqr». С другой стороны, можно передать методу *Concat* отдельные строки:

```
string str = string.Concat("abc", "def", "ghi", "jkl", "mno", "pqr");
```

Хотя в классе *String* определены версии *Concat* с двумя, тремя, четырьмя и с переменным числом аргументов, реально нужна только версия с ключевым словом *params*. Этот метод на самом деле включает в себя три других.

Другой набор методов *Concat* оперирует с аргументами типа *object*:

Статический метод *Concat* класса *String* (выборочно)

```
string Concat(object obj)
string Concat(object obj1, object obj2)
string Concat(object obj1, object obj2, object obj3)
string Concat(params object[] aobj)
```

Аргументы типа *object* преобразуются в строки с помощью собственного метода *ToString*. Вызов:

```
string.Concat(55, "-", 33, "=", 55 - 33)
```

возвращает строку «55-33=22».

Иногда нужно объединить массив строк, вставив между элементами какой-нибудь разделитель, Это позволяет сделать статический метод *Join*:

Статический метод *Join* класса *String*

```
string Join(string strSeparator, string[] astr)
string Join(string strSeparator, string[] astr, int iStartIndex, int iCount)
```

К примеру, если строковый массив определить:

```
string[] astr = { "abc", "def", "ghi", "jkl", "mno", "pqr" };
```

можно создать объединяющую строку с разделителями «конец строки» между каждой парой элементов с помощью:

```
string str = string.Join("\r\n", astr);
```

Результирующая строка представляет собой:

```
abc\r\ndef\r\nghi\r\njkl\r\nmno\r\npqr
```

После последней строки разделитель не добавляется.

Вторая версия *Join* позволяет указать последовательный набор строк из массива, которые и будут объединены.

Сравнение строк

String является классом (а не структурой), а тип *string* — ссылочный. В обычном случае это означало бы, что операторы сравнения (== и !=) неправильно бы работали для строк. Производилось бы сравнение ссылок на объекты, а не символов. Однако операторы == и != переопределены для строк и работают именно так, как этого можно ожидать. Выражения:

```
(str == "New York")
```

и:

```
(str != "New Jersey")
```

возвращают значения типа *bool*, зависящее от чувствительного к регистру посимвольного сравнения.

В классе *String* определены несколько методов, возвращающих *bool* в результате чувствительного к регистру строкового сравнения:

Методы *String* (выборочно)

```
bool Equals(string str)
bool Equals(object obj)
bool StartsWith(string str)
bool EndsWith(string str)
```

Например, если строка определена:

```
string str = "The end of time";
```

то:

```
str.StartsWith("The")
```

возвращает *true,* но:

```
str.StartsWith("the")
```

возвращает *false*.

Существует и статическая версия метода *Equals*:

Статические методы *String* (выборочно)

```
bool Equals(string str1, string str2)
```

Например, вместо:

```
if (str == "New York")
```

можно использовать:

```
if (Equals(str, "New York"))
```

Методы вроде этого в основном предназначены для языков, в которых не предусмотрены операторы сравнения,

Остальные методы сравнения строк, реализованные в классе *String,* возвращают целое значение в зависимости от того, является ли одна строка большей, меньшей или равной другой:

Значения, возвращаемые методами сравнения *String*

Возвращаемое значение	Что означает
Отрицательное	*str1* < *str2*
0	*str1* == *str2*
Положительное	*strl* > *str2*

Заметьте: возвращаются именно значения со знаком значения, а не -1, 0 или 1.

Обычно определять, меньше ли одна строка, чем другая, требуется при сортировке строк. И это приводит к тому, что часто сравнение не должно выполняться на основе жестко определенных кодов символов. Например, символы *е* и ё должны быть меньше, чем *F,* независимо от кодов их символов. Сравнение такого типа называется *лексическим* сравнением в отличие от *числового* сравнения.

Вот результат числового сравнения некоторых символов:

```
D < E < F < d < e < f < È < É < Ê < Ë < è < é < ê < ë
```

А вот результат лексического сравнения;

```
d < D < e < E < é < É < è < È < ê < Ê < ë < Ë < f < F
```

А что, лексическое сравнение нечувствительно к регистру? Обычно нет. Например, строка «New Jersey» считается меньшей, чем «new ferk», несмотря на прописную «n» во второй строке. Но если строки полностью идентичны за исключением регистра, буквы в нижнем регистре считаются меньшими, чем буквы в верхнем, т. е. «the» меньше, чем «The». Однако «Them» меньше, чем «then».

Иначе говоря, по умолчанию лексическое сравнение чувствительно к регистру, только если возвращаемое значение при нечувствительном к регистру сравнении было бы 0. В противном случае сравнение нечувствительно к регистру.

Лексическое сравнение вводит определенные соотношения между буквами, цифрами и другими символами. Обычно управляющие символы считаются меньшими, чем апостроф и тире, которые в свою очередь меньше пробельных символов. Затем идут знаки препинания и другие символы, цифры (от 0 до 9) и, наконец, буквы. Строка, равная *null,* меньше, чем пустая строка, а пустая строка меньше любой другой строки. Например:

```
"New" < "New York" < "Newark"
```

Нестатический метод *CompareTo* выполняет лексическое сравнение экземпляра строки и аргумента:

Метод *CompareTo* класса *String*

```
int CompareTo(string str2)
int CompareTo(object obj2)
```

Первой строкой является строковый объект, для которого вы вызываете *CompareTo,* например:

```
str1.CompareTo(str2)
```

Метод *CompareTo* с аргументом-объектом нужен для реализации интерфейса *IComparable*. Метод *CompareTo* используется статическими методами *Array.BinarySearch* и *Array.Sort*, которые я рассмотрю чуть позже.

Все прочие методы сравнения — статические. Метод *CompareOrdinal* выполняет строгое числовое сравнение на основе значения символов:

Статический метод *CompareOrdinal* класса *String*

```
int CompareOrdinal(string str1, string str2)
int CompareOrdinal(string str1, int iStartIndex1, string str2,
                   int iStartIndex2, int iCount)
```

Статический метод *Compare* выполняет лексическое сравнение:

Статический метод *Compare* класса *String*

```
int Compare(string str1, string str2)
int Compare(string str1, string str2, bool bIgnoreCase)
int Compare(string str1, string str2, bool bIgnoreCase, CultureInfo ci)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount)
int Compare(string str1, int iStartIndex1, string str2, int iStartIndex2,
            int iCount, bool bIgnoreCase)
int Compare(string stM, int iStartIndex1, string str2, int iStartIndex2,
            int iCount, bool bIgnoreCase, CultureInfo ci)
```

Аргумент *bIgnoreCase* влияет на возвращаемое значение, только если обе строки различаются лишь регистром. Чувствительные к регистру сравнения обычно полезнее при поиске, чем при сортировке. Вызовы:

```
string.Compare("e", "E")
```

II:

```
string.Compare("e", "E", false)
```

оба возвращают отрицательное значение, однако:

```
string.Compare("e", "E", true)
```

возвращает 0. Вызовы:

```
string.Compare("e", "e", bIgnoreCase)
```

И:

```
string.Compare("e", "E", bIgnoreCase)
```

всегда возвращают отрицательные значения независимо от аргумента *bIgnoreCase*.

Ни один из определенных в классе *String* методов ни при каких условиях не сообщит, что «André» равно «Andre».

Поиск в строках

Библиотечные функции языка C *strchr* и *strstr* ищут первое вхождение указанного символа или строки в строку и возвращают указатель. Эквивалент этих функций в C# — всевозможные версии метода *IndexOf* — возвращает индекс в исходной строке, а не указатель.

Метод *IndexOf* класса *String*

```
int IndexOf(char ch)
int IndexOf(char ch, int iStartIndex)
int IndexOf(char ch, int iStartIndex, int iCount)
int IndexOf(string str)
int IndexOf(string str, int iStartIndex)
int IndexOf(string str, int iStartIndex, int iCount)
```

Можно искать символ или другую строку. Поиск чувствителен к регистру. Метод возвращает -1, если символ или строка не найдены. Дополнительно можно указать начальный индекс и количество символов.

Допустим, строка определена так:

```
string str = "hello world";
```

тогда:

```
str.IndexOf('o')
```

вернет 4, и:

```
str.IndexOf("wo")
```

вернет 6.

Можно искать с конца строки:

Метод *LastIndexOf* класса *String*

```
int LastIndexOf(char ch)
int LastIndexOf(char ch, int iStartIndex)
int LastIndexOf(char ch, int iStartIndex, int iCount)
int LastIndexOf(string str)
int LastIndexOf(string str, int iStartIndex)
int LastIndexOf(string str, int iStartIndex, int iCount)
```

Хотя эти методы осуществляют поиск с конца строки, возвращаемый индекс исчисляется с начала строки. Например, для строки приведенный выше вызов;

```
str.LastIndexOf('o')
```

вернет 7, а:

```
str.LastIndexOf("wo")
```

вернет 6.

Первым аргументом следующих методов является массив символов. Эти методы позволяют выяснить индекс первого или последнего элемента в строке, совпадающего с любым символом из указанного массива:

Методы *String* (выборочно)

```
int IndexOfAny(char[] ach)
int IndexOfAny(char[] ach, int iStartIndex)
int IndexOfAny(char[] ach, int iStartIndex, int iCount)
int LastIndexOfAny(char[] ach)
int LastIndexOfAny(char[] ach, int iStartIndex)
int LastIndexOfAny(char[] ach, int iStartIndex, int iCount)
```

Например, если массив строк и строка определены так:

```
char[] achVowel = { 'a', 'e', 'i', 'o', 'u' };
string str = "hello world";
```

то:

```
str.IndexOfAny(achVowel)
```

возвращает 1, а:

```
str.LastIndexOfAny(achVowel)
```

возвращает 7.

Отсечение и заполнение

Иногда при обработке текстовых файлов (скажем, файлов исходных кодов программ) полезно удалить *пробельные символы* — невидимые символы, разделяющие другие элементы строки. В классе *String* есть методы, позволяющие это сделать. Они считают пробельными следующие символы Unicode:

Пробельные символы Unicode

0x0009 (табуляция)	0x2003 (эм)
0x000A (перевод строки)	0x2004 (треть эм)
0x000B (вертикальная табуляция)	0x2005 (четверть эм)
0x000C (прогон страницы)	0x2006 (шестая часть эн)
0x000D (возврат каретки)	0x2007 (пространство для рисунка)
0x0020 (пробел)	0x2008 (пространство для знака припинания)
0x00A0 (неразывный пробел)	0x2009 (узкий пробел)
0x2000 (учетверенное эн)	0x200A (нитяной пробел)
0x2001 (учетверенное эн)	0x200B (нулевой пробел)
0x2002 (эн)	0x3000 (идеографический пробел)

Можно либо использовать предопределенные пробельные символы, либо определить собственный набор символов:

Методы *String* (выборочно)

```
string Trim() .
string Trim(params char[] ach)
string TrimStart(params char[] ach)
string TrimEnd(params char[] ach)
```

Чтобы удалить предопределенные пробельные символы из начала и конца строки *str*, можно использовать:

```
str.Trim()
```

или:

```
str.Trim(null)
```

Кроме того, из начала строки можно удалить предопределенные пробельные символы с помощью:

```
str.TrimStart(null)
```

Можно удалить их из конца строки;

```
str.TrimEnd(null)
```

С другой стороны, можно указывать символы, которые желательно удалять из начала и конца строки (не обязательно пробельные). Можно либо определить массив символов и передать его методу *Trim* (или *TrimStart* или *TrimEnd*):

```
char[] achTrim = { ' ', '-', '_' };
str.Trim(achTrim)
```

либо перечислить символы в явном виде при вызове метода:

```
str.Trim(' ', '-', '_');
```

В обоих случаях эти три символа удалятся из начала и конца строки.

Кроме того, можно добавлять пробелы (или любые другие символы) в начало и конец строки для обеспечения соответствующей длины строки:

Методы *String* (выборочно)

```
string PadLeft{int iTotalLength)
string PadLeft(int iTotalLength, char ch)
string PadRight(int iTotalLength)
string PadRight(int iTotalLength, char ch)
```

Манипуляции со строками

Здесь я хотел бы рассказать о паре полезных методов, позволяющих вставить одну строку в другую, удалить диапазон символов, а также заменить какой-либо символ или строку на другую строку. В начале приложения я приводил примеры всех этих методов:

Методы *String* (выборочно)

```
string Insert(int iIndex, string strInsert)
string Remove(int ilndex, int iCount)
string Replace(char chOld, char chNew)
string Replace(string strOld, string strNew)
```

Возможно, вам приходилось использовать библиотечную функцию языка C *strtok*. Она предназначена для разбиения строки на фрагменты, т. е. на подстроки, разделенные определенными символами, обычно пробельными. В C вызов *strtok* приходится повторять, пока в строке не останется больше фрагментов. В O псе последовательные вызовы *strtok* заменяет единственный вызов метода *Split*:

Метод ***Split*** **класса** ***String***

```
string[] Split(params char[] achSeparators)
string[] Split(params char[] achSeparators, int iReturnCount)
```

Если первый параметр метода равен *null*, то в качестве разделительных символов будут использованы пробельные символы.

Форматирование строк

Как вы знаете из главы 1, первым аргументом методов *Console.Write* или *Console.WriteLine* может быть строка, описывающая форматирование остальных аргументов. Эти два метода являются в O эквивалентами функции *printf* языка C, в то время как статический метод *Format* класса *String* является эквивалентом функции *sprintf* языка C:

Статический метод *Format* класса *String* (выборочно)

```
string Format(string strFormat, object obj0)
string Format(string strFormat, object obj0, object obj1)
string Format(string strFormat, object objO, object obj1, object obj2)
string Format(string strFormat, params object[] aobj)
```

Например, такой вызов метода *Format*:

```
string str = String.Format("The sum of {0} and {1} is {2}", 2, 3, 2 + 3);
```

возвращает строку "The sum of 2 and 3 is 5»,

Сортировка и поиск в массивах

Реализация интерфейса *IComparable* в классе *String* требует реализации метода:

Метод интерфейса ***IComparable***

```
int CompareTo(object obj)
```

Этот метод вызывают два полезных статических метода *Array*. *Sort* и *BinarySearch*. Их можно использовать с массивами любых объектов, производных от классов, реализующих *IComparable*.

Далее приведены две базовые версии метода *Sort*:

Статические методы *Sort* класса *Array* (выборочно)

```
void Sort(Array arr)
void Sort(Array arr, int iStartIndex, int iCount)
```

Вторая версия позволяет задать подмножество в массиве. Например, массив определен так:

```
string[] astr = { "New Jersey", "New York", "new Mexico", "New Hampshire" };
```

Обратите внимание на строчную букву «*n*» в третьей строке. После вызова:

```
Array.Sort(astr);
```

элементы массива будут расположены в таком порядке: «New Hampshire». "New Jersey», «new Mexico». «New York*. Поскольку метод *Sort* вызывает метод *CompareTo* класса *String*, сортировка нечувствительна к регистру. Однако, если б одним из элементов массива являлась строка «New Mexico» (заглавная «*N*»), строка «New Mexico» после сортировки была бы расположена за строкой «new Mexico».

Следующие две версии метода *Son* требуют двух соответствующих массивов одинакового размера, возможно, с указанием стартового индекса и количества элементов:

Статические методы Sort класса *Array* (выборочно)

```
void Sort(Array arrKeys, Array arrItems)
void Sort(Array arrKeys, Array arrItems, int iStartIndex, int iCount)
```

Эти методы сортируют первый массив и соответствующим образом изменяют порядок элементов во втором массиве. Я использую эти версии метода *Sort* в программе SysInfoReflectionStrings из главы 4 для сортировки массива имен свойств *SystemInformation*, хранящегося в переменной *astrLabels*:

```
Array, Sort(astrLabels, astrValues);
```

Порядок элементов в массиве значений свойств *SystemInformation*, хранящийся в переменной *astrValues*, также изменяется, так что элементы двух массивов соответствуют друг другу.

Чтобы при выполнении сортировки вызывался другой метод, а не *CompareTo*, используйте следующие версии метода *Sort*:

Статические методы *Sort* класса *Array* (выборочно)

```
void Sort(Array arr, IComparer comp)
void Sort(Array arr, int iStartIndex, int iCount, IComparer comp)
void Sort(Array arrKeys, Array arrItems, IComparer comp)
void Sort(Array arrKeys, Array arrItems, int iStartIndex, iCount,
          IComparer comp)
```

Аргумент типа *IComparer* может быть экземпляром любого класса, реализующего интерфейс *IСompare*. Класс *String* не является таким классом! В нем реализован интерфейс *IComparable*, а не *IComparer*.

Интерфейс *IComparer* определен в пространстве имен *SystemCollections*. В классе, реализующем *Icomparer*, должен быть определен метод:

Метод интерфейса *IComparer*

```
int Compare(object obj1, object obj2)
```

Так как он нестатический, то не определен в классе *String*. (Единственный реализованный в классе *String* метод — *Compare* — статический.)

Пространство имен *SystemCollections* содержит два реализующих *IComparer* класса. Ими являются *Comparer* (для выполнения чувствительного к регистру сравнения, такого же. как по умолчанию) и *CaseInsensitiveComparer(для* нечувствительного к регистру сравнения). Оба содержат статический член *Default,* возвращающий экземпляр класса,

Например, для выполнения чувствительной к регистру сортировки строкового массива *astr* следует вызвать;

```
Array.Sort(astr);
```

или:

```
Array.Sort(astr, Comparer.Default);
```

Чтобы выполнить нечувствительную к регистру сортировку, надо вызвать:

```
Array.Sort(astr, CaseInsensitiveComparer.Default);
```

Нечувствительное к регистру сравнение гораздо полезнее в методе *BinarySearch* (или при выполнении сортировки в рамках подготовки к двоичному поиску), чем в методе *Sort:*

Статический метод *BinarySearch* класса *Array*

```
int BinarySearch(Array arr, object obj)
int BinarySearch(Array arr, int iStartIndex, int iCount, object obj)
int BinarySearch(Array arr, object obj, IComparer comp)
int BinarySearch(Array arr, int iStartIndex, int iCount, object obj,
                 IComparer comp)
```

Для двоичного поиска массив должен быть предварительно отсортирован. Допустим, отсортированный массив из четырех строчек (названий штатов Америки) содержит элементы:

```
"New Hampshire", "New Jersey", "new Mexico", "New York"
```

Вызов:

```
Array.BinarySearch(astr, "New York")
```

возвращает 3, поскольку строка совпадает с *astr[3].* Вызов:

```
Array.BinarySearch(astr, "New Mexico")
```

возвращает *-A.* Отрицательное значение указывает, что строка не содержится в массиве. (Напоминаю, сортировка по умолчанию чувствительна к регистру!) До-

полнение к возвращаемому значению равно 3, т. е. *astr[3]* является следующим наибольшим элементом массива,

Вызов:

```
Array.BinarySearch(astr, "new Mexico"));
```

возвращает 2. так как аргумент совпадает с *astr[2].* Вызов:

```
Array.BinarySearch(astr, "New Mexico", CaseInsensitiveComparer.Default));
```

выполняет нечувствительный к регистру поиск и также возвращает 2.

Класс *StringBuilder*

Возможно, вас беспокоит, не влияет ли многократное создание объектов *String* на производительность. Иногда влияет. Проанализируем приведенную далее программу, в которой большие строки создаются оператором += в 10 000 операций присоединения строк.

StringAppend.cs

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

class StringAppend
{
     const int iIterations = 10000;

     public static void Main()
     {
          DateTime dt  = DateTime.Now;
          string   str = String.Empty;

          for (int i = 0; i < iIterations; i++)
               str += "abcdefghijklmnopqrstuvwxyz\r\n";

          Console.WriteLine(DateTime.Now - tft);
     }
}
```

В этой программе для измерения времени выполнения в начале и конце работы вызывается метод *Now* класса *DateTime.* Время выводится в часах, минутах, секундах и сотнях наносекунд. (О классе *DateTime* и связанных с ним классами см. главу 10.) Выполнение каждой операции присоединения строк приводит к появлению нового объекта *Stnng.* что в свою очередь требует выделения памяти. Каждая предшествующая строка помечается как доступная для сборки мусора.

Скорость работы этой программы зависит от скорости компьютера, обычно это длится около 1 минуты.

Более удачным решением в нашем случае будет использование класса *StringBuilder,* определенного в пространстве имен *Svstem.Text.* В отличие от строк, хранящихся в объектах класса *String* class, строки в объектах класса *StringBuilder* могут изменяться. Класс *StringBuilder*динамически перераспределяет память, используемую строкой. Как только размер строки превышает размер буфера, объем буфера вырастает вдвое. Для преобразования объекта *StringBuilder* в объект *String* служит метод *ToString.*

Вот переработанный вариант программы, где применяется *StringBuilder.*

StringBuil d erAppend.cs

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

class StringBuilderAppend
{
     const int iIterations = 10000;

     public static void Main()
     {
          DateTime      dt = DateTime.Now;
          StringBuilder sb = new StringBuilder();

          for (int i = 0; i < iIterations; i++)
               sb.Append("abcdefghijklmnopqrstuvwxyz\r\n");

          string str = sb.ToString();

          Console.WriteLine( DateTime.Now - dt);
     }
}
```

Время выполнения этой программы — менее секунды. Похоже, что она работает в 1 000 раз быстрее своей предшественницы.

Другой эффективный способ заключается в использовании класса *StringWnter.* определенного в пространстве имен *SystemJO.* Я уже говорил в приложении А, что такие классы, как *StringWriter*и *StreamWriter*(используемые для записи текстовых файлов), являются производными от абстрактного класса *TextWriter*Как и *StringBuilder,* класс *StringWnter* соединяет строки. Его большое преимущество — возможность использования целого арсенала методов *Write* и *WriteLine,*определенных в классе *TextWriter.*Вот пример программы, выполняющей ту же задачу, что и две предыдущие, но использующей объект *StringWnter.*

StringWriterAppend.cs

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

class StringWriterAppend
{
     const int iIterations = 10000;

     public static void Main()
     {
          DateTime     dt = DateTime.Now;
          StringWriter sw = new StringWriter();

          for (int i = 0; i < iIterations; i++)
               sw.WriteLine("abcdefghijklmnopqrstuvwxyz");

          string str = sw.ToString();

          Console.WriteLine(DateTime.Now - dt);
     }
}
```

Скорость работы этой программы сравнима со скоростью StringBuilderAppend.

Из всего этого следует извлечь урок. Операционные системы, языки программирования, библиотеки классов и разнообразные инфраструктуры обеспечивают все увеличивающийся уровень абстракции, и иногда мы, программисты, теряем понимание механизмов, лежащих ниже этого уровня абстракции. То, что в коде может выглядеть как простое сложение, реально может быть связано с множеством операций более низкого уровня.

Мы можем абстрагироваться от этих низкоуровневых операций, но должны всегда помнить) об их наличии. Если какая-то конкретная операция кажется вам слишком медленной, требует слишком много памяти или связана с большим количеством непонятного сопутствующего кода, попробуйте выяснить, отчего это происходит, и найдите альтернативу. Скорее всего кто-то уже разработал нужные вам средства,

ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ MICROSOFT

(прилагаемый к книге компакт-диск)

ЭТО ВАЖНО - ПРОЧИТАЙТЕ ВНИМАТЕЛЬНО. Настоящее лицензионное соглашение (далее «Соглашение») является юридическим документом, оно заключается между Вами (физическим или юридическим лицом) и Microsoft Corporation (далее «корпорация Microsoft») на указанный выше продукт Microsoft, который включает программное обеспечение и может включать сопутствующие мультимедийные и печатные материалы, а также электронную документацию (далее «Программный Продукт»). Любой компонент, входящий в Программный Продукт, который сопровождается отдельным Соглашением, подпадает под действие именно того Соглашения, а не условий, изложенных ниже. Установка, копирование или иное использование данного Программного Продукта означает принятие Вами данного Соглашения. Если Вы не принимаете его условия, то не имеете права устанавливать, копировать или как-то иначе использовать этот Программный Продукт.

ЛИЦЕНЗИЯ НА ПРОГРАММНЫЙ ПРОДУКТ

Программный Продукт защищен законами Соединенных Штатов по авторскому праву и международными договорами по авторскому праву, а также другими законами и договорами по правам па интеллектуальную собственность.

1. **ОБЪЕМ ЛИЦЕНЗИИ.** Настоящее Соглашение дает Вам право:

   a) **Программный продукт.** Вы можете установить и использовать одну копию Программного Продукта на одном компьютере. Основной пользователь компьютера, на котором установлен данный Программный Продукт, может сделать только для себя вторую копию и использоватв ее на портативном компьютере.

   b) **Хранение или использование в сети.** Вы можете также скопировать или установить экземпляр Программного Продукта на устройстве хранения, например на сетевом сервере, исключительно для установки или запуска данного Программного Продукта на других компьютерах в своей внутренней сети, но тогда Вы должны приобрести лицензии на каждый такой компьютер. Лицензию на данный Программный продукт нельзя использовать совместно или одновременно па других компьютерах.

   c) **License Pak.** Если Вы купили эту лицензию в составе Microsoft License Pak, можете сделать ряд дополнительных копий программного обеспечения, входящего в данный Программный Продукт, и использовать каждую копию так, как было описано выше. Кроме того, Вы получаете право сделать соответствующее число вторичных копий для портативного компьютера в целях, также оговоренных выше.

   d) **Примеры** кода. Это относится исключительно к отдельным частям Программного Продукта, заявленным как примеры кода (далее «Примеры»), если таковые входят в состав Программного Продукта.

      i) **Использование и модификация.** Microsoft дает Вам право использовать и модифицировать исходный код Примеров при условии соблюдения пункта (d)(iii) ниже. Вы не имеете права распространять в виде исходного кода ни Примеры, ни их модифицированную версию.

      ii) **Распространяемые файлы.** При соблюдении пункта (d)(iii) Microsoft дает Вам право на свободное от отчислений копирование и распространение в виде объектного кода Примеров или их модифицированной версии, кроме тех часгей (или их модифицированных версий), которые оговорены в файле Readme, относящемся к данному Программному Продукту, как не подлежащие распространению.

      iii) **Требования *к* распространению файлов.** Вы можете распространять файлы, разрешенные к распространению, при условии, что: а) распространяете их в виде объектного кода только в сочетании со своим приложением и как его часть; б) не используете название, эмблему или товарные знаки Microsoft для продвижения своего приложения; в) включаете имеющуюся в Программном Продукте ссылку на авторские права в состав этикетки и застаики своего *приложения;* т) согласны освободить от ответственности и взять на себя защиту корпорации Microsoft от любых претензий или преследований по закону, включая судебные издержки, если таковые возникнут в результате использования или распространения Вашего приложения; и д) не допускаете дальнейшего распространения конечным пользователем своего приложения. По поводу отчислений и других условий лицензии применительно к иным видам использования или распространения распространяемых файлов обращайтесь в Microsoft.

2. **ПРОЧИЕ ПРАВА И ОГРАНИЧЕНИЯ**

- **Ограничения на реконструкцию, декомпиляцию и дизассемблирование.** Вы не имеете права реконструировать, декомпилировать или дизассемблировать данный Программный Продукт, кроме того случая, когда такая деятельность (только в той мере, которая необходима) явно разрешается соответствующим законом, несмотря на это ограничение.
- **Разделение компонентов.** Данный Программный Продукт лицензируется как единый продукт. Его компоненты нельзя отделять друг от друга для использования более чем на одном компьютере.
- **Аренда.** Данный Программный Продукт нельзя сдавать в прокат, передавать во временное пользование или уступать для использования в иных целях.
- **Услуги по технической поддержке.** Microsoft может (но не обязана) предоставить Вам услуги по технической поддержке данного Программного Продукта (далее «Услуги»). Предоставление Услуг регулируется соответствующими правилами и программами Microsoft, описанными в руководстве пользователя, электронной документации и/или других материалах, публикуемых Microsoft. Любой дополнительный программный код, предоставленный в рамках Услуг, следует считать частью данного Программного Продукта и подпадающим под действие настоящего Соглашения. Что касается технической информации, предоставляемой Вами корпорации Microsoft при использовании ее Услуг, то Microsoft может задействовать эту информацию в деловых целях, в том числе для технической поддержки продукта и разработки. Используя такую техническую информацию, Microsoft не будет ссылаться на Вас.
- **Передача прав на программное обеспечение.** Вы можете безвозвратно уступить все права, регулируемые настоящим Соглашением, при условии, что не оставите себе никаких копий, передадите все составные части данного Программного Продукта (включая компоненты, мультимедийные и печатные материалы, любые обновления, Соглашение и сертификат подлинности, если таковой имеется) и принимающая сторона согласится с условиями настоящего Соглашения,
- **Прекращение действия Соглашения.** Без ущерба для любых других прав Microsoft может прекратить действие настоящего Соглашения, если Вы нарушите его условия. В этом случае Вы должны будете уничтожить все копии данного Программного Продукта вместе со всеми его компонентами.

3. **АВТОРСКОЕ ПРАВО.** Все авторские права и право собственности на Программный Продукт (в том числе любые изображения, фотографии, анимации, видео, аудио, музыку, текст, примеры кода, распространяемые файлы и апплеты, включенные в состав Программного Продукта) и любые его копии принадлежат корпорации Microsoft или ее поставщикам. Программный Продукт охраняется законодательством об авторских правах и положениями международных договоров. Таким образом. Вы должны обращаться с данным Программным Продуктом, как с любым другим материалом, охраняемым авторскими правами, **с тем исключением,** что Вы можете установить Программный Продукт на один компьютер при условии, что храните оригинал исключительно как резервную или архивную копию. Копирование печатных материалов, поставляемых вместе с Программным Продуктом, запрещается,

ОГРАНИЧЕНИЕ ГАРАНТИИ

ДАННЫЙ ПРОГРАММНЫЙ ПРОДУКТ (ВКЛЮЧАЯ ИНСТРУКЦИИ ПО ЕГО ИСПОЛЬЗОВАНИЮ) ПРЕДОСТАВЛЯЕТСЯ БЕЗ КАКОЙ-ЛИБО ГАРАНТИИ. КОРПОРАЦИЯ MICROSOFT СНИМАЕТ С СЕБЯ ЛЮБУЮ ВОЗМОЖНУЮ ОТВЕТСТВЕННОСТЬ. В ТОМ ЧИСЛЕ ОТВЕТСТВЕННОСТЬ ЗА КОММЕРЧЕСКУЮ ЦЕННОСТЬ ИЛИ СООТВЕТСТВИЕ ОПРЕДЕЛЕННЫМ ЦЕЛЯМ. ВЕСЬ РИСК ПО ИСПОЛЬЗОВАНИЮ ИЛИ РАБОТЕ С ПРОГРАММНЫМ ПРОДУКТОМ ЛОЖИТСЯ НА ВАС.
НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ КОРПОРАЦИЯ MICROSOFT, ЕЕ РАЗРАБОТЧИКИ, А ТАКЖЕ ВСЕ, ЗАНЯТЫЕ В СОЗДАНИИ, ПРОИЗВОДСТВЕ И РАСПРОСТРАНЕНИИ ДАННОГО ПРОГРАММНОГО ПРОДУКТА, НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА КАКОЙ-ЛИБО УЩЕРБ (ВКЛЮЧАЯ ВСЕ, БЕЗ ИСКЛЮЧЕНИЯ, СЛУЧАИ УПУЩЕННОЙ ВЫГОДЫ, НАРУШЕНИЯ ХОЗЯЙСТВЕННОЙ ДЕЯТЕЛЬНОСТИ, ПОТЕРИ ИНФОРМАЦИИ ИЛИ ДРУГИХ УБЫТКОВ) ВСЛЕДСТВИЕ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ДАННОГО ПРОГРАММНОГО ПРОДУКТА ИЛИ ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ КОРПОРАЦИЯ MICROSOFT БЫЛА ИЗВЕЩЕНА О ВОЗМОЖНОСТИ ТАКИХ ПОТЕРЬ. ТАК КАК В НЕКОТОРЫХ СТРАНАХ НЕ РАЗРЕШЕНО ИСКЛЮЧЕНИЕ ИЛИ ОГРАНИЧЕНИЕ ОТВЕТСТВЕННОСТИ ЗА НЕПРЕДНАМЕРЕННЫЙ УЩЕРБ, УКАЗАННОЕ ОГРАНИЧЕНИЕ МОЖЕТ ВАС НЕ КОСНУТЬСЯ.

РАЗНОЕ

Настоящее Соглашение регулируется законодательством штата Вашингтон (США), кроме случаев (и лишь в той мере, насколько это необходимо) исключительной юрисдикции того государства, на территории которого используется Программный Продукт.
Если у Вас возникли какие-либо вопросы, касающиеся настоящего Соглашения, или если Вы желаете связаться с Microsoft по любой другой причине, пожалуйста, обращайтесь в местное представительство Microsoft или пишите по адресу Microsoft Sales Information Center, One Microsoft Way, Redmond, WA98052-6399.

Петцольд Чарльз

Программирование для Microsoft Windows на C#

Том 1

Перевод с английского под общей редакцией *В, Г. Вшивцева*

Переводчики: *С. А. Аристов, В. В. Краснов, Ю. С. Левашко, К, А. Меньшов, Н. Б. Соколова, А. Е. Соловченко*

Технический редактор *Н. Г. Тимченко*

Компьютерная верстка *В. Б. Хильченко*

Дизайн обложки *Е. В. Козлова*

Фотография Ч. Петцольда на обложке *DeidreSinott*
(фото опубликовано на сайте *www.charlespetzold.com*)

Оригинал-макет выполнен с использованием
издательской системы Adobe PageMaker 6.0

TypeMarketFontLibrary
легальный пользователь

пользователь
Para(-)Type
IN LEGAL USE

Подготовлено к печати
Издательско-торговым домом «Русская Редакция»

РУССКАЯ РЕДАКЦИЯ

Подписано в печать 25.05.02 г. Тираж 3 000 экз.
Формат 70x100/16. Физ. п. л. 36

Отпечатано с готовых диапозитивов в ОАО «Типография «Новости»
107105, Москва, ул. Фр. Энгельса, 46

Первый в Восточной Европе

ЦЕНТР ЗНАНИЙ Microsoft Press



В августе 2001 года Московское представительство Microsoft и российское издательство «Русская Редакция» объяви-ли об открытии **первого в Восточной Европе Центра Знаний Microsoft Press** (Microsoft Press Knowledge Center).



Главная задача Ценра Знаний в Москве — предоставить читателям весь спектр имеющейся на русском языке **литературы Microsoft Press.** Сегодня в России книги Microsoft Press, переведенные на русский язык, выпускают четыре издательства: **«Русская Редакция», «BHV — Санкт-Петербург», «Питер», «Эком».**



Центр знаний открыт **в специализированном магазине «Компьютерная и деловая книга»,** предлагающем наиболее полный ассортимент литературы, посвященной информационным технологиям.
В магазине собрано около 1 000 наименований русскоязычных книг по компьютерной тематике.

магазин «Компьютерная и деловая книга»
Адрес: Москва, Ленинский пр-т, стр. 38
Телефон: (095) 778-7269
Схема проезда, рейтинг продаж:
http://www.rusedit.ru

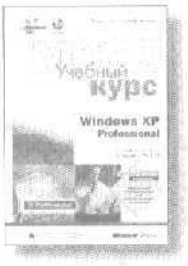


Издательство «Русская Редакция»

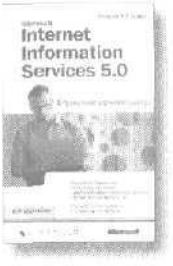
— партнер **Microsoft** Press в России —

предлагает широкий выбор литературы
по современным информационным технологиям.
Мы переводим на русский язык
бестселлеры ведущих издательств мира,
а также сотрудничаем с компетентными
российскими авторами.











Продажа книг
оптом: тел.: (095) 142-0571, e-mail: sale@rr.sedi.ru; **интернет-магазин:** http://lbook.ru; тел.: (095) 145-4519;
в розницу: магазин «КОМПЬЮТЕРНАЯ и ДЕЛОВАЯ КНИГА» Москва, Ленинский пр-т, стр. 38, тел.: (095) 778-7269
«КНИЖНАЯ ЯРМАРКА на Тульской», Варшавское ш., 9, тел.: 512-02

РУССКАЯ РЕДАКЦИЯ

ЕЖЕМЕСЯЧНЫЙ
НАУЧНО-ПОПУЛЯРНЫЙ
КОМПЬЮТЕРНЫЙ ЖУРНАЛ

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 и программных продуктов
- интернет и мультимедиа,игры и прикладные программы
- консультации экспертов,впречи с интересными людьми
- CD-приложение с полезными утилитами



e-mail:
info@hardnsoft.ru

НАШИ ИНДЕКСЫ:
Hard'n'Soft - 73140, Hara* n Soft + CD - 26067

профессиональный журнал

ПРОГРАММИСТ

Профессиональный журнал, посвященный исключительно **вопросам** разработки. Наши **авторы** - **профессиональные** программисты, которые делятся с читателями «секретами **мастерства**».

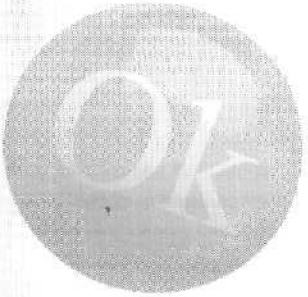
Мы публикуем материалы о самых **современных** технологиях и средствах **разработки,** статьи о принципах и методах, **теории** и практике **программирования.**

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.
1 Их привлекает имеющаяся возможность получения демо-версий и обновлений,
• В выборе программ им помогают каталог SoftLine-direct и сайт www.softiine.ru.
Ш 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).
Ш Совместная работа (Centura, Merant, Microsoft).
Ш Управление проектами (PlanisWare, PlanView, Microsoft).
Ш 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).
ffi Отладка и тестирование (NuMega, Intuitive Systems, Segue).
Ш Упаковка приложений (InstallShield, Wise Solutions).
Я 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 в 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
INFORMATION ANYWHERE


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



...теперь на русском языке

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) 146-4519, e-mail: emsdn@rusedit.ru, http://www.rusedit.ru
http://www.microsoft.com/rus/msdn/magazine