

Повысьте свой уровень с помощью
этого введения в C#!



C# 2005

ДЛЯ "ЧАЙНИКОВ"™

**Для
сомневающих**

Исходные тексты
демонстрационных
программ
и дополнительные
материалы
на прилагаемом
компакт-диске!

Стефан Рэнди Дэвис
Чак Сфер

 **ДИАЛЕКТИКА**
www.dialektika.com



С# 2005

ДЛЯ
"ЧАЙНИКОВ"™

C# 2005 FOR DUMMIES®

by Stephen Randy Davis
and Chuck Sphar



WILEY

Wiley Publishing, Inc.

C# 2005

ДЛЯ
"ЧАЙНИКОВ"™

Стефан Рэнди Дэвис
Чак Сфер



ДИАЛЕКТИКА

Москва ♦ Санкт-Петербург ♦ Киев
2008

ББК 32.973.26-018.2.75

Д94

УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского канд. техн. наук *И.В. Красикова, А.А. Мраморова*

Под редакцией канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:

info@dialektika.com, <http://www.dialektika.com>

115419, Москва, а/я 783; 03150, Киев, а/я 152

Дэвис, Стефан Рэнди, Сфер, Чак.

Д94 С# 2005 для "чайников".: Пер. с англ. — М.: ООО "И.Д. Вильямс", 2008. — 576 с.: ил. — Парал. тит. англ.

ISBN 978-5-8459-1068-4 (рус.)

Даже если вы никогда не имели дела с программированием, эта книга поможет вам освоить с нуля язык С#. Вы сможете писать на нем программы любой степени сложности. Если вы уже знакомы с каким-либо иным языком программирования, тогда процесс изучения С# только упростится, но наличие опыта программирования — условие совершенно необязательное.

Книга познакомит вас не только с типами, конструкциями и операторами языка С#, но и с ключевыми концепциями объектно-ориентированного программирования, реализованными в этом языке, который в настоящее время представляет собой один из наиболее приспособленных для создания программ для Windows-среды.

Если вы в начале большого пути в программирование — смелее покупайте эту книгу: она послужит вам отличным путеводителем, который облегчит вам первые шаги на этом длинном, но очень увлекательном пути.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства JOHN WILEY&Sons, Inc.

Copyright © 2008 by Dialektika Computer Publishing.

Original English language edition Copyright © 2006 by Wiley Publishing, Inc., Indianapolis, Indiana.

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Wiley Publishing, Inc.

Wiley, the Wiley Publishing logo, For Dummies, the Dummies Man logo, A Reference for the Rest of Us!, The Dummies Way, Dummies Daily, The Fun and Easy Way, Dummies.com, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc., and/or its affiliates in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

ISBN 978-5-8459-1068-4 (рус.)

ISBN 0-7645-9704-3 (англ.)

© Компьютерное изд-во "Диалектика", 2008,
перевод, оформление, макетирование

© by Wiley Publishing, Inc., 2006

Оглавление

Об авторах	1 ⁷
Введение	19
Часть I. Создание ваших первых программ на C#	27
Глава 1. Создание вашей первой Windows-программы на C#	29
Глава 2. Создание консольного приложения на C#	47
Часть II. Основы программирования в C#	55
Глава 3. Объявление переменных-значений	57
Глава 4. Операторы	73
Глава 5. Управление потоком выполнения	85
Часть III. Объектно-основанное программирование	и 3
Глава 6. Объединение данных — классы и массивы	115
Глава 7. Функции функций	141
Глава 8. Методы класса	177
Глава 9. Работа со строками в C#	199
Часть IV. Объектно-ориентированное программирование	2 23
Глава 10. Что такое объектно-ориентированное программирование	225
Глава 11. Классы	231
Глава 12. Наследование	261
Глава 13. Полиморфизм	283
Часть V. За базовыми классами	309
Глава 14. Интерфейсы и структуры	311
Глава 15. Обобщенное программирование	339
Часть VI. Великолепные десятки	373
Глава 16. Десять наиболее распространенных ошибок компиляции	375
Глава 17. Десять основных отличий C# и C++	385

Часть VII. Дополнительные главы	391
Глава 18. Эти исключительные исключения	393
Глава 19. Работа с файлами и библиотеками	419
Глава 20. Работа с коллекциями	445
Глава 21. Использование интерфейса Visual Studio	487
Глава 22. C# по дешевке	525
Предметный указатель	565

Содержание

Об авторах	17
Введение	19
Часть I. Создание ваших первых программ на C#	27
Глава 1. Создание вашей первой Windows-программы на C#	29
Введение в машинные языки, C# и платформу .NET	29
Что такое программа?	30
Что такое C#?	30
Что такое .NET?	31
Что такое Visual Studio 2005? Visual C#?	32
Создание Windows-приложения на языке C#	32
Создание шаблона	33
Компиляция и запуск вашей первой программы Windows Forms	36
Украшение программы	37
Учим форму трудиться	42
Проверка конечного продукта	43
Программисты на Visual Basic 6.0, берегитесь!	44
Глава 2. Создание консольного приложения на C#	47
Создание шаблона консольного приложения	47
Создание исходной программы	47
Пробная поездка	49
Создание реального консольного приложения	49
Изучение шаблона консольного приложения	51
Схема программы	51
Комментарии	51
Тело программы	52
Часть II. Основы программирования в C#	55
Глава 3. Объявление переменных-значений	57
Объявление переменной	57
Что такое int	58
Правила объявления переменных	59
Вариации на тему int	59
Представление дробных чисел	60
Работа с числами с плавающей точкой	61
Объявление переменной с плавающей точкой	62
Более точное преобразование температур	63

Ограничения переменных с плавающей точкой	63
Десятичные числа — комбинация целых и чисел с плавающей точкой	64
Объявление переменных типа decimal	64
Сравнение десятичных, целых чисел и чисел с плавающей точкой	65
Логичен ли логический тип?	65
Символьные типы	66
Тип char	66
Специальные символы	66
Тип string	67
Что такое тип-значение?	67
Сравнение string и char	68
Объявление числовых констант	69
Преобразование типов	70
Глава 4. Операторы	73
Арифметика	73
Простейшие операторы	73
Порядок выполнения операторов	74
Оператор присваивания	75
Оператор инкремента	76
Логично ли логическое сравнение?	77
Сравнение чисел с плавающей точкой	78
Составные логические операторы	79
Тип выражения	80
Вычисление типа операции	80
Типы при присваивании	82
Немного экзотики — тернарный оператор	83
Глава 5. Управление потоком выполнения	85
Управление потоком выполнения	86
Оператор if	86
Инструкция else	89
Как избежать else	90
Вложенные операторы if	90
Циклы	93
Цикл while	93
Цикл do...while	98
Операторы break и continue	98
Цикл без счетчика	99
Правила области видимости	103
Цикл for	104
Пример	104
Зачем нужны разные циклы	105
Вложенные циклы	106
Конструкция switch	109
Оператор goto	111

Часть III. Объектно-ориентированное программирование

и 3

Глава 6. Объединение данных — классы и массивы	115
Классы	115
Определение класса	116
Что такое объект	117
Доступ к членам объекта	117
Ссылки	120
Классы, содержащие классы	122
Статические члены класса	123
Определение константных членов-данных	124
Массивы C#	124
Зачем нужны массивы	125
Массив фиксированного размера	125
Массив переменного размера	127
Массивы объектов	130
Конструкция foreach	133
Сортировка массива объектов	134
Глава 7. Функции функций	141
Определение и использование функции	141
Использование функций в ваших программах	143
Аргументы функции	149
Передача аргументов функции	150
Передача функции нескольких аргументов	150
Соответствие определений аргументов их использованию	152
Перегрузка функции	153
Реализация аргументов по умолчанию	154
Передача в функцию типов-значений	156
Возврат значений из функции	162
Возврат значения оператором return	162
Возврат значения посредством передачи по ссылке	163
Когда какой метод использовать	163
Определение функции без возвращаемого значения	166
Передача аргументов в программу	167
Передача аргументов из приглашения DOS	169
Передача аргументов из окна	170
<i>Передача аргументов в Visual Studio 2005</i>	173
Глава 8. Методы класса	177
Передача объекта в функцию	177
Определение функций объектов и методов	179
Определение функций — статических членов	179
Определение метода	181
Полное имя метода	182
Обращение к текущему объекту	183
Ключевое слово this	185

Когда <code>this</code> используется явно	185
Что делать при отсутствии <code>this</code>	188
Помощь от Visual Studio — автоматическое завершение	190
Справка по встроенным функциям системной библиотеки	191
Помощь при использовании ваших собственных функций и методов	192
Внесение дополнений в справочную систему	193
Генерация XML-документации	197
Глава 9. Работа со строками в C#	199
Основные операции над строками	200
Объединение неразделимо!	200
Сравнение строк	201
Сравнение без учета регистра	205
Использование конструкции <code>switch</code>	205
Считывание ввода пользователя	206
Разбор числового ввода	207
Обработка последовательности чисел	210
Управление выводом программы	212
Использование методов <code>Trim()</code> и <code>Pad()</code>	212
Использование функции конкатенации	215
Использование функции <code>Split()</code>	217
Форматирование строки	218
Часть IV. Объектно-ориентированное программирование	223
Глава 10. Что такое объектно-ориентированное программирование	225
Объектно-ориентированная концепция №1 — абстракция	225
Приготовление блюд с помощью функций	226
Приготовление "объектно-ориентированных" блюд	226
Объектно-ориентированная концепция №2 — классификация	227
Зачем нужна классификация	227
Объектно-ориентированная концепция №3 — удобный интерфейс	228
Объектно-ориентированная концепция №4 — управление доступом	229
Поддержка объектно-ориентированных концепций в C#	229
Глава 11. Классы	231
Ограничение доступа к членам класса	231
Пример программы с использованием открытых членов	232
Прочие уровни безопасности	235
Зачем нужно управление доступом	235
Методы доступа	236
Пример управления доступом	237
Выводы	242
Определение свойств класса	242
Конструирование объектов посредством конструкторов	244
Конструкторы, предоставляемые C#	244

Конструктор по умолчанию	246
Создание объектов	247
Выполнение конструктора в отладчике	249
Непосредственная инициализация объекта — конструктор по умолчанию	252
Конструирование с инициализаторами	252
Перегрузка конструкторов	253
Устранение дублирования конструкторов	256
Фокусы с объектами	260
Глава 12. Наследование	261
Наследование класса	261
Зачем нужно наследование	263
Более сложный пример наследования	264
ЯВЛЯЕТСЯ или СОДЕРЖИТ	267
Отношение ЯВЛЯЕТСЯ	267
Доступ к BankAccount через содержание	268
Отношение СОДЕРЖИТ	269
Когда использовать отношение ЯВЛЯЕТСЯ, а когда — СОДЕРЖИТ	270
Поддержка наследования в C#	270
Изменение класса	270
Неверное преобразование времени выполнения	271
Ключевые слова is и as	272
Наследование и конструктор	274
Вызов конструктора по умолчанию базового класса	274
Передача аргументов конструктору базового класса	276
Обновленный класс BankAccount	278
Деструктор	281
Глава 13. Полиморфизм	283
Перегрузка унаследованного метода	283
Простейший случай перегрузки функции	284
Различные классы, различные методы	284
Соккрытие метода базового класса	285
Вызов методов базового класса	289
Полиморфизм	291
Что неверно в стратегии использования объявленного типа	292
Использование is для полиморфного доступа к скрытому методу	293
Объявление метода виртуальным	294
Абстракционизм в C#	297
Разложение классов	297
Голая концепция, выражаемая абстрактным классом	302
Как использовать абстрактные классы	302
Создание абстрактных объектов невозможно	304
Создание иерархии классов	304
Опечатывание класса	308

Часть V. За базовыми классами	309
Глава 14. Интерфейсы и структуры	311
Что значит МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК	311
Что такое интерфейс	312
Краткий пример	313
Пример программы, использующей отношение МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК	315
Создание собственного интерфейса	315
Предопределенные интерфейсы	316
Сборка воедино	317
Наследование интерфейса	323
Абстрактный интерфейс	323
Структуры C# и их отличия от классов	326
Структуры C#	327
Конструктор структуры	329
Методы структур	329
Пример применения структуры	330
Унификация системы типов	333
Предопределенные типы структур	333
Унификация системы типов с помощью структур	334
Упаковка типов-значений	337
Глава 15. Обобщенное программирование	339
Необобщенные коллекции	340
Необобщенные коллекции	340
Использование необобщенных коллекций	341
Обобщенные классы	343
Обобщенные классы безопасны	343
Обобщенные классы эффективны	344
Использование обобщенных коллекций	344
Понятие <T>	345
Использование List<T>	345
Создание собственного обобщенного класса	347
Очередь с приоритетами	348
Распаковка пакета	352
Функция Main()	353
Написание обобщенного кода	355
Обобщенная очередь с приоритетами	356
Незавершенные дела	358
Обобщенные методы	360
Обобщенные методы в необобщенных классах	362
Обобщенные методы в обобщенных классах	363
Ограничения для обобщенного метода	363
Обобщенные интерфейсы	364
Обобщенные и необобщенные интерфейсы	364
Использование (необобщенной) фабрики классов	365
Построение обобщенной фабрики	366

Объявление пространств имен	-	422
Важность пространств имен		424
Доступ к классам с использованием полностью квалифицированных имен		425
Директива using		426
Использование полностью квалифицированных имен		427
Объединение классов в библиотеки		430
Создание проекта библиотеки классов		430
Создание классов для библиотеки		431
Создание проекта драйвера	•	432
Хранение данных в файлах		434
Использование StreamWriter		435
Повышение скорости чтения с использованием StreamReader		440
Глава 20. Работа с коллекциями		445
Обход каталога файлов		445
Написание собственного класса коллекции: связанный список		451
Пример связанного списка		452
Зачем нужен связанный список		461
Обход коллекций: итераторы		461
Доступ к коллекции: общая задача		462
Использование foreach		464
Обращение к коллекциям как к массивам: индексаторы		465
Формат индексатора		465
Пример программы с использованием индексатора		465
Блок итератора		469
Итерация месяцев		473
Что такое коллекция		474
Синтаксис итератора		475
Блоки итераторов произвольного вида и размера		476
Где надо размещать итераторы		479
Глава 21. Использование интерфейса Visual Studio		487
Настройка расположения окон		487
Состояния окон		488
Скрытие окна		490
Перестановка окон		490
Наложение окон		491
Модные штучки		493
Работа с Solution Explorer		493
Упрощение жизни с помощью проектов и решений		494
Отображение проекта		495
Добавление класса		497
Завершение демонстрационной программы		498
Преобразование классов в программу		501
Как должен выглядеть код		502
Помогите мне!		506
F1		506

Предметный указатель	507
Поиск	509
Дополнительные возможности	510
Автоперечисление членов	511
Отладка	512
Жучки в программе: а дуством не пробовали?	512
Пошаговая отладка	514
Главное — вовремя остановиться	517
Стек вызовов	520
Я сделал это!	523
Глава 22. C# по дешевке	525
Работа без сети — но не без платформы .NET	526
Получение бесплатных компонентов	526
Обзор цикла разработки	527
Программирование на C# в программе SharpDevelop	528
Изучение SharpDevelop	528
Сравнение возможностей SharpDevelop и Visual Studio	529
Получение справочной информации	530
Настройка программы SharpDevelop	531
Добавление инструмента для запуска отладчика	531
Запуск отладчика из SharpDevelop	532
Отсутствующие возможности отладчика	534
Программирование на C# в TextPad	534
Создание класса документов .CS для языка C#	537
Добавление собственных инструментов: Build C# Debug	538
Настройка инструмента для компиляции финальной версии	540
Объяснение опций настройки инструментов Debug и Release	541
Работа над ошибками компиляции	545
Настройка остальных инструментов	545
Тестирование с помощью программы NUnit	548
Запуск программы NUnit	548
Тестирование	549
Написание тестов NUnit	550
Исправление ошибок в проверяемой программе	557
Написание исходного текста Windows Forms без Form Designer	559
Это всего лишь код	559
Работа в стиле визуального инструмента	560
Частичные классы	561
Самостоятельное написание	562
Убедитесь, что пользователи смогут запустить вашу программу	563
Visual Studio для бедных	564
Предметный указатель	565

Часть VI. Великолепные десятки	373
Глава 16. Десять наиболее распространенных ошибок компиляции	375
The name 'memberName' does not exist in the class or namespace 'className'	375
Cannot implicitly convert type 'x' into 'y'	377
'className.memberName' is inaccessible due to its protection level	379
Use of unassigned local variable 'n'	380
Unable to copy the file 'programName.exe' to 'programName.exe'. The process cannot...	380
'subclassName.methodName' hides inherited member 'baseclassName.methodName'.	
Use the new keyword if hiding was intended	381
'subclassName' : cannot inherit from sealed class 'baseclassName'	382
'className' does not implement interface member 'methodName'	383
'methodName' : not all code paths return a value	383
} expected	384
Глава 17. Десять основных отличий C# и C++	385
Отсутствие глобальных данных и функций	386
Все объекты размещаются вне кучи	386
Переменные-указатели запрещены	387
Обобщенные классы C# и шаблоны C++	387
Никаких включаемых файлов	388
Не конструирование, а инициализация	388
Корректное определение типов переменных	389
Нет множественного наследования	389
Проектирование хороших интерфейсов	389
Унифицированная система типов	389
Часть VII. Дополнительные главы	391
Глава 18. Эти исключительные исключения	393
Старый способ обработки ошибок	393
Возврат индикатора ошибки	395
Чем плохи коды ошибок	398
Использование механизма исключений для сообщения об ошибках	400
Пример	402
Создание собственного класса исключения	405
Использование нескольких catch-блоков	406
Как исключения протекают сквозь пальцы	408
Регенерация исключения	411
Как реагировать на исключения	412
Перекрытие класса Exception	413
Глава 19. Работа с файлами и библиотеками	419
Разделение одной программы на несколько исходных файлов	419
Разделение единой программы на сборки	421
Объединение исходных файлов в пространства имен	422
Содержание	13

Об авторах

Стефан Р. Дэвис (Stephen R. Davis) (более известный по второму имени — Ренди) живет со своей женой и сыном недалеко от Далласа, штат Техас. Он и его семейство написали множество книг, включая *C++ для чайников* (*C++ For Dummies*) и *C++ Weekend Crash Course*. Стефан работает в фирме L-3 Communications.

Чак Сфер (Chuck Sphar) ушел из подразделения Microsoft, работающего над документацией по языку C++, в 1997 году после шести лет тяжелой работы главным техническим писателем. Две его последние публикации были посвящены объектно-ориентированному программированию для Mac и библиотеке классов MFC. В настоящее время он заканчивает роман о древнем Риме (againstromerome.com) и работает с программированием в среде .NET. Пожелания и мелкие замечания можно отсылать Чаку по адресу csphar@chucksphar.com.

Посвящение

Пэм (Рат) и маме — Чак Сфер.

Благодарности

Я хотел бы поблагодарить Клодет Мур (Claudette Moore) и Дебби Маккенна (Debbie McKenna), которые заставили меня написать эту книгу. Я также хочу поблагодарить Ренди Дэвиса (Randy Davis) за его готовность передать своего "младенца" парню, которого он не знал. Я считаю, что это очень тяжело, и надеюсь, что был достаточно корректен, дополняя и расширяя первое издание его превосходной книги.

Должен также выразить благодарность прекрасным людям в издательстве Wiley, и в частности редактору Кейти Фелтман (Katie Feltman) и редактору проекта Киму Даросетту (Kim Darosett). Ким сумел поддержать меня в новой ипостаси — автора *для чайников*. Я также хотел бы поблагодарить Криса Боуера (Chris Bower) за его техническую консультацию и превосходное знание языка C#, Джона Эдвардса (John Edwards), которому книга обязана целостностью и согласованностью, а также художникам, специалистам по рекламе и другим людям, создавшим из моих файлов реальную книгу.

Выражаю сердечную благодарность Пэм за ее постоянную поддержку и помощь (much enabling). Она мне помогает во всем.

Чак Сфер.

Благодарности издательства

Издательский дом "Вильяме" благодарит Ерофеева Сергея и Кушенко Сергея за большой вклад в подготовку издания книги.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Информация для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

Введение

Язык программирования C# является мощным, относительно новым потомком более ранних языков C, C++ и Java. Программирование на нем доставляет много удовольствия, в чем можно будет убедиться при работе с этой книгой.

Язык C# был создан Microsoft как главная часть ее инициативы .NET. Возможно, из соображений политики компания Microsoft направила спецификации языка C# в комитет международных стандартов ассоциации ECMA (произносится как "эк-ма") летом 2000 года, задолго до внедрения платформы .NET. Теоретически любая компания может разработать свою собственную версию языка C#, предназначенную для работы в любой операционной системе и на любой машине, большей, чем калькулятор.

Когда вышло первое издание настоящей книги, компилятор языка C# Microsoft был единственным, и ее инструментальный пакет Visual Studio .NET предлагал единственную возможность программирования на языке C#. С тех пор, однако, Visual Studio претерпел два существенных изменения — появилась версия Visual Studio 2003 и, совсем недавно, Visual Studio 2005. И, по крайней мере, еще два игрока вступили в игру C#.

В настоящее время можно писать и компилировать программы на языке C# на множестве машин, работающих под управлением Unix, при помощи реализаций Mono или Portable .NET платформы .NET и языка C#.



✓ Mono (www.go-mono.com) является программным проектом с открытым исходным кодом, финансируемым компанией Novell Corporation. Версия 1.1.8 вышла в июне 2005 года. Хотя проект Mono и отстает от платформы .NET компании Microsoft (версию 1.1 Microsoft выпустила пару лет назад), он быстро развивается.

✓ Проект Portable .NET фирм Southern Storm Software и DotGNU (www.dotgnu.org/pnet.html) также является проектом с открытым исходным кодом. Во время написания этой книги текущей версией проекта Portable .NET была 0.7.0.

Оба проекта предназначены для выполнения программ C# в Windows и различных операционных системах семейства Unix, включая Linux и Macintosh компании Apple. Когда писалась эта книга, проект Portable .NET работал на большем количестве платформ, в то время как проект Mono гордится более полной реализацией платформы .NET. Так что выбор между ними может быть затруднен, в зависимости от вашего проекта, платформы и целей. (Книги по программированию для этих платформ уже становятся доступными. Посетите сайт www.amazon.com.)



Программное обеспечение с открытым исходным кодом создается сотрудничающими группами программистов-добровольцев и обычно является бесплатным для всех.

Переносимость языка C# и других языков платформы .NET выходит далеко за рамки настоящей книги. Но можно ожидать, что в течение нескольких лет программы C# для Windows, которые можно научиться создавать по этой книге, будут работать на различном аппаратном обеспечении и для всех типов операционных систем, что соответствует требованиям для языка Java компании Sun Microsystems — о работе на любой машине. Это, несомненно, хорошая вещь, даже для Microsoft. Переносимость — вопрос, над ко-

торым в настоящее время идет интенсивная работа, так что нет никаких сомнений, что все трудности и препятствия на пути к истинной универсальной переносимости языка C# будут преодолены. Но этот путь является уже не только путем Microsoft.

Однако в настоящий момент пакет Visual Studio компании Microsoft содержит наиболее развитые версии языка C# и платформы .NET, а также набор инструментов к ним с богатыми возможностями для программирования.



Если вам нужен только C#, то в одной из дополнительных глав вы узнаете, как практически бесплатно написать код C#. (Вы потеряете множество удобств, включая отличные средства визуального дизайна, обеспечиваемые Visual Studio 2005, но сможете создавать код Windows и без них — в особенности такой простой код, как рассматриваемый в этой книге.)

Что нового в C# 2.0

Хотя в версию 2.0 языка C# был внесен ряд изменений, в основном C# остается практически таким же, как и в предыдущей версии. В этой книге рассматриваются следующие значительные нововведения.

- S Блоки итераторов:** *итератор* представляет собой объект, который позволяет пройти по всем элементам *набора*, или *коллекции* объектов. Сделать это можно было всегда, но C# 2.0 значительно упрощает использование итераторов. Коллекции рассматриваются в главе 15, "Обобщенное программирование".
- S Обобщенное программирование** является важным нововведением! Новые возможности позволяют создавать более обобщенный и гибкий код, что является мечтой любого программиста. Из главы 15, "Обобщенное программирование", вы узнаете, как создавать более простой код с улучшенной *безопасностью типов* с помощью обобщенного программирования.

Оставляя в стороне более сложные нововведения, следует упомянуть о нескольких более простых элементах в соответствующих разделах книги. (Не нужно беспокоиться, если что-то в этом *Введении* вам непонятно. Все станет ясно в процессе чтения соответствующих глав книги.)

Об этой книге

Цель книги заключается в объяснении языка программирования C#, но для реального написания программ необходима специальная среда кодирования. Мы уверены, что большинство читателей будет использовать Microsoft Visual Studio, хотя предусмотрены и другие варианты. Основывая книгу на Visual Studio, мы попытались сделать долю Visual Studio минимально необходимой. Можно было бы просто сказать: "Запускайте свою программу каким угодно образом", но вместо этого мы говорим: "Запускайте свою программу C# в Visual Studio нажатием клавиши <F5>". Мы хотим, чтобы вы могли сосредоточиться на самом языке C#, а не на том, как работают простые вещи.

Вполне понятно, что многие читатели, если не большинство, захотят использовать C# для создания графических приложений Windows, поскольку язык C# является мощным средством разработки подобных программ, но это всего лишь одна из областей

применения C#. Данная же книга должна в первую очередь обращать внимание на C#, как на язык программирования. Графические программы Windows будут кратко рассмотрены в первой главе, но вы должны хорошо понять основы C#, прежде чем переходить к программированию для Windows. Также вполне понятно, что некоторые опытные пользователи будут применять C# для создания сетевых, распределенных приложений. Однако из-за издательских ограничений невозможно включить эту тему в данную книгу. В книге *C# для чайников* распределенное программирование не рассматривается. В ней совсем кратко рассмотрена платформа .NET — по той простой причине, что могущество языка C# во многом исходит из библиотек классов .NET Framework, которые используются этим языком.

Что необходимо для чтения книги

Для того чтобы просто запустить программы, сгенерированные C#, нужна, как минимум, *общезыковая исполняющая среда* (Common Language Runtime — CLR). Visual Studio 2005 копирует систему CLR на вашу машину во время процедуры установки. В качестве альтернативы можно загрузить весь пакет .NET, включая компилятор языка C# и множество других полезных инструментов, зайдя на Web-сайт компании Microsoft по адресу <http://msdn.microsoft.com>. Ищите там *набор инструментальных средств разработки программного обеспечения .NET* (Software Development Toolkit — SDK).



Большинство программ, приведенных в этой книге, можно при необходимости создавать и в среде Visual Studio 2003. Исключениями являются программы, содержащие новые возможности, доступные только в языке C# 2.0, прежде всего обобщения и блоки итераторов. Имеется также более дешевая версия системы Visual Studio 2005 — C# Express 2005, и другие недорогие альтернативы, рассматриваемые в дополнительных главах.

Как использовать книгу

При создании настоящей книги авторами преследовалась цель сделать ее максимально легкой в использовании, поскольку изучение нового языка и так достаточно трудное. Зачем же излишне его усложнять? Книга разделена на части. В первой части представлено введение в программирование на C# с использованием Visual Studio. В ней пошагово излагается создание двух различных типов программ. Авторы настоятельно рекомендуют начать с этой части и прочесть данные две главы, прежде чем перейти к другим частям книги. Даже если вы программировали ранее, базовая структура программы, созданная в первой части, постоянно применяется во всей книге.

Главы в частях со второй по пятую являются самостоятельными. Они написаны так, чтобы можно было открыть книгу на любой из них и начать чтение. Если вы новичок в программировании, то должны полностью прочесть вторую часть, прежде чем идти далее. Но если просто возвращаетесь назад, чтобы освежить свою память по некоторой определенной теме, у вас не возникнет никаких проблем при переходе к разделу, и вам не нужно будет повторно перечитывать предыдущие 20 страниц.

И, конечно же, книгу завершают традиционная часть о "великолепных десятках" и дополнительные главы; много интересного можно найти и на компакт-диске, прилагаемом к книге.

Структура книги

Ниже приведено краткое описание каждой части книги.

Часть I, "Создание ваших первых программ на C #"

В этой части шаг за шагом рассматривается написание минимального графического приложения Windows с использованием интерфейса Visual Studio 2005. В ней также показывается, как создать базовую структуру консольной программы C#, которая используется в других частях книги.

Часть II, "Основы программирования в C #"

На базовом уровне пьесы Шекспира — это всего лишь набор слов, связанных вместе. С этой же точки зрения 90% любой программы C#, которую вы когда-либо напишете, состоит из создания переменных, выполнения арифметических действий и управления ходом выполнения программы. Во второй части внимание уделяется этим основным операциям.

Часть III, "Объектно-основанное программирование"

Одно дело — объявлять переменные где-либо в программе, добавлять и убирать их. И совсем другое — создавать реальные программы для реальных людей. В третьей части внимание уделяется тому, как организовывать данные так, чтобы их было легче использовать при создании программы.

Часть IV, "Объектно-ориентированное программирование"

Вы можете соединять части самолета так, как пожелаете, но пока вы не сложите их правильно, у вас нет ничего, кроме кучи деталей. И только когда вы соберете самолет так, что сможете запустить двигатели и использовать подъемную силу крыла — только тогда вы сможете лететь куда угодно.

В четвертой части объясняется, как превратить набор данных в реальный объект — объект, который имеет внутренние члены и может моделировать свойства реальных вещей. В этой части представлена сущность объектно-ориентированного программирования.

Часть V, "За базовыми классами"

После того как самолет оторвется от земли, он должен куда-нибудь лететь. Изучение классов и основ объектно-ориентированного программирования — это только начало. В данной части сделан следующий шаг: в ней представлены структуры, интерфейсы и обобщения, открывающие доступ к более мощным объектно-ориентированным концепциям.

Часть VI, "Великолепные десятки"

Язык C# силен в поиске ошибок в ваших программах — иногда кажется, что он даже слишком хорошо указывает на недостатки. Однако верите вы в это или нет, но C# все же пытается принести вам пользу. Каждая проблема, им обнаруженная, могла бы привести к другим проблемам, которые вам пришлось бы находить и локализовывать самостоятельно.

К сожалению, сообщения об ошибках могут сбивать с толку. В одной из глав этой части представлено десять наиболее общих сообщений об ошибках C# времени компиляции, их значение, и как от них избавиться.

Многие читатели переходят в C# из других языков программирования. Во второй главе этой части описаны десять основных отличий между C# и его предком C++.

О прилагаемом CD-ROM

На прилагаемом компакт-диске содержится масса хороших вещей. Прежде всего на нем можно найти все исходные тексты из этой книги. Кроме того, на компакт-диске содержится набор полезных утилит. Утилита SharpDevelop не рекомендуется для полномасштабной разработки коммерческих программ, но она весьма полезна для написания небольших приложений или быстрого внесения изменений, чтобы не ждать, пока загрузится Visual Studio. Она полностью подходит для компиляции всех исходных текстов данной книги. Редактор TextPad представляет собой существенно усиленную версию стандартного Блокнота. Он предоставляет прекрасную дешевую платформу для программирования на C#. Инструмент тестирования NUnit, очень популярный среди программистов на C#, проводит проверку вашего кода легче, чем из Visual Studio, SharpDevelop или TextPad. Не пренебрегайте компакт-диском и имеющимися на нем программами.

И, конечно, не забудьте о файле **ReadMe**, содержащем всю наиболее свежую информацию.

Пиктограммы, используемые в книге

В книге используются следующие пиктограммы для выделения важной информации.



Этой пиктограммой помечен технический материал, который можно пропустить при первом чтении.



Данной пиктограммой выделены места, которые могут сохранить много вашего времени и усилий.



Это необходимо запомнить, так как это важно.



Это также следует запомнить. Иначе оно настигнет вас тогда, когда вы меньше всего ожидаете, и создаст одну из действительно трудно находимых ошибок.



Данная пиктограмма указывает код, который можно найти на прилагаемом к этой книге компакт-диске. Эта возможность предназначена, чтобы избавить вас от лишнего набора, если ваши пальцы начали дрожать. Но не злоупотребляйте ею — вы лучше поймете C#, если будете набирать текст программ самостоятельно.

Соглашения, используемые в книге

Чтобы помочь вам, в книге используется несколько соглашений. Термины, которые не являются "настоящими словами", такие как имена переменных, напечатаны **таким шрифтом**. Листинги программ выделены из текста следующим образом:

```
use System;
namespace MyNamespace
{
    public class MyClass
    {
    }
}
```

Каждый листинг сопровождается ясным и понятным пояснением. Полные исходные тексты программ помещены на прилагаемый компакт-диск, в отличие от небольших фрагментов.

Наконец, вы увидите стрелки, как, например, во фразе: "Выберите команду меню File^Open With^Notepad". Это означает, что необходимо выбрать меню File. Затем из появившегося раскрывающегося меню выбрать Open With, и наконец, из следующего подменю выбрать Notepad.

Что делать дальше

Очевидно, что первым шагом должно быть изучение языка C# (в идеале используя для этого книгу *C# 2005 для чайников*, конечно). Вы можете потратить несколько месяцев на написание простых программ C#, прежде чем сделать следующий шаг — освоить создание приложений Windows. Вам придется потратить еще много месяцев на приложения Windows, прежде чем вы начнете создавать программы, предназначенные для распространения через Интернет.

Тем временем вы можете поддерживать свои знания языка C# несколькими способами. Прежде всего, обратитесь к официальному источнику <http://msdn.microsoft.com/msdn>. Кроме того, на различных Web-сайтах для программистов имеется обширный материал по языку C#, включая живые обсуждения разных вопросов — от того, как сохранить исходный файл, и до сравнения свойств детерминистической и недетерминистической сборки мусора. Вот список нескольких больших сайтов по C#:

- **S** <http://msdn.microsoft.com>, который направит вас на соответствующие сайты групп разработчиков, включая C# и платформу .NET;
- S** <http://blogs.msdn.com/csharpfaq>, блог "Часто задаваемые вопросы по C#";
- S** <http://msdn.microsoft.com/vsharp/team/blogs>, который содержит личные блоги членов группы разработки C#;
- **S** www.cs2themax.com.

Один из авторов книги поддерживает Web-сайт www.chucksphar.com, содержащий ряд часто задаваемых вопросов (FAQ). Если вы столкнетесь с чем-то, чего не смо-

жете понять, попробуйте посетить этот сайт — возможно, в FAQ уже есть ответ на ваш вопрос. Кроме того, сайт содержит список ошибок, которые могли пробраться в книгу. И наконец — имеется в виду действительно крайний случай — вы можете найти ссылку на адреса электронной почты авторов и написать им, если не сможете найти ответ на свой вопрос на сайте.

Часть I

Создание ваших первых программ на C#



"Мы встретились в Интернете, и я с первого взгляда влюбилась в его синтаксис..."

В этой части...

Вы должны пройти длинный путь, прежде чем овладеете языком C#, так что немного отвлечитесь, прежде чем идти по нему. В первой части вы попробуете на вкус программирование графики Windows, пошагово создавая базовое приложение Windows при помощи интерфейса Visual Studio 2005. В этой части также будет показана разработка базовой структуры C# для демонстрационных программ, с которыми вы встретитесь в настоящей книге.

Глава 1

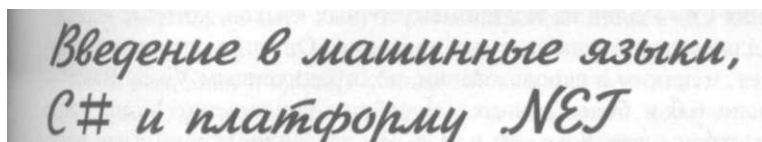
Создание вашей первой Windows-программы на C#

В этой главе...

- > Что такое программа? Что такое C#?
- > Создание Windows-программы
- > Настройка Visual Studio 2005



В этой главе будет немного рассказано о компьютерах, машинных языках, языке C# и Visual Studio 2005. Далее будет приведено пошаговое описание создания очень простой Windows-программы, написанной на C#.



Компьютер является удивительно быстрым, но невероятно глупым служащим. Компьютеры будут делать все, о чем их попросят (в разумных пределах, конечно), и сделают это чрезвычайно быстро, так как они постоянно становятся все быстрее и быстрее. Во время написания этих строк процессор обычного персонального компьютера может выполнять миллиард команд в секунду. Да, вы правильно поняли — именно "ард", а не "он".

К сожалению, компьютеры не понимают ничего похожего на человеческий язык. Вы, конечно, можете возразить: "Мой телефон позволяет позвонить моему другу, стоит мне всего лишь произнести его имя. А я знаю, что моим телефоном управляет крошечный компьютер. Значит, компьютер может говорить по-человечески". Но на самом деле ваши слова понимает компьютерная программа, а не сам компьютер.

Язык, который воспринимают компьютеры, называют *машинным языком*. Люди могут писать на нем, но это чрезвычайно трудно и приводит к частым ошибкам.



По историческим причинам машинный язык известен как ассемблер. В прошлом каждый изготовитель предоставлял программу, называемую ассемблером, которая преобразовывала специальные слова в отдельные машинные команды. Таким образом, вы могли бы написать нечто действительно загадочное, наподобие MOV AX, CX (между прочим, это реальная команда процессора Intel). Ассемблер преобразовал бы эту команду в шаблон битов, соответствующих единичной машинной команде.

сайте www.gotdotnet.com/team/lang). Однако C# является флагманским языком во флоте .NET. C# всегда будет первым языком, с помощью которого можно получить доступ к каждой новой возможности платформы .NET.



Платформа предыдущего поколения представляла собой смесь инструментов с загадочными названиями. Платформа .NET обновляет и объединяет их все в пакете Visual Studio 2005 с большей концентрацией на технологиях сети Интернет и баз данных, новейших версиях Windows и серверах .NET. Платформа .NET вместо частных форматов Microsoft поддерживает развивающиеся стандарты связи, такие как XML и SOAP. И, в заключение, платформа .NET поддерживает такую модную вещь, как *службы Web* (Web Services).

Что такое Visual Studio 2005? Visual C#?

Вы, безусловно, задаете очень много вопросов. Первым "визуальным" языком от Microsoft был Visual Basic, под кодовым названием Thunder ("Гром"). Первым популярным языком программирования, основанным на C, был Visual C++. Как и Visual Basic, он был назван "визуальным" из-за встроенного *графического интерфейса пользователя* (graphical user interface — GUI), который включил все, что необходимо для разработки отличных программ на C++.

В конечном итоге Microsoft упаковала все свои языки в единую среду — Visual Studio. Так как Visual Studio 6.0 начала немного устаревать, разработчики с нетерпением ожидали выхода седьмой версии пакета. Незадолго до выпуска Microsoft решила переименовать его в Visual Studio .NET, чтобы подчеркнуть связь новой среды разработки с платформой .NET.

На первый взгляд это звучало как маркетинговый ход, но при более тщательном рассмотрении оказалось, что пакет Visual Studio .NET отличался от своих предшественников совсем немного — но достаточно для того, чтобы обеспечить новое имя. Visual Studio 2005 является наследником исходного пакета Visual Studio .NET. Более мощные возможности пакета Visual Studio анализируются в дополнительных главах.



Компания Microsoft назвала свою реализацию языка Visual C#. Фактически, Visual C# является не более чем компонентом C# пакета Visual Studio. C# есть C#, независимо от того, входит он в Visual Studio или нет.

Хорошо, на этом все. Больше никаких вопросов.

Создание Windows-приложения на языке C#

Чтобы помочь вам с C# и Visual Studio, в этом разделе шаг за шагом рассматривается создание простой Windows-программы. Программы Windows обычно называются приложениями Windows, WinApps или приложениями WinForms для краткости.



Поскольку целью настоящей книги является рассмотрение языка C#, ее, по существу, нельзя считать ни книгой по Web-программированию, ни книгой по базам данных, ни книгой о программировании для Windows. В частности, визуальное программирование Windows Forms рассматривается только в этой главе. То есть вы всего лишь немного попробуете это на вкус.

Люди и компьютеры решили прийти к компромиссу. Программисты создают свои программы на языке, который не так свободен, как человеческая речь, но намного более гибок и легок в использовании, чем машинный язык. Такие языки, благодаря которым достигается компромисс (например, C#), называются компьютерными языками *высокого уровня*. (Хотя термин *высокий* является весьма относительным.)

Что такое программа?

Что такое программа? В известном смысле, программа Windows является исполняемым файлом, запускаемым двойным щелчком на его пиктограмме. Например, версия Microsoft Word, которая применялась для написания этой книги, является программой. Вы называете такую программу *исполняемой*. Имена исполняемых программных файлов обычно заканчиваются расширением `.exe`.

Но программа на самом деле — это нечто большее. Исполняемая программа состоит из одного или нескольких *исходных файлов*. Файл программы C# является текстовым файлом, содержащим последовательность команд C#, которые записываются вместе согласно правилам грамматики языка C#. Этот файл называют *исходным*, возможно, из-за того, что он служит источником расстройств и беспокойства программиста.

Что такое C#?

Язык программирования C# — один из тех промежуточных языков, которые используются программистами для создания исполняемых программ. Он занимает нишу между мощным, но сложным C++, и легким в использовании, но ограниченным Visual Basic — во всяком случае, в версии 6.0 и более ранних. (Новейшее воплощение Visual Basic .NET — язык, во многих отношениях похожий на C#. Но как лидирующий язык платформы .NET, именно C# имеет тенденцию первым представлять наиболее новые возможности.) Файл программы C# имеет расширение `.cs`.



Некоторые считают, что "до-дизз" — это то же, что и "ре-бемоль", но вы не должны называть этот новый язык таким именем — по крайней мере в пределах слышимости Редмонда, штат Вашингтон.

ку C# присущи следующие характеристики.



"ибкость: программы C# могут выполняться как на вашей машине, так и передаваться по сети и выполняться на удаленном компьютере.

Мощность: язык C# имеет фактически тот же набор команд, что и язык C++, но со сглаженными ограничениями.

Легкость в использовании: C# изменяет команды, ответственные за большинство ошибок в C++, так что вы потратите гораздо меньше времени на поиск этих ошибок.

визуальная ориентированность: библиотека кода .NET, применяемая языком C# для использования его возможностей, предоставляет помощь, необходимую для быстрого создания сложных визуальных форм с раскрывающимися списками, окнами с закладками, сгруппированными кнопками, полосами прокрутки и фоновыми изображениями.

Мужественность к Интернету: язык C# играет основную роль в системе .NET, которая является текущим подходом компании Microsoft к программированию для Windows и Интернета. .NET произносится как *dot-net*.

I / Безопасность: любой язык, предназначенный для использования в Интернете, должен включать серьезную защиту против злобных хакеров.

В заключение стоит отметить, что язык C# является неотъемлемой частью платформы .NET.

Что такое .NET?

Инициатива .NET появилась несколько лет назад в качестве стратегии Microsoft сделать всемирную сеть доступной простым смертным, таким как вы, например. Сегодня эта инициатива означает гораздо больше и включает в себя все, что делает Microsoft. В частности, она является новым способом программирования для Windows. Эта платформа предоставляет основанный на C язык — C#, а также простые визуальные инструменты, благодаря которым Visual Basic стал таким популярным. Краткое историческое описание поможет вам увидеть корни языка C# и платформы .NET.

Программирование для Интернета традиционно было очень трудным на более старых языках наподобие C и C++. Компания Sun Microsystems в ответ на эту проблему создала язык программирования Java. Для этого компания Sun взяла грамматику языка C++, сделала ее немного более дружелюбной и ориентировала на распределенную разработку.



Когда программисты говорят "распределенный", они имеют в виду географически рассредоточенные компьютеры, которые выполняют программы, общающиеся друг с другом — во многих случаях через Интернет.

Когда компания Microsoft занялась Java несколько лет назад, она столкнулась с компанией Sun на почве юриспруденции из-за изменений, которые она хотела сделать в языке. В результате Microsoft пришлось в какой-то степени отказаться от Java и начать искать способы конкурировать с этим языком.

Отказ от Java был к лучшему, потому что Java имел серьезную проблему: хотя он и является мощным языком, но вы должны написать вашу программу полностью на языке Java, чтобы получить все его преимущества. В Microsoft имеется достаточное количество разработчиков и написано слишком много миллионов строк исходного кода, так что компания Microsoft должна была придумать некоторый способ поддержки множества языков. Так появилась платформа .NET.

Платформа .NET представляет собой структуру, во многом сходную с библиотеками языка Java, поскольку язык C# подобен Java. Java является не только языком, но и обширной библиотекой кода. Точно так же и C# в действительности нечто намного большее, чем просто ключевые слова и синтаксис языка C#. Это еще и полностью объектно-ориентированная библиотека, содержащая тысячи программных элементов, упрощающих любой вид программирования, который только можно представить. Начиная с баз данных, ориентированных на работу в Интернете, и заканчивая криптографией и скромным диалоговым окном Windows.

Microsoft могла бы утверждать, что платформа .NET намного превосходит пакет Web-инструментов компании Sun, основанный на Java, но не в этом дело. В отличие от Java, в платформе .NET от вас не требуется переписывать уже имеющиеся программы. Программист на Visual Basic может добавить всего несколько строк, чтобы "познакомить" существующую программу с Web (это означает, что программа будет "знать", как получить данные из Интернета). Платформа .NET поддерживает все распространенные языки Microsoft и более сорока других языков, написанных третьими компаниями (самый последний список находится на

Кроме введения в Windows Forms, эта программа служит проверкой вашей среды Visual Studio. Это всего лишь тест. Если бы это действительно было программой для Windows... Впрочем, это и *есть* программа для Windows. Если вы сможете успешно написать, скомпоновать и выполнить эту программу, ваша среда Visual Studio настроена правильно, и вы готовы к созданию программ любой сложности.

Создание шаблона

Написание приложений Windows с нуля является, как известно, достаточно трудным процессом. С многочисленными дескрипторами и контекстами создание даже простой программы для Windows вызывает бесчисленные проблемы.

Visual Studio 2005 вообще и C# в частности значительно упрощают задачу по созданию базового приложения WinApp. Честно говоря, придется даже немного разочароваться, так как вы не будете с волнением создавать его вручную.

Поскольку Visual C# специально создан для работы в Windows, он может защитить от многих сложностей написания программ для Windows с нуля. Кроме того, Visual Studio 2005 включает в себя мастер приложений (Application Wizard), который формирует шаблоны программ.

Обычно *шаблоны программ* фактически ничего не делают — по крайней мере, ничего полезного. Однако они избавляют от начальных трудностей. Некоторые шаблоны программ достаточно сложны. Вы будете поражены тем, насколько много возможностей имеет мастер приложений.

После завершения установки Visual Studio 2005 выполните следующие действия для создания шаблона.

1. Для запуска Visual Studio 2005 выберите команду меню **StartoAll Programs^ Microsoft Visual Studio 2005^ Microsoft Visual Studio 2005**, как показано на рис. 1.1.

После похрипывания процессора и поскрипывания диска перед вами появится рабочий стол Visual Studio. Теплее, уже теплее...

2. Выберите в меню команду **File^New^Project**, как показано на рис. 1.2.
3. Visual Studio откроет диалоговое окно New Project, как продемонстрировано на рис. 1.3.



Проект является набором файлов, которые компонуются пакетом Visual Studio для создания единой программы. Вы будете создавать исходные файлы C#, имеющие расширение .CS. Расширение файла проекта — CSPROJ.

4. В панели **Project Types** выберите **Visual C#**, подпункт **Windows**. В панели **Templates** щелкните на пиктограмме **Windows Application**.

Если вы сразу же не увидите пиктограмму требуемого шаблона, не волнуйтесь. Возможно, необходимо немного прокрутить ползунок в панели Templates.

Пока не щелкайте на кнопке ОК.

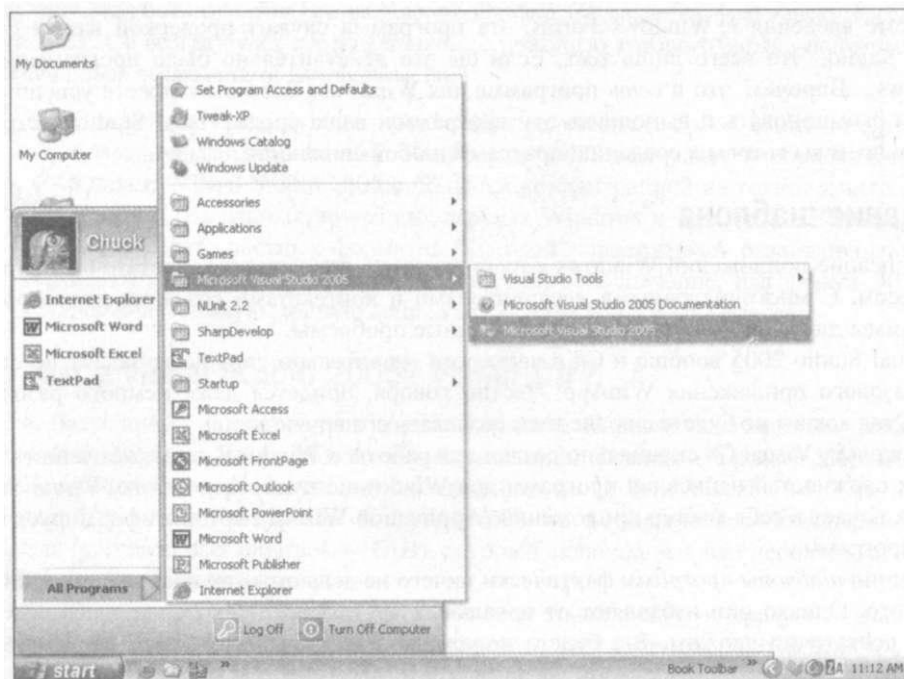


Рис. 1.1. Вот такую сеть мы плетем, когда задумываем написать программу на C#

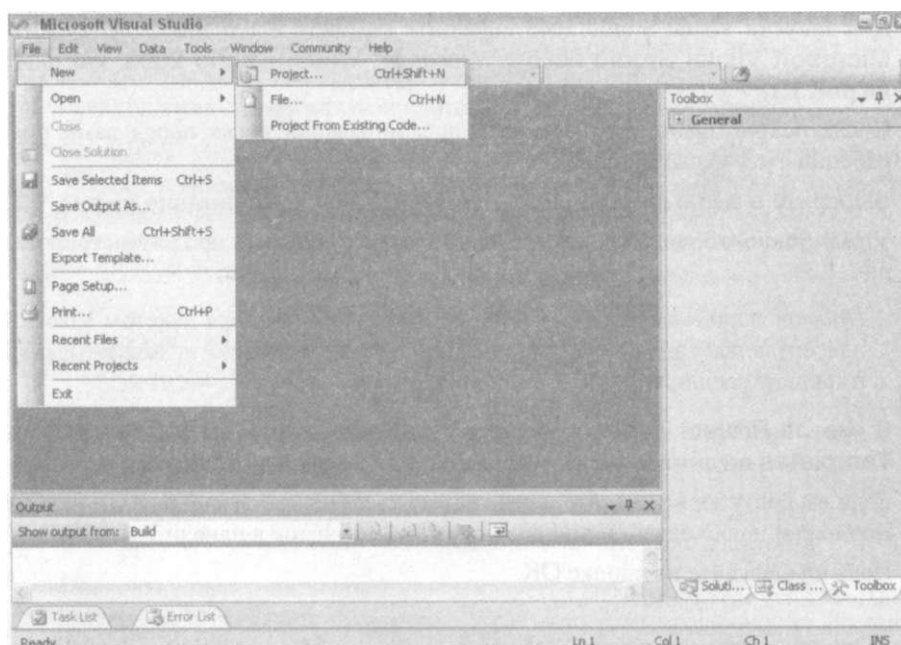


Рис. 1.2. Ваш путь к любому приложению Windows начинается с создания нового проекта

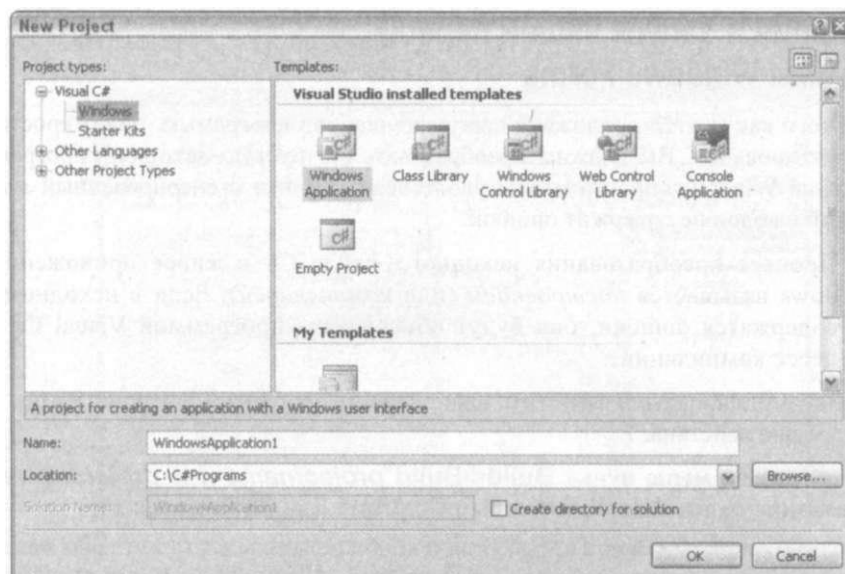


Рис. 1.3. Мастер приложений Visual Studio только и ждет, чтобы создать для вас программу Windows

5. В строке ввода Name введите название своего проекта (или оставьте предложенное по умолчанию).

Мастер приложений создаст папку, в которой сохранит различные файлы, включая первоначальные исходные файлы проекта C#. В качестве имени этого каталога мастер приложений использует название, которое вы ввели в поле Name. Исходным названием по умолчанию является **WindowsApplication1**. Если вы уже создавали новый проект, то начальным названием может быть **WindowsApplication2**, **WindowsApplication3** и т.д.

В данном примере можно использовать предложенные по умолчанию название и расположение для этого нового каталога: **My Documents\Visual Studio Projects\WindowsApplication1**. Я помещаю свои реальные программы там же, но для настоящей книги было изменено заданное по умолчанию расположение на более короткий путь к файлу. Чтобы изменить заданное по умолчанию расположение, выберите команду меню **Tools^Options^Projects and Solutions^General**. Укажите новое расположение— **C:\C#Programs** для этой книги — в окне **Visual Studio Projects Location** и щелкните на кнопке **OK**. (Вы можете одновременно создать новый каталог в диалоговом окне **Project Location**. Щелкните на пиктограмме папки с маленьким солнышком в верхней части диалогового окна. Каталог уже может существовать, если вы устанавливали примеры программ с компакт-диска).

6. Щелкните на кнопке **OK**.

Индикатор диска несколько секунд помигает, прежде чем в центре экрана откроется форма **Form1**.

Компиляция и запуск вашей первой программы Windows Forms

После того как мастер приложений загрузит шаблон программы, она откроется в режиме проектирования. Вы должны преобразовать эту пустую исходную программу C# в приложение Windows, просто чтобы удостовериться, что сгенерированный мастером приложений шаблон не содержит ошибок.



Процесс преобразования исходного файла C# в живое приложение Windows называется *построением* (или *компиляцией*). Если в исходном файле содержатся ошибки, они будут обнаружены программой Visual C# в процессе компиляции.

Чтобы скомпилировать и запустить вашу первую программу Windows Forms, выполните следующие действия.

1. Выберите в меню пункт **Build<=>Build projectname** (где *projectname*— это название наподобие **WindowsApplication1** или **MyProject**)

Должно открыться окно Output. Если оно не открылось, вы можете при желании это сделать до начала компиляции. Выберите пункт View^Other Windows^Output. Затем Build. В окне Output прокручивается ряд сообщений. Последнее сообщение должно быть следующего вида: **Build: 1 succeeded, 0 failed, 0 skipped** (или чем-то очень похожим на это). Это компьютерный эквивалент выражения "все в порядке". Если вы не используете окно Output, то должны увидеть сообщение **Build succeeded** или **Build failed** в статусной строке прямо над меню Start.

На рис. 1.4 показано, на что похожа программа Visual Studio после компиляции программы Windows, завершенной с окном Output. Не беспокойтесь насчет расположения окон. Вы можете перемещать их так, как вам необходимо. Важными составляющими являются окна Forms Designer и Output. Вкладка окна проектирования обозначена "**Form1.cs [Design]**".

2. Теперь вы можете запустить программу, выбрав в меню пункт **Debugs Start Without Debugging**.

Программа запустится и откроет окно, которое выглядит точно так же, как и окно Forms Designer, что проиллюстрировано на рис. 1.5.



В терминах языка C# такое окно называется *формой*. Форма имеет границу и строку заголовка вдоль верхней границы с маленькими кнопками Minimize, Maximize и Close.

3. Щелкните на маленькой кнопке Close в верхнем правом углу рамки для завершения программы.

Вот видите! Программирование на C# не такое уж и трудное.

Эта начальная программа являлась проверкой установки вашего пакета. Если у вас все получилось, следовательно, ваш пакет Visual Studio находится в хорошей форме и готов к программам, рассматриваемым в оставшейся части книги.



Обновите свое резюме, упомянув в нем, что вы являетесь программистом приложений для Windows. В крайнем случае — одного приложения точно...

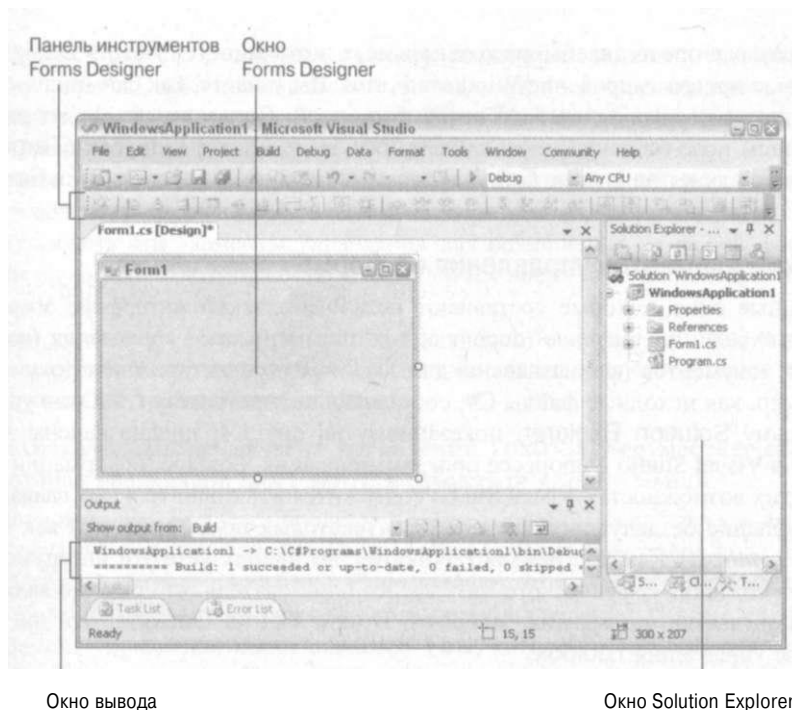


Рис. 1.4. Первоначальный шаблон программы Windows не очень впечатляет



Рис. 1.5. Шаблон приложения Windows работает, но он не убедит вашу супругу, что пакет Visual Studio 2005 стоит затраченных денег

Украшение программы

Заданная по умолчанию программа Windows не очень впечатляет, но вы можете немало ее улучшить. Вернитесь к Visual Studio и выберите окно со вкладкой **"Form1.cs [Design]"** (рис. 1.4). Это окно Forms Designer.

Forms Designer является мощным средством, дающим возможность "раскрасить" вашу программу в форме. Когда вы закончите, выберите команду меню Build, и Forms Designer создаст код C#, необходимый для построения приложения C#, с такой же красивой формой, как вы нарисовали.

В этом разделе представлены несколько новых возможностей Forms Designer, которые упрощают программирование Windows Forms. Вы узнаете, как скомпилировать приложение с двумя полями текстового ввода и кнопкой. Пользователь может ввести что-нибудь в одно поле (обозначенном как Source), но не может в другом (Target). Когда пользователь щелкнет на кнопке Copy, программа скопирует текст из поля Source в поле Target. Это все.

Размещение элементов управления на форме

Помеченные окна, которые составляют пользовательский интерфейс Visual Studio, называются *окнами документов* (document windows) и *окнами управления* (control windows). Окна документов предназначены для создания и редактирования документов, таких, например, как исходные файлы C#, составляющие программу C#. Окна управления, подобные окну Solution Explorer, показанному на рис. 1.4, предназначены для целей управления в Visual Studio в процессе программирования. Больше информации об окнах, меню и других возможностях Visual Studio содержится в дополнительных главах.

Все небольшие безделушки типа кнопок и текстовых полей известны как *элементы управления* (controls). Как программист Windows, вы используете эти инструментальные средства для создания графического интерфейса пользователя, что обычно является наиболее трудной частью программы Windows. В окне Forms Designer эти инструменты живут в окне управления Toolbox.

Если ваше окно Toolbox не открыто, выберите команду меню View¹^Toolbox. На рис. 1.6 показан пакет Visual Studio с открытым в правой части экрана окном Toolbox.

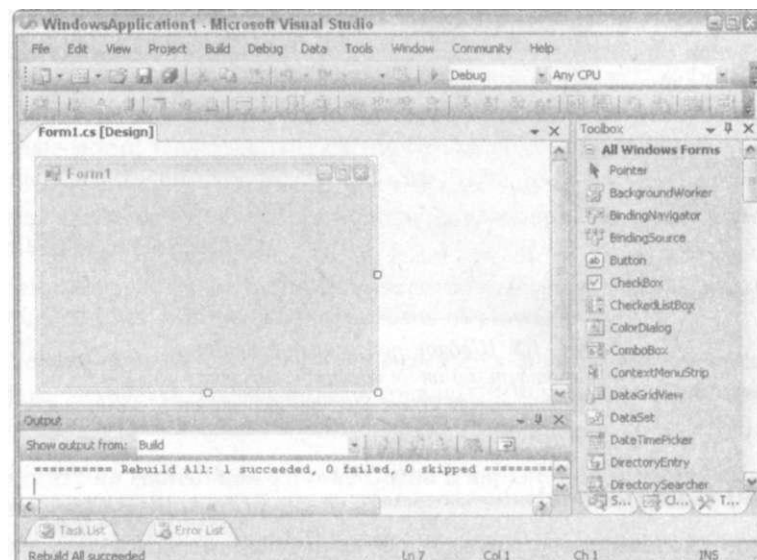


Рис. 1.6. Окно Toolbox битком набито интересными элементами управления



Не волнуйтесь, если ваши окна расположены не так, как на рис. 1.6. Например, ваше окно Toolbox может быть слева, справа или посередине. Вы можете перемещать любое из окон по рабочему столу как хотите. Как это делать, объясняется в дополнительных главах.

В окне **Toolbox** есть различные разделы, включая **Data**, **Components** и **Windows Forms**. Эти разделы, обычно называемые закладками, просто организуют элементы управления, чтобы вы не запутались в них. Окно **Toolbox** содержит множество элементов управления, к которым вы можете добавлять свои собственные.

Щелкните на знаке "плюс" рядом с надписью **Common Controls** (или **All Windows Forms**), чтобы открыть расположенные ниже элементы, как показано на рис. 1.6. Вы будете использовать эти элементы управления для размещения на вашей форме. Полоса прокрутки справа дает возможность перемещаться вверх и вниз по элементам управления, которые перечислены в окне **Toolbox**.

Элемент управления можно разместить где угодно на форме методом перетаскивания и опускания. Чтобы использовать окно **Toolbox** для создания двух текстовых полей и кнопки, нужно выполнить следующие действия.

1. **Захватите мышью элемент управления `Textbox`. переместите его на форму, которая обозначена как `Form1`, и отпустите кнопку мыши.**

Возможно, вам придется пролистать окно **Toolbox**. После того как вы перетащите требуемый элемент управления, на форме появится текстовое поле. В нем может содержаться текст `textBox1`. Это название, которое назначено мастером проектирования данному конкретному элементу управления. (Кроме свойства **Name**, элемент управления имеет свойство **Text**, которое не обязательно должно соответствовать свойству **Name**). Щелкая и перетаскивая углы текстового поля, можно изменять его размеры.



Текстовое поле можно сделать только шире. Его нельзя сделать более высоким, потому что по умолчанию эти текстовые поля являются однострочными. Небольшая указывающая вправо стрелка на текстовом поле позволяет изменить эту настройку, но не обращайте на нее внимания, пока не прочтете дополнительные главы.

2. **Снова захватите мышью элемент управления `Textbox` и поместите его под первым текстовым полем.**

Обратите внимание на появившиеся тонкие синие направляющие линии, которые помогают выровнять второе текстовое поле с другими элементами управления. Это отличная новая возможность.

3. **Теперь захватите мышью элемент управления `Button` и поместите его под двумя текстовыми полями.**

Под текстовыми полями появится кнопка.

4. **Изменяйте размеры формы и перемещайте элементы управления, используйте направляющие линии, пока форма не станет привлекательной.**

Полученная форма показана на рис. 1.7. Ваша форма может выглядеть несколько иначе.

Управление свойствами

Теперь самая важная проблема в вашем приложении — это обозначение кнопки. Название `button1` не очень наглядно. В первую очередь вы должны изменить именно его.



Рис. 1.7. Первоначальный вид формы

Каждый элемент управления имеет набор свойств, которые определяют его внешний вид и то, как он работает. Получить доступ к этим свойствам можно с помощью окна **Properties**. Для изменения свойств различных элементов управления выполните следующие действия.

1. Выберите кнопку, щелкнув на ней мышью.
2. Вызовите окно **Properties** с помощью команды меню **View ^ Properties Window**.

Кнопка имеет несколько наборов свойств: вверху перечислен набор свойств внешнего вида, ниже — свойства поведения и несколько других. Вы должны изменить свойство **Text**, которое находится в группе **Appearance**. (Чтобы отсортировать свойства в алфавитном порядке, а не по категориям, щелкните на обозначенной буквами **AZ** пиктограмме в верхней части окна.)

3. В окне **Properties** выберите поле в правой колонке рядом со свойством **Text**. Введите в него текст **Copy** и нажмите клавишу **<Enter>**.

Эти настройки проиллюстрированы на рис. 1.8 в окне **Properties** и в результирующей форме. Кнопка теперь помечена как **Copy**.

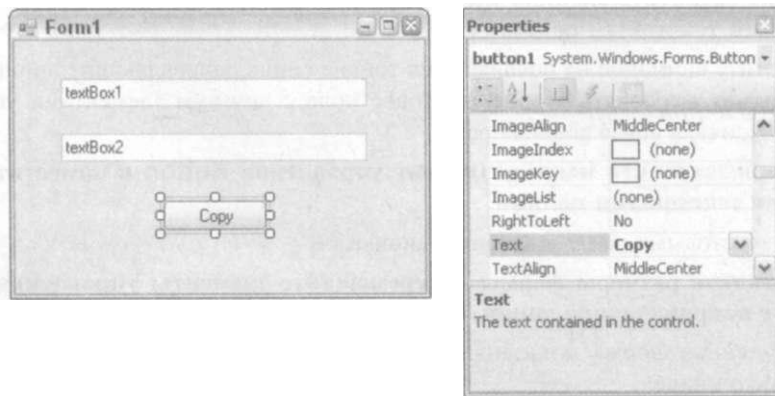


Рис. 1.8. Окно **Properties** позволяет управлять свойствами элементов

4. Измените начальное содержимое элементов управления **Textbox**. Выберите верхнее текстовое поле и повторите третий шаг, введя текст **Здесь пользователь вводит текст**. Сделайте то же самое для нижнего текстового поля, введя текст **Сюда программа скопирует текст**.

Благодаря этим надписям пользователь будет знать, что делать после запуска программы. Ничто не сбивает с толку пользователей сильнее, чем запутанное диалоговое окно.

5. Аналогично изменение свойства **Text** формы изменяет текст в заголовке окна. Щелкните где-нибудь на форме, введите новое название в свойство **Text** и нажмите клавишу **<Enter>**.

В данном примере установим название заголовка "Приложение копирования текста".

6. При изменении свойств формы щелкните на свойстве **AcceptButton** (в группе **Misc** окна **Properties**), а затем — в пустом месте справа от свойства **AcceptButton** для указания, какая кнопка должна реагировать, когда пользователь нажмет клавишу **<Enter>**. В данном случае выберите **button1**.

Надпись "Сору" является текстом на этой кнопке, но ее название остается **button1**. Его тоже можно изменить при желании. Это свойство **Name** элемента **Form**, т.е. свойство формы, а не кнопки.

7. Выберите нижнее текстовое поле и прокрутите группу свойств **Behavior**, пока не доберетесь до свойства **ReadOnly**. Установите его значение равным **True** путем выбора из раскрывающегося списка, как показано на рис. 1.9.

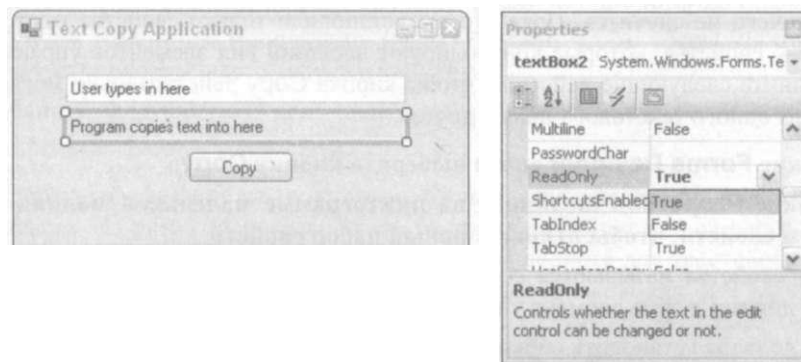


Рис. 1.9. Присвоение текстовому полю значения "только для чтения" не дает возможности редактировать это поле при выполнении программы

8. Щелкните на кнопке **Save** в панели инструментов **Visual Studio** для сохранения своей работы.



Во время работы пользуйтесь иногда кнопкой **Save**, чтобы не потерять слишком много, если ваша собака вдруг зацепит кабель питания компьютера. У несохраненных файлов изображена звездочка на вкладке в верхней части окна **Forms Designer**.

Компиляция приложения

Выберите команду меню **Build^Build Windows Application 1**, чтобы перекомпилировать приложение. Этот шаг скомпилирует новое приложение **Windows** с формой, которую вы только что создали. В окне **Output** вы должны увидеть сообщение **1 succeeded, 0 failed, 0 skipped**.

Теперь запустите программу, выбрав команду меню `Debug^Start Without Debugging`. Запущенная программа откроет форму, которая похожа на ту, которую вы редактировали, как показано на рис. 1.10. Вы можете вводить текст в верхнее текстовое поле, но не можете в нижнее (если вы не забыли изменить свойство `ReadOnly`).



Рис. 1.10. Окно программы выглядит так же, как и форма, которую вы только что скомпилировали

Учим форму трудиться

Программа выглядит правильно, но она ничего не делает. Если вы щелкнете на кнопке `Copy`, ничего не случится. Пока что вы установили только свойства из группы `Appearance`, — свойства, которые контролируют внешний вид элементов управления. Теперь выполните следующие действия, чтобы кнопка `Copy` действительно могла копировать текст из одного текстового поля в другое.

1. В окне **Forms Designer** снова выберите кнопку **Copy**.
2. В окне **Properties** щелкните на пиктограмме маленькой молнии над списком свойств, чтобы открыть новый набор свойств.

Эти свойства называются *событиями* элементов управления. Они определяют, что должен делать элемент управления при выполнении программы.

Вы должны установить событие `Click`. Оно указывает, что кнопка должна делать, когда пользователь щелкнет на ней. Это вполне логично.

3. Дважды щелкните на событии **Click** и посмотрите, как все изменится.

Режим `Design` — один из двух различных способов обзора приложения. Другим способом является режим `Code`, в котором показан исходный код `C#`, созданный незаметно для вас мастером проектирования. Пакет `Visual Studio` "знает", что для того, чтобы программа могла переносить текст, вы должны ввести код на языке `C#`.



Вместо пиктограммы молнии можно просто дважды щелкнуть на самой кнопке в окне `Forms Designer`.

При установке события `Click` экран в `Visual Studio` переключается в режим `Code` и создается новый *метод*. Этому методу присваивается описательное имя `button1_Click()`. Когда пользователь щелкнет на кнопке `Copy`, этот метод выполнит фактическое перемещение текста из источника `textBox1` в приемник `textBox2`.



Не нужно сильно беспокоиться о том, что такое метод. Методы описываются в главе 8, "Методы класса". Сейчас просто следуйте указаниям.

Данный метод просто копирует свойство `Text` из поля `textBox1` в поле `textBox2`.

4. Поскольку кнопка `button1` теперь обозначена "Сору", переименуйте метод с помощью команды меню **Refactor**. Дважды щелкните на названии `button1_Click` в окне `Code`. Выберите команду **Ref actors Rename**. В поле **New Name** введите `CopyClick`. Нажмите дважды клавишу `<Enter>` (следите при этом за полями в диалоговом окне).

Для текста элемента управления необходимо ясно отображать его цели.

Появившееся в пакете Visual Studio 2005 меню **Refactor** является самым надежным способом для выполнения некоторых модификаций кода. Например, ручное изменение названия метода `button1_Click` привело бы к потере ссылки на метод где-нибудь в коде, который Visual Studio сгенерировала для вас.

Второе диалоговое окно операции переименования показывает, что именно изменится: метод и любые ссылки на него в комментариях, текстовых строках или других местах в коде. Вы можете снять отметки в верхней панели, чтобы эти элементы не изменялись. В нижней панели **Preview Code Changes** можно увидеть, что и как изменится. Используйте меню **Refactor**, чтобы уберечься от большого количества работы, часто ведущей к ошибкам.

5. Добавьте следующую строку кода к методу `CopyClick()`:

```
textBox2.Text = textBox1.Text;
```



Обратите внимание на то, как язык C# пытается вам помочь в процессе ввода текста. На рис. 1.11 показан экран во время ввода последней части приведенной выше строки. Раскрывающийся список свойств для текстового поля дает возможность вспомнить, какие свойства доступны и как они используются. Эта функция автозавершения очень помогает во время программирования. (Если список автозавершения не появляется, нажмите комбинацию клавиш `<Ctrl+Space>`).

6. Выберите команду меню **Build^Build WindowsApplication1** для добавления к программе нового метода.

Проверка конечного продукта

Выберите команду меню **Debug>Start Without Debugging**, чтобы выполнить программу в последний раз. Введите какой-нибудь текст в исходное текстовое поле и затем щелкните на кнопке `Сору`. Текст волшебным образом скопируется в приемное текстовое поле, как показано на рис. 1.12. Весело повторяйте этот процесс, вводя какой угодно текст и копируя его, пока не устанете от этого.

Возвращаясь к процессу создания приложения, вы можете быть поражены тем, насколько это все основано на рисунках и работе с мышью. Захват элементов управления, размещение их в форме, установка свойств — и всего лишь одна строка кода C#!

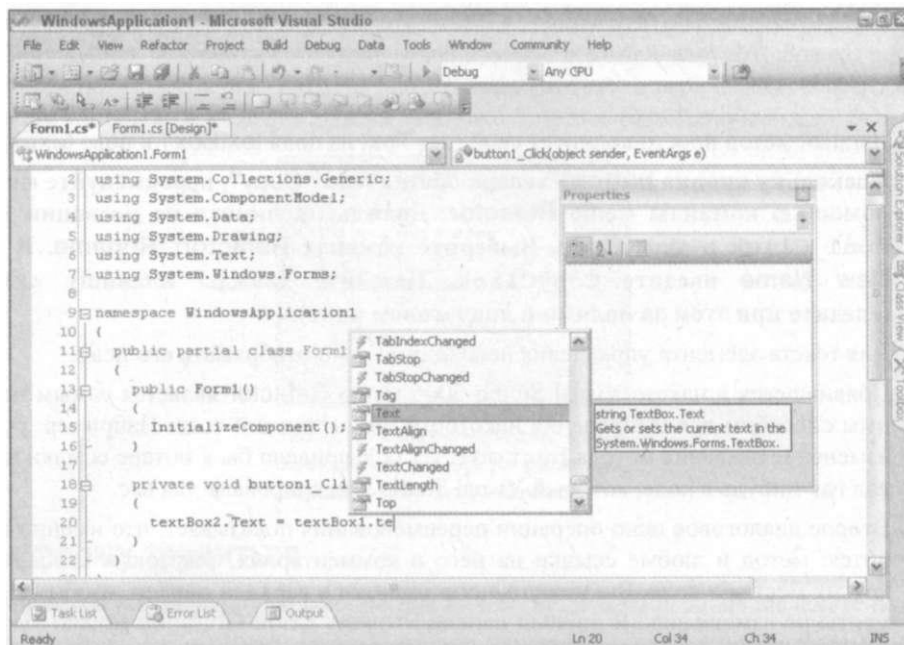


Рис. 1.11. Функция автозавершения отображает названия свойств во время набора текста

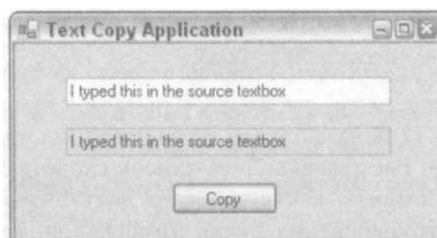


Рис. 1.12. Это работает!



Вы можете поспорить, что программа делает не так уж много, но позвольте с этим не согласиться. Посмотрите некоторые из более ранних книг по программированию для Windows, написанные до появления мастера приложений, и вы увидите, как много часов заняло бы написание даже такого простого приложения, как это.

Программисты на Visual Basic 6.0, берегитесь!

Для программистов на Visual Basic 6.0, которые есть среди вас, все это, возможно, кажется слишком приземленным. Действительно, Forms Designer работает почти так же, как и в поздних версиях Visual Basic. Однако .NET Forms Designer, применяемый языком Visual C#, намного более мощный, чем его двойник из Visual Basic 6.0. Платформа .NET и язык C# (а также и язык Visual Basic .NET в этом отношении) используют библиотеку подпрограмм .NET, которая является более мощной, обширной и целостной, чем старая библиотека Visual Basic. Кроме того, платформа .NET поддерживает разра-

ботку распределенных программ для сети, так же как и программ, применяющих множество языков, что не умеет Visual Basic. Но главное усовершенствование Forms Designer, используемого языками C# и Visual Basic .NET, по сравнению с предшественником Visual Basic 6.0, заключается в том, что весь код, сгенерированный для вас, является кодом, который можно легко изменять. В Visual Basic 6.0 вы должны были довольствоваться тем, что давал вам мастер проектирования.

Глава 2

Создание консольного приложения на C#

В этой главе...

- > Создание шаблона простого консольного приложения
- > Изучение шаблона простого консольного приложения
- > Составные части шаблона простого консольного приложения



Даже начинающие программисты на C# в состоянии писать программы для Windows. Не верите? Тогда обратитесь к главе 1, "Создание вашей первой Windows-программы на C#". Однако изучение основ C# лучше проводить, не отвлекаясь на графический пользовательский интерфейс, а создавая так называемые *консольные приложения*, для которых требуется писать существенно меньше кода и которые значительно проще понимать.

В этой главе Visual Studio понадобится для создания шаблона консольного приложения. Затем вручную этот шаблон будет несколько упрощен. Полученная в результате заготовка будет применяться для множества программ, рассматриваемых в данной книге.

Основное предназначение настоящей книги — помочь вам понять C#. Вы не сможете создать, например, красивую трехмерную игру без знания языка C#.

Создание шаблона консольного приложения



Описанные далее действия предусматривают использование Visual Studio. Если вы работаете не в Visual Studio, а в другой среде программирования, то должны обратиться к ее документации либо просто ввести исходный текст в вашей среде разработки C#.

Создание исходной программы

Выполните следующие шаги для создания консольного приложения C#.

1. Воспользуйтесь командой меню **File^New^Project** для формирования нового проекта.

Visual Studio откроет окно с пиктограммами, представляющими различные типы приложений, которые вы можете создать.

2. Выберите в окне **New Project** пиктограмму **Console Application** и щелкните на ней.



Убедитесь, что в панели **Project Types** вы выбрали **Visual C#** и **Windows**, иначе Visual Studio может создать неверный проект — например, приложения на языке программирования **Visual Basic** или **Visual C++**. Затем щелкните на пиктограмме **Console Application** в панели **Templates**.

Visual Studio требует создания проекта перед тем, как вы сможете начать вводить исходный текст вашей программы. Проект представляет собой что-то вроде корзины, в которой хранятся все файлы, необходимые для разработки программы. Когда вы дадите компилятору задание построить программу, он пересматривает эту корзину в поисках файлов, требуемых для сборки программы.

По умолчанию для вашего первого консольного приложения будет предложено имя **ConsoleApplication1**, но в этот раз измените его на **ConsoleAppTemplate**. В последующих главах книги вы сможете открывать шаблон, сохранять его под новым именем и сразу иметь все необходимое для начала работы.

По умолчанию место для хранения проектов находится в папке, спрятанной глубоко в папке **My Documents**. Лично я предпочитаю размещать свои программы не там, где меня заставляют, а там, где мне хочется. В главе 1, "Создание вашей первой Windows-программы на C#", было показано, как изменить место хранения проектов, предлагаемое по умолчанию, на **C:\C#Programs** (если вы хотите упростить себе работу с этой книгой).

2. Щелкните на кнопке **ОК**.

Немного пошуршав диском, Visual Studio сгенерирует файл **Program.cs**. (Если вы посмотрите в окно **Solution Explorer**, то увидите и другие файлы. Пока просто игнорируйте их существование. Если окно **Solution Explorer** отсутствует на экране, его можно вызвать командой **View>Solution Explorer**.) Расширение исходных файлов C# — **.CS**. Имя **Program** — это имя по умолчанию, присваиваемое файлу программы.

Содержимое вашего первого консольного приложения выглядит следующим образом:

```
using ...

namespace ConsoleAppTemplate
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```



Вдоль левой границы окна вы увидите несколько маленьких плюсов (+) и минусов (-) в квадратиках. Щелкните на знаке + возле **using...** Этим вы открываете область кода — эта весьма удобная возможность Visual Studio позволяет уменьшать неразбериху на экране, сворачивая области кода и пряча их долой с глаз программиста (но не компилятора!). После раскрытия области кода вы увидите такие директивы:

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

Области кода помогают сфокусироваться на коде, с которым вы работаете, скрывая код, который в данный момент не представляет интерес. Некоторые блоки кода, такие как блок пространств имен, блок классов, методов и т.п., получают значки +/- автоматически, без директивы **#region**. Вы можете включить в исходный текст собственные сворачиваемые области, добавляя директиву **#region** над интересующей частью кода, которую хотите иметь возможность сворачивать, и **#endregion** после нее. Это позволит дать имя вашей области, например, что-то вроде **Public methods**. Обратите внимание, что такие имена могут включать пробелы. Кроме того, области могут быть вложены одна в другую (еще одно преимущество над Visual Basic), но не могут перекрываться.

В настоящий момент вам нужна только одна директива **using System**. Можно убрать остальные; если вам будет не хватать какой-то из них, компилятор не преминет сообщить об этом.

Пробная поездка

Чтобы преобразовать исходный текст программы на C# в выполняемую программу, воспользуйтесь командой меню **Build^Build ConsoleAppTemplate**. Visual Studio ответит следующим сообщением:

```
-Build started: Project: ConsoleAppTemplate, Configuration:
  Debug Any CPU -
```

```
Csc.exe /noconfig /nowarn (and much more)
```

```
Compile complete -- 0 errors, 0 warnings
ConsoleAppTemplate -> C:\C#Programs\ ... (and more) == Build:
  1 succeeded or up-to-date, 0 failed, 0 skipped ==
```

Главное во всем этом — **1 succeeded** в последней строке.



Это общее правило в программировании: "succeeded" — это хорошо, "failed" — плохо.

Для запуска программы воспользуйтесь командой меню **Debug^Start Without Debugging**. Программа выведет на экран черное консольное окно и тут же завершится. Похоже, что она просто ничего не делает. Кстати, так оно и есть на самом деле. Шаблон — это всего лишь пустая оболочка.

Создание реального консольного приложения

Отредактируйте файл **Program.cs**, чтобы он выглядел следующим образом:

```
using System;

namespace ConsoleAppTemplate
{ // Фигурные скобки
```

```
// Класс Program — объект, содержащий наш код
public class Program
{
    // Это - начало программы
    // Каждая программа имеет метод Main()
    static void Main(string[] args)
    {
        // Немного кода, чтобы программа хоть что-то делала
        // Приглашение ввести имя пользователя
        Console.WriteLine("Пожалуйста, введите ваше имя:");
        // Считывание вводимого имени
        string sName = Console.ReadLine();

        // Приветствие пользователя с использованием введенного имени
        Console.WriteLine("Добрый день, " + sName);

        // Ожидание подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");

        Console.Read();
        // Код Main() на этом заканчивается
    } // Конец функции Main()
} // Конец класса Program
} // Конец пространства имен ConsoleAppTemplate
```



Не волнуйтесь о тексте, находящемся в программе после двойной или тройной косой черты (// или ///), и не беспокойтесь, если вы введете пару лишних пробелов или пустых строк. Однако обращайтесь внимание на строчные и прописные буквы.

Для того чтобы превратить исходный текст из файла **Program.cs** в выполняемую программу **ConsoleAppTemplate.exe**, примените команду меню **Builds Build ConsoleAppTemplate**.

Чтобы запустить программу из Visual Studio 2005, воспользуйтесь командой меню **Debug^Start Without Debugging**. При этом вы увидите черное консольное окно, в котором будет выведено приглашение ввести ваше имя (вам может потребоваться активировать окно, щелкнув на нем мышью). После того как вы введете свое имя, программа поприветствует вас и выведет надпись **Нажмите <Enter> для завершения программы...** Нажатие клавиши <Enter> приведет к закрытию окна.

Эту же программу можно выполнить из командной строки DOS. Для этого откройте окно DOS и введите следующее:

```
CD \C#Programs\ConsoleAppTemplate\bin\Debug
```

Теперь введите **ConsoleAppTemplate** для запуска программы. Вывод программы на экран будет точно таким же, как только что описано. Вы можете также перейти в папку **\C#Programs\ConsoleAppTemplate\bin\Debug** в Проводнике Windows и запустить программу двойным щелчком на файле **ConsoleAppTemplate . exe**.



Для того чтобы открыть окно DOS, попробуйте воспользоваться командой меню **Tools^Command Window**. Если эта команда недоступна в вашем меню Visual Studio Tools, воспользуйтесь командой меню **Starts All Programs^Microsoft Visual Studio 2005<=>Visual Studio Tools^Visual Studio 2005 Command Prompt**.

Изучение шаблона консольного приложения

В последующих подразделах будет рассмотрено, как работает созданное консольное приложение.

Схема программы

Базовая схема всех консольных приложений начинается со следующего кода:

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

```
namespace ConsoleAppTemplate
```

```
{  
    public class Program  
    {  
        // Стартовая точка программы  
        public static void Main(string[] args)  
        {  
            // Ваш код  
        }  
    }  
}
```

Программа начинает выполнение с первой строки, идущей после названия функции `Main()`, и заканчивается ее закрывающей фигурной скобкой. Пока что это все, что можно сказать по этому поводу.

Список директив **using** может находиться непосредственно до или после строки **namespace ConsoleAppTemplate** {. Порядок не имеет значения. В своем приложении вы можете использовать множество разных вещей из .NET. О том, что такое пространство имен и зачем нужна директива **using**, будет рассказано в одной из дополнительных глав.

Комментарии

Шаблон содержит массу строк, к которым добавляются другие строки, выглядящие следующим образом:

```
// Стартовая точка программы  
public static void Main(string[] args)
```

Первую строку этого фрагмента C# игнорирует — это строка *комментария*.



Любая строка, начинающаяся с символов `//` или `///`, представляет собой обычный текст, предназначенный для человека и полностью игнорируемый C#. Пока что `//` и `///` рассматриваются как эквивалентные символы начала комментария.

Зачем включать в программу строки, которые будут проигнорированы компилятором? Потому что комментарии помогают понять текст программы. Исходный текст — не

такая уж легкая для понимания штука. Помните, что язык программирования — это компромисс между тем, что понимает человек, и что понимает компьютер? Комментарии помогут вам при написании кода, а особенно тем (и это можете быть вы сами через какое-то время), кто будет заниматься вашей программой и пытаться понять ее логику. Добавление пояснений в программу сделает эту работу намного проще.



Не экономьте на комментариях. Комментируйте исходный текст сразу и часто. Это поможет вам и другим программистам легче разобраться, для чего предназначена та или иная инструкция C# в исходном тексте.

Тело программы

Ядро программы находится в блоке исходного текста, помеченного как **Main()**:

```
// Приглашение ввести имя пользователя
Console.WriteLine("Пожалуйста, введите ваше имя:");

// Считывание вводимого имени
string sName = Console.ReadLine();

// Приветствие пользователя с использованием введенного имени
Console.WriteLine("Добрый день, " + sName),•
```



Вы можете сэкономить массу времени, воспользовавшись при вводе новой возможностью — Code Snippets (фрагменты кода), которая облегчает ввод пространственных инструкций, наподобие **Console.WriteLine**. Нажмите комбинацию клавиш <Ctrl+K>, а затем <Ctrl+X> для появления раскрывающегося меню Code Snippets. Прокрутите его до **cw** и нажмите клавишу <Enter>. Visual Studio вставит в тело программы инструкцию **Console.WriteLine()** с точкой ввода между скобками.

Если у вас имеется ряд сокращений типа **cw**, **for** и **if**, которые легко запомнить, воспользуйтесь еще более быстрым методом: введите **cw** и нажмите клавишу <Tab>. (Попробуйте также выделить несколько строк исходного текста, и нажать клавиши <Ctrl+K>, а затем <Ctrl+S>. Выберите из списка **if** — и выделенный текст окажется внутри конструкции **if**.) Более того, вы можете даже создавать собственные фрагменты.

Программа начинает работу с первой инструкции C#: **Console.WriteLine**. Эта команда выводит на экран строку **Пожалуйста, введите ваше имя:.**

Следующая инструкция считывает вводимую пользователем строку и сохраняет ее в переменной с именем **sName** (о переменных будет рассказано в главе 3, "Объявление переменных-значений"). В последней строке выполняется объединение строк **Добрый день,** с введенным именем пользователя, а также вывод получившейся в результате объединения строки на экран.

Последние строки заставляют компьютер ожидать, пока пользователь не нажмет клавишу <Enter>. Эти строки обеспечивают приостановку выполнения программы, чтобы было время посмотреть на экране результаты ее работы:

```
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
```

В зависимости от того, как именно запущена программа, данный фрагмент может оказаться очень важным. В Visual Studio это можно сделать двумя способами. Если используется команда `Debug^Start`, Visual Studio закрывает окно программы сразу же по ее завершении. То же происходит и при запуске программы двойным щелчком на пиктограмме файла в Проводнике Windows.

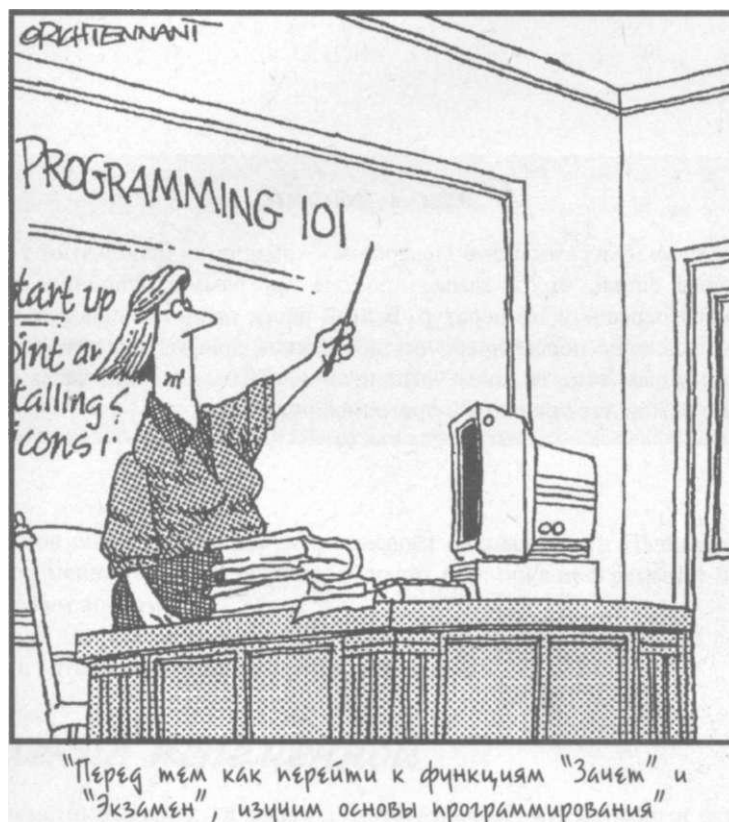


Вне зависимости от того, каким образом запускается программа, ожидание нажатия пользователем клавиши `<Enter>` перед завершением программы решает все проблемы.

Теперь вы можете удалить из шаблона строки от первого `Console.WriteLine` до предпоследнего, и получите пустой, чистый метод `Main ()` для использования в качестве шаблона для последующих консольных приложений. Но не убирайте последние инструкции `Console.WriteLine` и `Console.Read`. Они понадобятся вам в ваших консольных приложениях.

Часть II

Основы программирования в C



В этой части...

Новейшие программы для электронной коммерции используют те же базовые блоки, что и самые простые программы наподобие программы пересчета температур. В этой части представлены азы языка — создание переменных, осуществление арифметических операций и управление потоком выполнения программы. Эта часть особенно важна для новичков в программировании.

Глава 3

Объявление переменных-значений

В этой главе...

- > Создание места для хранения информации — переменные C#
- > Использование целых чисел
- > Работа с дробными числами
- > Объявление других типов переменных
- > Работа с числовыми константами
- > Изменение типов

n

Наиболее фундаментальной из всех концепций программирования является концепция переменной. Переменная C# похожа на небольшой ящик, в котором можно хранить разные вещи (в частности, числа) для последующего применения. Термин *переменная* пришел из мира математики. Например, математик может сказать следующее:

$n = 1$

Эта запись означает, что, начиная с этого момента, математик, используя *n*, подразумевает 1 — т.е. пока он не изменит *n* на что-то другое (число, уравнение и т.д.).

Значение термина *переменная* в программировании не сильно отличается от его значения в математике. Программист на C# может написать:

```
int n;  
n = 1;
```

Тем самым он определит "вещь" *n* и присвоит ей значение **1**. Начиная с этого места программы переменная *n* имеет значение **1** до тех пор, пока программист не заменит его некоторым другим числом.

К сожалению для программистов, C# накладывает ряд ограничений на переменные — ограничений, с которыми не сталкиваются математики.

Объявление переменной

Когда математик говорит: "*n* равно **1**", это означает, что термин *n* эквивалентен **1**. Математик свободен в своем выборе переменных — например, он может сказать или написать следующее:

$$x = y^2 + 2y + 1$$

$$\text{Если } k = y + 1, \text{ то } x = k^2$$

Здесь математик записал квадратное уравнение. Вероятно, переменные *x* и *y* были где-то определены ранее. Однако затем математик вводит еще одну переменную *k*, и дело не просто в том, что ее значение на **1** больше значения переменной *y*, нет, здесь *k* замещает собой поня-

тие "у плюс 1", представляя собой что-то вроде сокращенной записи. Если вы хотите разобраться в этом более детально и точно — обратитесь к учебникам по математике.

Программист должен быть гораздо педантичнее в использовании терминологии. Например, программист на C# может написать следующий код:

```
int n;  
n = 1;
```

Первая его строка означает: "Выделим небольшое количество памяти компьютера и назовем его *n*". Этот шаг аналогичен, например, абонированию почтового ящика в почтовом отделении и наклейке на него ярлыка. Вторая строка гласит: "Сохраним значение 1 в переменной *n*, тем самым заменяя им предыдущее хранившееся в ней значение". При использовании аналогии с почтовым ящиком это звучит как: "Откроем ящик, *n* выбросим все, что там было, и положим в него 1".



Знак равенства (=) называется *оператором присваивания*.



Математик говорит: "и равно 1". Программист на C# выражается более точно: "Сохраним значение 1 в переменной *n*". Операторы C# указывают компьютеру, что именно вы хотите сделать. Другими словами, операторы — это глаголы, а не существительные. Оператор присваивания берет значение справа от него и сохраняет его в переменной, указанной слева от него.

Что такое `int`

Математики работают с концепциями. Они могут в любой момент ввести любые переменные, которые только захотят, причем одна и та же переменная может иметь разное значение в одном и том же уравнении. В лучшем случае математики рассматривают переменную как некое аморфное значение, а в худшем — как некоторую расплывчатую концепцию.

Математик может написать следующее:

```
П = 1;  
П = 1.1;  
п = House;  
п = "Ну и глупость!"
```

Приведенные строки приравнивают переменную *n* к разнородным вещам, и математик об этом абсолютно не беспокоится.

C# и приблизительно не столь гибок. В нем каждая переменная имеет фиксированный тип. Когда вы абонируете почтовый ящик, вы выбираете ящик интересующего вас размера. Если вы выбрали ящик "для целых чисел", не стоит надеяться, что туда сможет поместиться строка.

В примере в предыдущем разделе вы выбрали ящик, созданный для работы с целыми числами — C# называет их `int`. Целые числа — это числа, применяющиеся для перечисления (1, 2, 3 и т.д.), а также 0 и отрицательные числа — 1, -2, -3...



Перед тем как использовать переменную, ее надо объявить. После того как вы объявили переменную как `int`, в нее можно помещать и извлекать из нее целые значения, что продемонстрировано в следующем примере:

```
// Объявляем переменную п  
int П;
```

```
// Объявляем переменную m и инициализируем ее значением 2
int m = 2;
// Присваиваем значение, хранящееся в t, переменной p
p = t,-
```

Первая строка после комментария является *объявлением*, которое создает небольшую область в памяти с именем *p*, предназначенную для хранения целых значений. Начальное значение *p* не определено до тех пор, пока этой переменной не будет *присвоено* некоторое значение. Второе объявление не только объявляет переменную *m* типа *int*, но и инициализирует ее значением 2.



Термин *инициализировать* означает присвоить начальное значение. Инициализация переменной заключается в первом присваивании ей некоторого значения. Вы ничего не можете сказать о значении переменной до тех пор, пока она не будет инициализирована.

Последняя строка программы присваивает значение, хранящееся в *m* (равное 2), переменной *p*. Переменная *p* будет хранить значение 2, пока ей не будет присвоено новое значение (в частности, она не потеряет свое значение при присваивании его переменной *t*).

Правила объявления переменных

Вы можете выполнить инициализацию переменной как часть ее объявления:

```
// Объявление переменной типа int с присваиванием ей
// начального значения 1
int o = 1;
```

Это эквивалентно помещению 1 в ящик *int* в момент его аренды, в отличие от его вскрытия и помещения туда 1 позже.



Инициализируйте переменные при их объявлении. Во многих, но не во всех, случаях C# инициализирует переменные за вас, но рассчитывать на это нельзя.

Вы можете объявить переменную в любом месте (ну, или почти в любом) программы. Однако вы не можете использовать переменную до того, как она будет объявлена, и присваивать ей какие-либо значения. Так, следующие два присваивания некорректны:

```
// Это присваивание неверно, поскольку переменной m не
// присвоено значение перед ее использованием
int m;
n = m;
// Следующее присваивание некорректно в силу того, что
// переменная p не была объявлена до ее использования
p = 2;
int p;
```

И последнее — нельзя *дважды* объявить одну и ту же переменную.

Вариации на тему int

Большинство простых переменных имеют тип *int*. Однако C# позволяет настраивать целый тип для конкретных случаев.

Все целочисленные типы переменных ограничены хранением только целых чисел, но! диапазоны этих чисел различны. Например, переменная типа `int` может хранить только! целые числа из диапазона примерно от -2 миллиардов до 2 миллиардов.

Два миллиарда сантиметров — это больше, чем диаметр Земли. Но если этой величи-ны вам не хватает, C# имеет еще один целочисленный тип, называемый `long` (сокращение от `long int`), который может хранить гораздо большие числа ценой уве-личения размера "ящика": он занимает 8 байт (64 бит) в отличие от 4-битового `int`.

В C# имеются и другие целочисленные типы, показанные в табл. 3.1.

Таблица 3.1. Размер и диапазон целочисленных типов C#			
Тип	Размер (байты)	Диапазон значений	Использование
<code>sbyte</code>	1	От -128 до 128	<code>sbyte sb = 12;</code>
<code>byte</code>	1	От 0 до 255	<code>byte b = 12;</code>
<code>short</code>	2	От -32 768 до 32 767	<code>short sn = 12345;</code>
<code>ushort</code>	2	От 0 до 65 535	<code>ushort usn = 62345;</code>
<code>int</code>	4	От -2 147 483 648 до 2 147 483 647	<code>int n = 1234567890;</code>
<code>uint</code>	4	От 0 до 4 294 967 295	<code>uint un = 3234567890U;</code>
<code>long</code>	8	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	<code>long l = 123456789012L;</code>
<code>ulong</code>	8	От 0 до 18 446 744 073 709 551 615	<code>ulong ul = 123456789012L;</code>

Как будет рассказано позже, фиксированные значения — такие как `1` — тоже имеют тип. По умолчанию считается, что простая константа наподобие `1` имеет тип `int`. Константы, отличные от `int`, должны явно указывать свой тип — так, например, `123U` (обратите внимание на `U`) — это константа типа `uint` — беззнакового целого.

Большинство целых значений — *знаковые* (signed), т.е. они могут представлять наряду с положительными и отрицательные значения. Беззнаковые (unsigned) целые числа могут представлять только неотрицательные значения, но зато их диапазон представления удваивается по сравнению с соответствующими знаковыми типами. Как видно из табл. 3.1, имена большинства беззнаковых типов образуются из знаковых путем добавления префикса `u`.

Представление дробных чисел

Целых чисел хватает для большинства вычислений. Я считал так до 6 класса¹ и даже не думал, что существуют какие-то другие числа. Я до сих пор не могу забыть свое потрясение в 6 классе, когда учительница рассказала о дробных числах.

Множество вычислений требуют применения дробных чисел, которые никак не могут быть точно представлены целыми числами. Общее уравнение для конвертации температуры в градусах Фаренгейта в температуру в градусах Цельсия демонстрирует это:

Напомним, что автор учился в американской школе. — *Примеч. пер.*

```
// Преобразование температуры 45°F
int nFahr = 41;
int nCelsius = (nFahr - 32) * (5/9);
```

Для некоторых значений данное уравнение работает совершенно корректно. Например, 41 °F равен 5°C. "Правильно, Девис!" — сказала бы мне учительница в 6 классе.

Попробуем теперь другое значение, например 100°F. Приступим к вычислениям: $100 - 32 = 68$; $68 (5/9)$ дает 37. "Нет, — сказала бы учительница, — правильный ответ — 37.78". И даже это не совсем верно, так как в действительности правильный ответ — 37.777..., где 7 повторяется до бесконечности, но, увы, невозможно написать бесконечную книгу.



Тип **int** может представлять только целые числа. Целый эквивалент числа 37.78 — 37. При этом, для того чтобы разместить число в целой переменной, дробная часть числа отбрасывается — такое действие называется **усечением (truncation)**.

Усечение — совсем не то же самое, что **округление (rounding)**. Усечение отбрасывает дробную часть, а при округлении получается ближайшее целое значение. Так, усечение 1.9 даст 1, а округление — 2.

Для температур 37 может оказаться вполне достаточно. Вряд ли ваша одежда при 37.78°C будет существенно отличаться от одежды при 37°C. Но для множества, если не большинства, приложений такое усечение неприемлемо.

На самом деле все еще хуже. Тип **int** не в состоянии хранить значение $5/9$ и преобразует его в 0. Соответственно, данная формула будет давать нулевое значение **nCelsius** для любого значения **nFahr**. Поэтому даже такой непритязательный человек, как я, сочтет это неприемлемым.



На прилагаемом к книге компакт-диске в каталоге **ConvertTemperatureWithRoundOff** имеется программа, использующая целочисленное преобразование температур. Пока что вы можете не разобрататься со всеми ее деталями, но можете посмотреть на уравнение преобразования и запустить программу **ConvertTemperatureWithRoundOff.exe**, чтобы увидеть результаты ее работы.

Работа с числами с плавающей точкой

Ограничения, накладываемые на переменные типа **int**, для многих приложений неприемлемы. Обычно главным препятствием является не диапазон возможных значений (двух квинтиллионов 64-битового **long** хватает, пожалуй, для подавляющего большинства задач), а невозможность представления дробных чисел.

В некоторых ситуациях нужны числа, которые могут иметь ненулевую дробную часть, и которые математики называют **действительными числами (real numbers)**. Всегда находятся люди, удивляющиеся такому названию — неужели целые числа — не действительные?



Обратите внимание на сказанное о том, что действительное число *может* иметь ненулевую дробную часть — т.е. число 1.5 является действительным так же, как и число 1.0. Например, $1.0 + 0.1 = 1.1$. Просто при чтении оставшейся части этой главы все время не забывайте о наличии точки.

К счастью, C# прекрасно понимает, что такое действительные числа. Они могут быть с плавающей точкой и с так называемым десятичным представлением. Гораздо более распространена плавающая точка.

Объявление переменной с плавающей точкой

Переменная с плавающей точкой может быть объявлена так, как показано в следующем примере:

```
float f = 1.0;
```

После того как вы объявите переменную как `float`, она остается таковой при всех естественных для нее операциях.

В табл. 3.2 рассматриваются использующиеся в C# типы с плавающей точкой. Все переменные этих типов — знаковые (т.е. не существует такой вещи, как переменная с плавающей точкой, не способная представлять отрицательные значения).

Таблица 3.2. Размеры и диапазоны представления типов переменных с плавающей точкой

Тип	Размер (в байтах)	Диапазон значений	Точность (количество цифр)	Использование
<code>float</code>	8	От $1.5 \cdot 10^{-45}$ до $3.4 \cdot 10^{38}$	6–7	<code>float f = 1.2F;</code>
<code>double</code>	16	От $5.0 \cdot 10^{-324}$ до $1.7 \cdot 10^{308}$	15–16	<code>double d = 1.2;</code>



Вы можете решить, что тип `float` — это тип по умолчанию для переменных с плавающей точкой, но на самом деле типом по умолчанию является `double`. Если вы не определите явно тип для, скажем, 12.3, C# сделает его `double`.

Столбец точности в табл. 3.2 указывает количество значащих цифр, которые может представлять такая переменная. Например, $5/9$ на самом деле равно 0.555... с бесконечной последовательностью пятерок. Однако переменная типа `float` имеет точность не более 6 цифр, что означает, что все цифры после шестой могут быть проигнорированы. Таким образом, $5/9$, будучи выражено в виде `float`, может выглядеть как

```
0.5555551457382
```

Не забывайте, что все цифры после шестой пятерки ненадежны.



На самом деле тип `float` имеет 6.5 значащих цифр. Дополнительные полцифры получаются из-за того, что точность представления чисел с плавающей точкой связана с такой функцией, как $10^{.82}$. Впрочем, вряд ли это должно сильно вас интересовать.

То же число $5/9$ может выглядеть при использовании типа `double` следующим образом:

```
0.55555555555555557823
```

Тип `double` имеет 15–16 значащих цифр.



Числа с плавающей точкой в C# по умолчанию имеют точность **double**, так что применяйте везде тип **double**, если только у вас нет веских причин поступить иначе. Однако используете ли вы **double** или **float** — все равно ваша программа будет считаться программой, работающей с числами с плавающей точкой.

Более точное преобразование температур

Вот как выглядит формула преобразования температур в градусах Фаренгейта в градусы Цельсия при использовании переменных с плавающей точкой:

```
dCelsius = (dFahr - 32.0) * (5.0 / 9.0);
```



На прилагаемом к книге компакт-диске имеется **double**-версия программы преобразования температур **ConvertTemperatureWithFloat**.

Приведенный далее пример показывает результат работы программы **ConvertTemperatureWithFloat**:

```
Введите температуру в градусах Фаренгейта: 100
Температура в градусах Цельсия равна: 37.77777777777778
Press Enter to terminate...
```

Ограничения переменных с плавающей точкой

Вы можете захотеть использовать переменные с плавающей точкой везде и всегда, раз уж они так хорошо решают проблему усечения. Да, конечно, они используют немного больше памяти, но ведь сегодня это не проблема? Но дело в том, что у чисел с плавающей точкой имеется ряд ограничений.

Преимущества

Нельзя использовать числа с плавающей точкой для перечисления. Некоторые структуры C# должны быть подсчитаны (1, 2, 3 и т.д.). И всем известно, что числа 1.0, 2.0, 3.0 можно применять для подсчета количества точно так же, как и 1, 2, 3, но C# ведь этого не знает. Например, при указанной выше точности чисел с плавающей точкой откуда C# знать, что вы не сказали в действительности 1.000001?

Находите ли вы эту аргументацию убедительной или нет — но вы не можете использовать числа с плавающей точкой для подсчета количества.

Сравнение чисел

Вы должны быть очень осторожны при сравнении чисел с плавающей точкой. Например, 12.5 может быть представлено как 12.500001. Большинство людей не волнуют такие мелкие добавки в конце числа, но компьютер понимает их буквально, и для C# 12.500000 и 12.500001 — это разные числа.

Так, если вы сложите 1.1 и 1.1, вы можете получить в качестве результата 2.2 или 2.200001. И если вы спросите: "Равно ли значение **dDoubleVariable**. 2.2?", то можете получить совсем не тот ответ, который ожидаете. Такие вопросы вы должны переформулировать, например, так: "Отличается ли абсолютное значение разности **dDoubleVariable** и 2.2 менее чем на 0.000001?", другими словами, равны ли два значения с некоторой допустимой ошибкой.



Процессоры Pentium используют небольшой трюк, который несколько снижает ушанную неприятность, — при работе они применяют специальный формат, в котором для числа с плавающей точкой выделяется 80 бит. При округлении такого числа к 64-битовому почти всегда получается результат, который вы ожидаете.

Скорость вычислений

Процессоры x86, использующиеся на старых компьютерах под управлением Windows, выполняли действия над целыми числами существенно быстрее, чем над числами с плавающей точкой. В настоящее время эта проблема так остро не стоит.

Отношение скоростей работы процессора Pentium III при простом (пожалуй, слишком простом) тесте, состоящем в 300000000 сложений и вычитаний целых чисел и чисел с плавающей точкой, оказалось равным примерно 3 к 1. То есть вместо одного сложения `double` можно сделать три сложения `int`. (Вычисления с применением умножения и делений могут привести к другим результатам.)

Ограниченность диапазона

В прошлом переменные с плавающей точкой могли представлять значительно больший диапазон чисел, чем целые. Сейчас диапазон представления целых чисел существенно вырос — стоит вспомнить о типе `long`.



Даже простой тип `float` способен хранить очень большие числа, но числа значащих цифр у него ограничено примерно шестью. Например, `123456789F` означает то же, что и `123456000F`. (О том, что такое `F` в приведенных записях, вы узнаете немного позже.)

Десятичные числа — комбинация целых и чисел с плавающей точкой

Как уже объяснялось в предыдущих разделах, и целые, и десятичные числа имеют свои недостатки. Переменным с плавающей точкой присущи проблемы, связанные с вопросами округления из-за недостаточной точности представления, целые переменные не могут представлять числа с дробной частью. Бывают ситуации, когда совершенно необходимо иметь возможность получить лучшее из обоих миров, а именно числа, которые:

- подобно числам с плавающей точкой, способны иметь дробную часть;
- подобно целым числам, должны представлять точные результаты вычислений — т.е. `12.5` должно быть равно `12.5`, и ни в коем случае не `12.500001`.

К счастью, в C# есть такой тип чисел, называющийся `decimal`. Переменная типа `decimal` в состоянии представлять числа от 10^{-28} до 10^{28} — вполне достаточный диапазон значений! И все это делается без проблем, связанных с округлением.

Объявление переменных типа `decimal`

Переменные типа `decimal` объявляются и используются так же, как и переменные других типов:

```
decimal m1;           // Хорошо
decimal m2 = 100;     // Лучше
decimal m3 = 100M;    // Идеально
```

Объявление `m1` выделяет память для переменной `m1` без ее инициализации. Пока вы не присвоите значение этой переменной, оно будет неопределенным. В этом нет особой проблемы, так как C# не позволит вам использовать переменную без начального присвоения ей какого-либо значения.

Второе объявление создает переменную `m2` и инициализирует ее значением `100`. В этой ситуации неприятным моментом оказывается то, что `100` имеет тип `int`. Поэтому C# вынужден конвертировать `int` в `decimal` перед инициализацией. К счастью, C# понимает, что именно вы добиваетесь, и выполняет эту инициализацию для вас.

Лучше всего использовать такое объявление, как объявление переменной `m3` с константой `100M` типа `decimal`. Буква `M` в конце числа указывает, что данная константа имеет тип `decimal`, так что никакого преобразования не требуется.

Сравнение десятичных, целых чисел и чисел с плавающей точкой

Создается впечатление, что числа типа `decimal` имеют лишь одни достоинства и лишены недостатков, присущих типам `int` и `double`. Переменные этого типа обладают широким диапазоном представления и не имеют проблем округления.

Однако у чисел `decimal` есть свои неприятности. Во-первых, поскольку они имеют дробную часть, они не могут использоваться в качестве счетчиков, например, в циклах, о которых пойдет речь в главе 5, "Управление потоком выполнения".

Вторая проблема не менее серьезна, и заключается в том, что вычисления с этим типом чисел гораздо медленнее, чем с простыми целыми числами или даже с числами с плавающей точкой. В уже упоминавшемся тесте с 300000000 сложений и вычитаний работа с числами `decimal` примерно в 50 раз медленнее работы с числами `int`. Это отношение становится еще хуже для более сложных операций. Кроме того, большинство математических функций, таких как синус или возведение в степень, не имеют версий для работы с числами `decimal`.

Понятно, что числа типа `decimal` наиболее подходят для финансовых приложений, где исключительно важна точность, но само количество вычислений относительно невелико.

Логичен ли логический `tnun`?

И наконец, о переменных логического типа. Тип `bool` имеет только два значения — `true` и `false`. Это не шутка — целый тип переменных придуман для работы только с двумя значениями.



Ранее программисты на C и C++ использовали нулевое значение переменной типа `int` для обозначения `false` и ненулевое — для обозначения `true`. В C# этот фокус не проходит.

Переменная типа `bool` объявляется следующим образом:

```
bool thisIsABool = true;
```

Нет никаких путей для преобразования переменных `bool` в другой тип переменных (даже если бы вы могли это делать, это бы не имело никакого смысла). В частности, вы

не можете преобразовать `bool` в `int` (чтобы, скажем, `false` превратилось в `0`) или в `string` (чтобы `false` стало `"false"`).

Символьные типы

Программа, которая в состоянии заниматься только вычислениями, могла бы устроить разве что математиков, страховых агентов и военных (да-да— первые вычислительные машины были созданы для расчета таблиц артиллерийских стрельб). Однако в большинстве приложений программы должны работать не только с цифрами, но и с буквами.

C# рассматривает буквы двумя различными путями — как отдельные символы типа `char` и как строки символов типа `string`.

Тип `char`

Переменная типа `char` способна хранить только один символ. Символьная константа выглядит как символ, окруженный парой одинарных кавычек:

```
char c = 'a';
```

Вы можете хранить любой символ из латинского алфавита, кириллицы, арабского, иврита, японских катаканы и хираганы и массы японских, китайских или корейских иероглифов.

Кроме того, тип `char` может использоваться в качестве счетчика, т.е. его можно применять в циклах, о которых вы узнаете в главе 5, "Управление потоком выполнения". У символов нет никаких проблем, связанных с округлением.



Переменные типа `char` не включают информации о шрифтах, так что в переменной `char` может храниться, например, вполне корректный иероглиф, но при выводе его без использования соответствующего шрифта вы увидите на экране только мусор.

Специальные символы

Некоторые символы являются непечатными в том смысле, что вы ничего не увидите при выводе их на экран или на принтер. Наиболее очевидным примером такого символа является пробел `' '` (кавычка, пробел, кавычка). Другие символы не имеют буквенного эквивалента — например, символ табуляции. Для указания таких символов C# использует обратную косую черту, как показано в табл. 3.3.

Таблица 3.3. Специальные символы	
Символьная константа	Значение
<code>'\n'</code>	Новая строка
<code>'\t'</code>	Табуляция
<code>'\0'</code>	Нулевой символ
<code>'\r'</code>	Возврат каретки
<code>'\\'</code>	Обратная косая черта

Тип string

Еще одним распространенным типом переменных является **string**. Приведенные далее примеры показывают, как объявляются и инициализируются переменные этого типа.

```
// Объявление с отложенной инициализацией
string someString1;
someString1 = "Это строка";
// Инициализация при объявлении
string someString2 = "Это строка";
```

Константа типа **string**, именуемая также строковым литералом, представляет собой набор символов, окруженный двойными кавычками. Символы в строке могут включать специальные символы, показанные в табл. 3.3. Строка не может быть перенесена на новую строку в исходном тексте на C#, но может содержать символ новой строки, как показано в следующем примере:

```
// Неверная запись строки
string someString = "Это строка
и это строка";
// А вот так - верно
string someString = "Это строка\nи это строка";
```

При выводе на экран при помощи вызова **Console.WriteLine** вы увидите текст, размещенный в двух строках:

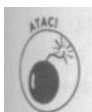
```
Это строка
и это строка
```

Строка не является ни перечислимым типом, ни типом-значением — в процессоре не существует встроенного типа строки. К строкам применим только один распространенный оператор — оператор сложения, который просто объединяет две строки в одну, например:

```
string s = "Это предложение." + " И это тоже.";
```

Приведенный код присваивает строке **s** значение:

```
"Это предложение. И это тоже."
```



Строка без символов, записанная как `""` (пара двойных кавычек), является корректной строкой для типа **string**, и называется *пустой строкой*. Пустая строка отличается от нулевого символа `'\0'` и от строки, содержащей любое количество пробелов (`" "`).

Кстати, все остальные типы данных в этой главе — *типы-значения* (value types). Строковый тип типом-значением не является.

Что такое тип-значение?



Все инструкции C# должны быть реализованы как машинные команды процессора — процессора Intel в случае PC. Эти процессоры также имеют собственную концепцию переменных. Например, процессор Intel содержит восемь внутренних хранилищ, именуемых *регистрами*, каждый из которых может хранить одно значение **int**. Не вдаваясь в детали функционирования процессора, можно сказать, что все типы, описываемые в данной главе, за исключением **decimal** и **string**, являются встроенными для процессора. Таким образом, существует машинная

команда, суть которой в следующем: прибавить один **int** к другому. Имеется и аналогичная команда и для сложения **double**.

Кроме того, переменные описанных типов (опять же, за исключением **string**) имеют фиксированную длину. Переменные типа с фиксированной длиной всегда занимают одно и то же количество памяти. Так что при присваивании **a = b** C# может! поместить значение **b** в **a** без каких-либо дополнительных мер, разработанных для типов переменной длины. Эта характеристика дает этим типам переменных имя *типы-значения*.



Типы **int**, **double**, **bool** и их "близкие родственники" наподобие беззнакового **int** являются встроенными типами. Встроенные типы переменных и тип **decimal** известны также как типы-значения. Тип **string** не относится ни к тем ни к другим.

Типы, о которых речь пойдет в главе 6, "Объединение данных — классы и массивы", определяемые программистом и известные как ссылки, не являются ни встроенными, ни типами-значениями. Тип **string** является ссылочным типом, хотя компилятор C# рассматривает его специальным образом в силу его широкой распространенности.

Сравнение *string* и *char*

Хотя строки имеют дело с символами, тип **string** существенно отличается от типа **char**. Понятно, что имеются некоторые тривиальные отличия. Так, символ помещается в одинарные кавычки, а строка — в двойные. Кроме того, тип **char** — это всегда один символ, так что следующий код не имеет смысла — ни в плане сложения, ни в плане конкатенации:

```
char c1 = 'a';
char c2 = 'b';
char c3 = c1 + c2;
```



На самом деле этот код почти компилируем, но его смысл существенно отличается от того, который мы ему приписываем. C# преобразует **c1** и **c2** в значения типа **int**, представляющие собой числовые значения соответствующих символов, после чего складывает полученные значения. Ошибка возникает при попытке сохранить полученный результат в **c3**, так как при размещении значения типа **int** в переменной меньшего размера **char** данные могут быть потеряны. В любом случае, эта операция не имеет смысла.

С другой стороны, строка может быть любой длины. Таким образом, конкатенация двух строк вполне осмысленна:

```
string s1 = "a";
string s2 = "b";
string s3 = s1 + s2;    // Результат — "ab"
```

В качестве части своей библиотеки C# определяет целый ряд строковых операций, которые будут описаны в главе 9, "Работа со строками в C#".



Соглашения по именованию

Программирование и так достаточно сложно, чтобы делать его еще сложнее. Чтобы код на С# было легче читать, обычно используются определенные соглашения по именованию переменных, которым желательно следовать, чтобы код был понятен другим программистам.

✓ **Имена всех объектов, кроме переменных, начинаются с прописной буквы, а имена переменных — со строчной.** Делайте эти имена как можно более информативными (зачастую это приводит к тому, что имена состоят из нескольких слов). Слова должны начинаться с прописной буквы, и лучше, если между ними не будет символов подчеркивания — например, `thisIsALongVariableName`.

✓ **Первая буква имени переменной указывает ее тип.** Большинство таких букв тривиальны — `f` для `float`, `d` для `double`, `s` для `string` и так далее. Единственным нарушающим правило символом является `p` для `int`. Есть еще одно исключение — по традиции, уходящей в программирование на Фортране, отдельные буквы `i`, `j` и `k` также используются как распространенные имена переменных типа `int`.

Венгерская запись постепенно выходит из моды, по крайней мере в кругах программистов .NET. Тем не менее я все еще остаюсь ее поклонником, поскольку она позволяет мне знать тип каждой переменной в программе, не обращаясь к ее объявлению. В последних версиях Visual Studio вы можете просто подвести курсор к переменной и получить информацию о ее типе в окне подсказки, что делает венгерскую запись менее полезной. Однако вместо того чтобы вступать в "религиозные войны" по поводу того или иного способа именования, выберите тот, который вам по душе, и следуйте ему.

Объявление числовых констант

В жизни очень мало абсолюта, но он присутствует в С#: любое выражение имеет значение и тип. В объявлении наподобие `int p` легко увидеть, что переменная `p` имеет тип `int`. Разумно предположить, что тип результата вычисления `p+1` также `int`. Но что можно сказать о типе константы `1`?

Тип константы зависит от двух вещей: ее значения и наличия необязательной буквы в конце. Любое целое величиной до примерно 2 миллиардов (см. табл. 3.1) рассматривается как `int`. Числа, превышающие это значение, трактуются как `long`. Любые числа с плавающей точкой рассматриваются как `double`.

В табл. 3.4 показаны константы, объявленные как имеющие конкретные типы, т.е., в частности, с буквенными дескрипторами в конце. Строчные эти буквы или прописные — значения не имеет, например записи `lu` и `1U` равноценны.

Таблица 3.4. Объявление констант с их типом

Константа	Тип
<code>1</code>	<code>int</code>
<code>1U</code>	<code>unsigned int</code>
<code>1L</code>	<code>long int</code> (избегайте использования строчной <code>l</code> — она слишком похожа на <code>1</code>)

Константа	Тип
1.0	double
1.0F	float
1M	decimal
true	bool
false	bool
'a'	char
'\n'	char (СИМВОЛ НОВОЙ строки)
'\x123'	char (символ с шестнадцатеричным числовым значением 123)
"a string"	string
	string (пустая строка)

Преобразование типов

Человек не рассматривает числа, используемые для счета, как разнотипные. Например, нормальный человек (не программист на C#) не станет задумываться, глядя на число 1, знаковое оно или беззнаковое, "короткое" или "длинное". Хотя для C# все эти типы различны, даже он понимает, что все они тесно связаны между собой. Например, в приведенном далее фрагменте исходного текста величина типа **int** преобразуется в **long**:

```
int nValue = 10;
long lvalue;
lvalue = nValue;    // Это присваивание корректно
```

Переменная типа **int** может быть преобразована в **long**, поскольку любое значение типа **int** может храниться в переменной типа **long** и оба типа представляют собой числа, пригодные для перечислений. C# выполняет такое преобразование автоматически, без каких-либо комментариев.

Однако обратное преобразование может вызвать проблемы. Например, приведенный далее фрагмент исходного текста содержит ошибку:

```
long lvalue = 10;
int nValue;
nValue = lvalue;    // Неверно!
```

Некоторые значения, которые могут храниться в переменной **long**, не помещаются в переменную типа **int** (ну, например, 4 миллиарда). C# в такой ситуации генерирует сообщение об ошибке, поскольку в процессе преобразования данные могут быть потеряны. Ошибку такого рода обычно довольно сложно обнаружить.

Но что, если вы точно знаете, что такое преобразование вполне допустимо? Например, несмотря на то что переменная **lvalue** имеет тип **long**, в данной конкретной программе ее значение не может превышать 100. В этом случае преобразование переменной **lvalue** типа **long** в переменную **nValue** типа **int** совершенно корректно.

Вы можете пояснить C#, что отлично понимаете, что делаете, посредством оператора приведения типов:

```
long lvalue = 10;
int nValue;
nValue = (int)lvalue;    // Все в порядке
```


При приведении вы размещаете имя требуемого типа в круглых скобках непосредственно перед преобразуемым значением. Приведенная выше запись гласит: "Не волнуйся и преобразуй **lvalue** в тип **int** — я знаю, что делаю, и всю ответственность беру на себя". Конечно, такое утверждение может показаться излишне самоуверенным, но зачастую оно совершенно справедливо.

Перечислимые числа могут быть преобразованы в числа с плавающей точкой автоматически, но обратное преобразование требует использования оператора приведения типов, например:

```
double dValue = 10.0;  
long lvalue = (long)dValue;
```

Все приведения к типу **decimal** и из него нуждаются в применении оператора приведения типов. В действительности все числовые типы могут быть преобразованы в другие числовые типы с помощью такого оператора. Однако ни **bool**, ни **string** не могут быть непосредственно приведены ни к какому иному типу.



Встроенные функции C# могут преобразовывать числа, символы или логические переменные в их строковые "эквиваленты". Например, вы можете преобразовать значение **true** типа **bool** в строку **"true"**; однако такое преобразование нельзя рассматривать как непосредственное. Эти два значения — совершенно разные вещи.

Глава 4

Операторы

В этой главе...

> Выполнение арифметических действий

I Логические операции

> Составные логические операторы



Математики создают переменные и выполняют над ними различные действия, складывая их, умножая, а иногда — представьте себе — даже интегрируя. В главе 3, "Объявление переменных-значений", описано, как объявлять и определять переменные, но там ничего не говорится о том, как их использовать после объявления, чтобы получить что-то полезное. В этой главе рассматриваются операции, которые могут быть произведены над переменными. Для выполнения операций требуются операторы, такие как $+$, $-$, $=$, $<$ или $\&$. Здесь речь пойдет об арифметических, логических и других операторах.

Арифметика

I Все множество арифметических операторов можно разбить на несколько групп: простые арифметические операторы, операторы присваивания и специальные операторы, предназначенные только программированию. После такого обзора арифметических операторов нужен обзор логических операторов, но о них будет рассказано несколько позже.

Простейшие операторы

• С большинством из этих операторов вы должны были познакомиться еще в школе. Они перечислены в табл. 4.1. Обратите внимание, что в программировании для обозначения умножения используется звездочка ($*$), а не крестик (\times).

Таблица 4.1. Простые операторы	
Оператор	Значение
- (унарный)	Отрицательное значение
*	Умножение
/	Деление
+	Сложение
% (бинарный)	Вычитание
	Деление по модулю

Большинство этих операторов называются *бинарными*, поскольку они выполняются над двумя значениями: одно из них находится с левой стороны от оператора, а другое — с правой. Единственным исключением является унарный минус, который столь же прост, как и остальные рассматриваемые здесь операторы:

```
int n1 = 5;  
int n2 = -n1; // Теперь значение n2 равно -5
```

Значение **-n** представляет собой отрицательное значение **n**.

Оператор деления по модулю может быть вам незнаком. Деление по модулю аналогично получению остатка после деления. Так, **5%3** равно **2** ($5/3=1$, остаток **2**), а **25%3** равен **1** ($25/3=8$, остаток **1**).



Строгое определение оператора **%** выглядит как $x = (x/y) * y + (x \% y)$.

Арифметические операторы (кроме деления по модулю) определены для всех типов переменных. Оператор же деления по модулю не определен для чисел с плавающей точкой, поскольку при делении значений с плавающей точкой не существует остатка.

Порядок выполнения операторов

Значение некоторых выражений может оказаться непонятным. Например, рассмотрим выражение:

```
int n = 5 * 3 + 2;
```

Что имел в виду написавший такую строку программист? Что надо умножить 5 на 3, а затем прибавить 2? Или сначала сложить 3 и 2, а результат умножить на 5?



C# обычно выполняет операторы слева направо, так что результатом приведенного примера будет значение, равное **17**.

C# вычисляет значение **n** в представленном далее выражении, сначала деля **24** на **6**, а затем деля получившееся значение на **2**:

```
int n = 24/6/2;
```

Однако у операторов есть своя иерархия, приоритеты, или проще — свой порядок выполнения. C# считывает все выражение и определяет, какие операторы имеют наивысший приоритет и должны быть выполнены до операторов с меньшим приоритетом. Например, приоритет умножения выше, чем сложения. Во многих книгах изложению этого вопроса посвящены целые главы, но сейчас не стоит забивать этим ни голову, ни вашу голову.



Никогда не полагайтесь на то, что вы (или кто-то иной) помните приоритеты операторов. Явно указывайте подразумеваемый порядок выполнения выражения посредством скобок.

Значение следующего выражения совершенно очевидно и не зависит от приоритета операторов:

```
int n = (7 % 3) * (4 + (6 / 3));
```

Скобки перекрывают приоритеты операторов, явно указывая, как именно компилятор должен интерпретировать выражение. С# ищет наиболее вложенную пару скобок и вычисляет выражение в ней — в данном случае это $6/3$, что дает значение **2**. В результате получается:

```
int n = (7 % 3) * (4 + 2); // 2 = 6 / 3
```

Затем С# продолжает поиск скобок и вычисляет значения в них, что приводит к выражению:

```
int n = 1 * 6; // 6 = 4 + 2, 1 = 7 % 3
```

Так что в конечном счете получается:

```
int n = 6;
```

Правило "всегда используйте скобки" имеет, пожалуй, одно исключение. Лично мне с этим сложно примириться, но многие программисты опускают скобки в выражениях *наподобие* приведенного ниже, поскольку очевидно, что умножение имеет более высокий приоритет, чем сложение:

```
int n = 7 + 2 * 3; // То же, что и 7 + (2 * 3)
```

Оператор присваивания

С# унаследовал одну интересную концепцию от С и С++: присваивание является бинарным оператором, возвращающим значение аргумента справа от него. Присваивание имеет тот же тип, что и оба аргумента (типы которых должны быть одинаковы).

Этот новый взгляд на присваивание никак не влияет на выражения, с которыми вы уже сталкивались:

```
n = 5 * 3;
```

В данном примере $5*3=15$ и имеет тип `int`. Оператор присваивания сохраняет это `int`-значение справа в `int`-переменной слева и возвращает значение **15**. То, что он возвращает значение, позволяет, например, сохранить это значение еще в одной переменной, т.е. написать:

```
m = n = 5 * 3;
```

При наличии нескольких присваиваний они выполняются справа налево. В приведенном выше выражении правый оператор присваивания сохраняет значение **15** в переменной `n` и возвращает **15**, после чего левый оператор присваивания сохраняет значение **15** в переменной `m` и возвращает **15** (это возвращенное значение в данном примере больше *никогда* не используется).

Такое странное определение присваивания делает корректным такой причудливый фрагмент, как показанный ниже (хотя я и предпочитаю воздерживаться от подобных вещей):

```
int n;  
int m;  
n = m = 2;
```

Старайтесь избегать цепочек присваиваний, поскольку они менее понятны человеку, читающему исходный текст программы. Всего, что может запутать человека, читающего исходный текст вашей программы (включая и лично вас), следует избегать. Любые неясности ведут к ошибкам. Огромная часть программирования — от правил языка и его конструкций до соглашений по именованию переменных и рекомендаций, основанных на опыте программистов — нацелены на одно: устранение ошибок в программах.

C# добавляет ко множеству простейших операторов небольшое подмножество операторов, построенных на основе других бинарных операторов. Например, выражение

```
П += 1;
```

эквивалентно следующему:

```
П = П + 1;
```

Такие операторы присваивания существуют почти для всех бинарных операторов. В табл. 4.2 показаны наиболее распространенные составные операторы присваивания.

Таблица 4.2. Составные операторы присваивания

Оператор	Значение
<code>a += b</code>	Присваивает значение <code>a + b</code> переменной <code>a</code>
<code>a -= b</code>	Присваивает значение <code>a - b</code> переменной <code>a</code>
<code>a *= b</code>	Присваивает значение <code>a * b</code> переменной <code>a</code>
<code>a /= b</code>	Присваивает значение <code>a / b</code> переменной <code>a</code>
<code>a %= b</code>	Присваивает значение <code>a % b</code> переменной <code>a</code>
<code>a &= b</code>	Присваивает значение <code>a & b</code> переменной <code>a</code> (& — логический оператор, будет рассмотрен позже)
<code>a = b</code>	Присваивает значение <code>a b</code> переменной <code>a</code> (— логический оператор)
<code>a ^= b</code>	Присваивает значение <code>a ^ b</code> переменной <code>a</code> (^ — логический оператор)

В табл. 4.2 опущено два более сложных составных оператора присваивания, `<=<` и `>>=`. Операторы побитового сдвига, на которых они основаны, будут рассмотрены позже в этой главе.

Оператор инкремента

Среди всех сложений, выполняемых в программах, добавление 1 к переменной — наиболее распространенная операция:

```
п = п + 1; // Увеличение п на 1
```

C# позволяет записать такую операцию сокращенно:

```
п += 1; // Увеличение п на 1
```

Но, оказывается, и это недостаточно кратко, и в C# имеется еще более краткое обозначение этого действия — оператор инкремента:

```
++п; // Увеличение п на 1
```

Все три приведенных выражения функционально эквивалентны, т.е. все они увеличивают значение `п` на 1.

Оператор инкремента достаточно странен, но еще больше странности добавляет ему то, что на самом деле имеется два оператора инкремента: `++п` и `п++`. Первый, `++п`, называется *префиксным*, а второй, `п++`, — *постфиксным*. Разница между ними достаточно тонкая, но очень важная.

Вспомните, что каждое выражение имеет тип и значение. В следующем фрагменте исходного текста и `++п`, и `п++` имеют тип `int`:

```
int П;
П = 1;
int p = ++П;
```

```
n = 1;
int m = n++;
```

Чему равны значения p и m после выполнения этого фрагмента? (Подсказка: можно выбирать 1 или 2.) Оказывается, значение p равно 2, а значение m — 1. То есть значение выражения $++p$ — это значение p *после* увеличения, а значение $p++$ равно значению p *до* увеличения. Значение самой переменной p в обоих вариантах равно 2.

Эквивалентные операторы декремента — $p--$ и $--p$ — используются для замены выражения $p = p - 1$. Они работают точно так же, как и операторы инкремента.



Откуда взялся оператор инкремента?

Причина появления оператора инкремента лежит в туманном прошлом — в наличии в 1970-х годах в машине PDP-8 машинной команды инкремента. Язык C, прямой предок C#, в свое время создавался для применения именно на этих машинах. Наличие соответствующей машинной команды позволяло уменьшить количество машинных команд при использовании $p++$ вместо $p=p+1$. В то время экономия даже нескольких машинных команд давала существенный выигрыш во времени работы.

В настоящее время компиляторы гораздо интеллектуальнее, и нет никакой разницы, написать ли в программе $p++$ или $p=p+1$. Однако программисты — люди привычки, так что оператор инкремента благополучно дожил до сегодняшних дней, и увидеть в программе на C или C++ выражение $p=p+1$ практически нереально.

Кроме того, чаще всего программистами используется постфиксная версия оператора. Впрочем, это дело вкуса.. ?

Логично ли логическое сравнение?

C# предоставляет к услугам программиста также целый ряд логических операторов сравнения, показанных в табл. 4.3. Эти операторы называются *логическими сравнениями* (logical comparisons), поскольку они возвращают результат сравнения в виде значения `true` или `false`, имеющего тип `bool`.

- Вот примеры использования логических сравнений:

```
int m = 5;
int n = 6;
bool b = m > n;
```

В этом примере переменной b присваивается значение `false`, поскольку 5 не больше, чем 6.

² Использование префиксного оператора инкремента дает определенный выигрыш (компилятору не приходится дополнительно хранить значение переменной до инкремента), так что лучше приобретать привычку применять префиксную форму там, где выбор вида оператора инкремента не принципиален. — *Примеч. ред.*

Таблица 4.3. Логические операторы сравнения

Оператор...	...возвращает true, если...
<code>a == b</code>	<code>a</code> имеет то же значение, что и <code>b</code>
<code>a > b</code>	<code>a</code> больше <code>b</code>
<code>a >= b</code>	<code>a</code> больше или равно <code>b</code>
<code>a < b</code>	<code>a</code> меньше <code>b</code>
<code>a <= b</code>	<code>a</code> меньше или равно <code>b</code>
<code>a != b</code>	<code>a</code> не равно <code>b</code>

Логические сравнения определены для всех числовых типов, включая `float`, `double`, `decimal` и `char`. Все приведенные ниже выражения корректны:

```
bool b;
b = 3 > 2;           // true
b = 3.0 > 2.0;       // true
b = 'a' > 'b';       // false - позже в алфавитном порядке
                        // означает "больше"
b = 'A' < 'a';       // true - прописное 'A' меньше
                        // строчного 'a'
b = 'A' < 'b';       // true - все прописные буквы меньше всех
                        // строчных
b = 10M > 12M;       // false
```

Операторы сравнения всегда дают в качестве результата величину типа `bool`. Операторы сравнения, отличные от `==`, неприменимы к переменным типа `string` (не волнуйтесь, C# предлагает другие способы сравнения строк).

Сравнение чисел с плавающей точкой

Сравнение двух чисел с плавающей точкой может легко оказаться не вполне корректным, так что тут нужна особая осторожность. Рассмотрим следующее сравнение:

```
float f1;
float f2;
f1 = 10;
f2 = f1 / 3;
bool b1 = (3 * f2) == f1;
f1 = 9;
f2 = f1 / 3;
bool b2 = (3 * f2) == f1;
```

Обратите внимание, что в пятой и восьмой строках примера сначала содержится оператор присваивания `=`, а затем оператор сравнения `==`. Это — разные операторы. C# сначала выполняет логическое сравнение, а затем присваивает его результат переменной слева от оператора присваивания.

Единственное отличие между вычислениями `b1` и `b2` состоит в исходном значении `f1`. Так чему же равны значения `b1` и `b2`? Очевидно, что значение `b1` равно `true`: $9/3$ равно 3 , $3*3$ равно 9 , 9 равно 9 . Никаких проблем!

Значение `b2` не столь очевидно: $10/3$ равно $3.3333\dots$, $3.3333\dots*3$ равно $9.9999\dots$. Но равны ли числа $9.9999\dots$ и 10 ? Это зависит от того, насколько сообразительны ваши

компилятор и процессор. При использовании процессора типа Pentium C# недостаточно умен для того, чтобы понять, что **Ы** надо присвоить значение **true**.



Для сравнения **f1** и **f2** можно воспользоваться функцией для вычисления абсолютного значения следующим образом:

Math.abs(f1-f2*3.0) < .0001; // ...или другая степень точности

Такая функция вернет значение **true** в обоих случаях. Вместо **.0001** можно использовать константу **Double.Epsilon**, чтобы получить максимальную точность. Эта константа представляет собой наименьшую возможную разницу между двумя неравными значениями типа **double**.

Чтобы узнать, какие еще возможности скрывает в себе класс **System.Math**, воспользуйтесь командой меню **Help¹^Index** и введите **Math** в поле **Look For**.

Составные логические операторы

Для переменных типа **bool** имеются специфичные для них операторы, показанные в табл. 4.4.

Таблица 4.4. Составные логические операторы	
Оператор...	...возвращает true, если...
!a	a равно false
a & b	a и b равны true
a b	Либо a, либо b, либо они обе равны true (a и/или b)
a ^ b	Либо a, либо b, но не обе одновременно равны true (либо a, либо b)
a && b	a и b равны true (сокращенное вычисление)
a b	Либо a, либо b, либо они обе равны true (сокращенное вычисление)

Оператор **!** представляет собой логический эквивалент знака "минус". Например, **!a** истинно, если **a** ложно, и ложно, если **a** истинно.

Следующие два оператора тоже вполне просты и понятны. **a&b** истинно тогда и только тогда, когда и **a**, и **b** одновременно равны **true**; **a|b** истинно тогда и только тогда, когда или **a**, или **b**, или оба они одновременно равны **true**. Оператор **^** (*исключающее или*) возвращает значение **true** тогда и только тогда, когда значения **a** и **b** различны — т.е. когда одно из значений **true**, а второе — **false**.

Все перечисленные операторы возвращают в качестве результата значение типа **bool**.



Операторы **&**, **|** и **^** имеют версии, называемые *побитовыми* (bitwise). При применении к переменным типа **int** эти операторы выполняют действия с каждым битом отдельно. Таким образом, **6&3** равно **2** (**0110²&0011²** равно **0010²**), **6|3** равно **7** (**0110²|0011²** равно **0111²**), а **6^Л3** равно **5** (**0110^{2Л}0011²** равно **0101²**). Бинарная арифметика — очень интересная вещь, но она выходит за рамки настоящей книги.

Последние два оператора очень похожи на предыдущие, но имеют одно едва уловимое отличие. В чем оно заключается, вы сейчас поймете. Рассмотрим следующий пример:
bool b = (ЛогическоеВыражение!) & (ЛогическоеВыражение2) ;

В этом случае C# вычисляет **ЛогическоеВыражение1** и **ЛогическоеВыражение2**, а затем смотрит, равны ли оба `true` или нет, чтобы найти, какое значение следует присвоить переменной `B`. Но может оказаться, что C# выполняет лишнюю работу — ведь если одно из выражений равно `false`, то каким бы ни было второе, результат не может быть `true` в любом случае.

Оператор `&&` позволяет избежать вычисления второго выражения, если после вычисления первого конечный результат становится очевидным:

```
bool B = (ЛогическоеВыражение1) && (ЛогическоеВыражение2);
```

В этой ситуации C# вычисляет значение **ЛогическоеВыражение1**, и если оно равно `false`, то переменной `B` присваивается значение `false` и **ЛогическоеВыражение2** не вычисляется. Если же **ЛогическоеВыражение1** равно `true`, то C# вычисляет **ЛогическоеВыражение2** и после этого определяет, какое значение присвоить переменной `B`.

Оператор `||` работает аналогично, как видно из следующего выражения:

```
bool b = (ЛогическоеВыражение1) || (ЛогическоеВыражение2);
```

В этой ситуации C# вычисляет значение **ЛогическоеВыражение1**, и если оно равно `true`, то переменной `b` присваивается значение `true` и **ЛогическоеВыражение2** не вычисляется. Если же **ЛогическоеВыражение1** равно `false`, то C# вычисляет **ЛогическоеВыражение2** и после этого определяет, какое значение присвоить переменной `b`.

Вы можете называть эти операторы "сокращенным и" и "сокращенным или".

Тип выражения

В вычислениях тип результата важен не менее, чем сам результат. Рассмотрим следующее выражение:

```
int П;  
П = 5 * 5 + 7;
```

Калькулятор утверждает, что результат вычислений равен 32. Но это выражение имеет не только значение, но и тип.

Будучи записано на "языке типов", оно принимает следующий вид:

```
int [=] int * int + int;
```

Для выяснения типа выражения нужно следовать тому же шаблону, что и при вычислении его значения. Умножение имеет более высокий приоритет, чем сложение. Умножение `int` на `int` дает `int`. Далее идет сложение `int` и `int`, что в результате тоже дает `int`. Итак, вычисление типа приведенного выражения происходит таким образом:

```
int * int + int  
int + int  
int
```

Вычисление типа операции

Выяснение типа выражения происходит в нисходящем направлении посредством выяснения типов *подвыражений*. Каждое выражение имеет тип, и типы левых и правых аргументов оператора должны соответствовать самому оператору:

```
type1 <op> type2 ⇒ type3
```

(Здесь стрелка означает "дает".) Типы `type1` и `type2` должны быть совместимы с оператором `op`.

Большинство операторов могут иметь несколько вариантов. Например, оператор умножения может быть следующих видов:

<code>int</code>	<code>*</code>	<code>int</code>	<code>=></code>	<code>int</code>
<code>uint</code>	<code>*</code>	<code>uint</code>	<code><=></code>	<code>uint</code>
<code>long</code>	<code>*</code>	<code>long</code>	<code>=></code>	<code>long</code>
<code>float</code>	<code>*</code>	<code>float</code>	<code>=></code>	<code>float</code>
<code>decimal</code>	<code>*</code>	<code>decimal</code>	<code><=></code>	<code>decimal</code>
<code>double</code>	<code>*</code>	<code>double</code>		<code>double</code>

Таким образом, `2*3` использует `int*int` версию оператора `*` и дает в результате `int 6`.

Неявное преобразование типов

Все хорошо, просто и понятно, если умножать две переменные типа `int` или две переменные типа `float`. Но что получится, если типы аргументов слева и справа будут различны? Что, например, произойдет в следующей ситуации:

```
int n1 = 10;
double d2 = 5.0;
double dResult = n1 * d2;
```

Во-первых, в C# нет оператора умножения `int*double`. C# может просто сгенерировать сообщение об ошибке и предоставить разбираться с проблемой программисту. Однако он пытается понять намерения программиста и помочь ему. У C# есть операторы умножения `int*int` и `double*double`. C# мог бы преобразовать `d2` в значение `int`, но такое преобразование привело бы к потере дробной части числа (цифр после десятичной точки). Поэтому вместо этого C# преобразует `n1` в значение типа `double` и использует оператор умножения `double*double`. Это действие известно как *неявное повышение типа* (implicit promotion).

Такое повышение называется *неявным*, поскольку C# выполняет его автоматически, и является *повышением*, так как включает естественную концепцию высоты типа. Список операторов умножения был приведен в порядке повышения — от `int` до `double`, или от `int` до `decimal` — от типа меньшего размера к типу большего размера. Между типами с плавающей точкой и `decimal` неявное преобразование не выполняется. Преобразование из более емкого типа, такого как `double`, в менее емкий, такой как `int`, называется *понижением* (demotion).

Повышение иногда называют *преобразованием вверх* (up conversion), а понижение — *преобразованием вниз* (down conversion).



Неявные понижения запрещены. В таких случаях C# генерирует сообщение об ошибке.

Явное преобразование типа

Но что, если C# ошибается? Если на самом деле программист хотел выполнить целое умножение?

Вы можете изменить тип любой переменной с типом-значением с помощью оператора *приведения типа* (cast), который представляет собой требуемый тип, заключенный в скобки, и располагаемый непосредственно перед приводимой переменной или выражением.

Таким образом, в следующем выражении используется оператор умножения `int*int`:

```
int n1 = 10;
double d2 = 5.0;
double nResult = n1 * (int)d2;
```

Приведение `d2` к типу `int` известно как *явное понижение* (explicit demotion) или *понижающее приведение* (downcast). Понижение является явным, поскольку программу! явно объявил о своих намерениях.

Вы можете осуществить приведение между двумя любыми типами-значениями, независимо от их взаимной высоты.



Избегайте неявного преобразования типов. Делайте все изменения типов значений явными с помощью оператора приведения.

Оставьте логику в покое

C# не позволяет преобразовывать другие типы в тип `bool` или выполнять преобразование типа `bool` в другие типы.

Типы при присваивании

Все сказанное о типах выражений применимо и к оператору присваивания.



Случайные несоответствия типов, приводящие к генерации сообщений об ошибках, обычно происходят в операторах присваивания, а не в точке действительного несоответствия.

Рассмотрим следующий пример умножения:

```
int n1 = 10;
int n2 = 5.0 * n1;
```

Вторая строка этого примера приведет к генерации сообщения об ошибке, связанной с несоответствием типов, но ошибка произошла при присваивании, а не при умножении. Вот что произошло: для того чтобы выполнить умножение, C# неявно преобразовал `n1` в тип `double`. Затем C# выполнил умножение двух значений типа `double`, получив в результате значение того же типа `double`.

Типы левого и правого аргументов оператора присваивания должны совпадать, но тип левого аргумента не может быть изменен. Поскольку C# не может неявно понизить тип выражения, компилятор генерирует сообщение о том, что он не может неявно преобразовать тип `double` в `int`.

При использовании явного приведения никаких проблем не возникнет:

```
int n1 = 10;
int n2 = (int) (5.0 * n1);
```

(Скобки необходимы, потому что оператор приведения имеет очень высокий приоритет.) Такой исходный текст вполне работоспособен, так как *явное* понижение разрешено. Здесь значение `n1` будет повышено до `double`, выполнено умножение, а результат типа `double` будет понижен до `int`. Однако в этой ситуации надо подумать о душевном здоровье программиста, поскольку написать просто `5*n1` было бы проще как для программиста, так и для C#.

Немного экзотики — тернарный оператор

Большинство операторов имеют два аргумента, меньшинство — один. И только один оператор — тернарный — имеет три аргумента. Лично я считаю, что это ненужная экзотика. Вот формат этого оператора:

Выражение типа bool ? Выражение1 : Выражение2

А это пример его применения:

```
int a = 1;
int b = 2;
int nMax = (a > b) ? a : b;
```

Если *a* больше *b* (условие в скобках), значение выражения равно *a*. Если *a* не больше *b*, значение выражения равно *b*.

Выражения 1 и 2 могут быть любой сложности, но это должны быть истинные выражения — они не могут содержать объявлений или других инструкций, не являющихся выражениями.

Тернарный оператор непопулярен³ по следующим причинам.

✓ **Он не является необходимым.** Использование оператора `if`, описанного в главе 5, "Управление потоком выполнения", дает тот же эффект, и его легче понять.

✓ **На тернарный оператор накладываются дополнительные ограничения.** Например, выражения 1 и 2 должны быть одного и того же типа. Это приводит к следующему:

```
int a = 1;
double b = 0.0;
int nMax = (a > b) ? a : b;
```

Такой исходный текст не будет компилироваться, несмотря на то что в конечном итоге `nMax` будет иметь значение *a*. Поскольку *a* и *b* должны быть одного и того же типа, *a* будет повышено до `double`, чтобы соответствовать *b*. Тип результирующего значения оператора `?:` оказывается `double`, и этот тип должен быть понижен до `int` перед присваиванием:

```
int a = 1;
double b = 0.0;
int nMax;
// Можно поступить так:
nMax = (int) ( (a > b) ? a : b );
// ...или так:
nMax = (a > b) ? a : (int)b;
```

Увидеть тернарный оператор в реальной программе — большая редкость.

³ Непопулярность этого оператора относится к C#, программистами на C и C++ он употребляется достаточно часто и не вызывает никаких отрицательных эмоций. — *Примеч. ред.*

Глава 5

Управление потоком выполнения

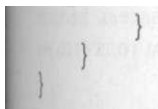
В этой главе...

- > Что делать, *если...*
- > Цикл **while**
- > Цикл **for**
- > Конструкция **switch**



ассмотрим следующую простую программу:

```
using System;
namespace HelloWorld
(
    public class Program
    {
        // Стартовая точка программы
        static void Main(string[] args)
        {
            // Приглашение для ввода имени
            Console.WriteLine("Введите ваше имя:");
            // Считывание введенного имени
            string sName = Console.ReadLine();
            // Приветствие с использованием введенного имени
            Console.WriteLine("Привет, " + sName);
            // Ожидание подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для "
                               "завершения программы...");
            Console.Read();
        }
    }
}
```



Толку от этой программы, помимо иллюстрации некоторых фундаментальных моментов программирования C#, очень мало. Она просто возвращает вам то, что вы ввели. Вы можете представить более сложный пример программы, в которой выполняются некоторые вычисления над введенными данными и генерируется какой-то более сложный вывод на экран (иначе для чего проводить вычисления?...), но и эта программа будет очень ограничена в своей функциональности.

Одним из ключевых элементов любого компьютерного процессора является его возможность принимать решения. Под выражением "принимать решения" имеется в виду, ! что процессор может пустить поток выполнения команд по одному или другому пути

в зависимости от того, истинно или ложно некоторое условие. Любой язык программирования должен обеспечивать такую возможность управления потоком выполнения.

Управление потоком выполнения

Основой возможности принятия решения в С# является оператор `if`:

```
if (Условие)  
i
```

II Этот код выполняется, если Условие истинно

```
}
```

```
// Этот код выполняется вне зависимости от  
// истинности Условия
```

Непосредственно за оператором `if` в круглых скобках содержится некоторое условное выражение типа `bool` (см. главу 4, "Операторы"), после чего следует код, заключенный в фигурные скобки. Если условное выражение истинно (имеет значение `true`), программа выполняет код, заключенный в фигурных скобках. Если нет — этот код программой опускается.

Работу оператора `if` проще понять, рассматривая конкретный пример:

```
// Гарантируем, что a - неотрицательно:
```

```
// Если a меньше 0 ...
```

```
if (a < 0)
```

```
{
```

```
    // ...присваиваем этой переменной значение 0
```

```
    a = 0;
```

```
}
```

В этом фрагменте исходного текста проверяется, содержит ли переменная `a` отрицательное значение, и если это так, переменной `a` присваивается значение `0`.



Если в фигурных скобках заключена только одна инструкция, то их можно не использовать, т.е. в приведенном выше фрагменте можно было бы написать `if (a < 0) a = 0;`. Но на мой взгляд, для большей удобочитаемости лучше всегда использовать фигурные скобки.

Оператор `if`

Рассмотрим небольшую программу, вычисляющую проценты. Пользователь вводит вклад и проценты, и программа подсчитывает сумму, получаемую по итогам года (это не слишком сложная программа). Вот как такие вычисления выглядят на С#:

```
// Вычисление суммы вклада и процентов
```

```
decimal mInterestPaid;
```

```
mInterestPaid = mPrincipal * (mInterest / 100);
```

```
// Вычисление общей суммы
```

```
decimal mTotal = mPrincipal + mInterestPaid;
```

В первом уравнении величина вклада `mPrincipal` умножается на величину процентной ставки `mInterest` (деление на `100` связано с тем, что пользователь вводит величину ставки в процентах). Получившаяся величина увеличения вклада сохраняется в переменной `mInterestPaid`, а затем суммируется с основным вкладом и сохраняется в переменной `mTotal`.



Программа должна предвидеть, что данные вводит всего лишь человек, которому свойственно ошибаться. Например, ошибкой должны считаться отрицательные величины вклада или процентов (конечно, в банке хотели бы, чтобы это было не так...), и в приведенной далее программе `CalculateInterest`, имеющейся на прилагаемом компакт-диске, выполняются соответствующие проверки.

```
//CalculateInterest
// Вычисление величины начисленных процентов для данного
// вклада. Если процентная ставка или вклад отрицательны,
// генерируется сообщение об ошибке.
using System;
namespace CalculateInterest
(
    public class Program
    {
        public static void Main(string[] args)
        {
            // Приглашение для ввода вклада
            Console.WriteLine("Введите сумму вклада:");
            string sPrincipal = Console.ReadLine();
            decimal mPrincipal =
                Convert.ToDecimal(sPrincipal);
            // Убеждаемся, что вклад не отрицателен
            if (mPrincipal < 0)
            {
                Console.WriteLine("Вклад не может "
                                   "быть отрицательным");
                mPrincipal = 0;
            }
            // Приглашение для ввода процентной ставки
            Console.WriteLine("Введите процентную ставку:");
            string sInterest = Console.ReadLine();
            decimal mInterest =
                Convert.ToDecimal(sInterest);
            // Убеждаемся, что процентная ставка не
            // отрицательна
            if (mInterest < 0)
            {
                Console.WriteLine("Процентная ставка не "
                                   "может быть отрицательна");
                mInterest = 0;
            }
            // Вычисляем сумму величины процентных
            // начислений и вклада
            decimal mInterestPaid;
            mInterestPaid = mPrincipal * (mInterest / 100);
            // Вычисление общей суммы
            decimal mTotal = mPrincipal + mInterestPaid;
            // Вывод результатов
            Console.WriteLine(); // skip a line
            Console.WriteLine("Вклад = " + mPrincipal);
```

```

        Console.WriteLine("Проценты = " + mInterest + "%");
        Console.WriteLine();
        Console.WriteLine("Начисленные проценты = "
            + mInterestPaid);
        Console.WriteLine("Общая сумма = " + mTotal);
        // Ожидание реакции пользователя
        Console.WriteLine("Нажмите <Enter> для "
            "завершения программы...");

        Console.Read();
    }
}

```

Программа **CalculateInterest** начинает свою работу с предложения пользователю ввести величину вклада. Это предложение выводится с помощью функции **WriteLine()**, которая выводит значение типа **string** на консоль.



Всегда точно объясняйте пользователю, чего вы от него хотите. Если возможно, укажите также требуемый формат вводимых данных. Обычно на неинформативные приглашения наподобие одного символа **>** пользователи отвечают совершенно некорректно.

В программе для считывания всего пользовательского ввода до нажатия клавиши **<Enter>** в переменную типа **string** используется функция **ReadLine()**. Поскольку программа работает с величиной вклада как имеющей тип **decimal**, введенную строку! следует преобразовать в переменную типа **decimal**, что и делает функция **Convert.ToDecimal()**. Полученный результат сохраняется в переменной **mPrincipal**.



Команды **ReadLine()**, **WriteLine()** и **ToDecimal()** служат примерами *вызовов функций*. Вызов функции делегирует некоторую работу другой части программы, именуемой функцией. Детально вызов функций будет описан в главе 7, "Функции функций", но приведенные здесь примеры очень просты и понятны. Если же вам что-то не ясно в вызовах функций, потерпите немного, и все будет объяснено детально.

В следующей строке выполняется проверка переменной **mPrincipal**. Если она отрицательна, программа выводит сообщение об ошибке. Те же действия производятся и для величины процентной ставки. После этого программа вычисляет общую сумму так, как уже было описано в начале раздела, и выводит конечный результат посредством нескольких вызовов функции **WriteLine()**.

Вот пример вывода программы при корректном пользовательском вводе:

Введите сумму вклада: 1234

Введите процентную ставку : 21

Вклад = 1234

Проценты = 21%

Начисленные проценты = 259.14

Общая сумма = 1493.14

Нажмите <Enter> для завершения программы...

А так выглядит вывод программы при ошибочном вводе отрицательной величины процентной ставки:

```
Введите сумму вклада: 1234
Введите процентную ставку 2-12.5
Процентная ставка не может быть отрицательна
```

```
Вклад      = 1234
Проценты   = 0%
```

```
Начисленные проценты = 0
Общая сумма           = 1234
Нажмите <Enter> для завершения программы...
```



Отступ внутри блока `if` повышает удобочитаемость исходного текста. С# игнорирует все отступы, но для человека они весьма важны. Большинство редакторов для программистов автоматически добавляют отступ при вводе оператора `if`. Для включения автоматического отступа в Visual Studio выберите команду меню `Tools>Options`, затем раскройте узел `Text Editor`, потом `C#`, а в конце щелкните на вкладке `Tabs`. На ней включите флаг `Smart Indenting` и установите то количество пробелов на один отступ, которое вам по душе. Установите то же значение и в поле `Tab Size`.

Инструкция `else`

Некоторые функции должны проверять взаимоисключающие условия. Например, в приведенном далее фрагменте исходного текста в переменной `max` сохраняется наибольшее из двух значений `a` и `b`:

```
// Сохраняем наибольшее из двух значений a и b
// в переменной max
int max;
// Если a больше b ...
if (a > b)
(
    // ...сохраняем значение a в переменной max
    max = a;
// Если a меньше или равно b ...
if (a <= b)
{
    // ...сохраняем значение b в переменной max
    max = b;
}
```

Второй оператор `if` является излишним, поскольку проверяемые условия взаимоисключающие. Если `a` больше `b`, то `a` никак не может быть меньше или равно `b`. Для таких случаев в С# предусмотрено ключевое слово `else`, позволяющее указать блок, который выполняется, если не выполняется блок `if`.

Вот как выглядит приведенный выше фрагмент кода при использовании `else`:

```
// Сохраняем наибольшее из двух значений a и b
// в переменной max
int max;
```

```
// Если а больше Ь ...
if (a > Ь)
{
    // ...сохраняем значение а в переменной max;
    // в противном случае
    max = a,-
}
else
{
    // ...сохраняем в переменной max значение b
    max = b;
}
```

Если **a** больше **Ь**, то выполняется первый блок; в противном случае выполняется второй блок. В результате в переменной **max** содержится наибольшее из значений **a** и **Ь**.

Как избежать else

При наличии многих **else** в исходном тексте можно легко запутаться, поэтому многие программисты предпочитают по возможности избегать использования **else**, если оно приводит к ухудшению удобочитаемости исходного текста. Так, рассмотренное выше вычисление максимального значения можно переписать следующим образом:

```
// Сохраняем наибольшее из двух значений а и b
// в переменной max
int max;
// Начнем с предположения, что а больше b
max = a,-
// Если же это не так ...
if (b > a)
{
    // ... то сохраняем в переменной max значение b
    max = b;
}
```

Другие программисты бегут от такого стиля написания программ, как от чумы, и **ml** можно понять. Это не значит, что следует поступать так же, как они, но оправдать их можно. Как поступать вам — дело ваше. В реальных программах встречаются оба стиля.

Вложенные операторы if

Программа **CalculateInterest** предупреждает пользователя о неверном вводе, но при этом продолжает вычисление начисленных процентов, несмотря на некорректность введенных значений. Вряд ли это правильное решение. Оно, конечно, не вызывает особых потерь процессорного времени, но только в силу простоты выполняемых программой подсчетов, в более же сложном случае это может привести к большим затратам. Кроме того, какой смысл запрашивать величину процентной ставки, если величина вклада уже введена неверно? Все равно результат придется проигнорировать, какое бы значение процентной ставки не было введено.

Программа должна запрашивать у пользователя величину процентной ставки только тогда, когда величина вклада введена верно, выполнять вычисления тогда и только тогда, когда оба введенных значения корректны. Для этого необходимы две конструкции **if** — одна внутри другой.



Оператор if, находящийся в теле другого оператора if, называется *встроенным (embedded)* или *вложенным (nested)*.



Приведенная далее программа CalculateInterestWithEmbeddedTest использует вложенный оператор if для того, чтобы избежать лишних вопросов при обнаружении некорректного ввода пользователя.

```
// CalculateInterestWithEmbeddedTest
//    Вычисление величины начисленных процентов для данного
//    вклада. Если процентная ставка или вклад отрицательны,
//    генерируется сообщение об ошибке и вычисления не
//    выполняются.
using System;

namespace CalculateInterestWithEmbeddedTest
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Определяем максимально возможное значение
            // процентной ставки
            int nMaximumInterest = 50;

            // Приглашение пользователю ввести величину исходного
            // вклада
            Console.Write("Введите сумму вклада:");
            string sPrincipal = Console.ReadLine();
            decimal mPrincipal = Convert.ToDecimal(sPrincipal);

            // Если исходный вклад отрицателен...
            if (mPrincipal < 0)
            {
                //...генерируем сообщение об ошибке...
                Console.WriteLine("Вклад не может быть отрицателен");
            }
            else
            {
                // ...в противном случае просим ввести процентную
                // ставку
                Console.Write("Введите процентную ставку:");
                string sInterest = Console.ReadLine();
                decimal mInterest = Convert.ToDecimal(sInterest);
                // Если процентная ставка отрицательна или слишком
                // велика...
                if (mInterest < 0 | mInterest > nMaximumInterest)
                {
                    // ...генерируем сообщение об ошибке
                    Console.WriteLine("Процентная ставка не может "
                                     "быть отрицательна " +
```

```

        "или превышать "
        + nMaximumInterest) ;

    mInterest = 0;
}
else
{
    // И величина вклада, и процентная ставка
    // корректны — можно приступить к вычислению
    // вклада с начисленными процентами
    decimal mInterestPaid;

    mInterestPaid = mPrincipal * (mInterest / 100);

    // Вычисляем общую сумму

    decimal mTotal = mPrincipal + mInterestPaid;

    // Выводим результат
    Console.WriteLine() , - // skip a line
    Console.WriteLine("Вклад = "
        + mPrincipal);
    Console.WriteLine("Проценты = "
        + mInterest + "%");
    Console.WriteLine();
    Console.WriteLine("Начисленные проценты = "
        + mInterestPaid);
} Console.WriteLine("Общая сумма      = "
    + mTotal);
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для "
    "завершения программы.. .") ;
Console.Read();
}
}
}

```

Программа начинает со считывания введенной пользователем величины исходного вклада. Если это значение отрицательно, она выводит сообщение об ошибке и завершает работу. Если же величина вклада не отрицательна, управление переходит к блоку `else`.

Проверка величины процентной ставки в этой программе немного усовершенствована. Программа требует не только неотрицательности введенного значения, но и чтобы оно было меньше некоторого максимального значения. Применяемый в программе оператор `if` использует следующий составной тест:

```
if (mInterest < 0 || mInterest > nMaximumInterest)
```

Выражение истинно, если `mInterest` меньше 0 или больше значения `nMaximumInterest`. Обратите внимание, что значение `nMaximumInterest` объявлено в начале программы, а не жестко закодировано в виде константы в исходном тексте условия.



Определяйте важные константы в начале программы с использованием символьных имен.

Такое кодирование констант в виде переменных служит следующим целям.

- ✓ Дает каждой константе поясняющее имя. `nMaximumInterest` более понятно, чем `50`.
- ✓ Облегчает поиск константы, если вам потребуется изменить ее.
- ✓ Облегчает процесс изменения константы. Обратите внимание, что та же переменная `nMaximumInterest` используется в сообщении об ошибке. Изменение `nMaximumInterest` на, например, `60`, приведет к корректной модификации как проверяемого условия, так и сообщения об ошибке.

Более подробно о константах будет рассказано в главе 6, "Объединение данных — классы и массивы".

Ввод корректной величины вклада и некорректной — процентной ставки, приводит к следующему выводу программы:

```
Введите сумму вклада :1234
Введите процентную ставку :-12.5
Процентная ставка не может быть отрицательна или превышать 50.
Нажмите <Enter> для завершения программы...
```

Только при вводе корректных значений и вклада, и процентной ставки программа приступит к вычислениям и выведет интересующий результат:

```
Введите сумму вклада :1234
Введите процентную ставку :12.5

Вклад      = 1234
Проценты   = 12.5%

Начисленные проценты = 154.250
Общая сумма           = 1388.250
Нажмите <Enter> для завершения программы...
```

Циклы

Конструкция `if` позволяет программе идти по коду различными путями в зависимости от результата вычисления значения типа `bool`. Ее наличие обеспечивает возможность создания программ существенно более интересных, чем те, которые могут быть написаны без использования `if`. Еще одним применением машинной команды условного перехода является возможность итеративного выполнения блока кода.

Рассмотрим еще раз программу `CalculateInterest` из раздела "Оператор `if` данной главы. Такие простые вычисления проще выполнить с помощью карманного калькулятора, чем писать для этого специальную программу.

Но что, если вы захотите вычислить проценты по вкладу для нескольких лет? Такая программа будет намного полезнее (конечно, простой макрос в Microsoft Excel все равно гораздо проще, чем требующаяся вам программа, но не стоит мелочиться).

Итак, нам надо выполнить некоторую последовательность инструкций несколько раз подряд. Это и называется *циклом* (loop).

Цикл while

Наиболее фундаментальный вид цикла создается с помощью ключевого слова `while` следующим образом:

```
while(Условие)
```

```
{
    // Код, повторно выполняемый до тех пор,
    // пока Условие не станет ложным
}
```

При первом обращении к циклу вычисляется условие в круглых скобках после ключевого слова **while**. Если оно истинно, выполняется следующий за ним блок кода — тело цикла. По окончании выполнения тела цикла программа вновь возвращается к началу цикла и вычисляет условие в круглых скобках, и все начинается сначала. Если же в какой-то момент условие становится ложным, тело цикла не выполняется, и управление передается коду, следующему за ним.



Если при первом обращении к циклу условие ложно, тело цикла не выполняется ни разу.



Программисты зачастую косноязычны и могут не совсем корректно выражаться. Например, говоря о цикле **while**, они могут сказать, что тело цикла выполняется до тех пор, пока условие не станет ложным. Я считаю, что такое определение некорректно, так как можно решить, что выполнение цикла прервется в тот же момент, как только условие станет ложным. Это не так. Программа не проверяет постоянно справедливость условия; проверка производится только тогда, когда управление передается в начало цикла.



Цикл **while** можно использовать для создания программы **CalculateInterestTable**, являющейся версией программы **CalculateInterest** с применением цикла. Она вычисляет таблицу величин вкладов по годам.

```
// CalculateInterestTable
// Вычисление величины начисленных процентов для данного
// вклада за определенный период времени
using System;
namespace CalculateInterestTable
{
    using System;
    public class Program
    {
        public static void Main(string[] args)
        {
            // Определяем максимально возможное значение
            // процентной ставки
            int nMaximumInterest = 50;

            // Приглашение пользователю ввести величину исходного
            // вклада
            Console.Write("Введите сумму вклада:");
            string sPrincipal = Console.ReadLine();
            decimal mPrincipal = Convert.ToDecimal(sPrincipal);

            // Если исходный вклад отрицателен...
            if (mPrincipal < 0)
            {
```

```

//...генерируем сообщение об ошибке...
Console.WriteLine("Вклад не может быть отрицателен");
}
else
{
    // ...в противном случае просим ввести процентную
    // ставку
    Console.Write("Введите процентную ставку:");
    string sInterest = Console.ReadLine();
    decimal mInterest = Convert.ToDecimal(sInterest);
    // Если процентная ставка отрицательна или слишком
    // велика...
    if (mInterest < 0 || mInterest > nMaximumInterest)
    {
        // ...генерируем сообщение об ошибке
        Console.WriteLine("Процентная ставка не может " +
            "быть отрицательна " +
            "или превышать " +
            nMaximumInterest);

        mInterest = 0;
    }
    else
    {
        // И величина вклада, и процентная ставка
        // корректны — запрашиваем у пользователя срок,
        // для которого следует вычислить величины вкладов
        // с начисленными процентами
        Console.Write("Введите количество лет:");
        string sDuration = Console.ReadLine();
        int nDuration = Convert.ToInt32(sDuration);

        // Выводим введенные величины
        Console.WriteLine(); // Пропуск строки
        Console.WriteLine("Вклад = "
            + mPrincipal);
        Console.WriteLine("Проценты = "
            + mInterest + "%");
        Console.WriteLine("Срок = "
            + nDuration + "years");
        Console.WriteLine();

        // Цикл по указанному пользователем количеству лет
        int nYear = 1;
        while(nYear <= nDuration)
        {
            // Вычисление вклада с начисленными процентами
            decimal mInterestPaid;
            mInterestPaid = mPrincipal * (mInterest / 100);

            // Вычисляем новое значение вклада
            mPrincipal = mPrincipal + mInterestPaid;
        }
    }
}

```

```

        // Округляем величину до копеек
        mPrincipal = decimal.Round(mPrincipal, 2);

        // Выводим результат
        Console.WriteLine(nYear + "-" + mPrincipal)

        // Переходим к следующему году
        nYear = nYear + 1;
    }
}
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для "
                  "завершения программы. ");
Console.Read();
}
}

```

Вот как выглядит вывод программы **CalculateInterestTable**:

Введите сумму вклада :1234
Введите процентную ставку :12.5
Введите количество лет:10

Вклад = 1234
Проценты = 12.5%
Срок = 10years

1-1388.25
2-1561.78
3-1757.00
4-1976.62
5-2223.70
6-2501.66
7-2814.37
8-3166.17
9-3561.94
10-4007.18

Нажмите <Enter> для завершения программы...

Каждое значение представляет общую сумму вклада по истечении указанного срока *i* предположении, что начисленные проценты добавляются к основному вкладу. Так, сумма в 1234 грн. при ставке 12.5% за 9 лет превращается в 3561.94 грн.



В большинстве значений для количества копеек выделяется две цифры. Однако в некоторых версиях C# завершающие нули могут не выводиться, и, например, сумма 2223.70 может оказаться выведенной как 2223.7. Это поведение 0 можно исправить с помощью специальных форматирующих символов, описываемых в главе 9, "Работа со строками в C#" (C# 2.0 выводит завершающие нули по умолчанию.)

Программа **CalculateInterestTable** начинает работу со считывания величины] вклада и процентной ставки и проверки их корректности. Затем программа считывает количество лет, для которых надо просчитать величины вкладов, и сохраняет их в переменной **nDuration**.

Перед тем как войти в цикл **while**, программа объявляет переменную **nYear**, инициализированную значением **1**. Эта переменная будет "текущим годом", т.е. ее значение будет увеличиваться с каждым выполнением тела цикла. Если номер года, хранящийся в переменной **nYear**, меньше общего количества лет, хранящегося в переменной **nDuration**, величина вклада для "этого года" вычисляется исходя из процентной ставки и величины вклада в "предыдущем году". Вычисленное значение программа выводит вместе со значением "текущего года".



Инструкция **decimal.Round()** округляет вычисленное значение до копеек.

Ключевая часть программы находится в последней строке тела цикла. Выражение **nYear = nYear + 1**; увеличивает переменную **nYear** на **1**. После увеличения значения года управление передается в начало цикла, где величина, хранящаяся в **nYear**, сравнивается с запрошенным количеством лет. В данном примере этот срок — **10** лет, так что когда значение переменной **nYear** станет равным **11**, т.е. превысит **10**, программа передаст управление первой строке после цикла **while**, и работа цикла на этом прекратится.



Большинство команд циклов используют этот базовый принцип увеличения значения счетчика, пока оно не достигнет некоторой предварительно заданной величины.

Переменная-счетчик **nYear** в программе **CalculateInterestTable** должна быть объявлена и инициализирована до цикла **while**, в котором она используется. Кроме того, переменная **nYear** должна увеличиваться, обычно в последнем выражении тела цикла. Как показано в примере, вы должны заранее позаботиться о том, какие переменные вам понадобятся в цикле. После того как вы напишете пару тысяч циклов **while**, все это будет делаться автоматически.



При написании цикла **while** не забывайте увеличивать значение счетчика. Взгляните на приведенный пример исходного текста:

```
int nYear = 1;
while (nYear < 10)
{
    // ... Какой-то код ...
}
```

(Я сознательно забыл написать **nYear = nYear + 1**;) Без увеличения счетчика переменная **nYear** всегда содержит значение **1**, так что цикл работает вечно. Такая ситуация называется *зацикливанием* (infinite loop). Единственный способ прекратить закливание — аварийно завершить программу извне (или перезагрузить компьютер).



Убедитесь, что условие прекращения работы цикла может быть достигнуто. Обычно это означает корректное увеличение счетчика цикла. В противном случае вы получите закливание программы, недовольных пользователей, падение продаж и много прочих неприятностей...



Зацикливание — достаточно распространенная ошибка, так что не слишком расстраивайтесь, допустив ее в своей программе.

Цикл `do...while`

Разновидностью цикла `while` можно считать цикл `do...while`. При его использовании условие не проверяется, пока не будет достигнут конец цикла:

```
int nYear = 1;
do
{
    // ... Некоторые вычисления ...
    nYear = nYear + 1;
} while (nYear < nDuration);
```

В противоположность циклу `while`, тело цикла `do...while` всегда выполняется крайней мере один раз, независимо от значения переменной `nDuration`. Этот цикл встречается в реальных программах гораздо реже, чем цикл `while`.

Операторы `break` и `continue`

Для управления циклом имеются два специальных оператора — `break` и `continue`. Оператор `break` вызывает прекращение выполнения цикла и передачу управления первому выражению непосредственно за циклом. Команда `continue` передает управление в начало цикла, к проверке его условия.



Лично я редко пользуюсь оператором `continue`, так что иногда просто забываю о его существовании. Думаю, что есть немало программистов, поступающих так же.

Предположим, вы хотите прекратить выполнение рассматривавшейся ранее программы, как только сумма вклада превысит начальную в некоторое заранее заданное число раз, независимо от того, какой срок прошел до этого момента. Это можно легко сделать, добавив в тело цикла следующие строки:

```
if (mPrincipal > (maxPower * mOriginalPrincipal))
{
    break;
}
```

Оператор `break` не будет выполняться, пока условие оператора `if` не станет истинным, т.е. пока вычисленная величина вклада не превысит исходную в `maxPower` раз. Оператор `break` передаст управление за пределы цикла `while (nYear <= nDuration)` и выполнение программы продолжится с выражения, следующего непосредственно за этим циклом.



Полный текст этой программы можно найти на прилагаемом компакт-диске в папке `CalculateInterestTableWithBreak` (он здесь не приводится для краткости изложения).

Введите сумму вклада i 100
 Введите процентную ставку : 25
 Введите количество лет: 100

Вклад = 100
 Проценты = 25%
 Срок = 100 years
 Выход по достижении коэффициента 10

1-125.00
 2-156.25
 3-195.31
 4-244.14
 5-305.18
 6-381.48
 7-476.85
 8-596.06
 9-745.08
 10-931.35
 11-1164.19

Нажмите <Enter> для завершения программы...

Программа прекращает работу, как только вычисленное значение вклада превышает 1000 — к счастью, этого не надо дожидаться 100 лет!

Цикл без счетчика

Программа `CalculateInterestTable` достаточно интеллектуальна для того, чтобы завершить работу, если пользователь ввел неверное значение вклада или процентной ставки. Однако трудно назвать дружественной программу, сразу же прекращающую работу, не давая пользователю ни одного шанса на исправление ошибки.



Комбинация `while` и `break` позволяет сделать программу немного более гибкой, что можно увидеть на примере исходного текста программы `CalculateInterestTableMoreForgiving`:

```
// CalculateInterestTableMoreForgiving
// Вычисление величины начисленных процентов для данного
// вклада за определенный период времени. Программа
// позволяет пользователю исправить ошибку ввода величины
// вклада и процентной ставки
using System;
namespace CalculateInterestTableMoreForgiving
{
    using System;

    public class Program
    {
        public static void Main(string[] args)
        {
            // Определяем максимально возможное значение
```

```

// процентной ставки
int nMaximumInterest = 50;

// Приглашение пользователю ввести величину исходного
// вклада; повторяем это приглашение до тех пор, пока
// не будет получено корректное значение
decimal mPrincipal;
while(true)
{
    Console.Write("Введите сумму вклада:");
    string sPrincipal = Console.ReadLine();
    mPrincipal = Convert.ToDecimal(sPrincipal);

    // Выход из цикла, если введенное значение корректно
    if (mPrincipal >= 0)
    {
        break;
    }

    // Генерируем сообщение о неверном вводе
    Console.WriteLine("Вклад не может быть отрицателен");
    Console.WriteLine("Повторите ввод");
    Console.WriteLine();
}

// Теперь вводим величину процентной ставки
decimal mInterest;
while(true)
{
    Console.Write("Введите процентную ставку:");
    string sInterest = Console.ReadLine();
    mInterest = Convert.ToDecimal(sInterest);

    // Если процентная ставка отрицательна или слишком
    // велика...
    if (mInterest >= 0 && mInterest <= nMaximumInterest)
    {
        break;
    }

    // ...генерируем сообщение об ошибке
    Console.WriteLine("Процентная ставка не может " +
        "быть отрицательна " +
        "или превышать " +
        nMaximumInterest);
    Console.WriteLine("Повторите ввод");
    Console.WriteLine();
}

// И величина вклада, и процентная ставка
// корректны — запрашиваем у пользователя срок,
// для которого следует вычислить величины вкладов
// с начисленными процентами

```

```

Console.Write("Введите количество лет:");
string sDuration = Console.ReadLine();
int nDuration = Convert.ToInt32(sDuration);

// Выводим введенные величины
Console.WriteLine(); // Пропуск строки
Console.WriteLine("Вклад = " + mPrincipal);
Console.WriteLine("Проценты = " +
    mInterest + "%");
Console.WriteLine("Срок      = " + nDuration
    " years");
Console.WriteLine();

// Цикл по указанному пользователем количеству лет
int nYear = 1;
while(nYear <= nDuration)
{
    // Вычисление вклада с начисленными процентами
    decimal mInterestPaid;
    mInterestPaid = mPrincipal * (mInterest / 100);

    // Вычисляем новое значение вклада
    mPrincipal = mPrincipal + mInterestPaid;

    // Округляем величину до копеек
    mPrincipal = decimal.Round(mPrincipal, 2);

    // Выводим результат
    Console.WriteLine(nYear + "-" + mPrincipal);

    // Переходим к следующему году
    nYear = nYear + 1;
}

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();

```



Данная программа во многом похожа на предыдущие примеры, за исключением исходного текста пользовательского ввода. В этот раз оператор `if`, выявлявший неверный ввод, заменен циклом `while`:

```

decimal mPrincipal;
while(true)
{
    Console.Write("Введите сумму вклада:");
    string sPrincipal = Console.ReadLine();
    mPrincipal = Convert.ToDecimal(sPrincipal);
}

```

```
// Выход из цикла, если введенное значение корректно
if (mPrincipal >= 0)
{
    break;
}

// Генерируем сообщение о неверном вводе
Console.WriteLine("Вклад не может быть отрицателен");
Console.WriteLine("Повторите ввод");
Console.WriteLine();
}
```

В представленном фрагменте кода пользовательский ввод выполняется в цикле. Если введенное значение корректно, программа выходит из цикла и продолжает выполнение. Однако если в пользовательском вводе имеется ошибка, пользователь получает сообщение о ней, и управление передается в начало цикла.



Обратите внимание на отсутствие условия в этом цикле, вернее, на то, что оно всегда истинно, а выход из цикла происходит при проверке условия в его теле. I

Обратите также внимание на обращение условия, поскольку теперь проблема не в том, чтобы вывести сообщение об ошибке при некорректном вводе, а в том, чтобы завершить цикл при корректном. Проверка условия

```
mInterest < 0 || mInterest > nMaximumInterest
```

превратилась в проверку

```
mInterest >= 0 && mInterest <= nMaximumInterest
```

Понятно, что условие `mInterest >= 0` противоположно условию `mInterest < 0`. Менее очевидна замена оператора ИЛИ (`|`) оператором И (`&&`). Теперь оператор `if` гласит: "Выйти из цикла, если процентная ставка не меньше нуля И не больше максимального значения (другими словами, имеет корректную величину)".

Кстати, каким образом можно изменить программу `CalculateInterestTableMoreForgiving` так, чтобы пользователь мог проводить вычисление за вычислением, вводя все новые значения вклада и процентной ставки, пока не захочет завершить работу с программой? Подсказка: для этого можно использовать еще один цикл `while (true)` со своим собственным условием выхода.

И последнее замечание: переменная `mPrincipal` должна быть объявлена за пределами цикла в соответствии с правилами видимости, которые будут объяснены в следующем разделе данной главы.



Это может звучать как тавтология, но вычисление выражения `true` дает значение `true`. Таким образом, `while (true)` представляет собой бесконечный цикл, и от заикливания спасает только наличие оператора `break` в теле цикла. При использовании цикла `while (true)` никогда не забывайте об операторе `break`, который должен прервать работу цикла по достижении заданного условия.

Вот как выглядит образец вывода программы:

```
Введите сумму вклада :-1000
Вклад не может быть отрицателен
Повторите ввод
```

Введите сумму вклада :1000
Введите процентную ставку :-10
Процентная ставка не может быть отрицательна или превышать 50
Повторите ввод

Введите процентную ставку:10
Введите количество лет:5

Вклад = 1000
Проценты = 10%
Срок , = 5 years

1-1100.0
2-1210.00
3-1331.00
4-1464.10
5-1610.51

Нажмите <Enter> для завершения программы...

Программа отказывается принимать отрицательные значения вклада и процентов, позволяя пользователю исправить ошибку ввода.



Всегда поясняйте пользователю, в чем именно он не прав, перед тем как дать ему возможность исправить допущенную ошибку. При ошибках, связанных с форматированием, неплохой идеей будет демонстрация примера корректного ввода. И будьте очень вежливы в своих сообщениях!

Правила области видимости

Переменная, объявленная в теле цикла, определена только внутри этого цикла. Рассмотрим следующий фрагмент исходного текста:

```
int nDays = 1;
while(nDays < nDuration)
{
    int nAverage = nValue / nDays;
    // ... Некоторая последовательность операторов ...
    nDays = nDays + 1;
}
```

Переменная **nAverage** не определена вне цикла **while**. Тому есть целый ряд причин, но рассмотрим одну из них: при первом выполнении цикла программа встречает объявление **int nAverage**. При втором выполнении цикла то же объявление встречается еще раз, так что если бы не было правила области видимости переменной, это привело бы к ошибке, так как переменная была бы уже определена.



Можно привести и другие, более убедительные причины правилу области видимости, но пока должно хватить и приведенного аргумента.

Достаточно сказать, что переменная **nAverage** прекращает свое существование при достижении программой закрывающей фигурной скобки и вновь создается при каждом выполнении тела цикла.



Опытные программисты говорят, что *область видимости* переменной `nAverage` ограничена циклом `while`.

Цикл for

Несмотря на свою простоту, цикл `while` все же является вторым по распространенности циклом в программах на C#. Пальму первенства прочно удерживает цикл `for` имеющий следующую структуру:

```
for(Выражение 1;    Условие;    Выражение2)
{
    // ... тело цикла ...
}
```

1

По достижении цикла `for` программа сначала выполняет *Выражение1*. Затем он вычисляет *Условие*, и если оно истинно, она выполняет тело цикла, которое заключен в фигурные скобки и следует сразу после оператора `for`. По достижении закрывающей скобки управление переходит *Выражению2*, после чего вновь вычисляется *Условие* и цикл повторяется.

Определение цикла `for` можно переписать как следующий цикл `while`:

```
Выражение 1 ;
while(Условие)
{
    // ... тело цикла ...
    Выражение2;
}
```

Пример

Возможно, вы лучше разберетесь, как работает цикл `for`, взглянув на конкретный пример:

```
// Некоторое выражение на C#
a = 1;
// Цикл
for(int nYear = 1; nYear < nDuration; nYear = nYear + 1)
{
    // ... тело цикла ...
}
// Здесь программа продолжается
a = 2;
```

Предположим, программа выполнила присваивание `a=1`; . После этого она объявляет переменную `nYear` и инициализирует ее значением `1`. Далее программа сравнивает значение `nYear` со значением `nDuration`. Если `nYear` меньше `nDuration`, выполняется тело цикла в фигурных скобках. По достижении закрывающей скобки программа возвращается к началу цикла и выполняет инструкцию `nYear=nYear+1`, перед тем как вновь перейти к проверке условия `nYear<nDuration`.



Переменная `nYear` не определена вне области видимости цикла `for`, которая включает как тело цикла, так и его заголовок.

Зачем нужны разные циклы

Зачем в C# нужен цикл `for`, если в нем уже есть такой цикл, как `while`? Наиболее простой и напрашивающийся ответ — он не нужен, так как цикл `for` не может сделать ничего такого, что нельзя было бы повторить с помощью цикла `while`.

Однако разделы цикла `for` повышают удобочитаемость исходных текстов, четко указывая три части, имеющиеся в каждом цикле: настройку, условие выхода и увеличение счетчика. Такой цикл не только проще для чтения и понимания, но и для проверки его корректности — вспомните, что основная ошибка при работе с циклом `while` — забытое увеличение счетчика или некорректный критерий завершения цикла. Недаром цикл `for` встречается в программах на порядок чаще других разновидностей циклов.



Так уж сложилось, что в основном в первой части цикла `for` выполняется инициализация переменной-счетчика, а в последней — ее увеличение. Но это не более чем традиция — C# не требует этого от программиста. В этих двух частях цикла `for` можно выполнять какие угодно действия, хотя, конечно же, для того чтобы отойти от традиционной схемы, нужны серьезные основания.

В цикле `for` особенно часто используется оператор инкремента (который вместе с другими операторами был описан в главе 4, "Операторы"). Обычно приведенный ранее цикл `for` записывается как

```
for(int nYear = 1; nYear < nDuration; nYear++)
{
    // ... тело цикла ...
}
```



Почти всегда в цикле `for` применяется постфиксная форма оператора инкремента, хотя в данном случае функционально она идентична префиксной.⁴

У цикла `for` имеется одно правило, которое я не в состоянии пояснить — если условие в цикле отсутствует, считается, что оно равно `true`.⁵ Таким образом, `for (; ;)` — такой же бесконечный цикл, как и `while (true)`.



В реальных программах `for (; ;)` в качестве бесконечного цикла используется значительно чаще, чем `while (true)`; но чем это вызвано, объяснить сложно.

⁴ К счастью, современные компиляторы достаточно интеллектуальны, чтобы понять, что в данном случае возвращаемое значение не играет никакой роли, и не выполнять дополнительной работы по сохранению начального значения переменной при использовании постфиксной формы оператора. — *Примеч. ред.*

⁵ Это правило легко пояснить тем, что если бы отсутствие условия воспринималось как `false`, то в таком цикле выполнялась бы исключительно первая часть заголовка, но не последняя и не тело цикла. — *Примеч. ред.*

Вложенные циклы

Один цикл может находиться в теле другого цикла:

```
for(... некоторое условие ...)  
{  
    for(... некоторое другое условие ...)  
    {  
        // ... какие-то действия ...  
    }  
}
```

Внутренний цикл выполняется полностью при каждом выполнении тела внешней цикла. Переменная-счетчик цикла, используемая во внутреннем цикле, является неопределенной вне области видимости внутреннего цикла.



Цикл, содержащийся внутри другого цикла, называется *вложенным* (nested). Вложенные циклы не могут "пересекаться", т.е. приведенный далее исходный текст некорректен:

```
do // Начало цикла do..while  
{  
    for( ... ) // Начало цикла for  
    {  
        while( ... ) // Конец цикла do..while  
    }  
} // Конец цикла for
```

Трудно даже предположить, что бы мог означать такой код, но это и неважно — равно такой код будет немедленно забракован компилятором.

Оператор `break` внутри вложенного цикла прекращает выполнение только этап вложенного цикла. В приведенном далее фрагменте исходного текста оператор `break` завершает работу цикла Б и возвращает управление циклу А:

```
// Цикл for А  
for(... некоторое условие ...)  
{  
    // Цикл for Б  
    for(... некоторое другое условие ...)  
    {  
        // ... некоторые действия ...  
        if (Истинное условие)  
        {  
            break; // Выход из цикла Б, но не из цикла А  
        }  
    }  
}
```



В C# нет команды, которая бы обеспечивала одновременный выход из обоих циклов. Это не настолько уж большое ограничение, как могло бы показаться. *На* практике зачастую *сложную* логику, содержащуюся внутри таких вложенных циклов, лучше инкапсулировать в виде функций. Выполнение оператора `return` в любом месте обеспечивает выход из функции, т.е. из всех циклов, какой бы ни была глубина вложенности. Функции и оператор `return` будут рассматриваться в главе 7, "Функции функций".



В приведенной далее программе `DisplayXWithNestedLoops`, которую можно назвать шуточной, пара вложенных циклов используется для того, чтобы вывести на экран большую букву *X*.

```
// DisplayXWithNestedLoops
// Использует пару вложенных циклов для вывода на экран
// большой буквы X
using System;

namespace DisplayXWithNestedLoops
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int nConsoleWidth = 40;
            // Итерация по строкам
            for(int nRowNum = 0;
                nRowNum < nConsoleWidth;
                nRowNum += 2)
            {
                // Итерация по столбцам
                for (int nColumnNum = 0;                                i
                    nColumnNum < nConsoleWidth; nColumnNum++)
                {
                    // По умолчанию используется пробел
                    char c = ' ';

                    // Если номер строки и столбца совпадают ...
                    if (nColumnNum == nRowNum)
                    {
                        // ... заменим пробел обратной косой чертой
                        c = '\\';
                    }

                    // Если столбец на противоположной
                    // стороне строки ...
                    int nMirrorColumn = nConsoleWidth - nRowNum;
                    if (nColumnNum == nMirrorColumn)
                    {
                        // ... заменим пробел косой чертой
                        c = '/';
                    }

                    // Вывод символа в текущей позиции
                    Console.Write(c);
                }
                Console.WriteLine();
            }

            // Ожидаем подтверждения пользователя
        }
    }
}
```

Сначала вычисляется выражение в круглых скобках после оператора **switch**. В данном случае это просто значение переменной **nMaritalStatus**. Затем вычисленное значение сравнивается со значениями у каждого из операторов **case**. Если нужное значение не найдено, управление передается операторам, следующим за меткой **default**.

Аргументом оператора **switch** может быть также строка **string**:

```
string s = "Davis";
switch(s)
{
    case "Mallory":
        // Некоторые действия
        break;
    case "Wells":
        // Некоторые действия
        break;
    case "Arturo":
        // Некоторые действия
        break;
    case "Brown":
        // Некоторые действия
        break;
    default:
        // Действия, если такой
        // фамилии нет в списке
}
```



При применении конструкции **switch** действует ряд ограничений.

- S** Аргумент оператора **switch()** должен иметь перечислимый тип или тип **string**.
- S** Нельзя использовать числа с плавающей точкой.
- ^** Значения **case** должны иметь тот же тип, что аргумент оператора **switch**.
- S** Значения **case** должны быть константами в том смысле, что их значения должны быть известны во время компиляции.
- S** Каждая конструкция **case** должна завершаться оператором **break** (или какой-то иной командой выхода, например, **return**). Оператор **break** передает управление за пределы конструкции **switch**.

Допускается наличие нескольких **case** для одного блока кода, как в следующем примере:

```
switch(nMaritalStatus)
{
    case 0.-
    case 2:
        // Действия для холостяков/незамужних и для
        // разведенных одни и те же
        break;
    case 1:
        // Действия для семейных
        break;
}
```

```

case 3:
    // Действия для вдов(цов)
    break;
case 4:
    // Действия для неизвестного семейного состояния
    break;
default:
    // Действия, когда переменная принимает значение,
    // отличающееся от всех перечисленных выше - по всей
    // видимости, это означает, что произошла какая-то
    // ошибка
    break;
}

```

Оператор goto

Управление может быть передано в другую точку при помощи оператора безусловного перехода **goto**. За этим оператором может следовать:

- ✓ метка;
- ✓ **case** в конструкции **switch**;
- ✓ ключевое слово **default**, обозначающее блок **default** в конструкции **switch**.

Два последних случая предназначены для перехода от одного блока **case** к другому в конструкции **switch**.

Вот пример применения оператора **goto**:

```

// Если условие истинно...
if (a > b)
{
    // Управление оператором goto передается коду,
    // расположенному за меткой exitLabel
    goto exitLabel;
}
// Некоторый программный код
exitLabel:
// Управление передается в эту точку

```

Оператор **goto** крайне непопулярен по той же причине, по которой он является очень мощным средством — в силу его полной неструктурированности. Отслеживание переходов в нетривиальных случаях, превышающих несколько строк кода, — крайне неблагоприятная задача. Это именно тот случай, когда говорят о "соплях" в программе.



Вокруг применения **goto** ведутся почти "религиозные войны". Доходит до критики C# просто за то, что в нем есть этот оператор. Но на самом деле в нем нет ничего ужасного или демонического. Другое дело, что его применения следует избегать, если в этом нет крайней нужды.

Часть III

Объектно-основанное программирование



В этой части...

Одно дело — объявить переменные там и сям, и складывать их, и вычитать. И совсем другое — писать реальные программы, которыми может пользоваться еще кто-то. В этой части речь пойдет о том, как группировать данные и работать с ними. Вы научитесь думать о программах как о наборе сотрудничающих *объектов* и приступите к созданию собственных объектов. Это заложит основу для всей вашей будущей деятельности в качестве программистов.

Глава 6

Объединение данных - классы и массивы

В этой главе...

- > Введение в классы C#
- > Хранение данных в объектах
- > Ссылки на объекты
- > Создание массивов объектов

Вы можете свободно объявлять и использовать все встроенные типы данных — такие как `int`, `double` или `bool` — для хранения информации, необходимой вашей программе. Для ряда программ таких простых переменных вполне достаточно, но большинству программ требуется средство для объединения связанных данных в аккуратные пакеты.

Некоторым программам надо собрать вместе данные, связанные логически, но имеющие разные типы. Например, приложение, работающее со списками студентов, должно хранить разнотипную информацию о них — имя, год рождения, успеваемость и т.п. Логически рассуждая, имя студента может иметь тип `string`, год рождения — `int` или `short`, средний балл — `double`. Такой программе необходима возможность объединить эти разнотипные переменные в единую структуру под именем `Student`. К счастью, в C# имеется структура, известная как *класс*, которая предназначена для облегчения группирования таких разнотипных переменных.

В других случаях программам требуются наборы однотипных объектов. Возьмем для примера программу, которая должна усреднять успеваемость. Тип `double` — естественный кандидат для представления индивидуальной успеваемости студента, но для того чтобы представлять успеваемость группы студентов, необходим тип, который является набором `double`. Для таких целей в C# существуют *массивы*.

И наконец, реальной программе для работы с информацией о студентах могут понадобиться как классы, так и массивы, причем объединенные в одно целое — массив студентов. C# позволяет получить желаемое.

Классы

Класс представляет собой объединение разнотипных данных и функций, логически организованных в единое целое. C# предоставляет полную свободу при создании классов, но хорошо спроектированные классы призваны представлять *концепции*.

Аналитик скорее всего скажет, что "класс отображает концепцию из предметной области задачи в программу". Предположим, например, что ваша задача — построения имитатора дорожного движения, который должен смоделировать улицы, перекрестка шоссе и т.п.

Любое описание такой задачи должно включать термин *транспортное средство*. Транспортные средства обладают определенной максимальной скоростью движения, имеют вес, и некоторые из них оснащены прицепами. Таким образом, имитатор дорожного движения должен включать класс **Vehicle**, у которого должны иметься свойства наподобие **dTopSpeed**, **nWeight** и **bTrailer**.

Поскольку классы — центральная концепция в программировании на C#, они будут гораздо детальнее рассмотрены в главах части IV, "Объектно-ориентированное программирование"; здесь же описаны только азы.

Определение класса

Пример класса **Vehicle** может выглядеть следующим образом:

```
public class Vehicle
{
    public string sModel;           // Название модели
    public string sManufacturer;    // Производитель
    public int nNumOfDoors;         // Количество дверей
    public int nNumOfWheels;        // Количество колес
}
```

Определение класса начинается словами **public class**, за которыми идет имя класса (в данном случае — **Vehicle**).



Как и все имена в C#, имена классов чувствительны к регистру. C# не имея никаких правил для именования классов, но неофициальная традиция гласит, что имена классов начинаются с прописной буквы.

За именем класса следует пара фигурных скобок, внутри которых могут быть не сколько *членов* (либо ни одного). Члены класса представляют собой переменные, образующие часть класса. В данном примере класс **Vehicle** начинается с члена **string sModel**, который содержит название модели транспортного средства. Второй член в этом примере — **string sManufacturer**, а последние два члена содержат количество дверей и колес в транспортном средстве.



Как и в случае обычных переменных, делайте имена членов максимально информативными. Хотя я и добавил к именам членов комментарии, они не являются обязательными с точки зрения C#. Обо всем должны говорить имена переменных.

Модификатор **public** перед именем класса делает класс доступным для всей программы. Аналогично, модификаторы **public** перед именами членов также делают их доступными для всей программы. Возможны и другие модификаторы, но более подробно о доступности речь пойдет в главе 11, "Классы".

Определение класса должно описывать свойства объектов решаемой задачи. Сделать это прямо сейчас вам будет немного сложно, поскольку вы не знаете, в чем именно состоит задача, так что просто следите за ходом изложения.

Что такое объект

Объект класса объявляется аналогично встроенным объектам **C#**, но не идентично им.



Термин *объект* используется для обозначения "вещей". Да, это не слишком полезное определение, так что приведем несколько примеров. Переменная типа **int** является объектом **int**. Автомобиль является объектом **Vehicle**.

Вот фрагмент кода, создающий автомобиль, являющийся объектом **Vehicle**:

```
Vehicle myCar;  
myCar = new Vehicle () ;
```

В первой строке объявлена переменная **myCar** типа **Vehicle**, так же как вы можете объявить переменную **nSomething** класса **int** (да, класс является типом, и все объекты **C#** определяются как классы). Команда **new Vehicle ()** создает конкретный объект типа **Vehicle** и сохраняет его местоположение в переменной **myCar**. Оператор **new** ("новый") не имеет ничего общего с возрастом автомобиля — он просто выделяет новый блок памяти, в котором ваша программа может хранить свойства автомобиля **myCar**.



В терминах **C#** **myCar** — это объект класса **Vehicle**. Можно также сказать, что **myCar** — экземпляр класса **Vehicle**. В данном контексте *экземпляр* (instance) означает "пример" или "один из". Можно использовать этот термин и как глагол, и говорить об *инстанцировании* **Vehicle** — это именно то, что делает оператор **new**.

Сравните объявление **myCar** с объявлением переменной **num** типа **int**:

```
int num;  
num = 1;
```

В первой строке объявлена переменная **num** типа **int**, а во второй созданной переменной присваивается значение типа **int**, которое вносится в память по месту расположения переменной **num**.



Переменная встроенного типа **num** и объект **myCar** хранятся в памяти по-разному. Константа **1** не занимает память, поскольку и процессор, и **C#** знают, что такое **1**. Но процессор понятия не имеет о такой концепции, как **Vehicle**. Выражение **new Vehicle** выделяет память, необходимую для описания транспортного средства, понятного процессору, **C#**, и вообще — всему миру.

Доступ к членам объекта

Каждый объект класса **Vehicle** имеет собственный набор членов. Приведенное далее выражение сохраняет число **1** в члене **nNumberOfDoors** объекта, на который ссылается **myCar**:

```
myCar.nNumberOfDoors = 1;
```



Каждый оператор **C#** имеет не только значение, но и тип. Объект **myCar** является объектом типа **Vehicle**. Переменная **Vehicle.nNumberOfDoors** имеет тип **int** (вернитесь к определению класса **Vehicle**). Константа **1** также имеет тип **int**, так что типы константы с правой стороны от оператора присваивания и переменной с левой стороны соответствуют друг другу.

Аналогично, в следующем фрагменте кода сохраняются ссылки на строки `string` описывающие модель и производителя `myCar`:

```
myCar.sManufacturer = "BMW";  
myCar.sModel = "Isetta";
```

(Isetta— небольшой автомобиль, который производился в 1950-х годах и имел одну дверь впереди.)

Пример объектно-основанной программы

Программа `VehicleDataOnly` очень проста и делает следующее:

- ✓ определяет класс `Vehicle`;
- ✓ создает объект `myCar`;
- ✓ указывает свойства `myCar`;
- ✓ получает значения свойств `myCar` и выводит их на экран.



Вот код программы `VehicleDataOnly`:

```
// VehicleDataOnly  
  
// Создает объект Vehicle, заполняет его члены информацией,  
// вводимой с клавиатуры, и выводит ее на экран  
using System;  
  
namespace VehicleDataOnly  
{  
    public class Vehicle  
    {  
        public string sModel;           // Модель  
        public string sManufacturer;    // Производитель  
        public int nNumOfDoors,-         // Количество дверей  
        public int nNumOfWheels,-       // Количество колес  
    }  
    public class Program  
    {  
        // Начало программы  
        static void Main(string[] args)  
        {  
            // Приглашение пользователю  
            Console.WriteLine("Введите информацию о машине");  
  
            // Создание экземпляра Vehicle  
            Vehicle myCar = new Vehicle ();  
  
            // Ввод информации для членов класса  
            Console.Write("Модель = ");  
            string s = Console.ReadLine();  
            myCar.sModel = s;  
        }  
    }  
}
```

```
// Можно присваивать значения непосредственно
Console.Write("Производитель = ");
myCar.sManufacturer = Console.ReadLine();

// Остальные данные имеют тип int
Console.Write("Количество дверей = ");
s = Console.ReadLine();
myCar.nNumOfDoors = Convert.ToInt32(s), -

Console.Write("Количество колес = ");
s = Console.ReadLine();
myCar.nNumOfWheels = Convert.ToInt32(s);

// Вывод полученной информации
Console.WriteLine("\nВаша машина: ");
Console.WriteLine(myCar.sManufacturer + " " +
myCar.sModel);
Console.WriteLine("с " + myCar.nNumOfDoors +
" дверями, "
+ "на " + myCar.nNumOfWheels
+ " колесах");

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
"завершения программы...");
Console.Read();
```



Листинг программы начинается с определения класса **Vehicle**.



Определение класса может находиться как до, так и после класса **Program** — это не имеет значения. Однако вам нужно принять какой-то один стиль и следовать ему.

Программа создает объект **myCar** класса **Vehicle**, а затем заполняет все его поля информацией, вводимой пользователем с клавиатуры. Проверка корректности входных данных не производится. Затем программа выводит введенные данные на экран в немногочисленном формате.

Вывод этой программы выглядит следующим образом:

```
Введите информацию о машине
Модель          = Metropolitan
Производитель   = Nash
Количество дверей = 2
Количество колес  = 4
```

Ваша машина:

Nash Metropolitan

с 2 дверями, на 4 колесах

Нажмите <Enter> для завершения программы...



Вызов **Read()**, в отличие от **ReadLine()**, оставляет курсор сразу за введенной строкой. Это приводит к тому, что ввод пользователя находится на той же строке, что и приглашение. Кроме того, добавление символа новой строки '\n' создает пустую строку без необходимости вызывать **WriteLine()**.

Отличие между объектами

Заводы в Детройте в состоянии выпускать множество автомобилей и отслеживать каждую выпущенную машину и при этом не путать их. Аналогично, программа может создать несколько объектов одного и того же класса, как показано в следующем фрагменте:

```
Vehicle car1 = new Vehicle();
car1.sManufacturer = "Studebaker";
car1.sModel = "Avanti";

// Следующий код никак не влияет на car1
Vehicle car2 = new Vehicle();
car2.sManufacturer = "Hudson";
car2.sModel = "Hornet";
```

Создание объекта **car2** и присваивание ему имени и производителя никак не влияет на объект **car1**.

Возможность различать объекты одного класса очень важна при программировании. Объект может быть создан, с ним могут быть выполнены различные действия — и он всегда выступает как единый объект, отличный от других подобных ему объектов.

Ссылки

Оператор "точка" и оператор присваивания — единственные два оператора, определенные для ссылочных типов. Рассмотрим следующий фрагмент исходного текста:

```
// Создание нулевой ссылки
Vehicle yourCar;
// Присваивание значения ссылке
yourCar = new Vehicle();
// Использование точки для обращения к члену
yourCar.sManufacturer = "Rambler";
// Создание новой ссылки, которая указывает на тот же объект
Vehicle yourSpousalCar = yourCar;
```

В первой строке создается объект **yourCar**, причем без присваивания ему значения. Такая неинициализированная ссылка называется *нулевым объектом* (null object). Любые попытки использовать неинициализированную ссылку приводят к немедленной генерации ошибки, которая прекращает выполнение программы.



Компилятор C# может перехватить большинство попыток использования неинициализированной ссылки и сгенерировать предупреждение в процессе компиляции программы. Если вам каким-то образом удалось провести компьютер то обращение к неинициализированной ссылке при выполнении программы приведет к ее аварийному останову.

Второе выражение создает новый объект **Vehicle** и присваивает его ссылке **yourCar**. И последняя строка кода присваивает ссылке **yourSpousalCar**

ссылку `yourCar`. Как показано на рис. 6.1, это приводит к тому, что `yourSpousalCar` ссылается на тот же объект, что и `yourCar`.



Рис. 6.1. Взаимоотношения между двумя ссылками на один и тот же объект

Эффект от следующих двух вызовов одинаков:

```
// Создание вашей машины
Vehicle yourCar = new Vehicle () ;
yourCar.sModel = "Kaiser";
// Эта машина принадлежит и вашей жене
Vehicle yourSpousalCar = yourCar;
// Изменяя одну машину, вы изменяете и другую
yourSpousalCar.sModel = "Henry J";
Console.WriteLine("Ваша машина - " + yourCar.sModel);
```

Выполнение данной программы приводит к выводу на экран названия модели **Henry J**, а не **Kaiser**. Обратите внимание, что `yourSpousalCar` не указывает на `yourCar`— вместо этого просто и `yourSpousalCar`, и `yourCar` указывают на один и тот же объект.

Кроме того, ссылка `yourSpousalCar` будет корректна, даже если окажется "потерянной" (например, при выходе за пределы области видимости), как показано в следующем фрагменте:

```
// Создание вашей машины
Vehicle yourCar = new Vehicle () ;
yourCar.sModel = "Kaiser";
// Эта машина принадлежит и вашей жене
Vehicle yourSpousalCar = yourCar;
// Когда она забирает себе вашу машину...
yourCar = null;           // yourCar теперь ссылается на "нулевой
                          // объект"
// ...yourSpousalCar ссылается на все ту же машину
Console.WriteLine("Ваша машина - " + yourSpousalCar.sModel);
```

Выполнение этого фрагмента исходного текста выводит на экран сообщение **"Ваша машина - Kaiser"** несмотря на то, что ссылка `yourCar` стала недействительной.



Объект перестал быть *достижимым* по ссылке `yourCar`. Но он не будет полностью *недостижимым*, пока не будут "потеряны" или обнулены обе ссылки — и `yourCar`, и `yourSpousalCar`.

После этого — вернее будет сказать, в некоторый непредсказуемый момент после этого — *сборщик мусора* (garbage collector) C# вернет память, использованную ранее под

объект, все ссылки на который утрачены. Дополнительные сведения о сборке мусора будут приведены в конце главы 12, "Наследование".

Классы, содержащие классы

Члены класса могут, в свою очередь, быть ссылками на другие классы. Например! транспортное средство имеет двигатель, свойствами которого являются, в частности! мощность и рабочий объем. Можно поместить эти параметры непосредственно в класс **Vehicle** следующим образом:

```
public class Vehicle
{
    public string sModel;           // Модель
    public string sManufacturer;    // Производитель
    public int nNumOfDoors;         // Количество дверей
    public int nNumOfWheels;        // Количество колес
    public int nPower;              // Мощность двигателя
    public double displacement;     // Рабочий объем
}
```

Однако мощность и рабочий объем двигателя не являются свойствами автомобиля,! так как, например, джип моего сына может поставляться с одним из двух двигателей! с совершенно разными мощностями.

Двигатель является самодостаточной концепцией и может быть описан отдельным! классом:

```
public class Motor
{
    public int nPower;              // Мощность
    public double displacement;     // Рабочий объем
}
```

Вы можете внести этот класс в класс **Vehicle** следующим образом:

```
public class Vehicle
{
    public string sModel;           // Модель
    public string sManufacturer;    // Производитель
    public int nNumOfDoors;         // Количество дверей
    public int nNumOfWheels;        // Количество колес
    public Motor motor;
}
```

Соответственно, создание **sonsCar** теперь выглядит так:

```
// Сначала создаем двигатель
Motor largerMotor = new Motor();
largerMotor.nPower = 230;
largerMotor.displacement = 4.0;
// Теперь создаем автомобиль
Vehicle sonsCar = new Vehicle();
sonsCar.sModel = "Cherokee Sport";
sonsCar.sManufacturer = "Jeep";
sonsCar.nNumOfDoors = 2;
sonsCar.nNumOfWheels = 4;
// Присоединяем двигатель к автомобилю
sonsCar.motor = largerMotor;
```

Доступ к рабочему объему двигателя из `Vehicle` можно получить в два этапа, как показано в приведенном фрагменте:

```
Motor m = sonsCar.motor;  
Console.WriteLine("Рабочий объем равен " + m.displacement);
```

Однако можно получить эту величину и непосредственно:

```
Console.WriteLine("Рабочий объем равен " +  
    sonsCar.motor.displacement);
```

В любом случае доступ к значению `displacement` осуществляется через класс `Motor`.



Этот фрагмент взят из исходного текста программы `VehicleAndMotor`, которую можно найти на прилагаемом компакт-диске.

Статические члены класса

Большинство членов-данных описывают отдельные объекты. Рассмотрим следующий класс `Car`:

```
public class Car  
{  
    public string sLicensePlate; // Номерной знак автомобиля
```

Номерной знак является *свойством объекта*, описывающим каждый автомобиль и уникальным для каждого автомобиля. Присваивание номерного знака одному автомобилю не меняет номерной знак другого:

```
Car cousinsCar = new Car();  
cousinsCar.sLicensePlate = "XYZ123";
```

```
Car yourCar = new Car();  
yourCar.sLicensePlate = "ABC789";
```

Однако имеются и такие свойства, которые присущи всем автомобилям. Например, количество выпущенных автомобилей является *свойством класса* `Car`, но не отдельного объекта. Свойство класса помечается специальным ключевым словом `static`, как показано в следующем фрагменте исходного текста:

```
public class Car  
{  
    public static  
        int nNumberOfCars; // Количество выпущенных автомобилей  
    public string sLicensePlate; // Номерной знак автомобиля  
}
```

Обращение к статическим членам выполняется не посредством объекта, а через сам класс, как показано в следующем фрагменте исходного текста:

```
// Создание нового объекта класса Car  
Car newCar = new Car();  
newCar.sLicensePlate = "ABC123";  
// Увеличиваем количество автомобилей на 1  
Car.nNumberOfCars++;
```




Обращение к члену объекта `newCar.sLicensePlate` выполняется посредством объекта `newCar`, в то время как обращение к (статическому) члену `Car.nNumberOfCars` осуществляется с помощью имени класса. Все объекты типа `Car` совместно используют один и тот же член `nNumberOfCars`.

Определение константных членов-данных

Одним специальным видом статических членов является член, представляющий собой константу. Значение такой константной переменной должно быть указано в объявлении, и не может изменяться нигде в программе, как показано в следующем фрагменте исходного текста:

```
class Program
{
    // Число дней в году (включая високосный год)
    public const int nDaysInYear = 366;
    public static void Main(string[] args)
    {
        // Это массив — о нем будет рассказано немного позже
        int[] nMaxTemperatures = new int[nDaysInYear];
        for(int index = 0; index < nDaysInYear; index++)
        {
            // Вычисление средней температуры для каждого дня года
        }
    }
}
```

Константу `nDaysInYear` можно использовать везде в вашей программе вместо значения `366`. Константные переменные очень полезны, так как позволяют заменить "магические числа" (в данном случае — `366`) описательным именем `nDaysInYear`, *повышает удобочитаемость программы и облегчает ее сопровождение.

C# имеет еще один способ объявления констант. Возможно, вам больше придется использовать не модификатор `const`, а модификатор `readonly`:

```
public readonly int nDaysInYear = 366;
```

Как и при применении модификатора `const`, после того как вы присвоите константе инициализирующее значение, оно не может быть изменено нигде в программе. Хотя причины этого совета носят слишком технический характер, чтобы описывать их в стоящей книге, при объявлении констант предпочтительно использовать модификатор `readonly`.

Для констант имеется собственное соглашение по именованию. Вместо именования их так же, как и переменных (как в примере с `nDaysInYear`) многие программы предпочитают использовать прописные буквы с разделением слов подчеркиваниями `DAYS_IN_YEAR`. Такое соглашение ясно отделяет константы от обычных переменных.

Массивы C#

В вашем распоряжении есть переменные, хранящие отдельные единственные значения. Классы могут использоваться для описания составных объектов. Но вам нужна еще

одна конструкция для хранения множества объектов, например, коллекции старинных автомобилей Билла Гейтса. Встроенный класс `Array` представляет собой структуру, которая может содержать последовательности однотипных элементов (чисел типа `int`, `double`, объекты `Vehicle` и т.п.).

Зачем нужны массивы

Рассмотрим задачу определения среднего из десяти чисел с плавающей точкой. Каждое из 10 чисел требует собственной переменной для хранения значения типа `double` (усреднение целых чисел может привести к ошибке округления, описанной в главе 3, "Объявление переменных-значений"):

```
double dO = 5;
double d1 = 2;
double d2 = 7;
double d3 = 3.5;
double d4 = 6.5;
double d5 = 8;
double d6 = 1;
double d7 = 9;
double d8 = 1;
double d9 = 3;
```

Теперь нужно просуммировать все эти значения и разделить полученную сумму на 10:

```
double dSum = dO + d1 + d2 + d3 + d4 +
              d5 + d6 + d7 + d8 + d9;
double dAverage = dSum / 10;
```

Перечислять все элементы — очень утомительно, даже если их всего 10. А теперь представьте, что вам надо усреднить 10000 чисел...

Массив фиксированного размера

К счастью, вам не нужно отдельно именовать каждый из элементов. C# предоставляет в распоряжение программиста массивы, которые могут хранить последовательности значений. Используя массив, вы можете переписать приведенный выше фрагмент следующим образом:

```
double[] dArray = {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
```



Класс `Array` использует специальный синтаксис, который делает его более удобным в применении. Двойные квадратные скобки `[]` указывают на способ доступа к отдельным элементам массива:

```
dArray[0] соответствует dO
dArray[1] соответствует d1
```

Нулевой элемент массива соответствует `dO`, первый — `d1` и так далее.



Номера элементов массива — 0, 1, 2, ... — известны как их *индексы*.



В C# индексы массивов начинаются с 0, а не с 1. Таким образом, элемент с индексом 1 не является первым элементом массива. Не забывайте об этом!

Использование `dArray` было бы слабым улучшением, если бы в качестве индекса массива нельзя было использовать переменную. Применять цикл `for` существенно проще, чем записывать каждый элемент вручную, что и демонстрирует программа `FixedArrayAverage`.

```
11 'FixedArrayAverage
// Усреднение массива чисел фиксированного размера с
// использованием цикла
namespace FixedArrayAverage

using System;

public class Program
{
    public static void Main(string[] args)

        double[] dArray =
            {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};

    // Накопление суммы элементов
    // массива в переменной dSum
    double dSum = 0,-
    for (int i = 0; i < 10; i++\
    |
        dSum = dSum + dArray[i];

    // Вычисление среднего значения
    double dAverage = dSum / 10;
    Console.WriteLine(dAverage);

    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
        "завершения программы...");
    Console.Read();
}
```

Программа начинает работу с инициализации переменной `dSum` значением 0. Затем программа циклически проходит по всем элементам массива `dArray` и прибавляет к `dSum`. По окончании цикла сумма всех элементов массива хранится в `dSum`. Разделив ее на количество элементов массива, получаем искомое среднее значение.

Проверка границ массива

Программа `FixedArrayAverage` должна циклически проходить по массиву из 10 элементов. К счастью, цикл разработан так, что проходит ровно по 10 элементам массива. Ну а если была бы допущена ошибка и проход был бы сделан не по 10 элементам, а по иному их количеству? Следует рассмотреть два основных случая.

Что произойдет при выполнении 9 итераций? C# не трактует такую ситуацию как ошибочную. Если вы хотите рассмотреть только 9 из 10 элементов, то как C# может указывать вам, что именно вам нужно делать? Конечно, среднее значение при этом будет неверным, но программе это неизвестно.

Что произойдет при выполнении 11 (или большего количества) итераций?

В этом случае C# примет свои меры и не позволит индексу выйти за дозволенные пределы, чтобы вы не смогли случайно переписать какие-нибудь важные данные в памяти. Чтобы убедиться в этом, измените сравнение в цикле `for`, заменив 10 на 11: `for (int i = 0; i < 11; i++)`

При выполнении программы вы получите диалоговое окно со следующим сообщением об ошибке:

```
IndexOutOfRangeException was unhandled  
Index was outside the bounds of the array.
```

Здесь C# сообщает о происшедшей неприятности — исключении `IndexOutOfRangeException`, из названия которого и из поясняющего текста становится понятна причина ошибки, — выход индекса за пределы допустимого диапазона. (Кроме этого, выводится детальная информация о том, где именно и что произошло, но пока то вы не настолько знаете C#, чтобы разобраться в этом.)

Массив переменного размера

Массив, используемый в программе `FixedArrayAverage`, сталкивается с двумя серьезными проблемами:

- ✓ его размер фиксирован и равен 10 элементам;
- ✓ что еще хуже, значения этих элементов указываются непосредственно в тексте программы.

Значительно более гибкой была бы программа, которая могла бы считывать переменной количество значений, вводимое пользователем, ведь она могла бы работать не только с определенными в программе `FixedArrayAverage` значениями, но и с другими множествами значений.

Формат объявления массива переменного размера немного отличается от объявления «ива фиксированного размера:

```
double[] dArray = new double[N];
```

Здесь `N` — количество элементов в выделяемом массиве.



Модифицированная версия программы `VariableArrayAverage` позволяет пользователю указать количество вводимых значений. Поскольку программа сохраняет введенные значения, она может не только вычислить среднее значение, но и вывести результат в удобном виде.

```
// VariableArrayAverage  
  
//Вычисление среднего значения массива, размер которого  
//указывается пользователем во время работы программы.  
// Накопление введенных данных в массиве позволяет  
//обращаться к ним неоднократно, в частности, для генерации  
//привлекательно выглядящего вывода на экран.
```

```

namespace VariableArrayAverage
{
    using System;

    public class Program
    {
        public static void Main(string[] args)
        {
            // Сначала считывается количество чисел типа double,
            // которое пользователь намерен ввести для усреднения
            Console.WriteLine("Введите количество усредняемых чисел: ");
            string sNumElements = Console.ReadLine();
            int numElements = Convert.ToInt32(sNumElements);
            Console.WriteLine();

            // Объявляем массив необходимого размера
            double[] dArray = new double[numElements];

            // Накапливаем значения в массиве
            for (int i = 0; i < numElements; i++)
            {
                // Приглашение пользователю для ввода чисел
                Console.WriteLine("Введите число типа double №" +
                                   (i + 1) + ": ");
                string sVal = Console.ReadLine();
                double dValue = Convert.ToDouble(sVal);

                // Вносим число в массив
                dArray[i] = dValue;
            }

            // Суммируем 'numElements' значений из массива в
            // переменной dSum
            double dSum = 0;
            for (int i = 0; i < numElements; i++)
            {
                dSum = dSum + dArray[i];
            }

            // Вычисляем среднее
            double dAverage = dSum / numElements;

            // Выводим результат на экран
            Console.WriteLine();
            Console.WriteLine(dAverage
                               + " является средним из ("
                               + dArray[0] ),-
            for (int i = 1; i < numElements; i++)
            {
                Console.WriteLine(" + " + dArray[i]);
            }
            Console.WriteLine(") / " + numElements);
        }
    }
}

```

```
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
```



Вот как выглядит вычисление среднего для чисел от 1 до 5:

Введите количество усредняемых чисел: 5

Введите число типа double №1: 1
 Введите число типа double №2: 2
 Введите число типа double №3: 3
 Введите число типа double №4: 4
 Введите число типа double №5: 5

3 является средним из $(1 + 2 + 3 + 4 + 5) / 5$
 Нажмите <Enter> для завершения программы...

Программа `VariableArrayAverage` начинается с вывода приглашения пользователя указать количество значений, которые будут введены далее и которые требуется усреднить. Введенное значение сохраняется в переменной `numElements` типа `int`. В представленном примере здесь оказывается введено число 5.

Затем программа выделяет память для нового массива `dArray` с указанным количеством элементов. В данном случае программа это делает для массива, состоящего из пяти элементов типа `double`. Программа выполняет `numElements` итераций цикла, считывая вводимые пользователем значения и заполняя ими массив.

После того как пользователь введет указанное им ранее число данных, программа использует тот же алгоритм, что и в программе `FixedArrayAverage` для вычисления среднего значения последовательности чисел.

В последней части генерируется вывод среднего значения вместе с введенными числами в привлекательном виде (по крайней мере с моей точки зрения).



Этот вывод не так уж и прост, как может показаться. Внимательно проследите, как именно программа выводит открывающую скобку, знаки сложения, числа последовательности и закрывающую скобку.



Программа `VariableArrayAverage`, возможно, не удовлетворяет вашим представлениям о гибкости. Может статься, что вам бы хотелось позволить пользователю вводить числа, а после ввода какого-то очередного числа дать команду вычислить среднее значение введенных чисел. Кроме массивов, C# предоставляет программисту и другие типы *коллекций*; некоторые из них могут при необходимости увеличивать или уменьшать свой размер. В главе 15, "Обобщенное программирование", вы познакомитесь с такими альтернативами массивам.

Свойство Length

В программе `VariableArrayAverage` для заполнения массива использован цикл `for`:

```
// Объявляем массив необходимого размера
double[] dArray = new double[numElements];

// Накапливаем значения в массиве
for (int i = 0; i < numElements; i++)
{
    // Приглашение пользователю для ввода чисел
    Console.WriteLine("Введите число типа double №" +
        (i + 1) + ": »);,;
    string sVal = Console.ReadLine();
    double dValue = Convert.ToDouble(sVal);

    // Вносим число в массив
    dArray[i] = dValue;
}
```

Массив `dArray` объявлен как имеющий длину `numElements`. Таким образом, понятно, почему цикл выполняет именно `numElements` итераций для прохода по массиву.

Вообще говоря, не слишком-то и удобно таскать повсюду вместе с массивом переменную, в которой хранится его длина. Но, к счастью, это не является неизбежным: у массива есть свойство `Length`, которое содержит его длину, так что `dArray.Length` в данном случае содержит то же значение, что и `numElements`.

Таким образом, предпочтительнее использовать такой вид цикла `for`:

```
// Накапливаем значения в массиве
for (int i = 0; i < dArray.Length; i++)
```

Отличие массивов фиксированной и переменной длины

С первого взгляда бросается в глаза, насколько отличается синтаксис массивов фиксированной и переменной длины:

```
double[] dFixedLengthArray = {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
double[] dVariableLengthArray = new double[10];
```

Однако C# пытается сэкономить ваши усилия, а потому позволяет использовать тот же код для выделения памяти для массива с его одновременной инициализацией:

```
double[] dFixedLengthArray =
    new double[10] {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
```

Здесь память выделяется как для массива переменной длины, но его инициализация выполняется так же, как и для массива фиксированной длины.

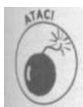
Массивы объектов

Программистам очень часто приходится работать с наборами пользовательских объектов (классов). Скажем, программе для университета требуется некая структура, описывающая студентов, например, такая:

```
public class Student
{
    public string sName;
    public double dGPA; // Средний балл
}
```

Этот простейший класс содержит только имя студента и его успеваемость. В следующем фрагменте объявляется массив из `num` ссылок на объекты типа `Student`:

```
Student[] students = new Student[num];
```



Выражение `new Student[num]` объявляет *не* массив объектов `Student`, а массив *ссылок* на объекты `Student`.

Итак, каждый элемент `students[i]` представляет собой ссылку на нулевой объект, так как C# инициализирует новые, неопределенные объекты значением `null`. Можно сформулировать это и иначе, сказав, что ни один из элементов `students[i]` после выполнения рассматриваемого оператора `new` не ссылается на объект типа `Student`. Так что сначала вы должны заполнить массив следующим образом:

```
for (int i = 0; i < students.Length; i++)
{
    students[i] = new Student();
}
```

Теперь в программе можно вводить свойства отдельных студентов таким образом:

```
students[i] = new Student();
students[i].sName = "My Name";
students[i].dGPA = dMyGPA;
```



Все это можно увидеть в программе `AverageStudentGPA`, которая получает информацию о студентах и вычисляет их среднюю успеваемость.

```
// AverageStudentGPA
// Вычисляет среднюю успеваемость множества студентов.
using System;

namespace AverageStudentGPA
{
    public class Student
    {
        public string sName;
        public double dGPA;           // Средний балл
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            // Определяем количество студентов
            Console.WriteLine("Введите количество студентов");
            string s = Console.ReadLine();
            int nNumberOfStudents = Convert.ToInt32(s);
            // Выделяем массив объектов Student

            Student[] students = new Student[nNumberOfStudents];

            // Заполняем массив
            for (int i = 0; i < students.Length; i++)
```



```

{
    // Приглашение для ввода информации. Единица
    // прибавляется в связи с тем, что индексация
    // массивов в C# начинается с нуля
    Console.Write("Введите имя студента "
        + (i + 1) + ": ");
    string sName = Console.ReadLine();

    Console.Write("Введите средний балл студента: ");
    string sAvg = Console.ReadLine();
    double dGPA = Convert.ToDouble(sAvg);

    // Создаем объект на основе введенной информации
    Student thisStudent = new Student();
    thisStudent.sName = sName;
    thisStudent.dGPA = dGPA;

    // Добавляем созданный объект в массив
    students[i] = thisStudent;
}

// Усредняем успеваемость студентов
double dSum = 0.0;
for (int i = 0; i < students.Length; i++)
{
    dSum += students[i].dGPA;
}
double dAvg = dSum/students.Length;

// Выводим вычисленное значение
Console.WriteLine();
Console.WriteLine("Средняя успеваемость по "
    + students.Length
    + " студентам - " + dAvg);

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}
}
}

```

Программа предлагает пользователю ввести количество рассматриваемых студентов после чего создает массив соответствующего размера, элементами которого являю ссылки на объекты типа `Student`.

После этого программа входит в цикл `for`, в котором происходит заполнение массива. Пользователю предлагается ввести имя каждого студента и его средний балл — данные используются для создания объекта `Student`.

После заполнения массива программа входит во второй цикл, в котором усредняется успеваемость студентов, а после вычислений полученный средний балл выводится на экран.

Вот как выглядит типичный результат работы программы:

```
Введите количество студентов
3
Введите имя студента 1: Randy
Введите средний балл студента: 3.0
Введите имя студента 2: Jeff
Введите средний балл студента: 3.5
Введите имя студента 3: Carrie
Введите средний балл студента: 4.0

Средняя успеваемость по 3 студентам - 3.5
Нажмите <Enter> для завершения программы...
```



Имя ссылочной переменной лучше делать состоящим из одного слова, как, например, `student`. В имя переменной желательно каким-то образом включить имя класса, как, например, `badStudent`, `goodStudent` и т.п. Имя массива (или другой коллекции) предпочтительнее делать простым и очевидным, таким как `students`, `phoneNumbers` или `phoneNumbersInMyPalmPilot`. Как обычно, этот совет всего лишь отражает личное мнение автора— C# совершенно безразлично, какие именно имена вы будете давать вашим переменным.

Конструкция *foreach*

Рассмотрим еще раз, как именно вычисляется средняя успеваемость студентов:

```
public class Student
{
    public string sName;
    public double dGPA; // Средний балл
}

public class Program
(
    public static void Main(string[] args)
    {
        // ... Создаем массив ...
        // Усредняем успеваемость
        double dSum = 0.0;
        for (int i = 0; i < students.Length; i++)
        {
            dSum += students[i].dGPA;
        }

        double dAvg = dSum / students.Length;
        // ... Прочие действия с массивом ...
    }
} Цикл for проходит по всем элементам массива.
```



Переменная `students.Length` содержит количество элементов в массиве.

C# предоставляет программистам особую конструкцию `foreach`, которая спроектирована специально для итеративного прохода по контейнерам, таким как массивы. Он работает следующим образом:

```
// Усредняем успеваемость
double dSum = 0.0;
foreach (Student stud in students)
{
    dSum += stud.dGPA;
}
double dAvg = dSum / students.Length;
```

При первом входе в цикл из массива выбирается первый объект типа `Student` и сохраняется в переменной `stud`. При каждой последующей итерации цикл `foreach` выбирает из цикла и присваивает переменной `stud` очередной элемент массива. Управление покидает цикл `foreach`, когда все элементы массива оказываются обработанными.

Обратите внимание, что в выражении `foreach` нет никаких индексов. Это позволяет существенно снизить вероятность появления ошибки в программе.



Программистам на C, C++ или Java цикл `foreach` покажется на первый взгляд неудобным, однако этот уникальный оператор C# (точнее, .NET) — простейший способ организации циклической обработки всех элементов массива.

На самом деле цикл `foreach` мощнее, чем можно представить из приведенного примера. Кроме массивов, он работает и с другими видами коллекций (о которых рассказывается, например, в главе 15, "Обобщенное программирование"). Кроме того, `foreach` в состоянии работать и с многомерными массивами (т.е. массивами массивов), но эта тема выходит за рамки настоящей книги.

Сортировка массива объектов

Сортировка элементов в массиве — весьма распространенная программистская задача. То, что массив не может расти или уменьшаться, еще не означает, что его элементы не могут перемещаться, удаляться или добавляться. Например, обмен местами двух элементов типа `Student` в массиве может быть выполнен так, как показано в следующем фрагменте исходного текста:

```
Student temp = students[i]; // Сохраняем i-го студента
students[i] = students[j];
students[j] = temp;
```

Здесь сначала во временной переменной сохраняется ссылка на объект в *i*-ой позиции массива `students`, чтобы она не была потеряна при обмене, затем ссылка в *i*-ой позиции заменяется ссылкой в *j*-ой позиции. После этого в *j*-ую позицию помещается ранее сохраненная во временной переменной ссылка, которая изначально находилась в *i*-ой позиции. Происходящее схематично показано на рис. 6.2.



Некоторые коллекции данных более гибки, чем массивы, и поддерживают добавление и удаление элементов. С такими коллекциями вы познакомитесь в главе 15, "Обобщенное программирование".

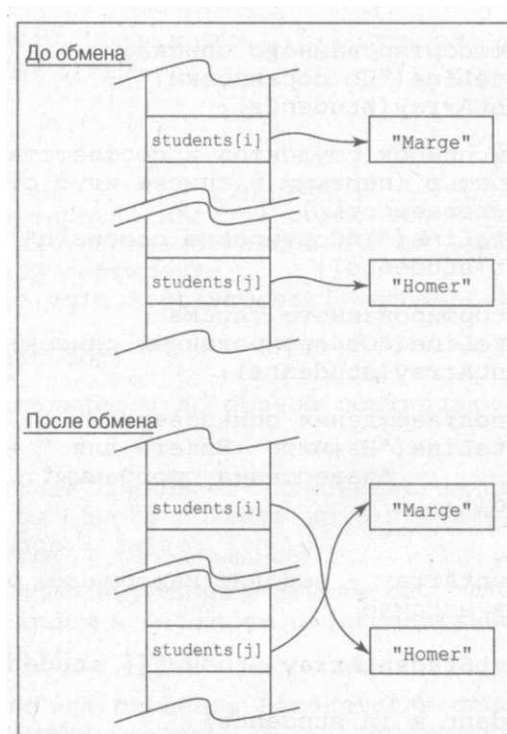


Рис. 6.2. "Обмен двух объектов" на самом деле означает "обмен ссылок на два объекта"



Приведенная ниже программа демонстрирует, как использовать возможность манипуляции элементами массива для их сортировки. В программе применен алгоритм *пузырьковой сортировки*. Он не слишком эффективен и плохо подходит для сортировки больших массивов с тысячами элементов, но зато очень прост и вполне применим для небольших массивов.

```
// SortStudents
// Демонстрационная программа для сортировки массива
// объектов
using System;

namespace SortStudents
{
    class Program
    {
        public static void Main(string[] args)
        {
            // Создание массива студентов
            Student[] students = new Student[5];
            students[0] = Student.NewStudent("Homer", 0);
            students[1] = Student.NewStudent("Lisa", 4.0);
            students[2] = Student.NewStudent("Bart", 2.0);
            students[3] = Student.NewStudent("Marge", 3.0);
            students[4] = Student.NewStudent("Maggie", 3.5);
        }
    }
}
```

```

// Вывод неотсортированного списка:
Console.WriteLine("До сортировки:");
OutputStudentArray(students);

// Сортируем список студентов в соответствии с их
// успеваемостью (первыми в списке идут студенты с
// лучшей успеваемостью)
Console.WriteLine("\nСортировка списка\n");
Student.Sort(students);

// Вывод отсортированного списка
Console.WriteLine("Отсортированный список:");
OutputStudentArray(students);

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}

// OutputStudentArray - выводит информацию о всех
// студентах в массиве
public static
    void OutputStudentArray(Student [] students)
{
    foreach(Student s in students)
    {
        Console.WriteLine(s.GetString());
    }
}

// Student - описание студента, включающее его имя и
// успеваемость
class Student
{
    public string sName;
    public double dGrade = 0.0;

    // NewStudent - возвращает новый инициализированный
    // объект
    public static Student NewStudent(string sName,
        double dGrade)
    {
        Student student = new Student();
        student.sName = sName;
        student.dGrade = dGrade;
        return student;
    }

    // GetString - преобразует текущий объект типа Student в
    // строку
    public string GetString()
    {

```

```

        string s = "";
        s += dGrade;
        s += " - ";
        s += sName;
        return s;
    }

    // Sort - сортировка массива студентов в порядке
    // убывания их успеваемости при помощи алгоритма
    // пузырьковой сортировки
    public static void Sort(Student[] students)
    {
        bool bRepeatLoop;
        // Цикл выполняется до полной сортировки списка
        do
        {
            // Этот флаг принимает значение true при наличии
            // хотя бы одного объекта не в порядке сортировки
            bRepeatLoop = false;

            // Цикл по всему списку студентов
            for(int index = 0; index < (students.Length - 1);
                index++)
            {
                // Если два студента находятся в списке в неверном
                // порядке...
                if (students[index].dGrade <
                    students[index + 1].dGrade)
                {
                    // ...меняем их местами...
                    Student to = students[index];
                    Student from = students[index + 1];
                    students[index] = from;
                    students[index + 1] = to;

                    // ...и присваиваем флагу значение true, чтобы
                    // программа выполнила очередной проход по
                    // списку (итерации продолжают, пока список не
                    // будет полностью отсортирован)
                    bRepeatLoop = true;
                }
            }
        } while (bRepeatLoop);
    }
}

```

Рассмотрим вывод данной программы, просто чтобы убедиться в ее работоспособности:

До сортировки:

0 - Homer
4 - Lisa
2 - Bart

```
3 - Marge
3.5 - Maggie
```

Сортировка списка

Отсортированный список:

```
4 - Lisa
3.5 - Maggie
3 - Marge
2 - Bart
0 - Homer
```

Нажмите <Enter> для завершения программы...

Чтобы сберечь ваше и мое время, создание списка из пяти студентов просто ~~закончи~~ровано непосредственно в программе. Метод `NewStudent()` выделяет память и ~~созда~~ет новый объект типа `Student`, инициализирует его и возвращает вызывающей ~~функ~~функции. Для вывода информации о всех студентах в списке используется функция `OutputStudentArray()`.

Затем программа вызывает функцию `Sort()`. После сортировки программа ~~повтор~~ет процесс вывода списка, чтобы вы могли убедиться, что теперь он упорядочен.

Само собой, ключевым моментом в программе `SortStudents` является ~~мето~~д `Sort()`. Этот алгоритм выполняет проходы по списку до тех пор, пока список не ~~буд~~полностью отсортирован, и при каждом таком проходе сравнивает два соседних ~~объ~~элемента массива. Если они находятся в неверном порядке, функция обменивает их местами и ~~от~~мечает этот факт в специальной переменной-флаге, которая затем используется в ~~уел~~условии цикла, указывая, полностью ли отсортирован массив. На рис. 6.3-6.6 показано, и ~~вы~~глядит список студентов после каждого прохода.

HOMER	0
LISA	4
BART	2
MARGE	3
MAGGIE	3,5

Рис. 6.3. Список студентов до сортировки

LISA	4
BART	2
MARGE	3
MAGGIE	3,5
HOMER	0

-*— Homer проделал свой путь в конец списка

Рис. 6.4. Список студентов после первого прохода

Lisa	4	← Lisa остается на верху списка
Marge	3	
Maggie	3,5	
Bart	2	← Bart опустился в конец списка, но не в последнюю позицию
Homer	0	

Рис. 6.5. Список студентов после второго прохода

В конечном итоге лучшие студенты, как пузырьки в воде, "всплывают" в верх списка в то время как наихудшие "тонут" и падают на дно. Потому такая сортировка и называется *пузырьковой*.

Lisa	4	
Maggie	3,5	↙
Marge	3	↘
Bart	2	
Homer	0	

Maggie и Marge меняются местами

Рис. 6.6. Список студентов после предпоследнего прохода оказывается полностью отсортирован. Последний проход завершает сортировку, так как никаких изменений при нем не вносится



Ключевым моментом любой функции сортировки является возможность перепорядочения элементов внутри массива, присваивая ссылку из одного элемента другому. Заметим, что такое присваивание не создает копии объекта, а следовательно, является очень быстрой операцией.

Глава 7

Функции функций

В этой главе...

- > Определение функции
- > Передача аргументов в функцию
- > Получение результата из функции
- > Передача аргументов программе

П

рограммисты нуждаются в возможности разбить большую программу на более мелкие части, с которыми легче совладать. Например, программы, содержащиеся в предыдущих главах, уже достигли своего предельного размера, с которым как с единым целым может справиться один человек.

C# позволяет разделить код на куски, известные как *функции*. Правильно спроектированные и реализованные функции существенно облегчают работу по написанию сложных программ.

Определение и использование функции

Рассмотрим следующий пример:

```
class Example

public int nInt;                // Не статический член
public static int nStaticInt    // Статический член
public void MemberFunction()    // Не статическая функция
{
    Console.WriteLine("Это функция-член" );
}
public static void ClassFunction() // Статическая функция
{
    Console.WriteLine("Это функция класса");
}
```

Элемент `nInt` является членом-данными, с которыми вы познакомились в главе 6, "Объединение данных — классы и массивы". Однако элемент `MemberFunction()` для вас нов. Он известен как *функция-член*, представляющая собой набор кода C#, который может быть выполнен с помощью ссылки на имя этой функции. Честно говоря, такое определение смущает даже меня самого, так что лучше рассмотреть, что такое функция, на примерах.

Примечание: отличие между статическими и нестатическими функциями крайне важно. Частично эта тема будет раскрыта в данной главе, но более подробно об этом

речь пойдет в главе 8, "Методы класса", где будет введен термин *метод*, очень распространенный в объектно-ориентированных языках программирования наподобие C# л обозначения нестатических функций классов.

В следующем фрагменте присваиваются значения члену объекта `nInt` и члену класса (статическому члену) `nStaticInt`:

```
Example example      new Example(); // Создание объекта
example.nInt          1; // Инициализация члена с
                        // использованием объекта
Example.nStaticInt    = 2; // Инициализация члена с
                        // использованием класса
```

Практически аналогично в приведенном далее фрагменте происходит обращение (путем вызова) к функциям `MemberFunction()` и `ClassFunction()`:

```
Example example = new Example(); // Создание объекта
example.MemberFunction(); // Вызов функции-члена
                        // с указанием объекта
Example.ClassFunction(); // Вызов функции класса с
                        // указанием класса

// Приведенные далее строки не компилируются
example.ClassFunction(); // Нельзя обращаться к функции
                        // класса с указанием объекта
Example.MemberFunction(); // Нельзя обращаться к функции-
                        // члену с указанием класса
```



Отличие между функциями класса (статическими) и функциями-членами (нестатическими), или методами, отражает различие между переменными класса (статическими) и переменными-членами (нестатическими), описанное ранее! в главе 6, "Объединение данных — классы и массивы".

Выражение `example.MemberFunction()` передает управление коду, содержащемуся внутри функции. Процесс вызова `Example.ClassFunction()` практически такой же. В результате выполнения приведенного выше фрагмента кода получается следующий вывод на экран:

```
Это функция-член
Это функция класса
```

После того как функция завершает свою работу, она передает управление в точку из которой она была вызвана.



Я включаю круглые скобки при указании функций в тексте — например `Main()` — чтобы функции было легче распознать. Без этого упоминания их в тексте можно легко спутать с переменными или классами.

В приведенном примере код функций не делает ничего особенного, кроме вывода на экран единственной строки, но в общем случае функции выполняют различные сложные операции, такие как вычисление математических функций, объединение строк, сортировка массивов или отправка электронных писем, словом, сложность решаемых функциями задач ничем не ограничена. Функции могут быть любого размера и любой степени сложности, но все же лучше, если они будут небольшими по размеру, так как с такими функциями гораздо легче работать.

Использование функций в ваших программах

В этом разделе в качестве демонстрации того, как разумное определение функций может помочь сделать программу проще для написания и понимания, будет взята монолитная программа `CalculateInterestTable` из главы 5, "Управление потоком выполнения", и разделена на несколько функций. Такой процесс переделки рабочего кода при сохранении его функциональности называется *рефакторингом*, и Visual Studio 2005 обеспечивает удобное меню `Refactor`, которое автоматизирует большинство распространенных задач рефакторинга.



Определение функций и их вызовы будут детально рассмотрены немного позже в этой главе, а пока считайте данный пример просто кратким обзором.

Чтение комментариев при опущенном программном коде должно способствовать пониманию намерений программиста. Если это не так — значит, вы плохо комментируете ваши программы. (И наоборот, если вы не можете, опустив большинство комментариев, понять, что делает программа на основании имен функций, значит, вы недостаточно ясно именуете функции и/или делаете их слишком большими).

"Скелет" программы `CalculateInterestTable` выглядит следующим образом:

```
public static void Main(string[] args)
{
    // Приглашение ввести начальный вклад.
    // Если вклад отрицателен, генерируется сообщение об
    // ошибке.
    // Приглашение для ввода процентной ставки.
    // Если процентная ставка отрицательна, генерируется
    // сообщение об ошибке.
    // Приглашение для ввода количества лет.
    // Вывод введенных данных.
    // Цикл по введенному количеству лет.
    while(nYear <= nDuration)
    {
        // Вычисление значения вклада с начисленными
        // процентами.
        // Вывод результата вычислений.
    }
}
```

Это пример хорошего метода проектирования функций. Если вы вернетесь и изучите программу, то увидите, что она состоит из следующих трех частей:

- I / часть начального ввода данных, в которой пользователи вводят вклад, процентную ставку и срок;
- | / раздел, выводящий введенную информацию на экран, чтобы пользователь мог убедиться в корректности ввода;
- I J последняя часть кода, создающая и выводящая таблицу на экран.

Это хорошее начало для выполнения рефакторинга. Кроме того, если вы внимательно рассмотрите часть ввода начальной информации, то увидите, что код для ввода



вклада,
процентной ставки и
срока

практически один и тот же. Это наблюдение дает еще одну точку для рефакторинга.



Все перечисленное позволяет выполнить рефакторинг программы `CalculateInterestTable` и создать программу `CalculateInterestTableWithFunctions`.

```
// CalculateInterestTableWithFunctions
// Генерация таблицы роста вклада по тому же алгоритму, что
// и ранее рассматривавшихся программах, однако в этой
// программе работа распределена между несколькими
// функциями.
using System;
namespace CalculateInterestTableWithFunctions
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Раздел 1 - ввод данных, необходимых для создания
            // таблицы
            decimal mPrincipal = 0;
            decimal mInterest = 0;
            decimal mDuration = 0;
            InputInterestData(ref mPrincipal,
                             ref mInterest,
                             ref mDuration);

            // Раздел 2 - проверка введенных данных путем вывода
            // их пользователю на экран
            Console.WriteLine(); // skip a line
            Console.WriteLine("Вклад          = " +
                              mPrincipal);
            Console.WriteLine("Процентная ставка = " +
                              mInterest + "%");
            Console.WriteLine("Срок          = " +
                              mDuration + " лет");
            Console.WriteLine();

            // Раздел 3 - и, наконец, вывод таблицы вкладов по
            // годам
            OutputInterestTable(mPrincipal, mInterest, mDuration);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для ");
        }
    }
}
```

```

        "завершения программы...");
    Console.Read();
}

// InputInterestData - клавиатурный ввод вклада,
// процентной ставки и срока для расчета таблицы
// Эта функция реализует раздел 1, разбивая его на три
// компонента
public static
    void InputInterestData(ref decimal mPrincipal,
                           ref decimal mInterest,
                           ref decimal mDuration)
{
    // 1а Получение вклада
    mPrincipal = InputPositiveDecimal("вклад");

    // 1б Получение процентной ставки

    mInterest = InputPositiveDecimal("процентная ставка");

    // 1в Получение срока
    mDuration = InputPositiveDecimal("срок");
}

// InputPositiveDecimal - возвращает положительное число
// типа decimal с клавиатуры.
// Выполняется только одна проверка - на
// неотрицательность введенного значения.
public static
    decimal InputPositiveDecimal(string sPrompt)
{
    // Цикл выполняется, пока не будет введено верное
    // значение
    while(true)
    {
        // Приглашение для ввода
        Console.Write("Введите " + sPrompt + " :");

        // Получение значения типа decimal с клавиатуры
        string sInput = Console.ReadLine();
        decimal mValue = Convert.ToDecimal(sInput);

        // Выход из цикла при вводе корректного значения
        if (mValue >= 0)
        {
            // Возврат введенного значения
            return mValue;
        }

        // В противном случае генерируется и выводится
        // сообщение об ошибке
        Console.WriteLine(sPrompt +

```

```

        " не может иметь отрицательное значение");
        Console.WriteLine("Попробуйте еще раз");
        Console.WriteLine();
    }
}

// OutputInterestTable - для заданных значений вклада,
// процентной ставки и срока генерирует и выводит на
// экран таблицу роста вклада
// Реализация раздела 3 основной программы
public static
    void OutputInterestTable(decimal mPrincipal,
                             decimal mInterest,
                             decimal mDuration)
{
    for (int nYear = 1; nYear <= mDuration; nYear++)
    {
        // Вычисление начисленных процентов
        decimal mInterestPaid;
        mInterestPaid = mPrincipal * (mInterest / 100);
        // Вычисление значения нового вклада путем
        // добавления начисленных процентов к основному
        // вкладу
        mPrincipal = mPrincipal + mInterestPaid;

        // Округление вклада до копеек
        mPrincipal = decimal.Round(mPrincipal, 2);

        // Вывод результата
        Console.WriteLine(nYear + "-" + mPrincipal);
    }
}
}
}

```

Раздел Main () разделен на три очевидные части, каждая из которых выделена и ментарием с полужирным шрифтом. Кроме того, первая часть, в свою очередь, подела на три подраздела — 1а, 1б и 1в.



Вам не следует пытаться выделять в своих исходных текстах комментарии лужирным шрифтом или номерами разделов. Исходный текст реальной программы и без того сложная и запутанная штука, чтобы вносить в него искусственные усложнения. На практике для понимания достаточно ясных и информативных имен функций, указывающих их предназначение.

В первой части для ввода значений трех переменных, необходимых для работы программы (mPrincipal, mInterest и mDuration), вызывается функция InputInterestData(). Во второй части полученные значения выводятся на экран так же, и в предыдущих версиях программы. Последняя часть строит и выводит на экран таблицу цувкладовспомощьюфункцииOutputInterestTable().

Начнем с конца, с функции OutputInterestTable(). В ней содержится цш в котором выполняется вычисление начисленных процентов — в точности так же, ю

это делалось в программе `CalculateInterestTable` в главе 5, "Управление потоком выполнения". Преимущество данной версии заключается в том, что при разработке этой части кода не нужно сосредотачиваться на деталях ввода и верификации данных. При написании этой функции следует просто думать о том, как вычислить и вывести таблицу для уже полученных значений. После выполнения функции управление вернется в строку, следующую за вызовом функции `OutputInterestTable()`.



`OutputInterestTable()` — хороший повод для того, чтобы воспользоваться новым меню **Refactoring** в Visual Studio 2005. Для этого надо выполнить следующие действия.

1. **Воспользоваться в качестве стартовой точки примером `CalculateInterestTableMoreForgiving` из главы 5, "Управление потоком выполнения", выбрав исходный текст от объявления переменной `nYear` до конца цикла `while`:**

```
int nYear = 0;           // Переменная цикла
while(nYear <= nDuration) // и весь цикл while
{
    //...
}
```

2. **Выбрать команду меню `Refactor^Extract Method`.**

3. **В диалоговом окне `Extract Method` ввести `OutputInterestTable`, затем просмотреть поле `Preview Method Signature` и щелкнуть на кнопке `OK`.**

Обратите внимание, что предложенная "сигнатура" нового "метода" начинается с ключевых слов `private static` и включает `mPrincipal`, `mInterest` и `nDuration` в круглых скобках. О ключевом слове `private`, как альтернативе `public`, будет рассказано в главе 11, "Классы". Пока же вы можете при желании сделать эту функцию `public`.

Результат такого рефакторинга заключается в следующем.

/ Ниже функции `Main()` добавляется новая `private static` функция `OutputInterestTable()`.

/ Там, где в функции `Main()` находился выбранный код, появляется следующая строка:

```
mPrincipal = OutputInterestTable(mPrincipal,
                                mInterest, nDuration);
```

Точно такой же подход "разделяй и властвуй" применим и для функции `InputInterestData()`. Однако в этом случае требуется более сложный рефакторинг, так что я выполнил его вручную и все его этапы здесь не показаны. Все же искусство рефакторинга выходит за рамки настоящей книги.

В функции `InputInterestData()` вы сосредотачиваетесь только на вводе трех значений типа `decimal`. В данном случае, несмотря на три различные переменные, действия по их вводу оказываются идентичны и могут быть размещены в функции `InputPositiveDecimal()`, которая одинаково применима как для ввода вклада, так и для ввода процентной ставки и срока, для которого выполняется расчет. Заметьте, как три цикла `while` в исходной программе превратились в один в теле функции `InputPositiveDecimal()`. Тем самым устранено дублирование кода, которое всегда нежелательно.



Внесение в программу модификаций, делающих ее понятнее и проще, не гаяняя при этом ее функциональность и внешнее поведение, называется *рефакрингом*. Большое количество информации о нем можно найти на Web-сайте www.refactoring.com.

Функция `InputPositiveDecimal()` выводит приглашение и ожидает ввода пользователя. Если введенное пользователем значение неотрицательно, она возвращает вызвавшей ее функции. Если же введенное значение отрицательно, функция выводит! сообщение об ошибке и повторяет цикл ввода.

С точки зрения пользователя получается та же программа, что и раньше (в главе "Управление потоком выполнения"), поскольку работает она в точности так же:

```
Введите вклад:100
Введите процентную ставку:-10
Процентная ставка не может быть отрицательна
Попробуйте еще раз
```

```
Введите процентную ставку:10
Введите срок:10
```

```
Вклад           = 100
Процентная ставка = 10%
Срок             = 10 лет
```

```
1-110.0
2-121.00
3-133.10
4-146.41
5-161.05
6-177.16
7-194.88
8-214.37
9-235.81
10-259.39
```

Нажмите <Enter> для завершения программы...

Итак, взяв длинную, запутанную программу и применив рефакторинг, можно почитать программу меньшего размера, со сниженным дублированием кода, а главное - более понятную.

Зачем беспокоиться о функциях?

Когда концепция функции появилась в 1950-е годы в Фортране, ее единственной целью было избежать дублирования кода. Предположим, вы пишете программу, которая должна вычислять и выводить на экран некоторое отношение во многих местах. В этом случае программа может просто вызывать в этих местах функцию `DisplayRatio()`, позволяющую избежать дублирования кода. Такая экономия может показаться не слишком большой, если функция состоит всего из пары строк, но функции бывают разные; они могут быть очень сложными и большими. Кроме того, распространенные функции, такие как `WriteLine()`, `MessageBox()` используются в сотнях различных мест.

Второе преимущество применения функций также очевидно: проще корректно написать и отладить одну функцию, чем десяток фрагментов кода, и вдвойне проще сделать это, если функция невелика. Функция `DisplayRatio()` включает проверку то-

го, что знаменатель в отношении не равен нулю. Если у вас имеется множество фрагментов кода, а не одна функция, то скорее всего в некоторых местах программы вы просто забудете вставить эту проверку.

Менее очевидно третье преимущество: хорошо спроектированные функции снижают сложность программы. Каждая функция должна соответствовать некоторой концепции. Вы должны быть способны указать назначение каждой функции без использования слов *и* и *или*. Вы должны следовать принципу: одна функция — одна задача.

Функция `calculateSin()` служит идеальным примером. Программист, реализующий сложные вычисления, совершенно не должен беспокоиться, как именно будут применены их результаты. Прикладной программист может использовать функцию `calculateSin()`, не интересуясь, как именно она устроена и работает. Этот подход существенно снижает количество вещей, о которых должен думать и переживать прикладной программист. Большую работу гораздо проще сделать, если разделить ее на несколько частей.

Большие программы, как, например, текстовый редактор, строятся из множества функций разного уровня абстракции. Например, функция `DisplayDocument()` должна вызывать функцию `Paragraph()` для вывода абзацев документа. Эта функция, в свою очередь, должна вызывать функцию `CalculateWordWrap()` для вычисления длин отдельных строк абзаца. Функция `CalculateWordWrap()` может вызывать функцию `LookUpWordBreak()`, определяющую, как должно быть разбито для переноса слово, стоящее в конце строки. Каждая из перечисленных функций решает одну задачу, которую можно сформулировать простым предложением (кстати, обратите внимание и на информативность названий функций).

Без возможности **абстрагирования** сложных концепций написание программы даже средней сложности становится практически нереализуемым, не говоря уж о создании операционных систем, игр, офисного программного обеспечения и тому подобных больших и сложных программ.

Аргументы функции

Метод, подобный приведенному ниже, полезен примерно так же, как и зубная щетка, которой может пользоваться только один человек. Это связано с тем, что никакие данные приведенной функции не передаются и ею не возвращаются:

```
public static void Output()
{
    Console.WriteLine("Это функция");
}
```

Сравним этот пример с реальными функциями. Например, функция вычисления синуса требует определенных входных данных — в конце концов, вы ведь вычисляете синус чего-то? Аналогично, при конкатенации двух строк нужно передать функции две строки — не так ли? И получить от функции результаты ее работы? Следовательно, возникает крайняя необходимость в механизме обмена информацией с функцией.

Передача аргументов функции

Значения, передаваемые функции, называются *аргументами функции* (другое часто используемое название — *параметры*). Большинство функций требуют для работы аргументы определенного типа. Вы передаете аргументы функции, перечисляя их в *скобках* после ее имени. Проанализируем следующее небольшое добавление к рассматривавшемуся ранее классу `Example`:

```
public class Example
{
    public static void Output(string funcString)
    {
        Console.WriteLine("Функция Output() получила аргумент: 11
                           + funcString);
    }
}
```

П

Эту функцию можно вызвать в самом классе следующим образом:

```
Output("Hello");
```

и получить в результате вывод на экран

```
Функция Output() получила аргумент: Hello
```

Программа передает функции `Output()` ссылку на строку `"Hello"`. Функция получает эту строку и присваивает ей имя `funcString`. В теле функции `Output()` переменная `funcString` может использоваться точно так же, как и любая другая переменная типа `string`.

Можно немного изменить пример:

```
string myString = "Hello";
Output(myString);
```

В этом фрагменте переменной `myString` присваивается ссылка на строку `"Hello"`. Вызов `Output(myString)` передает функции объект, на который ссылается переменная `myString`, т.е. ту же строку `"Hello"`, что и ранее. Этот процесс изображен на рис. 7.1. Результат работы фрагмента исходного текста тот же, что и до внесения в него изменений.

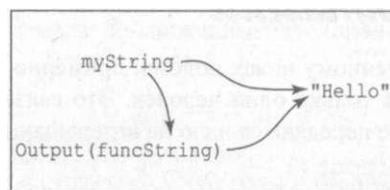


Рис. 7.1. Вызов **Output** (*myString*) копирует значение *myString* в переменную *funcString*

Передача функции нескольких аргументов

Когда я прошу сына помыть машину, он приводит сразу несколько аргументов, почему он не может это сделать. Их у него такое множество, что несколько штук всегда наготове. Но речь сейчас пойдет не о детях, а о нескольких аргументах, которые могут использоваться при вызове функции.



Вы можете определить функцию с несколькими аргументами различных типов. Рассмотрим в качестве примера функцию `AverageAndDisplay()`:

```
// AverageAndDisplay
using System;

namespace Example
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Обращение к функции-члену
            AverageAndDisplay("оценки 1", 3.5, "оценки 2", 4.0);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы. . . ");

            Console.Read();
        }
        // AverageAndDisplay - усредняет два числа и выводит
        // результат с использованием переданных меток
        public static
            void AverageAndDisplay(string si, double d1,
                                   string s2, double d2)
        {
            double dAverage = (d1 + d2) / 2;
            Console.WriteLine("Среднее " + si
                              + ", равной " + d1
                              + ", и " + s2
                              + ", равной " + d2
                              + ", равно " + dAverage);
        }
    }
}
```



Вот как выглядит вывод этой программы на экран:

```
Среднее оценки 1, равной 3.5, и оценки 2, равной 4, равно 3.75
Нажмите <Enter> для завершения программы. . .
```

Функция `AverageAndDisplay()` объявлена с несколькими аргументами в том порядке, в котором они в нее передаются.

Как обычно, выполнение программы начинается с первой команды после `Main()`. Первая строка после `Main()`, не являющаяся комментарием, вызывает функцию `AverageAndDisplay()`, передавая ей две строки и два значения типа `double`.

Функция `AverageAndDisplay()` вычисляет среднее переданных значений типа `double`, `d1` и `d2`, переданных в функцию вместе с их именами (содержащимися в переменных `si` и `s2`), и сохраняет полученное значение в переменной `dAverage`.



Изменение значений аргументов внутри функции может привести к ошибкам. Разумнее присвоить эти значения временным переменным и модифицировать уже их.

Соответствие определений аргументов их использованию



Каждый аргумент в вызове функции должен соответствовать определена функции как в смысле типа, так и в смысле порядка. Приведенный далее *ш* ходный текст некорректен и вызывает ошибку в процессе компиляции.

```
// AverageWithCompilerError - эта версия не компилируется!
using System;

namespace Example
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Обращение к функции-члену
            AverageAndDisplay("оценки 1", "оценки 2", 3.5, 4.0);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }

        // AverageAndDisplay - усредняет два числа и выводит
        // результат с использованием переданных меток
        public static
            void AverageAndDisplay(string s1, double d1,
                                   string s2, double d2)
        {
            double dAverage = (d1 + d2) / 2;
            Console.WriteLine("Среднее " + s1
                              + ", равной " + d1
                              + ", и " + s2
                              + ", равной " + d2
                              + ", равно " + dAverage);
        }
    }
}
```

C# обнаруживает несоответствие типов передаваемых функции аргументов с аргументами в определении функции. Строка "оценки 1" соответствует типу `string` определению функции; однако согласно определению функции вторым аргумента должно быть число типа `double`, в то время как при вызове вторым аргументом функции оказывается строка `string`.

Это случилось потому, что я просто обменял местами второй и третий аргумент! функции. И это как раз то, за что я не люблю компьютеры — именно за то, что они понимают все совершенно буквально.

Чтобы исправить ошибку, достаточно поменять местами второй и третий аргументы вызове функции `AverageAndDisplay()`.

Перегрузка функции



В одном классе может быть две функции с одним и тем же именем — при условии различия их аргументов. Это явление называется *перегрузкой* (overloading) имени функции.

Вот пример программы, демонстрирующей перегрузку.

```
// AverageAndDisplayOverloaded - эта версия демонстрирует
// возможность перегрузки функции вычисления и вывода
// среднего значения
using System;

namespace AverageAndDisplayOverloaded
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Вызов первой функции-члена
            AverageAndDisplay("моей оценки", 3.5,
                              "твоей оценки", - 4.0);

            Console.WriteLine();
            // Вызов второй функции-члена
            AverageAndDisplay(3.5, 4.0);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");

            Console.Read();
        }

        // AverageAndDisplay - вычисление среднего значения двух
        // чисел и вывод его на экран с переданными функции
        // метками этих чисел
        public static
            void AverageAndDisplay(string s1, double d1,
                                   string s2, double d2)

        {
            double dAverage = (d1 + d2) / 2;
            Console.WriteLine("Среднее " + s1
                              + ", равной " + d1);
            Console.WriteLine("и " + s2
                              + ", равной " + d2
                              + ", равно " + dAverage);
        }

        public static void AverageAndDisplay(double d1,
                                              double d2)
        {
            double dAverage = (d1 + d2) / 2;
```

```

        Console.WriteLine("среднее " + d1
                           + " и "      + d2
                           + " равно "  + dAverage);
    }
}

```

В программе определены две версии функции `AverageAndDisplay()`. Программист вызывает одну из них после другой, передавая им соответствующие аргументы. C# в состоянии определить по переданным функции аргументам, какую из версий следует вызвать, сравнивая типы передаваемых значений с определениями функций. Программа корректно компилируется и выполняется, выводя на экран следующие строки:

```

Среднее моей оценки, равной 3.5
и твоей оценки, равной 4, равно 3.75

```

```

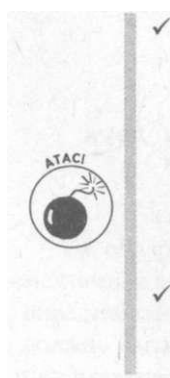
Среднее 3.5 и 4 равно 3.75
Нажмите <Enter> для завершения программы...

```

Вообще говоря, C# не позволяет иметь в одной программе две функции с одинаковыми именами. В конце концов, как тогда он сможет разобраться, какую из функций следует вызвать? Но дело в том, что C# в имя функции во внутреннем представлении компилятора включает не только имя функции, но и количество и типы ее аргументов. Поэтому C# в состоянии отличить функцию `AverageAndDisplay(string, double, string, double)` от функции `AverageAndDisplay(double, double)`. Если рассматривать эти функции с их аргументами, становится очевидным, что они разные.

Реализация аргументов по умолчанию

Зачастую оказывается желательно иметь две (или более) версии функции, имеющие следующие отличия.



- ✓ Одна из функций представляет собой более сложную версию, обеспечивающую большую гибкость, но требующую большого количества аргументов от вызывающей программы, причем некоторые из них для пользователя могут быть просто непонятны.

Под пользователем функции зачастую подразумевается программист, применяющий ее в своих программах, так что пользователь функции и пользователь готовой программы — это разные люди. Еще один термин, применяемый для обозначения такого рода пользователя — клиент.

- ✓ Вторая версия функции гораздо проще для применения, она работает же, как и первая, в которой часть аргументов принимает некоторые предопределенные значения по умолчанию.

Такое поведение легко реализуется с использованием перегрузки функций.



Рассмотрим следующую пару функций `DisplayRoundedDecimal()`:

```

// FunctionsWithDefaultArguments - две версии одной и той же
// функции, причем одна из них представляет версию второй с
// использованием значений аргументов по умолчанию

```

```

using System;

namespace FunctionsWithDefaultArguments
(
    public class Program
    {
        public static void Main(string[] args)
        {
            // Вызов функции-члена
            Console.WriteLine("{0}"11,
                DisplayRoundedDecimal(12.345678M, 3));

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                "завершения программы...");

            Console.Read();
        }

        // DisplayRoundedDecimal - преобразует значение типа
        // decimal в строку с определенным количеством значащих
        // цифр
        public static
            string DisplayRoundedDecimal(decimal mValue,
                int nNumberOfSignificantDigits)
        {
            // Сначала округляем число до указанного количества
            // значащих цифр
            decimal mRoundedValue =
                decimal.Round(mValue,
                    nNumberOfSignificantDigits);

            // и преобразуем его в строку
            string s = Convert.ToString(mRoundedValue);

            return s;
        }

        public static
            string DisplayRoundedDecimal(decimal mValue)
        {
            // Вызываем DisplayRoundedDecimal(decimal, int) с
            // указанием количества значащих цифр по умолчанию
            string s = DisplayRoundedDecimal(mValue, 2);
            return s;
        }
    }
}

```



Функция `DisplayRoundedDecimal()` преобразует значение типа `decimal` в значение типа `string` с определенным количеством значащих цифр после десятичной точки. Поскольку числа типа `decimal` часто применяются в финансовых расчетах, наиболее распространенными будут вызовы этой функции со вторым аргументом, равным 2. В анализируемой программе это предусмотрено, и вызов `DisplayRoundedDecimal()` с одним аргументом округляет значение этого аргумента до двух цифр после десятичной

точки, позволяя пользователю не беспокоиться о смысле и числовом значении **второй** аргумента функции.



Обратите внимание, что версия функции `DisplayRoundedDecimal(decimal)` в действительности осуществляет вызов функции `DisplayRoundedDecimal(decimal, int)`. Такая практика позволяет избежать ненужного дублирования кода. Обобщенная версия функции может использовать существенно большее количество аргументов, которые ее разработчик может даже не включить документацию.



Аргументы по умолчанию не просто сберегают силы ленивого программиста. Программирование — работа, требующая высочайшей степени концентрации и излишние аргументы функции, для выяснения назначения и рекомендуем значений которых необходимо обращаться к документации, затрудняют программирование, приводят к перерасходу времени и повышают вероятность внесения ошибок в код. Автор функции хорошо понимает взаимосвязи между аргументами функции и способен обеспечить несколько корректных перегруженных версий функции, более дружественных к клиенту.

Примечание для программистов на Visual Basic и C/C++: в C# единственный способ реализации аргументов по умолчанию — это перегрузка функций. C# не позволяет иметь необязательные аргументы.

Передача в функцию типов-значений

Базовые типы переменных, такие как `int`, `double`, `decimal`, известны как тип **значения**. Переменные таких типов могут быть переданы в функцию одним из двух способов. По умолчанию эти переменные передаются в функцию **по значению** (by value) альтернативная форма — передача по ссылке.

Программисты часто не совсем точны в употреблении терминов. Если речь идет о типах-значениях, то когда программист говорит о "передаче переменной в функцию", это обычно означает "передача значения переменной в функцию".

Передача по значению

В отличие от ссылок на объекты, переменные с типами-значениями (например `id` или `double`) обычно передаются в функцию **по значению**, т.е. функции передается значение, содержащееся в этой переменной, но не сама переменная.



При такой передаче изменение значения соответствующей переменной **внутри** функции не вызовет изменения значения переданной переменной в вызывающей программе, что и демонстрируется следующей программой:

```
// PassByValue - программа для демонстрации семантики
// передачи аргумента по значению
using System;

namespace PassByValue
{
    public class Program
    {
```



```
// Update - функция пытается модифицировать значения
// аргументов, переданные ей; обратите внимание, что
// функции в классе могут быть объявлены в любом порядке
public static void Update(int i, double d)
{
    i = 10;
    d = 20.0;
}

public static void Main(string[] args)
{
    // Объявляем и инициализируем, две переменные
    int i = 1;
    double d = 2.0;
    Console.WriteLine("Перед вызовом Update(int, double):");
    Console.WriteLine("i = " + i + ", d = " + d);

    // Вызываем функцию
    Update(i, d);

    // Обратите внимание — значения 1 и 2.0 не изменились
    Console.WriteLine("После вызова Update(int, double) :");
    Console.WriteLine("i = " + i + ", d = " + d);

    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
        "завершения программы...");
    Console.Read();
}
```



Выполнение программы дает такой вывод на экран:

```
Перед вызовом Update (int, double) :
i=1, d = 2
После вызова Update (int, double) :
i = 1, d = 2
Нажмите <Enter> для завершения программы...
```

Вызов `Update ()` передает функции значения 1 и 2.0, а не ссылки на переменные `i` и `d`. Таким образом, изменение их значений в функции никак не влияет на значения исходных переменных в вызывающей программе.

Передача по ссылке

Передача функции переменных с типами-значениями по ссылке имеет ряд преимуществ — этот метод передачи используется, в частности, когда вызывающая программа хочет предоставить функции возможность изменять значение передаваемой в качестве аргумента переменной. Приведенная далее программа `PassByReference` демонстрирует эту возможность.



C# дает программисту возможность передачи аргументов по ссылке с использованием ключевых слов `ref` и `out`. Слегка измененная программа `PassByValue` демонстрирует, как это делается.

```

// PassByReference - программа для демонстрации семантики
// передачи аргумента по ссылке
using System;

namespace PassByValue
{
    public class Program
    {
        // Update - функция пытается модифицировать значения
        // аргументов, переданные ей; обратите внимание, на
        // передачу аргументов как ref и out
        public static void Update(ref int i, out double d)
        {
            i = 10;
            d = 20.0;
        }

        public static void Main(string[] args)
        {
            // Объявляем две переменные и одну инициализируем
            int i = 1;
            double d;
            Console.WriteLine("Перед вызовом " +
                             "Update(ref int, out double):");
            Console.WriteLine("i = " + i +
                              ", d неинициализирована");

            // Вызываем функцию
            Update(ref i, out d);

            // Обратите внимание — значение i равно 10, значение d
            // равно 20.0
            Console.WriteLine("После вызова "
                              "Update(ref int, out double):",-
                              "i = " + i + ", d = " + d);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}

```

Ключевое слово `ref` указывает C#, что в функцию следует передать ссылку на `i`, а не просто значение этой переменной. Следовательно, изменения, выполненные с этой переменной, оказываются экспортированы обратно вызывающей программе.

Подобно этому, ключевое слово `out` говорит: "Передай ссылку на эту переменную, но можешь никак не заботиться о ее значении, поскольку оно все равно не будет использоваться и будет перезаписано в процессе работы функции". Это ключевое слово годится тогда, когда переменная применяется исключительно для того, чтобы вернуть значение вызывающей программе.

Выполнение рассмотренной программы `PassByReference` приводит к следующему выводу на экран:

Перед вызовом `Update(ref int, out double)` :

`i = 1`, `d` неинициализирована

После вызова `Update(ref int, out double)` :

`i = 10`, `d = 20`

Нажмите <Enter> для завершения программы. . .



Аргумент, передаваемый как `out`, всегда считается передаваемым так же как `ref`, т.е. писать `ref out` — это тавтология. Кроме того, при передаче аргументов по ссылке вы должны всегда передавать только *переменные* — передача литеральных значений, например просто числа 2 вместо переменной типа `int`, в этом случае приводит к ошибке.

Обратите внимание, что начальные значения `i` и `d` переписываются в функции `Updated`. После возврата в функцию `Main()` эти переменные остаются с измененными в функции `Update()` значениями. Сравните это поведение с программой `PassByValue`, где внесенные изменения не сохраняются при выходе из функции.

Не передавайте переменную по ссылке дважды



Никогда не передавайте по ссылке одну и ту же переменную дважды в одну функцию, поскольку это может привести к неожиданным побочным эффектам. Описать эту ситуацию труднее, чем просто продемонстрировать пример программы. Внимательно взгляните на функцию `Update()` в приведенном листинге.

```
// PassByReferenceError - демонстрация потенциально
// ошибочной ситуации при вызове функции с передачей
// аргументов по ссылке
using System;

namespace PassByReferenceError
{
    public class Program
    {
        // Update - эта функция пытается изменить значения
        // переданных ей аргументов
        public static void DisplayAndUpdate(ref int nVar1,
                                           ref int nVar2)
        {
            Console.WriteLine("Начальное значение nVar1 - " +
                              nVar1);
            nVar1 = 10;
            Console.WriteLine("Начальное значение nVar2 - " +
                              nVar2);
            nVar2 = 20;
        }

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

```

// Объявляем и инициализируем переменную
int n = 1;
Console.WriteLine("Перед вызовом " +
    "Update(ref n, ref n):");
Console.WriteLine("n = " + n);
Console.WriteLine();

// Вызываем функцию
DisplayAndUpdate(ref n, ref n);

// Обратите внимание на то, как изменяется значение n
// - не так, как ожидалось от этой переменной и
// функции, в которую она была передана
Console.WriteLine();
Console.WriteLine("После вызова " +
    "Update(ref n, ref n):");
Console.WriteLine("n = " + n);

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}
}

```

В этом примере функция `Update(ref int, ref int)` объявлена как функция с двумя целыми аргументами, передаваемыми по ссылке — и в этом нет ничего *некорректного* или необычного. Проблема возникает тогда, когда в функцию передается одна *и та же переменная* как в качестве первого, так и второго аргумента. Внутри функции происходит изменение переменной `nVar1`, которая ссылается на переменную `n`, от ее начального значения 1 до значения 10. Затем происходит изменение переменной `nVar2`, которая тоже ссылается на переменную `n`, от ее начального значения 1 до значения 20; и побочное действие заключается в том, что переменная `n` теряет ранее присвоенное значение 10 и получает новое значение 20.

Это видно из приведенного ниже вывода программы на экран:

```

Перед вызовом Update(ref n, ref n):
n = 1

Начальное значение nVar1 - 1
Начальное значение nVar2 - 10

После вызова Update(ref n, ref n):
n = 20
Нажмите <Enter> для завершения программы...

```



Понятно, что ни программист, который писал функцию `Update()`, ни программист, ее использовавший, не рассчитывали на такой экзотический результат. Вся проблема оказалась в том, что одна и та же переменная была передана в одну и ту же функцию по ссылке больше одного раза. Никогда так не поступайте, если только вы не абсолютно уверены в том, чего именно вы хотите добиться таким действием.



Вы можете передавать одно и то же значение в функцию сколько угодно раз, если передаете его по значению.

Почему некоторые аргументы используются только для возврата значений?

C# по мере возможности старается защитить программиста от всех глупостей, которые он может вольно или невольно сделать. Одна из этих глупостей — программист может забыть проинициализировать переменную перед ее первым применением (особенно часто это случается с переменными-счетчиками). C# генерирует ошибку, когда вы пытаетесь использовать объявленную, но не инициализированную переменную:

```
int nVariable;  
Console.WriteLine("Это ошибка " + nVariable);  
nVariable = 1;  
Console.WriteLine("А это - нет " + nVariable);
```

Однако C# не в состоянии отслеживать переменные в функции:

```
void SomeFunction(ref int nVariable)  
{  
    Console.WriteLine("Ошибка или нет? " + nVariable);  
},
```

Откуда функция `SomeFunction()` может знать, была ли переменная `nVariable` инициализирована перед вызовом функции? Это невозможно. Вместо этого C# отслеживает переменные при вызове функции — например, вот такой вызов функции приведет к сообщению об ошибке:

```
int nUninitializedVariable;  
SomeFunction(ref nUninitializedVariable);
```

Если бы C# позволил осуществить такой вызов, то функция `SomeFunction()` получила бы ссылку на неинициализированную переменную (т.е. на *мусор* (garbage)) в памяти. Ключевое слово `out` позволяет функции и вызывающему ее коду договориться о том, что передаваемая по ссылке переменная может быть не инициализирована — функция обещает не использовать ее значение до тех пор, пока оно не будет каким-либо образом присвоено самой функцией. Следующий фрагмент исходного текста компилируется без каких-либо замечаний:

```
int nUninitializedVariable;  
SomeFunction(out nUninitializedVariable);
```

Передача инициализированной переменной как `out`-аргумента также является корректным действием:

```
int nInitializedVariable = 1;  
SomeFunction(out nInitializedVariable);
```

В этом случае значение переменной `nInitializedVariable` будет просто проигнорировано функцией `SomeFunction()`, но никакой опасности для программы это не представляет.

Возврат значений из функции

Многие реальные операции создают некоторые значения, которые должны быть *возвращены* тому, кто вызвал эту операцию. Например, функция `sin()` получает аргумент и возвращает значение тригонометрической функции "синус" для данного аргумента. Функция может вернуть значение вызывающей функции двумя способами. Наиболее распространенный — с помощью команды `return`; второй способ использует возможности передачи аргументов по ссылке.

Возврат значения оператором `return`

Приведенный далее фрагмент исходного текста демонстрирует небольшую функцию возвращающую среднее значение переданных ей аргументов,

```
public class Example
{
    public static double Average(double d1, double d2)
    {
        double dAverage = (d1 + d2) / 2;
        return dAverage;
    }
    public static void TestO
    {
        double v1 = 1.0;
        double v2 = 3.0;
        double dAverageValue = Average(v1, v2);
        Console.WriteLine("Среднее для " + v1
                           + " и " + v2 + " равно "
                           + dAverageValue);
        // Такой метод также вполне работоспособен
        Console.WriteLine("Среднее для " + v1
                           + " и " + v2 + " равно "
                           + Average(v1, v2));
    }
}
```

Прежде всего обратите внимание, что функция объявлена как `public static double Average()` — тип `double` перед именем функции указывает на тот факт, что функция `Average()` возвращает вызывающей функции значение типа `double`.

Функция `Average()` использует имена `d1` и `d2` для значений, переданных ей в качестве аргументов. Она создает переменную `dAverage`, которой присваивает среднее значение этих переменных. Затем значение, содержащееся в переменной `dAverage`, возвращается вызывающей функции.



Программисты иногда говорят, что "функция возвращает `dAverage`". Это некорректное сокращение. Говорить, что передается или возвращается `dAverage` или иная переменная — неточно. В данном случае вызывающей функции возвращается значение, содержащееся в переменной `dAverage`.

Вызов `Average()` из функции `Test()` выглядит точно так же, как и вызов любой другой функции; однако значение типа `double`, возвращаемое функцией `Average`, сохраняется в переменной `dAverageValue`.



Функция, которая возвращает значение (как, например, `Average()`), не может завершиться просто по достижении закрывающей фигурной скобки, поскольку C# совершенно непонятно, какое же именно значение должна будет вернуть эта функция? Для этого обязательно наличие оператора `return`.

Возврат значения посредством передачи по ссылке

Функция может также вернуть одно или несколько значений вызывающей программе с помощью ключевых слов `ref` или `out`. Рассмотрим пример `Update()` из раздела "Передача по ссылке" данной главы:

```
// Update - функция пытается модифицировать значения
// аргументов, переданные ей; обратите внимание, на
// передачу аргументов как ref и out
public static void Update(ref int i, out double d)
{
    i = 10;
    d = 20.0;
}
```

Эта функция объявлена как `void`, так как она не возвращает никакого значения вызывающей функции. Однако поскольку переменная `i` объявлена как `ref`, а переменная `d` — как `out`, любые изменения значений этих переменных, выполненные в функции `Update()`, сохранятся при возврате в вызывающую функцию. Другими словами, значения этих переменных вернутся вызывающей функции.

Когда какой метод использовать

Вы можете задуматься: "Функция может возвращать значение как с использованием оператора `return`, так и посредством переменных, переданных по ссылке. Так какой же метод мне лучше применять в своих программах?" В конце концов, ту же функцию `Average()` вы могли написать и так:

```
public class Example
{
    // Примечание: параметр, передаваемый как 'out', лучше
    // сделать последним в списке
    public static void Average(double l, double d2,
                              out double dResults)
    {
        dResults = (d1 + d2) / 2;
    }

    public static void Test()
    {
        double v1 = 1.0;
        double v2 = 3.0;
        double dAverageValue;
        Average(dAverageValue, v1, v2);
        Console.WriteLine("Среднее " + v1
                          + " и " + v2 + " равно "
                          + dAverageValue);
    }
}
```

Обычно значение вызывающей функции возвращается с помощью оператора `return`, а не посредством `out`-аргумента, хотя обосновать преимущество такого подхода очень трудно.



• **^S^** Возврат значения посредством передачи аргумента по ссылке иногда требует дополнительных действий, которые будут описаны в главе 14, "Интерфейсы! структуры". Однако обычно эффективность — не главный фактор при принятии решения о способе возврата значения из функции.

Как правило, "метод `out`" используется, если требуется вернуть из функции несколько значений — например, как в следующей функции:

```
public class Example
{
    public static
        void AverageAndProduct(double d1, double d2,
                                out double dAverage,
                                out double dProduct)
    {
        dAverage = (d1 + d2) / 2;
        dProduct = d1 * d2;
    }
}
```



Возврат из функции нескольких значений встречается не так часто, как может показаться. Функция, которая возвращает несколько значений, обычно делается это путем возврата одного объекта класса, который инкапсулирует несколько значений, или путем возврата массива значений. Оба способа приводят к более ясному и понятному исходному тексту.

Нулевая ссылка и ссылка на ноль

Ссылочные переменные, в отличие от переменных типов-значений, при создании инициализируются значением по умолчанию `null`. Однако нулевая ссылка (т.е. ссылка, инициализированная значением `null`) — это не то же, что ссылка на ноль. Например, две следующие ссылки совершенно различны:

```
class Example
{
    int nValue, -
}
// Создание нулевой ссылки ref1
Example ref1;
// Создание ссылки на нулевой объект
Example ref2 = new Example();
ref2.nValue = 0;
```

Переменная `ref1` пуста, как мой бумажник. Она указывает в "никуда", т.е. не указывает ни на какой реальный объект. Ссылка же `ref2` указывает на вполне конкретный объект, значение которого равно нулю.

Возможно, эта разница станет понятнее после следующего примера:

```
string s1;
string s2 = "";
```


По сути, возникает аналогичная ситуация— `si` указывает на *нулевой объект*, в то время как `s2` — на *пустую строку* (на сленге программистов пустая строка иногда называется нулевой строкой). Это очень существенное отличие, как становится ясно из следующего исходного текста:

```
// Test - тестовая программа
namespace Test
(
    using System;
    public class Program
    {
        public static void Main(string [] strings)
        {
            Console.WriteLine("Эта программа исследует " +
                              "функцию TestString()");
            Console.WriteLine();
            Example exampleObject = new Example();
            Console.WriteLine("Передача нулевого объекта:");
            string s = null;
            exampleObject.TestString(s);
            Console.WriteLine();
            // Теперь передаем в функцию нулевую (пустую) строку
            Console.WriteLine("Передача пустой строки:");
            exampleObject.TestString("");
            Console.WriteLine();
            // Наконец, передаем реальную строку
            Console.WriteLine("Передача реальной строки:");
            exampleObject.TestString("test string");
            Console.WriteLine();
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read() ,-

class Example
{
    public void TestString(string sTest)
    {
        // Проверка, не нулевой ли объект (эта проверка должна
        // быть первой!)
        if (sTest == null)
        {
            Console.WriteLine("sTest == null");
            return;
        }
        // Мы знаем, что sTest не указывает на нулевой объект,
        // но все еще может указывать на пустую строку.
        // Проверяем, не указывает ли sTest на нулевую
        // (пустую) строку
        if (String.Compare(sTest, "") == 0)
        {
            Console.WriteLine("sTest - ссылка на пустую строку");
            return;
        }
    }
}
```

```

    {
        // Строка не пустая, выводим ее
    } Console.WriteLine("sTest указывает на: '" + sTest + "'");
}
}

```

Функция `TestString O` использует сравнение `sTest == null` для проверки, не нулевой ли объект указывает ссылка. Для проверки, не указывает ли ссылка на пустую строку, функция `TestString()` использует функцию `Compare()` (функция `Compare()` возвращает 0, если переданные ей строки равны. В главе 9, "Работа со строками в C#", вы более детально познакомитесь со сравнением строк).

Вот как выглядит вывод этой программы на экран:

```

Эта программа исследует функцию TestString O
Передача нулевого объекта:
sTest == null
Передача пустой строки:
sTest - ссылка на пустую строку
Передача реальной строки:
sTest указывает на: 'test string'
Нажмите <Enter> для завершения программы...

```

Определение функции без возвращаемого значения

Выражение `public static double Average O` объявляет функцию `Average` как возвращающую значение типа `double`. Однако бывают функции, не возвращающие ничего. Ранее вы сталкивались с примером такой функции — `AverageAndDisplay O`, которая выводила вычисленное среднее значение на экран, ничего не возвращая вызывающей функции. Вместо того чтобы опустить в объявлении такой функции тип возвращаемого значения, в C# указывается `void`:

```
public void AverageAndDisplay(double, double)
```

Ключевое слово `void`, употребленное вместо имени типа, по сути, означает *отсутствие типа*, т.е. указывает, что функция `AverageAndDisplay()` ничего не возвращает вызывающей функции. (В C# любое объявление функции обязано указывать возвращаемый тип, даже если это `void`.)



Функция, которая не возвращает значения, программистами называется *void-функцией*, по использованному ключевому слову в ее описании.

Функции, не являющиеся `void`-функциями, возвращают управление вызывающей функции при выполнении оператора `return`, за которым следует возвращаемое вызывающей функции значение. Поскольку `void`-функция не возвращает никакого значения, выход из нее осуществляется посредством оператора `return` без какого бы то ни было значения либо (по умолчанию) при достижении закрывающей тело функции фигурной скобки.

Рассмотрим следующую функцию `DisplayRatio()`:

```
public class Example
{

```

```

public static void DisplayRatio(double dNumerator,
                                double dDenominator)
{
    // Если знаменатель равен 0...
    if (dDenominator == 0.0)
    {
        // ... вывести сообщение об ошибке и вернуть
        // управление вызывающей функции ...
        Console.WriteLine("Знаменатель не может быть нулем");
        // Выход из функции
        return;
    }
    // Эта часть функции выполняется только в том случае,
    // когда знаменатель не равен нулю
    double dRatio = dNumerator / dDenominator;
    Console.WriteLine("Отношение " + dNumerator
                      + " к " + dDenominator
                      + " равно " + dRatio);
} // Если знаменатель не равен нулю, выход из функции
// выполняется здесь

```

Функция `DisplayRatio()` начинает работу с проверки, не равно ли значение `dDenominator` нулю.

- ✓ Если значение `dDenominator` равно нулю, программа выводит сообщение об ошибке и возвращает управление вызывающей функции, не пытаясь вычислить значение отношения. При попытке вычислить отношение произошла бы ошибка деления на 0 с аварийным остановом программы в результате.
- ✓ Если значение `dDenominator` не равно нулю, программа выводит на экран значение отношения. При этом закрывающая фигурная скобка после вызова функции `WriteLine()` является закрывающей скобкой функции `DisplayRatio()` и, таким образом, представляет собой точку возврата из функции в вызывающую программу.

Передача аргументов в программу

Взгляните на любое консольное приложение в этой книге. Выполнение программы всегда начинается с функции `Main()`. Рассмотрим аргументы в следующем объявлении функции `Main()`:

```

public static void Main(string[] args)
{
    // ... Исходный текст программы ...
}

```

Функция `Main()` — статическая функция класса `Program`, определенная Visual Studio AppWizard. Она не возвращает значения и принимает в качестве аргументов массив объектов типа `string`. Что же это за строки?

Для запуска консольного приложения пользователь вводит имя программы в командной строке. При необходимости он может ввести в этой же строке после имени программы дополнительные аргументы. Вы видели это при использовании команд наподобие `copy myFile C:\myDir`, копирующей файл `myFile` в каталог `C:\myDir`.



Как показано в приведенной далее демонстрационной программе `Display Arguments`, массив объектов `string`, передаваемый в качестве параметра в функцию `Main()`, и представляет собой аргументы текущего вызова программы.

```
// DisplayArguments - вывод аргументов, переданных программе
using System;

namespace DisplayArguments
{
    class Test // Класс, содержащий функцию Main(), не обязан
               // называться Program
    {
        public static void Main(string[] args)
        {
            // Количество аргументов
            Console.WriteLine("У программы {0} аргументов",
                              args.Length);

            // Это аргументы:
            int nCount = 0;
            foreach (string arg in args)
            {
                Console.WriteLine("Аргумент {0} - {1}",
                                   nCount++, arg);
            }

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");

            Console.Read();
            return 0; // Другие программы, запущенные в
                    // консольном окне, могут проверить это
        } // возвращаемое значение; ненулевое
    } // значение обычно означает ошибку
}
```

Обратите внимание, что функция `Main()` может возвращать значение несмотря **4** то, что она объявлена как `void`, как и везде в настоящей книге. Однако она непременно должна быть объявлена как `static`.

Программа начинает свою работу с вывода длины массива `args`. Это значение ~~со~~ответствует количеству аргументов, переданных функции. Затем программа циклически|проходит по всем элементам массива `args`, выводя каждый его элемент на экран.

Ниже приведен пример вывода данной программы (в первой строке показан вид **ю**мандной строки, введенной при запуске программы):

```
DisplayArguments /c arg1 arg2
```

```
У программы 3 аргументов
Аргумент 0 - /c
Аргумент 1 - arg1
Аргумент 2 - arg2
Нажмите <Enter> для завершения программы...
```

Как видите, имя самой программы в список аргументов не входит (для того чтобы динамически выяснить имя программы, имеется другая функция). Пользователь ввел тра

параметра, и все они являются строками. Первый параметр — `"/с"`. Обратите внимание, что он обрабатывается и передается в массив `args` точно так же, как и все остальные аргументы. Только сама программа может определить, что именно означает каждый из аргументов и как он влияет на работу программы.

Передача аргументов из приглашения DOS

Чтобы запустить программу `DisplayArguments` из приглашения DOS, выполните следующие шаги.

1. Выберите команду `Starts Programs^Accessories^Command Prompt` (ПускО Программы^Стандартные^Командная строка).

После этого вы должны увидеть на экране черное окно с мигающим курсором рядом с приглашением `C: \>` (приглашение может включать каталог).

2. Перейдите в каталог, содержащий проект `DisplayArguments`, введя команду `cd \C#Programs\DisplayArguments\bin\Debug`.

(По умолчанию корневой каталог для демонстрационных программ из данной книги — `C#Programs`. Если выбранный вами каталог другой, введите его.)

Приглашение изменится и примет вид `C:\C#Programs\DisplayArguments\bin\Debug>`.



Если что-то идет не так — воспользуйтесь средствами Windows для поиска нужной папки. В Проводнике Windows щелкните правой кнопкой мыши на папке `C:\C#Programs` и выберите в раскрывающемся меню команду `Search` (Найти...), как показано на рис. 7.2.

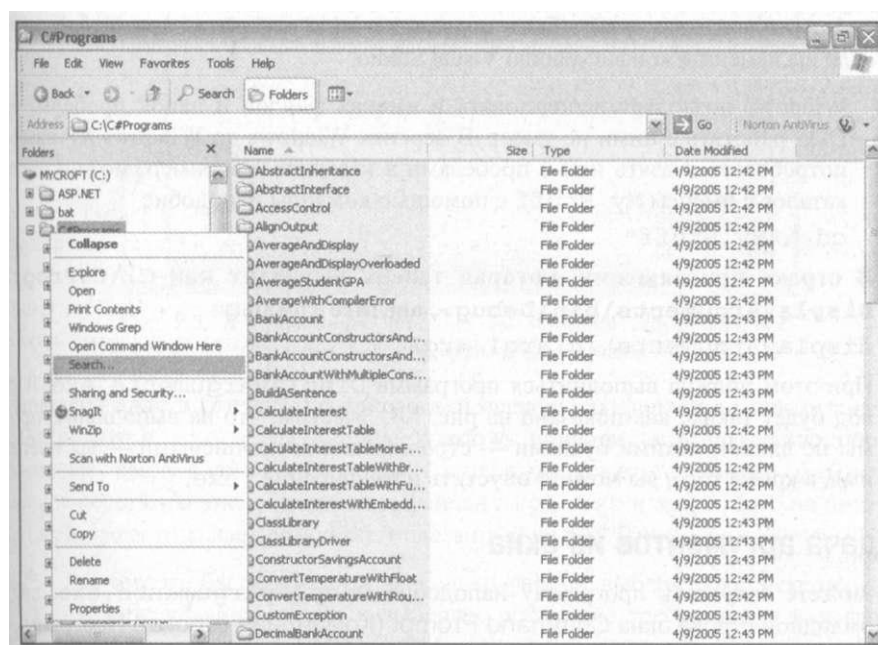


Рис. 7.2. Проводник Windows позволяет легко найти нужную папку на жестком диске

В появившемся диалоговом окне введите `DisplayArguments.exe` и щелкните кнопке **Search** (Найти). Найденные имена файлов появятся в правой части диалогового окна **Search Result** (Результаты поиска), как показано на рис. 7.3. Пропишите файл `DisplayArguments.exe` в каталоге `obj\Debug`; вам нужен файл в каталоге `bin\Debug`. Теперь, после того как вы нашли точное расположение файла и его полное имя, вернитесь в консольное окно и перейдите в требуемую папку.

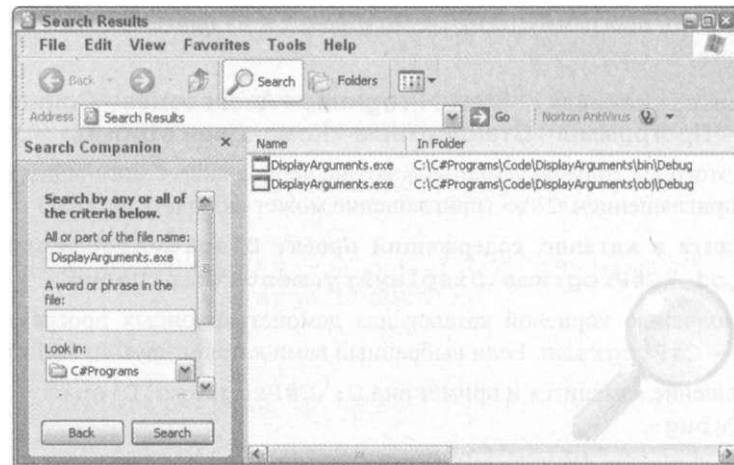


Рис. 7.3. Файл найден; имя соответствующей папки можно найти в правой части диалогового окна

Обычно Visual Studio 2005 размещает выполнимые файлы в подкаталоге `bin Debug`; однако это может быть подкаталог `bin\release` или любой другой, если вы измените конфигурацию Visual Studio.



Windows позволяет использовать в именах файлов и папок пробелы, одна DOS работать с ними не умеет. В версиях Windows до Windows XP вам может потребоваться взять имя с пробелами в кавычки. Например, можно перейти каталог с именем `My Stuff` с помощью команды наподобие

```
cd "\"My Stuff"
```

3. в строке приглашения, которая теперь выглядит как `C:\C#\Programs DisplayArguments\bin\Debug>`, введите команду `displayarguments /c arg1 arg2`.

При этом должна выполняться программа `DisplayArguments.exe`, и ее ввод будет таким, как показано на рис. 7.4. Заметим, что на выполнение программы не влияет, какими буквами — строчными или прописными — вы набрали имя, а кроме того, вы можете опустить и расширение `.exe`.

Передача аргументов из окна

Вы можете запустить программу наподобие `DisplayArguments.exe`, введя имя в командной строке окна **Command Prompt** (Командная строка). Программу можно также запустить и с использованием интерфейса Windows, дважды щелкнув на имени в окне или в Проводнике Windows.

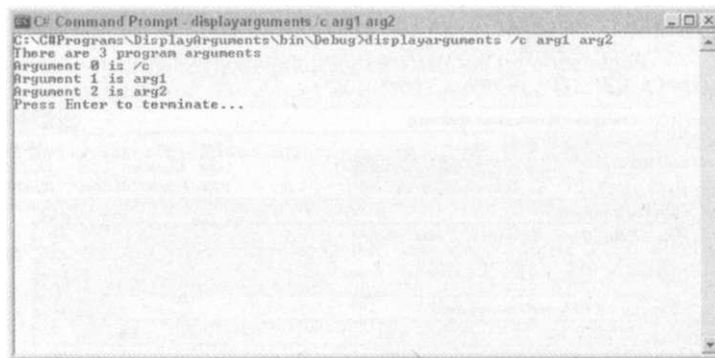


Рис. 7.4. Выполнение программы *DisplayArguments.exe* из приглашения DOS приводит к выводу информации о ее аргументах

Как показано на рис. 7.5, двойной щелчок на имени *DisplayArguments* приводит к запуску программы без передачи ей аргументов.

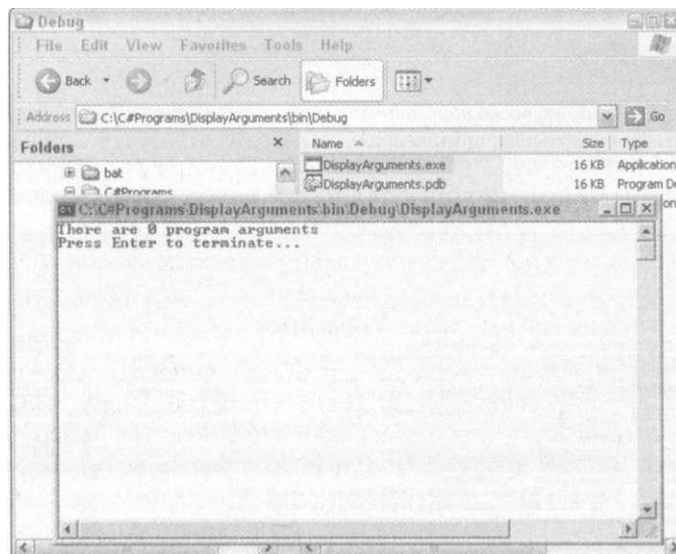


Рис. 7.5. В Проводнике Windows вы можете запустить программу посредством двойного щелчка на ее имени

Перетаскивание и отпускание одного или нескольких файлов на пиктограмму *DisplayArguments.exe* в Проводнике Windows приводит к выполнению программы, аналогичному вводу в командной строке *DisplayArguments имена файлов*. Одновременное перетягивание и отпускание файлов *arg1.txt* и *arg2.txt* на пиктограмму *DisplayArguments* дает результат, показанный на рис. 7.6.



Для того чтобы перетащить несколько файлов, выберите в списке первый файл, нажмите клавишу <Ctrl> и выберите остальные требующиеся вам файлы, как показано на рис. 7.6. Теперь нажмите кнопку мыши, перетяните все множество файлов и отпустите их на пиктограмму приложения *DisplayArguments*.

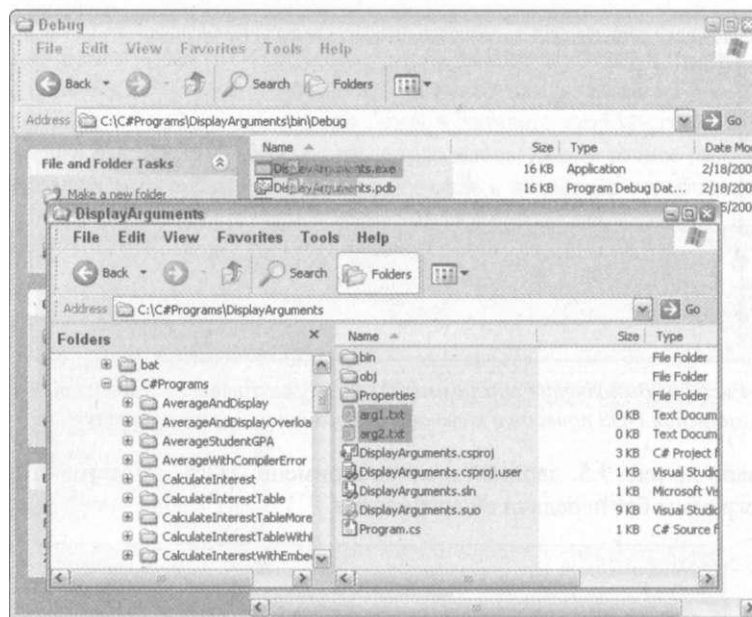


Рис. 7.6. Windows позволяет перетащить и отпустить файлы на пиктограмму консольного приложения

Вывод программы `DisplayArguments` в этом варианте запуска показан на рис. 7.7.

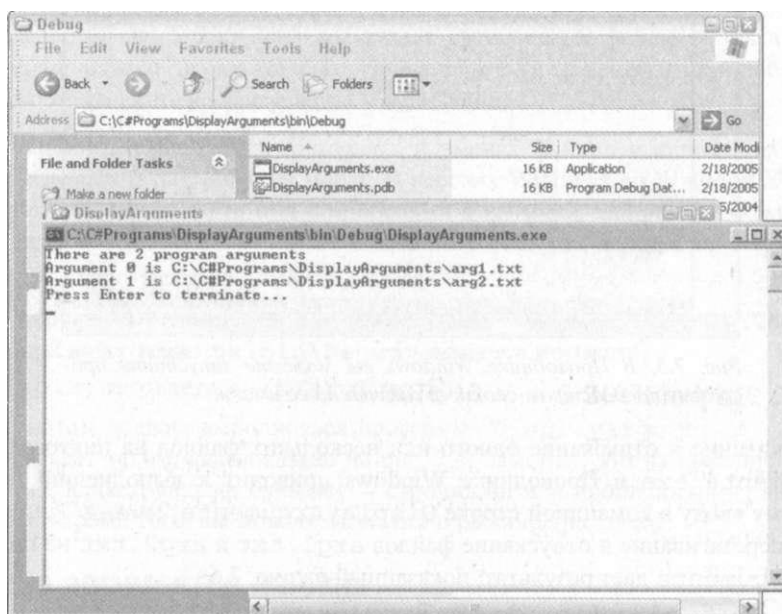


Рис. 7.7. Перетаскивание файлов на пиктограмму приложения дает тот же эффект, что и указание их имен в командной строке



Обратите внимание, что конкретный порядок передачи файлов приложению при использовании технологии перетаскивания не определен.

Функция `WriteLine ()`

Вы могли заметить, что функция `WriteLine ()`, использовавшаяся в рассматриваемых программах, представляет собой не более чем вызов функции класса `Console`:

```
Console.WriteLine("Это — вызов функции");
```

Функция `WriteLine ()` — одна из множества предопределенных функций, предоставляемых библиотекой `.NET`. `Console` — предопределенный класс, предназначенный для использования в консольных приложениях.

Аргументом функции `WriteLine ()`, применявшимся в примерах выше, является строка `string`. Оператор `+` позволяет программисту собрать эту строку из нескольких строк или строк и переменных встроенных типов, например, так:

```
string s = "Маша";  
Console.WriteLine ("Меня зовут " + s +  
                    " и мне " + 3 + " года");
```

В результате вы увидите выведенную на экран строку "Меня зовут Маша и мне Ев'года".

Второй вид функции `WriteLine ()` допускает наличие более гибкого множества аргументов, например:

```
Console.WriteLine ("Меня зовут {0} и мне {1} года",  
                  "Маша", 3);
```

Первый аргумент такого вызова называется форматной строкой. В данном примере строка "Маша" вставляется вместо символов `{0}` — ноль указывает на первый аргумент после командной строки. Целое число 3 вставляется в позицию, помеченную как `{1}`. Этот вид функции более эффективен, поскольку конкатенация строк не так проста, как это звучит, и не столь эффективна.

Кроме того, в этом варианте в форматной строке может использоваться ряд управляющих элементов, которые указывают, как именно должны выводиться аргументы функции `WriteLine ()`. Вы познакомитесь с ними в главе 9, "Работа со строками в C#".

Передача аргументов в Visual Studio 2005

Для того чтобы запустить программу в Visual Studio 2005, сначала удостоверьтесь, что она собрана без ошибок. Выберите команду меню **Builds Build имя программы** и убедитесь в отсутствии в окне Output сообщений об ошибках. Корректное сообщение в этом окне должно выглядеть как

```
Build: 1 succeeded, 0 failed, 0 skipped
```

Если в окне Output вы видите что-то другое — ваша программа не запустится.

Выполнить программу без передачи аргументов — дело одного щелчка. После того как программа успешно собрана, выберите команду меню **Debug^Start Debugging** (или нажмите клавишу `<F5>`) или **Debug^Start Without Debugging** (клавиши `<Ctrl+F5>`) и получите желаемое.

По умолчанию Visual Studio выполняет программу, не передавая ей аргументов. Если это не то, что вам нужно, вы должны указать Visual Studio, какие аргументы следует передавать. Для этого выполните такие шаги.

1. Откройте окно **Solution Explorer**, для чего воспользуйтесь командой меню **View: Solution Explorer**.

Окно **Solution Explorer** содержит описание вашего *решения*. Решение состоит из одного или нескольких проектов. Каждый проект описывает программу. Например, проект **DisplayArguments** гласит, что **Program.cs** — один из файлов вашей программы, и что **ваша** программа является консольным приложением. Проект также содержит описание других свойств, включая аргументы, используемые при запуске программы **DisplayArguments** из Visual Studio.

2. Щелкните правой кнопкой мыши на **DisplayArguments** в **Solution Explorer** и выберите из раскрывающегося меню команду **Properties**, как показано на рис. 7.8.

При этом перед вами появится окно вида, представленного на рис. 7.9, в котором можно указать множество различных настроек вашего проекта — только вводить этого без глубокого понимания, что именно вы настраиваете, ни в коем случае не следует.

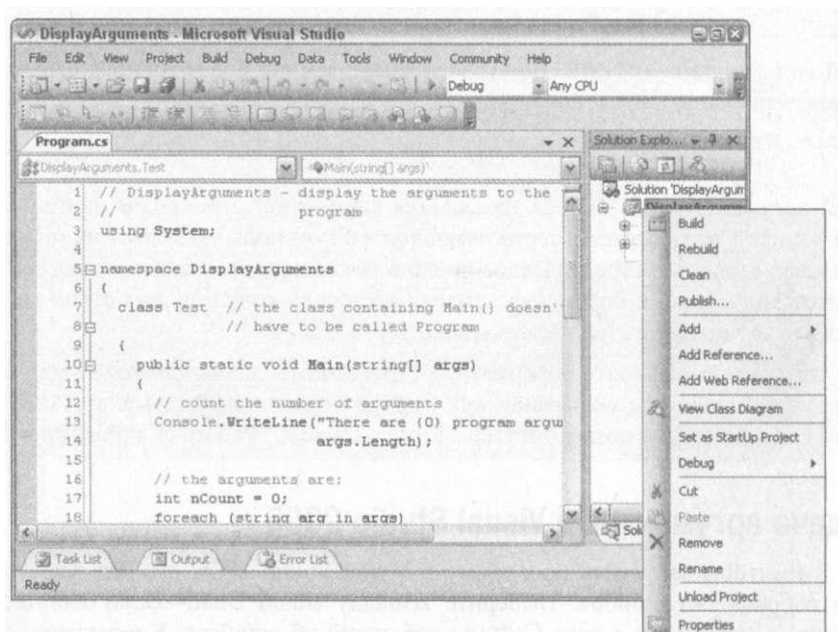


Рис. 7.8. Обращение к свойствам проекта посредством щелчка правой кнопкой мыши в **Solution Explorer**

3. На вкладке **DisplayArguments** выберите в списке вкладок в левой части **Debug**.
4. В поле **Command Line Arguments** группы **Start Options** введите аргументы, которые вы хотите передать в программу при запуске ее из Visual Studio.

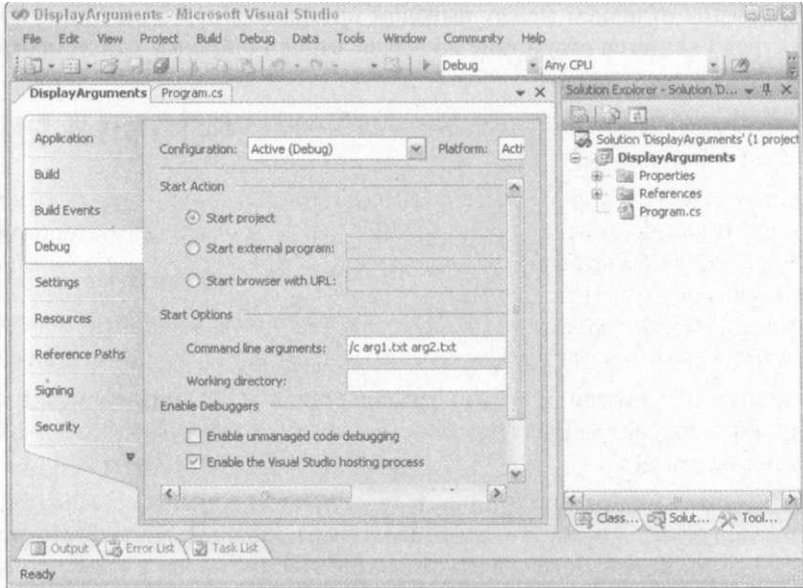


Рис. 7.9. Введите аргументы программы в поле *Command Line Arguments* на вкладке *Debug*

5. Сохраните и закройте окно *Properties*, а затем выполните программу с помощью команды меню *Debug>Start*.

Как показано на рис. 7.10, Visual Studio откроет окно DOS с ожидаемым результатом выполнения программы.

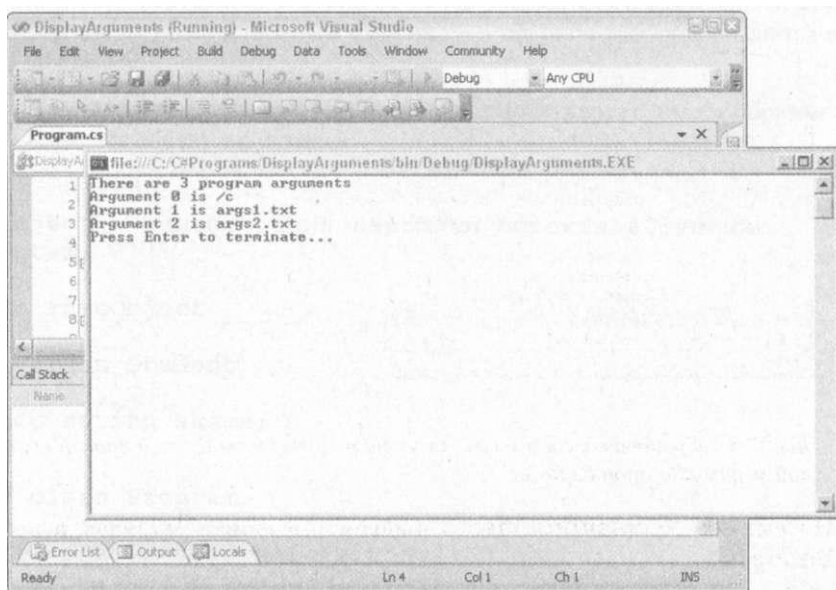


Рис. 7.10. Передача аргументов консольному приложению в *Visual Studio*

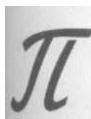
Единственным отличием между выводом программы из Visual Studio 2005 и из **Я** мандной строки является отсутствие на экране строки с именем самой программы и переданными ей аргументами.

Глава 8

Методы класса

В этой главе...

- > Передача объекта в функцию
- > Преобразование функции класса в метод
- > Что такое `this`
- > Генерация документации



После статических функций, рассматривавшихся в главе 7, "Функции функций", мы перейдем к нестатическим *методам* класса. Статические функции принадлежат всему классу, в то время как методы — экземплярам класса. Кстати, многие программисты предпочитают называть все одним словом — либо методами, либо функциями, не делая того различия между ними, на которое обращено ваше внимание здесь. Однако имеющееся отличие между статическими и нестатическими функциями очень важно.

Передача объекта в функцию

Ссылка на объект передается в функцию точно так же, как и переменная, принадлежащая типу-значению, с единственным отличием — объекты всегда передаются в функцию только по ссылке.



Следующая маленькая программа продемонстрирует, каким образом можно передать объект в функцию:

```
// PassObject - демонстрация передачи объекта в функцию
using System;

namespace PassObject
{
    public class Student
    {
        public string sName;
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            Student student = new Student();
        }
    }
}
```

```

// Присваиваем имя путем непосредственного обращения к
// полю объекта
Console.WriteLine("Сначала:");
student.sName = "Madeleine";
OutputName(student);

// Изменяем имя с использованием функции
Console.WriteLine("После изменения:");
SetName(student, "Willa");
OutputName(student);

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
                  "завершения программы...");
Console.Read();
}

// OutputName - Вывод имени студента
public static void OutputName(Student student)
{
    // Вывод текущего имени студента
    Console.WriteLine("Student.sName = {0}",
                      student.sName);
}

// SetName - изменение имени студента
public static void SetName(Student student,
                           string sName)
{
    student.sName = sName;
}
}

```

Программа создает объект `student`, в котором не содержится ничего, кроме имени. Она сначала присваивает имя непосредственно и выводит его с помощью функции `OutputName()`.

Затем программа изменяет имя посредством функции `SetName()`. Поскольку все объекты в C# передаются в функции по ссылке, изменения, внесенные в объект `student` в функции, остаются и после возврата из нее. Когда функция `Main()` опять вызывает функцию для вывода имени студента, последняя выводит измененное имя, что видно из вывода программы на экран:

Сначала:

`Student.sName = Madeleine`

После изменения:

`Student.sName = Willa.`

Нажмите <Enter> для завершения программы.



Обратите внимание, что при передаче ссылочного объекта в функцию ключевое слово `ref` не используется. Функция, которой объект передается по ссылке, может посредством этой ссылки изменить только *содержимое* объекта, в не в состоянии присвоить новый объект, как показано в следующем фрагмент исходного текста:

```

Student student = new Student();
student.Name = "Madeleine";
OutputName (student);
Console.WriteLine (student.Name); // Вывод еще "Madeleine"

// Исправленная функция OutputName() :
public static void OutputName (Student student)
{
    student = new Student(); // Не приводит к изменению
                              // объекта student вне
                              // OutputName()
    student.Name = "Pam";
}

```

Определение функций объектов и методов

Класс представляет собой набор элементов, описывающий объект или концепцию реального мира. Например, класс `Vehicle` может содержать данные о максимальной скорости, максимальном разрешенном весе, количестве пассажирских мест и т.д. Однако транспортное средство имеет и активные свойства: возможность тронуться с места, остановиться и т.п. Эти действия можно описать функциями, работающими с данными транспортного средства. Эти функции представляют собой такую же часть класса `Vehicle`, как и его члены-данные.

Определение функций - статических членов



Например, вы можете переписать программу из предыдущего раздела немного иначе:

```

// PassObjectToMemberFunction - для работы с полями объекта
// используется статическая функция-член
using System;

namespace PassObjectMemberToFunction
{
    public class Student
    {
        public string sName;

        // OutputName - вывод имени студента
        public static void OutputName(Student student)
        {
            // Вывод текущего имени студента
            Console.WriteLine("Student.sName = {0}",
                              student.sName);
        }

        // SetName - изменяем имя студента
        public static void SetName(Student student,

```

```

        string sName)
    {
        student.sName = sName;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Student student = new Student();
        // Присваиваем имя непосредственным обращением к
        // объекту-
        Console.WriteLine("Сначала:");
        student.sName = "Madeleine";
        Student.OutputName(student); // Теперь функция
                                    // принадлежит классу
                                    // Student

        // Изменяем имя с помощью функции
        Console.WriteLine("После изменения:");
        Student.SetName(student, "Willa");
        Student.OutputName(student);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
        Console.Read();
    }
}

```

По сравнению с программой **PassObject** данная программа имеет только одно важное изменение: функции **OutputName ()** и **SetName ()** перенесены в класс **Student**.

Из-за этого изменения функция **Main ()** вынуждена обращаться к означенным функциям с указанием класса **Student**. Эти функции теперь являются членами класса **Student**, а не **Program**, которому принадлежит функция **Main ()**

Это маленький, но достаточно важный шаг. Размещение функции **OutputName!** в классе приводит к повышению степени повторного использования: внешние функции, которым требуется выполнить вывод объекта на экран, могут найти функцию **OutputName ()** вместе с другими функциями в классе, следовательно, писать такие функции для каждой программы, применяющей класс **Student**, не требуется.

Указанное решение лучше и с философской точки зрения. Класс **Program** не должен беспокоиться о том, как инициализировать имя объекта **Student**, или о том, как вывести его имени на экран. Вся эту информацию должен содержать сам класс **Student**. Объекты отвечают сами за себя.



В действительности функция **Main ()** не должна инициализировать объем именем **Madeleine** непосредственно — в этом случае также следует использовать функцию **SetName ()**.

Внутри самого класса `Student` одна функция-член может вызывать другую **без** явного указания имени класса. Функция `SetName()` может вызвать функцию `OutputName()`, не указывая имени класса. Если имя класса не указано, **C#** считает, что вызвана функция из того же класса.

Определение метода

Обращение к членам данных объекта — *экземпляра* класса — выполняется посредством указания объекта, а не класса:

```
Student student = new Student(); // Создание экземпляра Student
student.sName = "Madeleine";    // Обращение к члену
```

C# позволяет вызывать *нестатические* функции-члены аналогично:

```
student.SetName("Madeleine");
```



Следующий пример демонстрирует это:

```
//InvokeMethod - вызов функции-члена с указанием объекта
using System;
```

```
namespace InvokeMethod
{
    class Student
    {
        // Информация об имени студента
        public string sFirstName;
        public string sLastName;

        // SetName - сохранение информации об имени
        public void SetName(string sFName, string sLName)
        {
            sFirstName = sFName;
            sLastName = sLName;
        }

        // ToNameString - преобразует объект класса Student
        // строку для вывода
        public string ToNameString()
        {
            string s = sFirstName + " " + sLastName;
            return s;
        }
    }
}

public class Program
{
    public static void Main()
    {
        Student student = new Student();
        student.SetName("Stephen", "Davis");
    }
}
```

```

        Console.WriteLine("Имя студента "
                           + student.ToNameString());

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");

        Console.Read();
    }
}
}

```

Вывод данной программы состоит из одной строки:

Имя студента Stephen Davis

Эта программа очень похожа на программу `PassObjectToMemberFunction`. В введенной версии используются *нестатические* функции для работы с именем и фамилией,

Программа начинает с создания нового объекта `student` класса `Student`, после чего вызывает функцию `SetName()`, которая сохраняет строки "Stephen" и "Davis" в членах-данных `sFirstName` и `sLastName`. И наконец, программа вызывает функцию-член `ToNameString()`, возвращающую имя студента, составленное из двух а



По историческим причинам, никак не связанным с C#, нестатические функции называются *методами*. В книге будет использоваться термин *метод* для функций-членов, и термин *функция* — для функций всех типов. Некоторые программисты применяют термины *метод экземпляра* (нестатический) и *метод класса* (статический).

Вернемся вновь к методу `SetName()`, предназначенному для изменения значений лей объекта класса `Student`. Какой именно объект модифицирует метод `SetName`? Рассмотрим, как работает следующий пример:

```

Student christa = new Student(); // Создаем двух совершенно
Student sarah   = new Student(); // разных студентов
christa.SetName("Christa", "Smith");
sarah.SetName("Sarah", "Jones");

```

Первый вызов `SetName()` изменяет поля объекта `christa`, а второй — объект `sarah`.



Программисты на C# говорят, что метод работает с *текущим* объектом. В первом вызове текущим объектом является `christa`, во втором — `sarah`.

Полное имя метода

Имеется тонкая, но важная проблема, связанная с описанием имен методов. Рассмотрим следующий фрагмент исходного текста:

```

public class Person
{
    public void Address()
    {
        Console.WriteLine("Hi");
    }
}

```



```

public class Letter
{
    string sAddress;
    // Сохранение адреса
    public void Address(string sNewAddress)
}

    sAddress = sNewAddress;
}
}

```

Любое обсуждение метода `Address()` после этого становится неоднозначным. Метод `Address()` класса `Person` не имеет ничего общего с методом `Address()` класса `Letter`. Если кто-то скажет, что в этом месте нужен вызов метода `Address()`, то какой именно метод `Address()` имеется в виду?

Проблема не в самих методах, а в описании. Метод `Address()` как независимая самодостаточная сущность просто не существует — существуют методы `Person.Address()` и `Letter.Address()`. Путем добавления имени класса в начало имени метода явно указывается, какой именно метод имеется в виду.

Это описание имени метода очень похоже на описание имени человека. К примеру, в семье меня знают как Стефана (честно говоря, в семье меня зовут несколько иначе, но это уже несущественные подробности). В семье больше нет Стефанов (по крайней мере в моей семье), но вот на работе есть еще два Стефана.

Когда я обедаю с коллегами в компании, где нет этих других Стефанов, имя *Стефан* однозначно относится ко мне. Но когда все оказываются на рабочих местах, чтобы избежать неоднозначности, следует добавлять к имени и фамилию и обращаться к Стефану Дэвису, Стефану Вильямсу или Стефану Лейе.

Таким образом, `Address()` можно рассматривать как имя метода, а его класс — как фамилию.

Обращение к текущему объекту

Рассмотрим следующий метод `Student.SetName()`:

```

class Student

    // Информация об имени студента
    public string sFirstName;
    public string sLastName;

    // SetName - сохранение информации об имени
    public void SetName(string sFName, string sLName)
    {
        sFirstName = sFName;
        sLastName = sLName;
    }
}

public class Program

```

Шва 8. Методы класса

```

public static void Main()

```

```

{
    Student student1 = new Student();
    student1.SetName("Joseph", "Smith");
    Student student2 = new Student();
    student2.SetName("John", "Davis");
}
}

```

Функция `Main()` использует метод `SetName()` для того, чтобы обновить поля объектов `student1` и `student2`. Но внутри метода `SetName()` нет ссылки ни на какой объект типа `Student`. Как уже было выяснено, метод работает "с текущим объектом". Но как он знает, какой именно объект — текущий?

Ответ прост. Текущий объект передается при вызове метода как неявный аргумент-1. Например, вызов

```
student1.SetName("Joseph", "Smith");
```

эквивалентен следующему:

```
Student.SetName(student1, "Joseph", "Smith");
// Это - эквивалентный вызов (однако этот вызов не будет
// корректно скомпилирован)
```

Я не хочу сказать, что вы можете вызвать метод `SetName()` двумя различными способами, я просто подчеркиваю, что эти два вызова семантически эквивалентны. Объем являющийся текущим — скрытый первый аргумент — передается в функцию так же, и другие аргументы. Оставьте эту задачу компилятору.

А что можно сказать о вызове одного метода из другого? Этот вопрос иллюстрируется следующим фрагментом исходного текста:

```

public class Student
{
    public string sFirstName, -
    public string sLastName;
    public void SetName(string sFirstName, string sLastName)

        SetFirstName(sFirstName);
        SetLastName(sLastName);

    public void SetFirstName(string sName)

        sFirstName = sName;

    public void SetLastName(string sName)

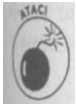
        sLastName = sName;
}

```

В вызове `SetFirstName()` не видно никаких объектов. Дело в том, что при вызове одного метода объекта из другого в качестве неявного текущего объекта передается тот же объект, что и для вызывающего метода. Обращение к любому члену в методе рассматривается как обращение к текущему объекту, так что метод знает сам, какому именно объекту он принадлежит.

Ключевое слово `this`

В отличие от других аргументов, текущий объект в список аргументов функции не попадает, а значит, программист не назначает ему никакого имени. Однако `C#` не оставляет этот объект безымянным и присваивает ему не слишком впечатляющее имя `this`, которое может пригодиться в ситуациях, когда вам нужно непосредственно обратиться к текущему объекту.



`this` — ключевое слово `C#`, так что оно не может использоваться в программе ни для какой иной цели, кроме описываемой.

Итак, можно переписать рассматривавшийся выше пример следующим образом:

```
public class Student
(
    public string sFirstName;
    public string sLastName;
    public void SetName(string sFirstName, string sLastName)
    {
        // Явная ссылка на "текущий объект" с применением
        // ключевого слова this
        this.SetFirstName(sFirstName);

        this.SetLastName(sLastName);

    public void SetFirstName(string sName)

        this.sFirstName = sName;

    public void SetLastName(string sName)

        this.sLastName = sName;
```

Обратите внимание на явное добавление ключевого слова `this`. Добавление `this` к ссылкам на члены не привносит ничего нового, поскольку наличие `this` подразумевается и так. Однако когда `Main()` делает показанный ниже вызов, `this` означает `student1` как в функции `SetName()`, так и в любом другом методе, который может быть вызван: `student1.SetName("John", "Smith");`

Когда `this` используется явно

Обычно явно использовать `this` не требуется, так как компилятор достаточно разумен, чтобы разобраться в ситуации. Однако имеются две распространенные ситуации, когда это следует делать. Например, ключевое слово `this` может потребоваться при инициализации членов данных:

```
class Person
(
    public string sName;
    public int nID;
    public void Init(string sName, int nID)
```

```

    {
        this.sName = sName; // Имена аргументов те же,
                             // что имена членов-данных
        this.nID = nID;
    }
}

```

Аргументы метода `Init()` носят имена `sName` и `nID`, которые совпадают с именами соответствующих членов-данных. Это повышает удобочитаемость, поскольку очевидно, в какой переменной какой аргумент следует сохранить. Единственная проблема, что имя `sName` имеется как в списке аргументов, так и среди членов-данных. Эта ситуация оказывается слишком сложной для компилятора.

Добавление `this` проясняет ситуацию, четко определяя, что именно подразумевая под `sName`. В `Init()` имя `sName` обозначает аргумент функции, в то время и `this.sName` — член объекта.



Ключевое слово `this` требуется также при сохранении текущего объекта для последующего применения или использования в некоторой другой функции. Рассмотрим следующую демонстрационную программу `ReferencingThisExplicitly`:

```

// ReferencingThisExplicitly - программа демонстрирует явное
// использование this
using System;

namespace ReferencingThisExplicitly
{
    public class Program
    {
        public static void Main(string[] strings)
        {
            // Создание студента
            Student student = new Student();
            student.Init("Stephen Davis", 1234);
            // Внесение курса в список
            Console.WriteLine("Внесение в список Stephen Davis " +
                              "курса Biology 101");
            student.Enroll("Biology 101");

            // Вывод прослушиваемого курса
            Console.WriteLine("Информация о студенте:");
            student.DisplayCourse();

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }

    // Student - класс, описывающий студента
    public class Student

```

```

// Все студенты имеют имя и идентификатор
public string sName;
public int nID;

// Курс, прослушиваемый студентом
CourseInstance courseInstance;

// Init - инициализация объекта
public void Init(string sName, int nID)
{
    this.sName = sName;
    this.nID = nID;

    courseInstance = null;
}

// Enroll - добавление в список
public void Enroll(string sCourseID)
{
    courseInstance = new CourseInstance();
    courseInstance.Init(this, sCourseID);

// Вывод имени студента и прослушиваемых курсов
public void Display-Course ()
{
    Console.WriteLine(sName) ;
    courseInstance.Display() ;
}

// CourseInstance - объединение информации о студенте и
// прослушиваемом курсе
public class CourseInstance
{
    public Student student;
    public string sCourseID;

    // Init - связь студента и курса
    public void Init(Student student, string sCourseID)
    (
        this.student = student;
        this.sCourseID = sCourseID;

// Display - вывод имени курса
public void Display ()
{
    Console.WriteLine(sCourseID);
}
}

```

В этой программе в объекте `Student` имеются поля для имени и идентификатора студента и один экземпляр курса (да, студент не очень ретивый...). Функция создает экземпляр `student`, после чего вызывает функцию `Init()` для его инициализации. В этот момент ссылка `courseInstance` равна `null`, поскольку студенту назначен ни один курс.

Метод `Enroll()` назначает студенту курс путем инициализации ссылки `seInstance` новым объектом. Однако метод `CourseInstance.Init()` использует экземпляр класса `Student` в качестве первого аргумента. Какой экземпляр должен быть передан? Очевидно, что вы должны передать текущий объект класса `Student` именно тот, ссылкой на который является `this`.

Что делать при отсутствии `this`

Смешивать статические функции классов и нестатические методы объектов не из лучших, но тем не менее C# и здесь может прийти на помощь.



Чтобы понять, в чем состоит суть проблемы, давайте рассмотрим исходный текст:

```
// MixingFunctionsAndMethods - совмещение функций класса и
// методов объектов может привести к проблемам
using System;

namespace MixingFunctionsAndMethods
{
    public class Student
    {
        public string sFirstName;
        public string sLastName;

        // InitStudent - инициализация объекта student
        public void InitStudent(string sFirstName, string
sLastName)

        {
            this.sFirstName = sFirstName;
            this.sLastName = sLastName;

        }

        // OutputBanner - вывод начальной строки
        public static void OutputBanner()

        {
            Console.WriteLine("Никаких хитростей:");
            // Console.WriteLine(? какой объект используется ?);

        }

        public void OutputBannerAndName()

        {
            // Используется класс Student, но статическому методу
            // не передаются никакие объекты
            OutputBanner();
        }
    }
}
```



```

        // Явная передача объекта
        OutputName(this);
    }

    // OutputName - вывод имени студента
    public static void OutputName(Student student)
    {
        // Здесь объект указан явно
        Console.WriteLine("Имя студента - {0}",
                           student.ToNameString());
    }

    // ToNameString - получение имени студента
    public string ToNameString()
    {
        // Здесь текущий объект указан неявно; можно
        // использовать this:
        // return this.sFirstName + " " + this.sLastName;
        return sFirstName + " " + sLastName;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Student student = new Student();
        student.InitStudent("Madeleine", "Cather");
        // Вывод заголовка и имени
        Student.OutputBanner();
        Student.OutputName(student);
        Console.WriteLine();

        // Вывод заголовка и имени еще раз
        student.OutputBannerAndName();

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
        Console.Read();
    }
}

```



Следует начать с конца программы, с функции `Main()`, чтобы вы лучше разглядели имеющиеся проблемы. Программа начинается с создания объекта класса `Student` и инициализации его имени. Затем она хочет всего лишь вывести имя студента с небольшим заголовком.

Функция `Main()` вначале выводит заголовок и сообщение с использованием статических функций. Программа вызывает функцию `OutputBanner()` для вывода строки заголовка и `OutputName()` для вывода имени студента. Функция `OutputBanner()`

просто выводит строку на консоль. Функция `OutputName()` получает в качестве аргумента объект класса `Student`, так что она может получить и вывести имя студента.

Далее функция `Main()` использует для решения той же задачи функцию объекта и вызывает `student.OutputBannerAndName()`.

Метод `student.OutputBannerAndName()` сначала вызывает статическую функцию `OutputBanner()`. При этом вызове, поскольку не указан класс, к которому принадлежит функция, подразумевается, что она принадлежит тому же классу, что и вызывание функция, т.е. классу `Student`. После этого выполняется вызов `OutputName()`. Это тоже функция класса, и точно так же устанавливается ее принадлежность классу `Student`. Однако эта функция требует передачи ей объекта класса `Student`. Метод `student.OutputBannerAndName()` передает ей в качестве этого аргумента `this`.

Более интересная ситуация возникает при вызове `ToString()` из `OutputName()`. Функция `OutputName()` объявлена как `static`, таким образом, не имеет `this`. У нее есть переданный ей объект класса `Student`, который она и использует для осуществления вызова.



Статическая функция не может вызывать нестатические методы без явного указания объекта. Нет объекта — нет и вызова. В общем случае статическая функция не может обратиться ни к одному нестатическому элементу класса. Однако нестатические методы могут обращаться как к статическим, так и к нестатическим членам класса — данным и функциям.

Помощь от Visual Studio — автоматическое завершение

В среде Visual Studio программисту доступна одна важная возможность, известные как автозавершение (auto-complete). Когда вы вводите имя класса или объекта в ~~вашем~~ исходном тексте, Visual Studio пытается предвидеть, какое имя класса или метода вы намерены ввести.

Описать автозавершение в Visual Studio проще на конкретном примере. Для демонстрации того, как работает эта возможность Visual Studio, представлен следующий фрагмент исходного текста из программы `Mixing Functions And Methods`:

```
// Вывод заголовка и имени
Student.OutputBanner();
Student.OutputName(student);
Console.WriteLine();
// Повторный вывод заголовка и имени
student.OutputBannerAndName();
```



Автозавершение — удобная вещь, но если вам требуется большая помощь, их пользуйтесь командой `Help^Index` для вызова справки. Выводимый предметный указатель можно ограничить, например, только темами, связанными с C# или .NET. Воспользуйтесь этим предметным указателем так же, как вы пользуетесь предметным указателем книги. К вашим услугам также команды поиска по текстам статей справки и содержания справочной системы. Нужные вам статьи можно пометить с тем, чтобы в будущем можно было быстро к ним возвращаться.

Справка по встроенным функциям системной библиотеки

При использовании фрагмента программы `Mixing Functions And Methods` в качестве примера, как только вы вводите в редакторе `Console.`, Visual Studio сразу же выводит список всех методов `Console`. Стоит ввести `W`, как Visual Studio переходит в списке к первому методу, начинающемуся на `W` (а именно — `Write()`). Если вы нажмете клавишу `<1>`, то перейдете к методу `WriteLine()`, а справа от списка появится справка по этому методу (рис. 8.1). В справке, в частности, указано, что имеется 19 перегруженных версий данного метода — естественно, каждая со своим набором аргументов.

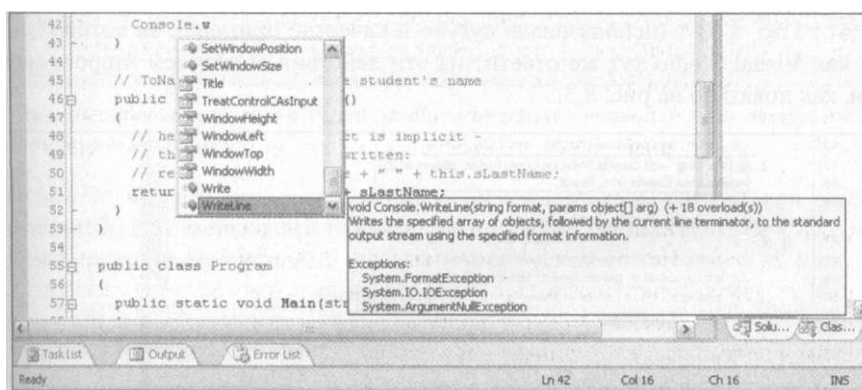


Рис. 8.1. Автозавершение в Visual Studio позволяет правильно выбрать требуемый метод

Вы завершаете ввод имени `WriteLine`. Как только вы введете после имени открывающую скобку, Visual Studio изменит выводимое окно подсказки — теперь в нем будут показаны возможные аргументы функции (рис. 8.2).

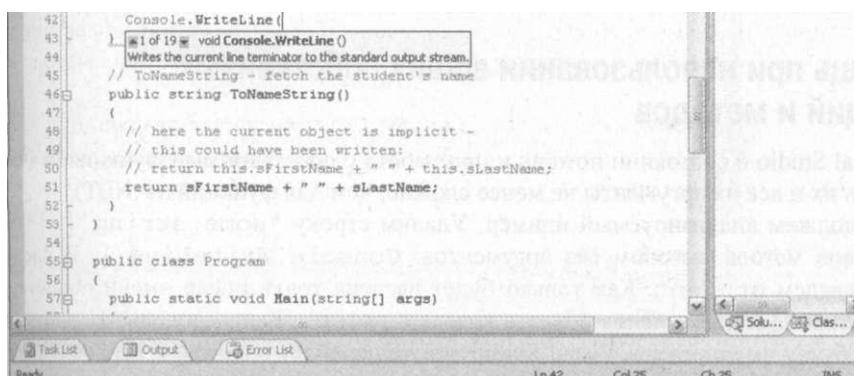


Рис. 8.2. Visual Studio подсказывает, какие возможные аргументы может принимать функция



Не нужно полностью вводить имя функции. Предположим, вы ввели `WriteL`, чего вполне достаточно, чтобы однозначно идентифицировать нужный вам метод. Не завершая ввода имени, введите открывающую скобку, и Visual Studio завершит ввод имени метода за вас. Вы можете также воспользоваться клавишами `<Ctrl+Space>` для появления раскрывающегося меню автозавершения.

Справка по встроенным функциям системной библиотеки

При использовании фрагмента программы `Mixing Functions And Methods` в качестве примера, как только вы вводите в редакторе `Console.`, Visual Studio сразу же выводит список всех методов `Console`. Стоит ввести `W`, как Visual Studio переходит в список к первому методу, начинающемуся на `W` (а именно — `Write()`). Если вы нажмете клавишу `<I>`, то перейдете к методу `WriteLine()`, а справа от списка появится справка по этому методу (рис. 8.1). В справке, в частности, указано, что имеется 19 перегруженных версий данного метода — естественно, каждая со своим набором аргументов.

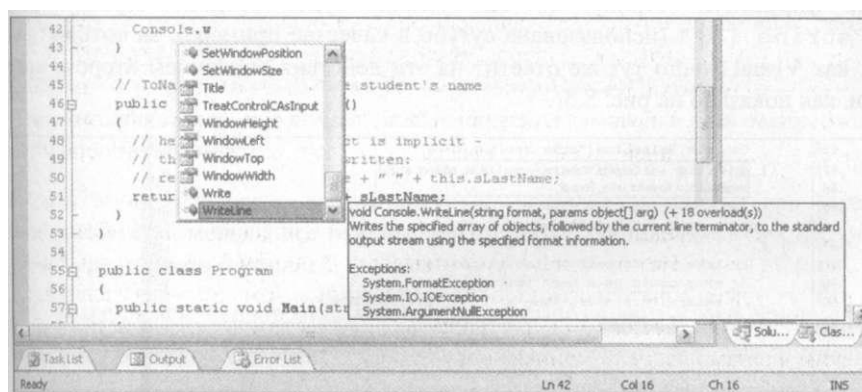


Рис. 8.1. Автозавершение в Visual Studio позволяет правильно выбрать требуемый метод

Вы завершаете ввод имени `WriteLine`. Как только вы введете после имени открывающую скобку, Visual Studio изменит выводимое окно подсказки — теперь в нем будут показаны возможные аргументы функции (рис. 8.2).

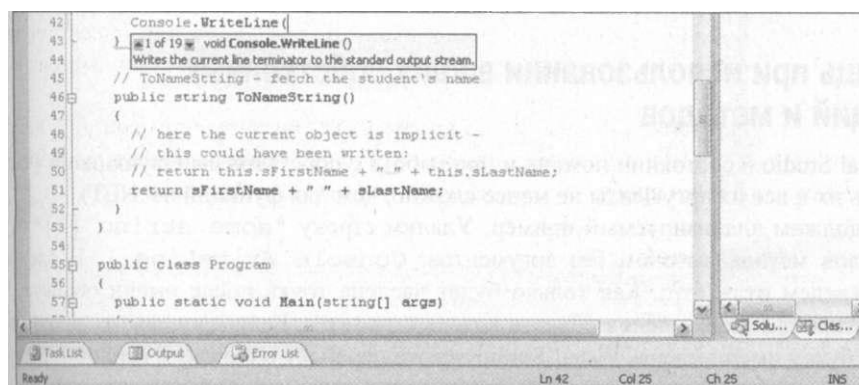


Рис. 8.2. Visual Studio подсказывает, какие возможные аргументы может принимать функция



Не нужно полностью вводить имя функции. Предположим, вы ввели `WriteL`, чего вполне достаточно, чтобы однозначно идентифицировать нужный вам метод. Не завершая ввода имени, введите открывающую скобку, и Visual Studio завершит ввод имени метода за вас. Вы можете также воспользоваться клавишами `<Ctrl+Space>` для появления раскрывающегося меню автозавершения.



Можно щелкнуть мышью в левой части всплывающего окна справки для и чтобы найти интересующую версию перегрузки функции `WriteLine()`. описанием функции вы увидите описание ее первого аргумента (если такой имеется). Функция `WriteLine()` имеет 19 вариантов перегрузки для разных типов данных. Первый из них, который вы увидите во всплывающем окне, не требует аргументов. Посредством клавиш `<↓>` и `<↑>` можно прокручивать список перегрузок.

Если воспользоваться версией `WriteLine`, которая в качестве первого аргумента получает форматную строку, то как только будет введена строка, например `"some string {0}"` (использована сугубо в качестве примера), за которой вводится запятая, как Visual Studio тут же ответит на эти действия описанием второго аргумента функции, как показано на рис. 8.3.

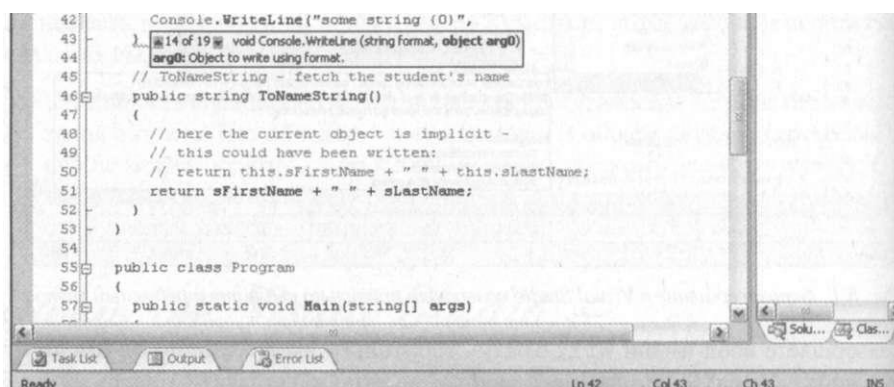


Рис. 8.3. Visual Studio предоставляет информацию по каждому аргументу функции

Помощь при использовании ваших собственных функций и методов

Visual Studio в состоянии помочь и при работе с собственными функциями (часто помнить их и все их аргументы не менее сложно, чем для функций из .NET).

Продолжим анализируемый пример. Удалим строку `"some string {0}"` и за ним вызов метода вызовом без аргументов: `Console.WriteLine()`. В следующей строке введем `student`. Как только будет введена точка после имени объекта, Visual Studio откроет список членов объекта класса `Student`. Если продолжить ввод и ввести первую букву имени члена, Visual Studio тут же перейдет в списке к первому члену с этой буквой, как показано на рис. 8.4. Будет также показано и объявление метода, что вы могли вспомнить, как им пользоваться.

По пиктограмме слева от имени в списке автозавершения можно узнать, имеете ли дело с членами-данными или с методами.



Гораздо проще определить это по цвету: члены-данные имеют цвет морских волн, а методы — розовый.

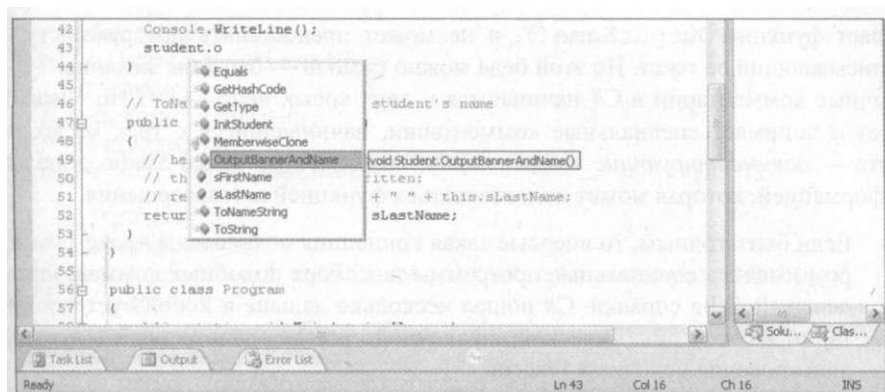


Рис. 8.4. Автозавершение в Visual Studio может работать и с пользовательскими классами и методами

Вы можете встретиться с неизвестными вам методами, такими как Equals или GetHashCode. Эти методы все без исключения объекты получают от C# (совершенно бесплатно) для технических целей.



Повторимся — ввод открывающей скобки позволяет Visual Studio автоматически завершить ввод имени метода.

Та же возможность автозавершения работает и с функциями. Если вы введете имя класса Student, за которым следует точка, Visual Studio откроет список членов этого класса. Как только вы введете OutputN, Visual Studio отреагирует на это списком аргументов функции OutputName (), как показано на рис. 8.5.

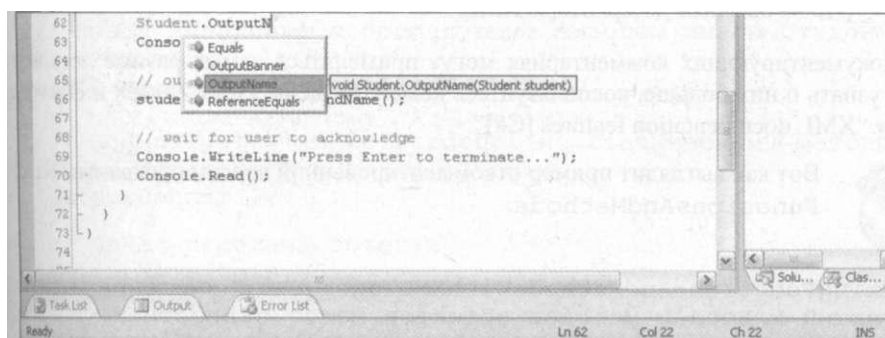


Рис. 8.5. Visual Studio обеспечивает вас информацией независимо от того, работаете вы с методами объекта или функциями класса

Внесение дополнений в справочную систему

Функция автозавершения Visual Studio существенно облегчает работу программиста, однако, к огромному сожалению, когда дело касается пользовательских классов или функций, ее возможности оказываются весьма ограниченными.

Visual Studio в состоянии предоставить только ограниченную справочную информацию по пользовательским классам и функциям. Например, Visual Studio не знает о том,

что делает функция `OutputName()`, и не может предоставить программисту какой-либо описывающий ее текст. Но этой беде можно помочь — было бы желание.

Обычные комментарии в C# начинаются с двух косых черт — `//`. Но Visual Studio выделяет и понимает специальные комментарии, начинающиеся с трех косых черт — `///`. Это — *документирующие комментарии*, снабжающие Visual Studio дополнительной информацией, которая может использоваться функцией автозавершения.



Если быть точным, то впервые такая концепция появилась в языке Java, в котором имелись специальные программы для сбора подобных комментариев в отдельный файл справки. C# пошел несколько дальше и использует эти комментарии в своей динамической справочной системе, работающей в процессе редактирования исходных текстов.

Документирующие комментарии могут содержать любую комбинацию элементов, показанных в табл. 8.1.

Таблица 8.1. Основные команды документирующих комментариев	
Команда	Значение
<code><summary></summary></code>	Описание самой функции. Выводится, когда вы вводите имя функции в процессе редактирования
<code><param></param></code>	Описание аргументов функции. Выводится после того, как вы введете имя функции и открывающую скобку. Для каждого аргумента используется своя пара дескрипторов <code><param></code>
<code><returns></returns></code>	Описывает возвращаемое функцией значение



Документирующие комментарии следуют правилам XML7HTML: команда **начинается** с дескриптора `<command>` и заканчивается дескриптором `</command>`. Это — обычные дескрипторы XML.

В документирующих комментариях могут применяться самые разные дескрипторы. Чтобы узнать о них больше, воспользуйтесь командой меню `Help^Index` и обратитесь к разделу "XML documentation features [C#]".



Вот как выглядит пример откомментированной версии программы `Mixing FunctionsAndMethods`:

```
// MixingFunctionsAndMethodsWithXMLTags - - совмещение
// функций класса и методов объектов может привести к
// проблемам
using System;

namespace MixingFunctionsAndMethods
{
    /// <summary>
    /// Простое описание студента
    /// </summary>
    public class Student
    {
        /// <summary>
        /// Имя студента
```

```

/// </summary>
public string sFirstName;
/// <summary>
/// Фамилия студента
/// </summary>
public string sLastName;

// InitStudent
/// <summary>
/// Инициализация студента перед его использованием.
/// </summary>
/// <param name="sFirstName">Имя студента</param>
/// <param name="sLastName">Фамилия студента</param>
public void InitStudent(string sFirstName,
                        string sLastName)
{
    this.sFirstName = sFirstName;
    this.sLastName = sLastName;
}

// OutputBanner
/// <summary>
/// Вывод заголовка перед выводом информации о студенте
/// </summary>
public static void OutputBanner()
{
    Console.WriteLine("Никаких хитростей:");
    // Console.WriteLine(? какой объект используется ?);
}

// OutputBannerAndName
/// <summary>
/// Вывод заголовка с последующим выводом имени студента
/// </summary>
public void OutputBannerAndName()
{
    // Используется класс Student, но статическому методу
    // не передаются никакие объекты
    OutputBanner();
    // Явная передача объекта
    OutputName(this, 5);
}

// OutputName
/// <summary>
/// Выводит имя студента на консоль
/// </summary>
/// <param name="student">Студент, имя которого должно
/// быть выведено</param>
/// <param name="nIndent">К^Н4ес^Т^В^О пробелов в
/// отступе</param>
/// <returns>Вывод строка</returns>
public static string OutputName(Student student,
                                int nIndent)

```



```

{
    // Здесь объект указан явно
    string s = new String(' ', nlindent);
    s += String.Format("Имя студента - {0}",
                       student.ToNameString());
    Console.WriteLine(s);
    return s;
}

// ToNameString
/// <summary>
/// Преобразует имя студента в строку для вывода
/// </summary>
/// <returns>СТРОКУ с именем студента</returns>
public string ToNameString()
{
    // Здесь текущий объект указан неявно; можно
    // использовать this:
    // return this.sFirstName + " " + this.sLastName;
    return sFirstName + " " + sLastName;
}
}

/// <summary>
/// Класс, использующий класс Student
/// </summary>
public class Program
{
    /// <summary>
    /// Стартовая точка программы.
    /// </summary>
    /// <param name="args">Аргументы командной
    /// строки</param>
    public static void Main(string[] args)
    {
        Student student = new Student();
        student.InitStudent("Madeleine", "Cather");

        // Вывод заголовка и имени
        Student.OutputBanner();
        string s = Student.OutputName(student, 5);
        Console.WriteLine();

        // Вывод заголовка и имени еще раз
        student.OutputBannerAndName();

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");

        Console.Read();
    }
}
}

```

Комментарии непосредственно предшествуют описываемой ими функции. Они поясняют предназначение ее и каждого из ее аргументов, тип возвращаемых данных и ссыл-

ки на связанные функции. Вот что происходит на практике при добавлении вызова `Student.OutputName()` в функцию `Main()`.

1. Visual Studio предлагает список функций. После того как вы выберете `OutputName()`, Visual Studio выводит краткое описание из раздела `<summary>` `</summary>`, как показано на рис. 8.6. Текст появляется чуть ниже функции в желтом окне автозавершения.
2. После того как вы выберете функцию или введете ее имя, Visual Studio выводит описание первого параметра, которое получает из раздела `<param /param>`. Это описание появляется на том же месте, что и предыдущее, замещая собой описание функции.
3. Затем Visual Studio повторяет процесс для второго аргумента, `indent`.

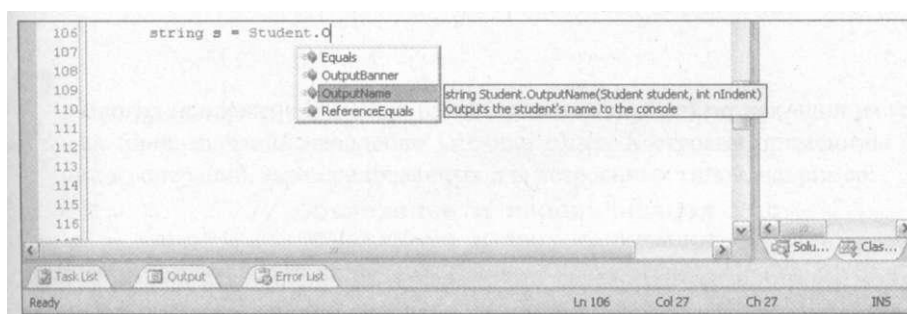


Рис. 8.6. Visual Studio может описывать пользовательские функции и их аргументы с применением XML

Конечно, вводить такого рода комментарии — занятие скучное, но зато потом ваша работа по использованию своих же классов и методов существенно облегчится.



Visual Studio можно настроить так, чтобы документирующие комментарии не отображались на экране.

Генерация XML-документации

Можно легко заставить Visual Studio вывести всю документацию в виде XML-файла.



Все, сказанное в данном разделе, — специфическая техническая информация. Если вы не знаете, что такое XML-файл, можно просто пропустить этот раздел, не читая.

Для генерации XML-документации выполните следующие шаги.

1. Выберите команду меню **Project^Имя проекта Properties**.
2. В разделе **Build** прокрутите раздел **Output** и найдите свойство под названием **XML Documentation File**. Отметьте соответствующий флаг и введите имя файла.
3. Сохраните и закройте вкладку **Properties**.



К свойствам проекта можно обратиться, щелкнув правой кнопкой мыши имени проекта в Solution Explorer.

4. Теперь выберите команду меню **Build: Rebuild Solution** для полной сборки проекта, независимо от того, требуется она или нет.
5. Просмотрите подкаталог `bin\Debug` проекта `MixingFunctionsAndMethodsWithXMLTags` (или другой подкаталог, который был указан вами для XML-документации).

В этом подкаталоге должен находиться файл с указанным вами в п. 2 имени, в котором содержится описание всех функций, документированных посредством XML-дескрипторов.

Работа со строками в С#

В этой главе...

- > Основные операции со строками
- } Разбор считанной строки •
- > Форматирование выводимых строк



В многих приложениях можно рассматривать тип `string` как один из встроенных типов-значений наподобие `int` или `char`. К строкам применимы некоторые из операций, зарезервированных для встроенных типов, например:

```
int i = 1;           // Объявление и инициализация int
string s = "abc";    // Объявление и инициализация string
```

С другой стороны, как видно из приведенного далее фрагмента, строки можно рассматривать как пользовательский класс:

```
string si = new String();
string s2 = "abcd";
int nLengthOfString = s2.Length;
```

Так что же такое `string` — тип-значение или класс? На самом деле `String` — это класс, который в силу его широкой распространенности С# трактует специальным образом. Например, ключевое слово `string` представляет собой синоним имени класса `String`, как видно из следующего фрагмента исходного текста:

```
String si = "abcd"; // Присваивание строкового литерала
                  // объекту класса String
string s2 = si;     // Присваивание объекта класса String
                  // строковой переменной
```

В этом примере переменная `si` объявлена как объект класса `String` (обратите внимание на прописную `S` в начале имени), в то время как `s2` объявлена как просто `string` (сострочной `s` в начале имени). Однако эти два присваивания демонстрируют, что `string` и `String` — это одинаковые (или совместимые) типы.



В действительности то же самое справедливо и для других встроенных типов. Даже тип `int` имеет соответствующий класс `Int32`, `double` — класс `Double` и т.д.; отличие в том, что `string` и `String` — это действительно одно и то же.

Основные операции над строками

Имеется ряд операций над строками, без которых не обходится практически никакая программа на C#. Например, почти во всех программах используется оператор "сложения" строк, показанный в следующем примере:

```
string sName = "Randy";  
Console.WriteLine("Его имя - " + sName); // Конкатенация
```

Этот специальный оператор обеспечивается классом `String`. Однако класс `String` предоставляет и другие методы для работы со строками. Их полный список можно найти в разделе "`String class`" предметного указателя справочной системы.

Объединение неразделимо!

Вам нужно запомнить одну до сих пор неизвестную вам вещь: после того как объект `string` создан, изменить его нельзя. Несмотря на то что можно говорить о модификации строки, в C# нет операции, модифицирующей реальный объект `string`. Внешне создается впечатление, что масса операторов модифицируют объекты `string`, с которыми работают, но это не так — они всегда возвращают модифицированную строку как новый объект `string`.

Например, операция `"Его имя - " + "Randy"` не изменяет ни одну из объединяемых строк, а генерирует новую строку `"Его имя - Randy"`. Одним из побочных эффектов такого поведения является то, что вы не должны беспокоиться, что кто-то изменит строку без вашего ведома.



Рассмотрим следующую демонстрационную программу:

```
// ModifyString - методы класса String не модифицируют сам  
// объект (s.ToUpperO не изменяет строку s; вместо этого он  
// возвращает новую преобразованную строку)  
using System;  
  
namespace ModifyString  
{  
    class Program  
    {  
        public static void Main(string[] args)  
        {  
            // Создание объекта Student  
            Student si = new Student();  
            si.sName = "Jenny";  
            // Создаем новый объект с тем же именем  
            Student s2 = new Student();  
            s2.sName = si.sName;  
  
            // "Изменение" имени объекта si не изменяет сам  
            // объект, поскольку ToUpperO возвращает новую
```

```

// строку, не влияя на оригинал
s2.sName = si.sName.ToUpper();

Console.WriteLine("si - {0}, s2 - {1}",
                  s1.sName,
                  s2.sName);

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
                  "завершения программы...");

Console.Read();

}

```

II Student - простейший класс, содержащий строку

```

class Student
{
public String sName;
}
}

```

Объекты класса Student si и s2 созданы так, что их члены sName указывают на одни и те же строковые данные. Вызов метода ToUpper() преобразует строку si.sName, ~~измени~~ все ее символы на прописные. Никаких проблем, связанных с тем, что si и s2 указывают на один объект, не возникает, поскольку метод ToUpper() не изменяет sName, а создает новую строку, записанную прописными буквами.

Вот что выводит на экран приведенная выше программа:

SI - Jenny, s2 - JENNY

Нажмите <Enter> для завершения программы...

Это свойство строк называется **неизменностью** (или неизменяемостью).



Неизменность строк очень важна для строковых констант. Строка наподобие "Это строка" представляет собой вид строковой константы, как 1 представляет собой константу типа int. Компилятор, таким образом, может заменить все обращения к одинаковым константным строкам обращением к одной константной строке, что снижает размер получающейся программы. Такое поведение компилятора было бы невозможным, если бы строки могли изменяться.

Сравнение строк

Множество операций рассматривает строку как единый объект— например, метод Compare(), который сравнивает две строки, вводя для них отношение "меньше-больше".



- Если левая строка **больше** правой, Compare() возвращает 1.
- Если левая строка **меньше** правой, Compare() возвращает -1.
- Если строки равны, Compare() возвращает 0.

Вот как выглядит алгоритм работы Compare(), записанный с использованием псевдокода

```

compare(string s1, string s2)
{

```

```

// Циклический проход по всем символам строк, пока один из
// символов одной строки не окажется больше
// соответствующего ему символа второй строки
foreach (для каждого) символа более короткой строки
    if (числовое значение символа строки s1 > числового
        значения символа строки s2)
        return 1
    if (числовое значение символа строки s1 < числового
        значения символа строки s2)
        return -1
// Если все символы совпали, но строка s1 длиннее строки
// s2, то она считается больше строки s2
if В строке s1 остались символы
    return 1
// Если все символы совпали, но строка s2 длиннее строки
// s1, то она считается больше строки s1
if В строке s2 остались символы
    return -1
// Если все символы строк попарно одинаковы, и строки
// имеют одну и ту же длину — то это одинаковые строки
return 0
}

```

Таким образом, "abed" больше "abbd", а "abede" больше "abed". Как правило, в реальных ситуациях интересует не то, какая из строк больше, а равны ли две строки друг другу или нет



Сравнение строк "больше-меньше" необходимо для их сортировки.

Операция Compare () возвращает значение 0, если две строки идентичны. В приведенной далее тестовой программе это значение используется для выполнения ряда операций, когда программа встречает некоторую строку или строки.



Программа BuildASentence предлагает пользователю ввести несколько строк текста. Эти строки объединяются с предыдущими для построения единого предложения. Программа завершает работу, если пользователь вводит слово EXIT, exit, QUIT или quit.

```

// BuildASentence - данная программа конструирует
// предложение путем конкатенации пользовательского ввода,
// до тех пор пока пользователь не введет команду
// завершения. Эта программа демонстрирует использование
// проверки равенства строк
using System;
namespace BuildASentence
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Каждая введенная вами строка "
                              "будет добавляться в предложение, "

```

```

        "пока вы не введете EXIT or QUIT");
// Запрашиваем пользовательский ввод и соединяем
// вводимые пользователем фразы в единое целое, пока
// не будет введена команда завершения работы
string sSentence = "";
for(;;)
{
    // Получение очередной строки
    Console.WriteLine("Введите строку");
    string sLine = Console.ReadLine();
    // Выход при команде завершения
    if (IsTerminateString(sLine))
    {
        break;
    }
    // В противном случае добавление введенного к строке
    sSentence = String.Concat(sSentence, sLine);
    // Обратная связь
    Console.WriteLine("\nВы ввели: {0}", sSentence);
}
Console.WriteLine("\nПолучилось:\n{0J", sSentence);
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы. ...");
Console.Read();
}

// IsTerminateString - возвращает true, если введенная
// строка представляет собой команду завершения работы
public static bool IsTerminateString(string source)
{
    string[] sTerms = { "EXIT", "exit", "QUIT", "quit" };
    // Сравниваем введенную строку с командами завершения
    // работы
    foreach (string sTerm in sTerms)
    {
        // Возвращаем true при совпадении
        if (String.Compare(source, sTerm) == 0)
        {
            return true;
        }
    }
    return false;
}

```

После краткого описания своих действий программа создает пустую строку для предложения — sSentence, после чего входит в "бесконечный" цикл.



Конструкции while (true) и for(;;) представляют собой бесконечные циклы, выход из которых осуществляется посредством операторов break (выход из цикла) или return (выход из программы). Эти два цикла эквивалентны, и оба встречаются в реальных программах. О циклах подробно рассказывается в главе 5, "Управление потоком выполнения".

Далее программа предлагает пользователю ввести строку текста, которую затем считывает с помощью метода `ReadLine()`. После прочтения строки программа проверяет, не является ли введенная строка командой завершения работы, пользуясь для этого функцией `IsTerminateString()`. Эта функция возвращает `true`, если переданная строка — одна из команд завершения работы, и `false` в противном случае.



По соглашению имя функции, проверяющей некоторое свойство и возвращающей значение `true` или `false`, начинается с `Is`, `Has`, `Can` или иного подобного слова. Само собой, это соглашение предназначено исключительно для людей, для облегчения понимания и удобочитаемости программы — C# без различия, какое имя используется для функции.

Если введенная строка не является командой завершения работы, она добавляется в конец предложения посредством функции `String.Concat()`. Полученный результат тут же выводится на экран, чтобы пользователь мог видеть, что у него получается.

Метод `IsTerminateString()` определяет массив строк `sTerms`. Каждый член этого массива представляет собой один из вариантов команды завершения работы. Любая из перечисленных в массиве строк, будучи введенной пользователем, приводит к завершению работы программы.



В массив включены как строка `"EXIT"`, так и строка `"exit"`, поскольку функция `Compare()` по умолчанию рассматривает эти строки как различные (так же, как и другие варианты написания этого слова, такие как `"exit"` или `"Exit"`, не будут восприняты программой в качестве команд завершения).

Функция `IsTerminateString()` циклически просматривает все элементы массива команд завершения работы и сравнивает их с переданной строкой. Если функция `Compare()` сообщает о соответствии строки одному из образцов команд завершения, функция тут же возвращает значение `true`; если же до завершения цикла соответствие не найдено, то по выходе из цикла функция возвращает значение `false`.



Итерации по массиву — отличный способ проверки на равенство одному из возможных значений.

Вот пример вывода программы `BuildASentence`.

Каждая введенная вами строка будет добавляться в предложение, пока вы не введете `EXIT` or `QUIT`
Введите строку

Программирование на C#

Вы ввели: Программирование на C#

Введите строку

- сплошное удовольствие

Вы ввели: Программирование на C# - сплошное удовольствие

Введите строку

EXIT

Получилось:

Программирование на C# - сплошное удовольствие
Нажмите <Enter> для завершения программы...

Пользовательский ввод здесь выделен полужирным шрифтом.

Сравнение без учета регистра

Метод `Compare()`, использованный в функции `IsTerminateString()`, рассматривает строки "EXIT" и "exit" как различные. Однако имеется перегруженная версия функции `Compare()`, которой передается три аргумента. Третий аргумент этой функции указывает, следует ли при сравнении игнорировать регистр букв (значение `true`) или нет (значение `false`). О перегрузке функций рассказывалось в главе 7, "Функции функций".

Следующая версия функции `IsTerminateString()` возвращает значение `true`, какими бы буквами не была введена команда завершения.

```
// IsTerminateString - возвращает значение true, если
// исходная строка соответствует одной из команд завершения
// программы
public static bool IsTerminateString(string source)
{
    • // Проверяет, равна ли переданная строка строкам exit
      // или quit, независимо от регистра используемых букв
      return (String.Compare("exit", source, true) == 0)
             (String.Compare("quit", source, true) == 0);
}
```

Эта версия функции `IsTerminateString()` проще предыдущей версии с использованием цикла. Ей не надо заботиться о регистре символов, и она может обойтись всего лишь двумя условными выражениями, так как ей достаточно рассмотреть только два варианта команды завершения программы.



Обратите внимание, что приведенная версия функции `IsTerminateString()` не применяет оператор `if`. Вычисленное значение логического выражения просто непосредственно передается пользователю, что позволяет избежать применения `if`.

Использование конструкции switch

Мне не нравится этот способ, но вы можете использовать конструкцию `switch` для поиска действий для конкретной строки. Обычно `switch` применяется для сравнения значения переменной с некоторым набором возможных значений, однако эту конструкцию можно применять и для объектов `string`. Вот как выглядит версия функции `IsTerminateString()` с использованием конструкции `switch`.

```
//IsTerminateString - возвращает значение true, если
// исходная строка соответствует одной из команд завершения
// программы
public static bool IsTerminateString(string source)
{
    switch (source)
    {
        case "EXIT":
        case "exit":

```

```

        case "QUIT":
        case "quit":
            return true;
    }
    return false;
}

```

Такой подход работает постольку, поскольку выполняется сравнение только ~~пред~~ определенного ограниченного количества строк. Цикл `for ()` представляет собой ~~сущ~~ ственно более гибкий подход, а применение функции `Compare ()`, нечувствительно! к регистру, существенно повышает возможности программы по "пониманию" ~~введенно~~ го пользователем.

Считывание ввода пользователя

Программа может считывать ввод пользователя по одному символу, но тогда вы ~~ами~~ должны заботиться о символах новой строки и прочих деталях. Более простым подходом оказывается считывание строки с ее последующим разбором.



Ваша программа может считывать строки, как если бы это были массива символов, с использованием оператора `foreach` или оператора индекса `[]`. Приведенный ниже исходный текст программы `StringToCharAccess` демонстрирует данную методику.

```

// StringToCharAccess - обращение к символам строки, как
// если бы строка была массивом
using System;

namespace StringToCharAccess
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Считывание строки с клавиатуры
            Console.WriteLine("Введите произвольную " +
                             "строку символов.");
            string sRandom = Console.ReadLine();
            Console.WriteLine();
            // Выводим строку
            Console.WriteLine("Вывод строки: " + sRandom);
            Console.WriteLine();

            // Выводим ее как последовательность символов
            Console.Write("Вывод с использованием foreach: " );
            foreach(char c in sRandom)
            {
                Console.Write(c);
            }

            Console.WriteLine(); // Завершение строки

            // Пустая строка-разделитель

```

```

Console.WriteLine();

// Вывод строки как последовательности символов
Console.Write("Вывод с использованием for: " );
for(int i = 0; i < sRandom.Length; i++)
{
    Console.Write(sRandom[i]);
}
Console.WriteLine(); // Завершение строки

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();

```



Эта программа выводит некоторую совершенно случайную строку, вводимую пользователем. Сначала строка выводится обычным способом с применением метода `WriteLine(string)`. Затем для вывода строки используется цикл `foreach`, который выводит ее символы по одному. И наконец, для той же цели применяется цикл `for` наряду с оператором индекса `[]`. В результате на экране получается примерно следующее:

Введите произвольную строку символов.

Вывод строки: djn dfv ckexfqyfz cnhjrf

Вывод с использованием `foreach`: djn dfv ckexfqyfz cnhjrf

Вывод с использованием `for`: djn dfv ckexfqyfz cnhjrf

Нажмите <Enter> для завершения программы...

Зачастую требуется убрать все пробельные символы с обоих концов строки (под термином **пробельный символ** (white space) подразумеваются символы, обычно не отображаемые на экране, например, пробел, символ новой строки или табуляции). Для этого можно воспользоваться методом `Trim()`:

```

// Удаляем пробельные символы с концов строки
sRandom = sRandom.Trim();

```



`String.Trim()` возвращает новую строку. Применяя этот метод так, как показано во фрагменте исходного текста выше, первоначальная версия строки с пробельными символами оказывается потерянной и больше не используется.

Разбор числового ввода

Функция `ReadLine()` используется для считывания объекта типа `string`. Программа, которая ожидает числовой ввод, должна эту строку соответствующим образом преобразовать в числа. C# предоставляет программисту класс `Convert` со всем необходимым для этого инструментарием, в частности, методами для преобразования строки в каждый из встроенных числовых типов. Так, следующий фрагмент исходного текста считывает число с клавиатуры и сохраняет его в переменной типа `int`.

```
string s = Console.ReadLine(); // Данные вводятся как строка,
int n = Convert.ToInt32(s);    // и преобразуются в число
```

Другие методы для преобразования еще более очевидны: `.ToDouble()`, `ToFloat()`, `ToBoolean()`.



Метод `ToInt32()` выполняет преобразование в 32-битовое знаковое целое число (вспомните, что 32 бита — это размер обычного `int`), так что эта функция выполняет преобразование строки в число типа `int`; для преобразования строки в число типа `long` используется функция `ToInt64()`.

Когда функции преобразования попадает «неправильный» символ, она может выдать некорректный результат, так что вам следует убедиться, что строка содержит именно те данные, которые ожидаются.

Приведенная далее функция возвращает значение `true`, если переданная ей строка состоит только из цифр. Такая функция может быть вызвана перед функцией преобразования строки в целое число, поскольку число может состоять только из цифр.



Вообще-то для чисел с плавающей точкой в строке может быть эта самая `T` (точка), а кроме того, перед числом может находиться знак минус — но сейчас интерес представляют не эти частности, а сама идея.

Итак, вот эта функция:

```
// IsAllDigits - возвращает true, если все символы строки
// являются цифрами
public static bool IsAllDigits(string sRaw)
{
    // Убираем все лишнее с концов строки. Если при этом в
    // строке ничего не остается — значит, эта строка не
    // представляет собой число
    string s = sRaw.Trim(); // Игнорируем пробельные символы
    if (s.Length == 0)
    {
        return false;
    }
    // Циклически проходим по всем символам строки
    for(int index = 0; index < s.Length; index++)
    {
        // Наличие в строке символа, не являющегося цифрой,
        // говорит о том, что это не число
        if (Char.IsDigit(s[index]) == false)
        {
            return false;
        }
    }
    // Все в порядке: строка состоит только из цифр
    return true;
}
```

Функция `IsAllDigits` сначала удаляет все ненужные пробельные символы с обоих концов строки. Если после этого строка оказывается пуста — значит, она состояла целиком из пробельных символов и числом не является. Если строка остается не пустой, функция проходит по всем ее символам. Если какой-то из символов оказывается не цифрой,

той, функция возвращает `false`, указывая, что переданная ей строка не является числом. Возврат функцией значения `true` означает, что все символы строки — цифры, так что строка, по всей видимости, представляет собой некоторое числовое значение.



Следующая демонстрационная программа считывает вводимое пользователем число и выводит его на экран:

```
// IsAllDigits - демонстрационная программа, иллюстрирующая
// применение функции IsAllDigits
using System;

namespace IsAllDigits
{
    class Program
    {
        public static void Main(string[] args)
        {
            // Ввод строки с клавиатуры
            Console.WriteLine("Введите целое число");
            string s = Console.ReadLine();
            // Проверка, может ли эта строка быть числом
            if (!IsAllDigits(s))
            {
                Console.WriteLine("Это не число!");
            }
            else
            {
                // Преобразование строки в целое число
                int n = Int32.Parse(s);
                // Выводим число, умноженное на 2
                Console.WriteLine("2 * {0} = {1}", n, 2 * n);
            }
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                "завершения программы...");
            Console.Read();
        }
        // IsAllDigits - возвращает true, если все символы
        // строки
        // являются цифрами
        public static bool IsAllDigits(string sRaw)
        {
            // Тело функции было рассмотрено ранее и здесь оно для
            // краткости опущено
        }
    }
}
```



Программа считывает строку, вводимую пользователем с клавиатуры, после чего выполняет ее проверку с помощью функции `IsAllDigits`. Если функция **возвращает `false`**, программа выводит пользователю предупреждающее сообщение. Если же нет,

программа преобразует строку в число с помощью функции `Int32.Parse()`, код представляет собой альтернативу `Convert.ToInt32()`. И наконец, программа выводит полученное число и его удвоенное значение (что должно доказывать корректность преобразования строки в число).

Вот как выглядит пример вывода рассматриваемой программы:

Введите целое число

1A3

Это не число!

Нажмите <Enter> для завершения программы...



Можно просто попытаться использовать функцию класса `Convert` для выполнения преобразования строки в число, и обработать возможные исключения, генерируемые функцией преобразования. Однако имеется немалая вероятность, что при этом функция не сгенерирует исключения, а вернет некорректный результат — например, в приведенном выше примере с вводом в качестве числа 1A3 вернет значение 1.

Обработка последовательности чисел

Зачастую программы получают в качестве вводимых данных строку, состоящую из нескольких чисел. Воспользовавшись методом `String.Split()`, вы сможете разделить строку на несколько подстрок, по одной для каждого числа, и работать с ними отдельно.

Функция `Split()` преобразует единую строку в массив строк меньшего размера с применением указанного символа-разделителя. Например, если вы скажете функции `Split()`, что следует использовать в качестве разделителя запятую, строка "1,2,3" превратится в три строки — "1", "2" и "3".



Приведенная далее демонстрационная программа применяет метод `Split()` для ввода последовательности чисел для суммирования.

```
// ParseSequenceWithSplit - считывает последовательность
// разделенных запятыми чисел, разделяет ее на отдельные
// целые числа и суммирует их
namespace ParseSequenceWithSplit
{
    using System;

    class Program
    {
        public static void Main(string[] args)
        {
            // Приглашение пользователю ввести последовательность
            // целых чисел
            Console.WriteLine("Введите последовательность целых" +
                              " чисел, разделенных запятыми:");

            // Считывание строки текста
            string input = Console.ReadLine();
            Console.WriteLine();
        }
    }
}
```

```

// Преобразуем строку в отдельные подстроки с
// использованием в качестве символов-разделителей
// запятых и пробелов
char[] cDividers = {',', ' '};
string[] segments = input.Split(cDividers);

// Конвертируем каждую подстроку в число
int nSum = 0;
foreach(string s in segments)
{
    // (Пропускаем пустые подстроки)
    if (s.Length > 0)
    {
        // Пропускаем строки, не являющиеся числами
        if (IsAllDigits(s))
        {
            // Преобразуем строку в 32-битовое целое число
            int num = Int32.Parse(s);
            Console.WriteLine("Очередное число = {0}", num);

            // Добавляем полученное число в сумму
            nSum += num;
        }
    }
}

// Вывод суммы
Console.WriteLine("Сумма = {0}", nSum);

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();

// IsAllDigits - возвращает true, если все символы строки
// являются цифрами
public static bool IsAllDigits(string sRaw)
{
    // Убираем все лишнее с концов строки. Если при этом в
    // строке ничего не остается - значит, эта строка не
    // представляет собой число
    string s = sRaw.Trimf(); // Игнорируем пробельные символы
    if (s.Length == 0)
    {
        return false;
    }
    // Циклически проходим по всем символам строки
    for(int index = 0; index < s.Length; index++)
    {
        // Наличие в строке символа, не являющегося цифрой,
        // говорит о том, что это не число
        if (Char.IsDigit(s[index]) == false)
        {
            return false;
        }
    }
}

```



```

    }
    // Все в порядке: строка состоит только из цифр
    return true;
}
}
}

```

Программа `ParseSequenceWithSplit` начинает со считывания строки с клавиатуры. Затем методу `Split()` передается массив символов `cDividers`, представляющий собой символы-разделители, используемые при отделении отдельных чисел в строке.

Далее программа циклически проходит по всем "подмассивам", созданным функцией `Split()`, применяя для этой цели цикл `foreach`. Программа пропускает все подстроки нулевой длины, а для непустых строк вызывает функцию `IsAllDigits()` для того чтобы убедиться, что строка представляет собой число. Корректные строки преобразуются в целые числа и суммируются с аккумулятором `nSum`. Некорректные числа игнорируются (я предпочел не генерировать сообщения об ошибках).

Вот как выглядит типичный вывод данной программы:

Введите последовательность целых чисел, разделенных запятыми:

1, 2, а, 3, 4

Очередное число = 1

Очередное число = 2

Очередное число = 3

Очередное число = 4

Сумма = 10

Нажмите <Enter> для завершения программы...

Программа проходит по списку, рассматривая запятые, пробелы (или оба символа вместе) как разделительные. Она пропускает "число" а и выводит общую сумму корректных чисел, равную 10. В реальных программах, однако, вряд ли можно просто игнорировать некорректные числа, никак не сообщая об этом пользователю. При чтении во входном потоке любой программы "мусора" обычно требуется тем или иным способом привлечь к этому факту внимание пользователя.

Управление выводом программы

Управление выводом программы представляет собой важный аспект работы со строками. Подумайте сами: вывод программы — это именно то, что видит пользователь. Он имеет значения, насколько элегантна внутренняя логика и реализация программы — вряд ли впечатлит пользователя; куда важнее для него корректность и внешнее представление выводимых программой данных.

Класс `String` предоставляет программисту ряд методов для форматирования выводимой строки. В следующих разделах будут рассмотрены такие методы, как `Trim()`, `PadRight()`, `PadLeft()`, `Substring()` и `Concat()`.

Использование методов `Trim()` и `Pad()`

Методом `Trim()` можно воспользоваться для удаления ненужных пробельных символов с обоих концов строки. Обычно этот метод применяется для удаления пробелов при выравнивании выводимой строки.

Еще один распространенный метод, часто используемый при форматировании — функции `Pad`, которые добавляют к строке пробелы с тем, чтобы ее длина стала равной некоторому предопределенному значению. Например, так вы можете добавить к строке пробелы слева или справа, чтобы обеспечить выравнивание вывода по правому или левому краю.



В приведенной далее небольшой демонстрационной программе `AlignOutput` для выравнивания списка имен применяются обе упомянутые функции.

```
// AlignOutput - выравнивание множества строк для улучшения
// внешнего вида вывода программы
namespace AlignOutput
{
    using System;

    class Program
    {
        public static void Main(string[] args)
        {
            string[] names = {"Christa ",
                               "  Sarah",
                               "Jonathan",
                               "Sam",
                               " Schmekowitz "};

            // Вывод имен в том виде, в котором они получены
            Console.WriteLine("Имена имеют разную длину");

            foreach(string s in names)
            {
                Console.WriteLine("Имя '{0}' до обработки", s);
            }
            Console.WriteLine();

            II Выравниваем строки по левому краю и делаем их
            // равной длины
            string[] sAlignedNames = TrimAndPad(names);

            // Выводим окончательный результат на экран
            Console.WriteLine("Те же имена выровнены и имеют " +
                              "одинаковую длину");
            foreach(string s in sAlignedNames)

                Console.WriteLine("Имя ' {0} ' после обработки", s);
        }
        II Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
        Console.Read();
    }
}
```

```

// TrimAndPad - для данного массива строк удаляются
// пробелы с обеих сторон строки, после чего выполняется
// дополнение пробелов таким образом, чтобы все строки
// оказались выровнены с наибольшей строкой в массиве
public static string[] TrimAndPad(string[] strings)
{
    // Копируем исходный массив в массив, с которым будем
    // работать
    string[] stringsToAlign = new String[strings.Length];

    // Удаляем ненужные пробелы с обеих сторон каждой
    // строки
    for(int i = 0; i < stringsToAlign.Length; i++)
    {
        stringsToAlign[i] = strings[i].Trim();
    }

    // Находим наибольшую длину строки в массиве
    int nMaxLength = 0;
    foreach(string s in stringsToAlign)
    {
        if (s.Length > nMaxLength)
        {
            nMaxLength = s.Length;
        }
    }

    // Выравниваем все строки к длине самой длинной
    for(int i = 0; i < stringsToAlign.Length; i++)
    {
        stringsToAlign[i] =
            stringsToAlign[i].PadRight(nMaxLength + 1);
    }

    // Возвращаем результат вызывающей функции
    return stringsToAlign;
}
}
}

```

Демонстрационная программа `AlignOutput` определяет массив имен, которые ~~ши~~ разные выравнивание и длину (вы можете переписать программу так, чтобы эти имена ~~сип~~ вались с клавиатуры или из файла). Функция `Main()` сначала выводит эти имена на ~~зн~~ том виде, в котором они получены программой. Затем вызывается функция `TrimAndPad` существенно улучшающая внешний вид выводимых программой строк:

```

Имена имеют разную длину
Имя 'Christa ' до обработки
Имя ' Sarah' до обработки
Имя 'Jonathan' до обработки
Имя 'Sam' до обработки
Имя ' Schmekowitz ' до обработки

```

```

Те же имена выровнены и имеют одинаковую длину
Имя 'Christa      ' после обработки
Вид 'Sarah        ' после обработки
та 'Jonathan     ' после обработки
Имя 'Sam          ' после обработки
Имя 'Schmekowitz  ' после обработки
Нажмите <Enter> для завершения программы...

```

Метод `TrimAndPad()` начинает с создания копии переданного ему массива `strings`. В общем случае функция, работающая с переданными ей аргументами, должна вернуть новые модифицированные значения, а не изменять переданные ей.

`TrimAndPad()` начинается с цикла, вызывающего `Trim()` для каждого элемента массива, чтобы удалить лишние пробельные символы с обоих концов строки. Затем выполняется второй цикл, в котором происходит поиск самого длинного элемента массива. И наконец, в последнем цикле для элементов массива вызывается метод `PadRight()`, удлиняющий строки, делая их равными по длине.

Метод `PadRight(10)` увеличивает строку так, чтобы ее длина была как минимум 10 символов. Например, если длина исходной строки — 6 символов, то метод `PadRight(10)` добавит к ней справа еще 4 пробела.

Метод `TrimAndPad()` возвращает массив выровненных строк для вывода. Функция `Main()` проходит по полученному списку строк, выводя их на экран. Вот и все.

Использование функции конкатенации

Зачастую программисты сталкиваются с задачей разбивки строки или вставки некоторой подстроки в середину другой строки. Заменить один символ другим проще всего (помощью метода `Replace()`):

```

string s = "Danger NoSmoking";
a.Replace(s, ' ', '!')

```

Этот фрагмент исходного текста преобразует начальную строку в "Danger!NoSmoking".

Замена всех вхождений одного символа (в данном случае — пробела) другим (восклицательным знаком) особенно полезна при генерации списка элементов, разделенных запятыми для упрощения разбора. Однако более распространенный и сложный случай включает разбиение единой строки на подстроки, отдельную работу с каждой подстрокой с последующим объединением их в единую модифицированную строку.



Рассмотрим, например, функцию `RemoveSpecialChars()`, которая удаляет все встречающиеся специальные символы из передаваемой ей строки. Демонстрационная программа `RemoveWhiteSpace` использует функцию `RemoveSpecialChars()` для удаления из строки пробельных символов (пробелов, табуляций и символов новой строки).

```

// RemoveWhiteSpace - определение функции
// RemoveSpecialChars(), которая может удалять из
// передаваемой ей строки произвольный предопределенный
// набор символов. В данной демонстрационной программе
// функция используется для удаления из тестовой строки всех
// пробельных символов
namespace RemoveWhiteSpace

```

```

using System;

public class Program
{
    public static void Main(string[] args)
    {
        // Определение множества пробельных символов
        char[] cWhiteSpace = {' ', '\n', '\t'};
        // Начинаем работу со строкой, в которой имеются
        // пробельные символы
        string s = " this is a\nstring";
        Console.WriteLine("До:" + s);

        // Выводим строку с удаленными пробельными символами
        Console.WriteLine("После:" +
            RemoveSpecialChars(s, cWhiteSpace))

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");
        Console.Read();
    }

    // RemoveSpecialChars - удаляет из строки все указанные
    // символы
    public static string RemoveSpecialChars(string sInput,
        char[] cTargets)
    {
        // В sOutput будет содержаться возвращаемая строка
        string sOutput = sInput;

        // Начинаем поиск пробельных символов
        for(;;)
        {
            // Ищем позиции искомых символов; если таковых в
            // строке больше нет - выходим из цикла
            int nOffset = sOutput.IndexOfAny(cTargets);
            if (nOffset == -1)
            {
                break;
            }

            // Разбиваем строку на две части - до найденного
            // символа и после него
            string sBefore = sOutput.Substring(0, nOffset);
            string sAfter = sOutput.Substring(nOffset + 1);

            // и объединяем эти части, но уже без найденного
            // символа
            sOutput = String.Concat(sBefore, sAfter);
        }
    }
}

```

```
return sOutput;
```



Ключевой в этой демонстрационной программе является функция `RemoveSpecialChars()`. Она возвращает строку, которая представляет собой исходную строку, но с **указанными** вхождением всех символов, содержащихся в массиве `cTargets`. Чтобы **лучше** понять эту функцию, представьте, что ей передана строка "ab, cd, e", а массив **специальных** символов содержит единственный символ `,`.

Функция `RemoveSpecialChars()` начинается с входа в цикл, выход из которого **происходит** только тогда, когда в строке не останется ни одной запятой. Функция `IndexOfAny()` возвращает позицию первой найденной запятой (значение `-1` указывает, что ни одна запятая не найдена).

После первого вызова `IndexOfAny()` возвращает `2` (позиция `a` ¹ равна `0`, позиция `b` ¹ — `1`, а позиция `,` — `2`). Два следующих вызова функции разбивают строку на две **подстроки** в указанном месте. Вызов `Substring(0, 2)` создает подстроку, содержащую **два** символа, начиная с символа в позиции `0`, т.е. "ab". Второй вызов `Substring(3)` создает подстроку из символов с позиции `3` исходной строки и до ее **конца**, т.е. "cd, e" (`+1` в вызове позволяет пропустить найденную запятую). Затем функция `Concat()` объединяет эти подстроки вместе, создавая строку "abed, e".

Управление выполнением передается после этого в начало цикла. Очередная итерация **находит** запятую в позиции `4`, так что в результате получается строка "abede". Поскольку в ней нет ни одной запятой, возвращаемая при последнем проходе позиция равна `-1`.

Демонстрационная программа сначала выводит строку, содержащую пробельные символы, затем использует функцию `RemoveSpecialChars()` для их удаления и **выводит** получившуюся в результате строку:

```
lo: this is a
Wring
После: thisisastring
Зажмите <Enter> для завершения программы...
```

Использование функции SplitQ



В программе `RemoveWhiteSpace` было продемонстрировано применение методов `Concat()` и `IndexOf()`; однако использованный способ решения поставленной задачи не самый эффективный. Стоит только немного подумать, и можно получить существенно более эффективную функцию с использованием уже знакомой функции `Split()`. Соответствующая программа имеется на прилагаемом компакт-диске в каталоге `RemoveWhiteSpaceWithSplit`. Вот код функции `RemoveSpecialChars()` из этой программы.

```
// RemoveSpecialChars - удаляет из строки все указанные
// символы
public static string RemoveSpecialChars(string sInput,
                                         char[] cTargets)
```

```
// Разбиваем входную строку с использованием указанных
// символов в качестве разделителей
string[] sSubStrings = sInput.Split(cTargets);

// В sOutput будет содержаться возвращаемая строка
string sOutput = "";

// Цикл по всем подстрокам
foreach(string substring in sSubStrings)
{
    sOutput = String.Concat(sOutput, substring);
}
return sOutput;
}
```

В этой версии для разбиения входной строки на множество подстрок используется функция `Split()` с удаляемыми символами в качестве символов-разделителей, поскольку разделители не включаются в подстроки, создается эффект их удаления, логика гораздо проще и менее подвержена ошибкам при реализации.

Цикл `foreach` в этой версии функции собирает части строки в единое целое. В программы остается неизменным.

Форматирование строки

Класс `String` предоставляет в распоряжение программиста метод `Format()` форматирования вывода, в основном — чисел. В своей простейшей форме `Format` позволяет вставлять строки, числа, логические значения в середину формируемой строки. Рассмотрим, например, следующий вызов:

```
string myString =
    String.Format("{0} умножить на {1} равно {2}", 2, 3, 2*3);
```

Первый аргумент `Format()` — **форматная строка** (строка формата). Элементы в ней указывают, что *i*-ый аргумент, следующий за форматной строкой, должен вставлен в этой точке. `{0}` означает первый аргумент (в данном случае — 2), `{1}` второй (3) и так далее.

В приведенном фрагменте получившаяся строка присваивается переменной `myStr` и имеет следующий вид:

```
2 умножить на 3 равно 6
```

Пока не указано иное, функция `Format()` использует формат по умолчанию для **каждого** типа аргумента. Для указания формата вывода можно размещать в фигурных скобках к номеру аргумента дополнительные модификаторы, которые показаны в табл. 9.1.

Таблица 9.1. Модификаторы, используемые функцией `String.Format()`

Модификатор	Пример	Результат	Примечания
C — денежные единицы	{0:C} для 123.456	\$123.45	Символ валюты зависит от настроек региональных стандартов
	{0:C} для -123.456	(\$123.45)	
D — десятичное число	{0:D5} для 123	00123	Только для целых чисел

Модификатор	Пример	Результат	Примечания
E — число в экспоненциальной форме	{0:E} для 123.45	1.2345E+002	Известна также как научная запись
F — число с плавающей точкой	{0:F2} для 123.4567	123.45	Число после F указывает количество цифр после десятичной точки
N — число	{0:N} для 123456.789	123,456.79	Добавляет запятые и округляет число до ближайших сотых
	{0:N1} для 123456.789	123,456.8	Указывает количество цифр после десятичной точки
	{0:N0} для 123456.789	123,457	
X — шестнадцатеричное число	{0:X} для 123	0x7B	Шестнадцатеричное число 7B равно десятичному числу 123. Применяется только для целых чисел
{0:0...}	{0:000.00} для 12.3	012.30	Вносит 0 там, где нет реальных цифр
{0:#...}	{0:###.##} для 12.3	12.3	Вносит пробелы; полезно при выравнивании по десятичной точке
	{0:##0.0#} для 0	0.0	Комбинация 0 и # заставляет вносить пробелы на местах # и обеспечивает наличие как минимум одной цифры, даже если число равно 0.
{0:# или 0%}	{0:#00.##} для .1234	12.3%	% заставляет выводить число как проценты (умножая на 100 и добавляя символ %)
	{0:#00.##} для .0234	02.3%	



Все эти модификаторы могут показаться слишком запутанными, но вы всегда можете получить информацию о них в справочной системе С#. Чтобы увидеть модификаторы в действии, взгляните на приведенную далее демонстрационную программу `OutputFormatControls`, позволяющую ввести не только число с плавающей точкой, но и модификатор формата, который будет использован при выводе введенного числа обратно на экран.

```
//OutputFormatControls - позволяет пользователю посмотреть,
// как влияют модификаторы форматирования на вывод чисел.
//Модификаторы вводятся в программу так же, как и числа - в
// процессе работы программы
namespace OutputFormatControls
{
    using System;

    public class Program
    {
        public static void Main(string[] args)
        {
            // Бесконечный цикл для ввода чисел, пока пользователь
            // не введет вместо числа пустую строку, что является
            // сигналом к окончанию работы программы
        }
    }
}
```



```

for(;;)
{
    // Ввод числа и выход из цикла, если введена пустая]
    // строка
    Console.WriteLine("Введите число с плавающей точкой")
    string sNumber = Console.ReadLine();
    if (sNumber.Length == 0)
    {
        break;
    }
    double dNumber = Double.Parse(sNumber);

    // Ввод модификаторов форматирования, разделенных
    // пробелами
    Console.WriteLine("Введите модификаторы " +
        "форматирования, разделенные " + ,
        "пробелами");
    char[] separator = { ' ' };
    string sFormatString = Console.ReadLine();
    string[] sFormats = sFormatString.Split(separator);

    // Цикл по введенным модификаторам
    foreach(string s in sFormats)
    {
        if (s.Length != 0)
        {
            // Создание управляющего элемента форматирования!
            // из введенного модификатора
            string sFormatCommand = "{0:" + s + "}";
            // Вывод числа с применением созданного
            // управляющего элемента форматирования
            Console.Write(
                "Модификатор {0} дает ", sFormatCommand);
            try
            {
                Console.WriteLine(sFormatCommand, dNumber);
            }
            catch(Exception)
            {
                Console.WriteLine("<Неверный модификатор>");
            }
            Console.WriteLine();
        }
    }
}
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}
}
}

```

Программа `OutputFormatControls` считывает вводимые пользователем числа с плавающей точкой в переменную `dNumber` до тех пор, пока не будет введена пустая строка — это является признаком окончания ввода. Обратите внимание, что программа не выполняет никаких тестов для проверки корректности введенного числа с плавающей точкой. Программа считает пользователя достаточно интеллектуальным и знающим, как вводить числа с плавающей точкой (довольно смелое допущение!).

Затем программа считывает ряд модификаторов форматирования, разделенных пробелами. Каждый из них далее комбинируется со строкой `{0}` в переменной `sFormatCommand`. Например, если вы ввели `N4`, программа создаст управляющий элемент `{0:N4}`. После чего введенное пользователем число выводится на экран с применением этого элемента:

```
Console.WriteLine(sFormatCommand, dNumber);
```

В рассмотренном только что случае модификатора `N4` команда по сути превращается в

```
tasole.WriteLine("{0:N4}", dNumber);
```

как выглядит типичный вывод программы на экран (полужирным шрифтом выделен ввод пользователя):

Введите число с плавающей точкой

1234 5.6789

Введите модификаторы форматирования, разделенные пробелами

CE F1 N0 0000000.00000

Модификатор `{0:C}` дает \$12,345.68

Модификатор `{0:E}` дает 1.234568E+004

Модификатор `{0:F1}` дает 12345.7

Модификатор `{0:N0}` дает 12,346

Модификатор `{0:0000000.00000}` дает 0012345.67890

Введите число с плавающей точкой

.12345

Введите модификаторы форматирования, разделенные пробелами

00.0%

Модификатор `{0:00.0%}` дает 12.3%

Введите число с плавающей точкой

Нажмите <Enter> для завершения программы...

Будучи примененным к числу 12345.6789, модификатор `N0` добавляет в нужное место ноль (часть N) и убирает все цифры после десятичной точки (часть 0), что дает строку, 346 (последняя цифра — результат округления, а не отбрасывания).

Аналогично, будучи примененным к числу 0.12345, модификатор `00.0%` даст 12.31. Этот % приводит к умножению числа на 100 и добавлению символа % к выводимому числу. `00.0` указывает, что в выводимой строке должно быть по меньшей мере две цифры слева от десятичной точки, и только одна — справа. Если тот же модификатор применить к числу 0.01, будет выведена строка `01.0%`.



Непонятная конструкция `try...catch` предназначена для перехвата всех потенциальных ошибок при вводе некорректных чисел. Однако об этом рассказывается совсем в другой главе.

Часть IV

Объектно-ориентированное программирование



В этой части...

Объектно-ориентированное программирование — термин, вызывающий у программистов наибольший выброс адреналина в кровь. Так, объектно-ориентированным языком программирования является C++ — и в этом его главное отличие от старого доброго C. К объектно-ориентированным языкам определенно относится и Java, как и еще добрая сотня языков, придуманных за последний десяток лет. Но что же это такое — *объектно-ориентированный*? Зачем это надо? и надо ли вообще? стоит ли использовать это в своих программах?

В этой части вы столкнетесь с возможностями C#, которые делают его объектно-ориентированным языком программирования. Объектно-ориентированное программирование — это не просто работа с объектами; это — мощь и гибкость программ при меньших затрачиваемых усилиях, это элегантность проекта, словом — это все, изложенное в данной части книги.

Глава 10

Что такое объектно-ориентированное программирование

В этой главе...

- > Основы объектно-ориентированного программирования
- > Абстракция и классификация
- > Важность объектно-ориентированного программирования

В

этой главе будут даны ответы на два основных вопроса — какие концепции лежат в основе объектно-ориентированного программирования и чем они отличаются от уже рассмотренных концепций функционального программирования.

Объектно-ориентированная концепция №1 — абстракция

Когда мы с сыном смотрим футбол, я подчас испытываю непреодолимую тягу к вредным для здоровья, но таким вкусным кулинарным изыскам, в частности, к мексиканским блюдам. Достаточно бросить на тарелку чипсы, бобы, сыр, приправы и пять минут зажарить эту массу в микроволновой печи.

Для того чтобы воспользоваться печью, следует открыть ее дверцу, поместить внутрь полуфабрикат и нажать несколько кнопок на передней панели. Через пару минут блюдо готово (только не стойте перед печью, а то ваши глаза начнут светиться в темноте).

Обратите внимание на то, чего *не делалось* при использовании микроволновой печи.

✓ Ничего не переключалось и не изменялось внутри печи. Чтобы установить для нее рабочий режим, существует интерфейс — лицевая панель с кнопками и небольшой индикатор времени; это все, что нужно.

✓ Не перепрограммировался процессор внутри печи, даже если прошлый раз готовилось абсолютно другое блюдо.

✓ Не было необходимости смотреть внутрь печи.

✓ При приготовлении блюд не надо было беспокоиться о внутреннем устройстве печи — даже если вы работаете главным инженером по производству таких печей.



Это не просто пространные рассуждения. В повседневной жизни нас постоянно преследуют стрессы. Чтобы уменьшить их число, мы начинаем обращать внимание только на события определенного масштаба. В объектно-ориентированном программировании уровень детализации, на котором вы работаете, называется **уровнем абстракции**. И объяснить этот термин можно на примере **абстрагирования** от подробностей внутреннего устройства микроволновой печи.

К счастью, ученые-кибернетики открыли объектную ориентированность и ряд других концепций, снижающих уровень сложности, с которым должен работать программно. Использование абстракций делает программирование более простым и уменьшает количество возможных ошибок. Именно в этом направлении как минимум полстолетия движется прогресс в программировании — работа со все более сложными концепциями а все меньшим количеством ошибок.

Во время приготовления блюда многие смотрят на микроволновую печь просто как на железный ящик. И пока ею можно управлять с помощью интерфейса, ее нельзя сжечь, "подвесить" или, что еще хуже, превратить готовящееся блюдо в уголь.

Приготовление блюд с помощью функций

Представьте себе, что я попросил бы своего сына написать алгоритм приготовления мною закусок. Поняв наконец, чего я от него добиваюсь, он бы, наверное, написал что-то вроде "открыть банку бобов, натереть сыр, посыпать перцем" и т.д. Когда дело дошло бы непосредственно до приготовления в печи, он в лучшем случае написал бы нечто типа "готовить в микроволновой печи не более пяти минут".

Этот рецепт прост и верен. Но с помощью подобного алгоритма "функциональный" программист не сможет написать программу приготовления закусок. Программисты, работающие с функциями, живут в мире, лишенном таких объектов, как микроволновая печь и прочие удобства. Они заботятся о последовательности операций в функциях. В "функциональном" решении проблемы закусок управление будет передано от пальцев рук кнопкам передней панели, а затем внутрь печи. После этого программе придется решать, какое время должна работать печь и когда следует включить звуковой сигнал готовности.

При таком подходе очень трудно отвлечься от сложностей внутреннего устройства печи. В этом мире нет объектов, за которые можно спрятать всю присущую микроволновой печи сложность.

Приготовление "объектно-ориентированных" блюд

Применяя объектно-ориентированный подход к приготовлению блюд, я первым делом определяю объекты, используемые в задаче: сыр, бобы, чипсы и микроволновая печь. После этого я начинаю моделировать их в программе, не задумываясь над деталями их применения.

При этом я работаю (и думаю) на уровне базовых объектов. Главное, думать о том, что приготовить блюдо, не волнуясь о деталях работы микроволновой печи — над этим уже подумали ее создатели (которым абсолютно нет дела до ваших кулинарных пристрастий).

После создания и проверки всех необходимых объектов можно переключиться на следующий уровень абстракции, т.е. мыслить на уровне процесса приготовления закуски не отвлекаясь на отдельные куски сыра или банку бобов. При таком подходе я легко перевидал рецепт моего сына на язык C#.

Объектно-ориентированная концепция №2 — классификация

В концепции уровней абстракции очень важной частью является классификация. Опять-таки, если бы я спросил моего сына: "Что такое микроволновая печь?" — он бы наверняка ответил: "Это печь, которая...". Если бы затем последовал вопрос: "А что такое печь?" — он бы ответил что-то вроде: "Ну, это кухонный прибор, который...". (При попытке выяснить у него, что же такое кухонный прибор, он наверняка бы спросил, сколько можно задавать дурацких вопросов.)

Из детских ответов становится ясно, что ими печь воспринимается как один из экземпляров вещей, называемых микроволновыми печами. Кроме того, печь является подклассом духовок, а духовки относятся к типу кухонных приборов.



В объектно-ориентированном программировании конкретная микроволновая печь является **экземпляром** класса микроволновых печей. Класс микроволновых печей является подклассом печей, который, в свою очередь, является подклассом кухонных приборов.

Люди склонны заниматься классификацией. Все вокруг увешано ярлыками. Мы делаем все для того, чтобы уменьшить количество вещей, которые надо запомнить. Вспомните, например, когда вы первый раз увидели "Пежо" или "Рено". Возможно, в рекламе и говорилось, что это суперавтомобиль, но мы-то с вами знаем, что это не так. Это ведь просто машина. Она имеет все свойства, которыми обладает автомобиль. У нее есть руль, колеса, сиденья, мотор, тормоза и т.д. И можно поспорить, что многие водили бы такую штуку без всяких инструкций.

Но не будем тратить место в книге на описание того, чем этот автомобиль похож на другие. Следует знать лишь то, что это "машина, которая...", и то, чем она отличается от других машин (например, ценой). Теперь можно двигаться дальше. Легковые автомобили являются таким же подклассом колесных транспортных средств, как грузовики и пикапы. При этом колесные транспортные средства входят в состав транспортных средств наравне с кораблями и самолетами.

Зачем нужна классификация

Зачем вообще нужна эта классификация, это объектно-ориентированное программирование? Ведь оно влечет за собой массу трудностей. Тем более, что уже имеется готовый механизм функций. Зачем же что-то менять?

Иногда может показаться, что легче разработать и создать микроволновую печь специально для какого-то блюда и не строить универсальный прибор на все случаи жизни. Тогда на лицевую панель не нужно было бы помещать никаких кнопок, кроме кнопки СТАРТ. Блюдо всегда готовилось бы одинаковое время, и можно было бы избавиться от всех этих бесполезных кнопок типа РАЗМОРОЗКА или ТЕМПЕРАТУРА ПРИГОТОВЛЕНИЯ. Все, что требовалось бы от такой печи, — это чтобы в нее помещалась одна тарелка с полуфабрикатом. Да, но что же тогда получится? Ведь при этом кубический фут пространства использовался бы для приготовления всего одной тарелки закуски!

Чтобы сэкономить место, можно освободиться от этой глупой концепции — "микрон| новая печь". Для приготовления закуски хватит и внутренностей печи. Тогда в инструкции достаточно написать примерно следующее: "Поместите полуфабрикат в ящик. Соедини красный и черный провод. Установите на трубе излучателя напряжение в 3000 вольт. Дол появиться негромкий гул. Постарайтесь не стоять близко к установке, если хотите иметь ц|тей". Простая и понятная инструкция!

Но такой функциональный подход создает некоторые проблемы.

- ✓ **Слишком сложно.** Нежелательно, чтобы фрагменты микроволновой печи па мешивались с фрагментами закуски при разработке программы. Но поскольку **ц** данном подходе нельзя создавать объекты и упрощать написание, работая с и| цым из них в отдельности, приходится держать в голове все нюансы каждого о| екта одновременно.
- ✓ **Не гибко.** Когда-нибудь потребуются замена имеющейся микроволновой **печ**и печь другого типа. Это делается без проблем, если интерфейс печи можно б|д оставить старым. Без четко очерченных областей действия, а также без разделен| интерфейса и внутреннего содержимого становится крайне трудно убрать стар| объект и поставить на его место новый.
- ✓ **Невозможно использовать повторно.** Печи предназначены для пригото|влен| разных блюд. Вряд ли кому-то захочется создавать новую печь всякий раз при а| обходимости приготовить новое блюдо. Если задача уже решена, неплохо ищи| зовать ее решение и в других программах.

Объектно-ориентированная концепция №3 — удобный интерфейс

Объект должен быть способен спроектировать внешний интерфейс максимально **пр**с| стым при полной достаточности для корректного функционирования. Если интер|фа| устройства будет недостаточен, все кончится битьем кулаком или чем-то более тяже|ла по верхней панели такого устройства или просто разборкой для того, чтобы добраться| его внутренностей (что наверняка окажется нарушением законодательства об интел|ла туальной собственности). С другой стороны, если интерфейс слишком сложен, **вс**ы сомнительно, что кто-то купит такое устройство (как минимум, вряд ли кто-то будет Н| пользоваться все предоставляемые интерфейсом возможности).

Люди постоянно жалуются на сложность видеомagniтофонов (впрочем, с перехо|ду на управление с помощью экрана количество жалоб несколько уменьшилось). В э|к устройствах слишком много кнопок с различными функциями. Зачастую одна и та | кнопка выполняет разные функции — в зависимости от того, в каком именно состояи| находится в этот момент видеомagniтофон. Кроме того, похоже, невозможно найти **я** видеомagniтофона различных марок с одинаковым интерфейсом.

Теперь рассмотрим ситуацию с автомобилями. Вряд ли можно сказать (и доказат|ь что автомобиль проще видеомagniтофона. Однако, похоже, люди не испытывают та| трудностей с его вождением, как с управлением видеомagniтофоном.

В каждом автомобиле в наличии примерно одни и те же элементы управления и га|ти на одних и тех же местах. Если же управление отличается... Ну вот вам реальная ж| тория из моей жизни — у моей сестры был французский автомобиль, в котором упрас|

ние фарами оказалось там, где у всех "нормальных" автомобилей находится управление сигналами поворота. Отличие вроде бы небольшое, но я так и не научился поворачивать на этом автомобиле влево, не выключив при этом фары...

Кроме того, при хорошо продуманном дизайне автомобиля один и тот же элемент управления никогда не будет использоваться для выполнения более одной операции в зависимости от состояния автомобиля.

Объектно-ориентированная концепция №4 — управление доступом

Микроволновая печь должна быть сконструирована таким образом, чтобы никакая комбинация кнопок или рукояток не могла ее сломать или навредить вам. Конечно, определенные комбинации не будут выполнять никаких функций, но главное, чтобы ни одна из них не привела к следующему.

- ✓ **Поломке устройства.** Какие бы рукоятки ни крутил ваш ребенок и какие бы кнопки не нажимал — микроволновая печь не должна от этого сломаться. После того как вы вернете все элементы управления в корректное состояние, она должна нормально работать.
- ✓ **К пожару или прочей порче имущества или нанесению вреда здоровью потребителя.** Мы живем в сутяжном мире, и если бы что-то похожее могло произойти — компании пришлось бы продать все вплоть до автомобиля ее президента, чтобы рассчитаться с подающими на нее в суд и адвокатами.

Однако чтобы эти два правила выполнялись, вы должны принять на себя определенную ответственность. Вы ни в коем случае не должны вносить изменения в устройство, в частности, отключать блокировки.

Почти все кухонное оборудование любой степени сложности, включая микроволновые печи, имеет пломбы, препятствующие проникновению пользователя внутрь. Если ~~так~~ пломба повреждена, это указывает, что крышка устройства была снята, и вся ответственность с производителя тем самым снимается. Если вы каким-либо образом изменили внутреннее устройство печи, вы сами несете ответственность за все последующие неприятности, которые могут произойти.

Аналогично, класс должен иметь возможность контролировать доступ к своим членам-данным. Никакая последовательность вызовов членов класса не должна приводить программу к аварийному завершению, однако класс не в состоянии гарантировать это, если внешние объекты имеют доступ к внутреннему состоянию класса. Класс должен иметь возможность ~~принять~~ критические члены-данные и делать их недоступными для внешнего мира.

Поддержка объектно-ориентированных концепций в C#

Итак, как же C# реализует объектно-ориентированное программирование? Впрочем, это не совсем корректный вопрос. C# является объектно-ориентированным языком про-

граммирования, но не реализует его — это делает программист. Как и на любом другом языке, вы можете написать на C# программу, не являющуюся объектно-ориентированной (например, вставив весь код Word в функцию Main()). Иногда нужно писать и такие программы, но все же главное предназначение C# — создание объектно-ориентированных программ.

C# предоставляет программисту следующие необходимые для написания объектно-ориентированных программ возможности.

- ✓ **Управляемый доступ.** C# управляет обращением к членам класса. Ключевые слова C# позволяют объявить некоторые члены открытыми для всех, а другие защищенными или закрытыми. Подробнее эти вопросы рассматриваются в главе 11, "Классы".
- ✓ **Специализация.** C# поддерживает специализацию посредством механизма, известного как *наследование классов*. Один класс при этом наследует члены другого класса. Например, вы можете создать класс Car, как частный случай класса Vehicle. Подробнее эти вопросы рассматриваются в главе 12, "Наследование".
- ✓ **Полиморфизм.** Эта возможность позволяет объекту выполнить операцию так, как это требуется для его корректного функционирования. Например, класс Rocket унаследованный от Vehicle, может реализовать операцию Start совершенно иначе, чем Car, унаследованный от того же Vehicle. По крайней мере, будем надеяться, что это справедливо хотя бы по отношению к вашему автомобилю хотя с некоторыми автомобилями никогда ни в чем нельзя быть уверенным...].просыг полиморфизма рассматриваются в главах 13, "Полиморфизм", и 14, "Интерфейсы и структуры".

Классы

В этой главе...

- У Защита класса посредством управления доступом
- > Инициализация объекта с помощью конструктора
- > Определение нескольких конструкторов в одном классе
- > Конструирование статических членов и членов класса



Класс должен сам отвечать за свои действия. Так же как микроволновая печь не должна вспыхнуть, объята пламенем, из-за неверного нажатия кнопки, так и класс не должен скончаться (или прикончить программу) при предоставлении некорректных данных.

Чтобы нести ответственность за свои действия, класс должен убедиться в корректности своего начального состояния и в дальнейшем управлять им так, чтобы оно всегда оставалось корректным. C# предоставляет для этого все необходимое.

Ограничение доступа к членам класса

Простые классы определяют все свои члены как `public`. Рассмотрим программу `BankAccount`, которая поддерживает член-данные `balance` для хранения информации о балансе каждого счета. Сделав этот член `public`, вы допускаете любого в святая святых банка, позволяя каждому самому указывать сумму на счету.

Неизвестно, в каком банке храните свои сбережения вы, но мой банк и близко не настолько открыт и всегда строго следит за моим счетом, самостоятельно регистрируя каждое снятие денег со счета и вклад на счет. В конце концов, это позволяет уберечься от всяких недоразумений, если вас вдруг подведет память.



Управление доступом дает возможность избежать больших и малых ошибок в работе банка. Обычно программисты, привыкшие к функциональному программированию, говорят, что достаточно лишь определить правило, согласно которому никакие другие классы не должны обращаться к члену `balance` непосредственно. Увы, теоретически это, может быть, и так, но на практике такой подход никогда не работает. Да, программисты начинают работу, будучи переполненными благими намерениями, которые вскоре непонятно куда исчезают под давлением сроков сдачи проекта...

Пример программы с использованием открытых членов



В приведенной демонстрационной программе класс `BankAccount` объявляет все методы как `public`, в то же время члены-данные `nAccountNumber` и `dBalance` сделаны `private`. Эта демонстрационная программа корректна и не будет компилироваться, так как создана исключительно в дидактических целях.

```
// BankAccount - создание банковского счета с использованием
// переменной типа double для хранения баланса счета (она
// объявлена как private, чтобы скрыть баланс от внешнего
// мира)
// Примечание: пока в программу не будут внесены
// исправления, она не будет компилироваться, так как
// функция Main() обращается к private-члену класса
// BankAccount.
using System;

namespace BankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("В текущем состоянии эта " +
                              "программа не компилируется.");
            // Открытие банковского счета
            Console.WriteLine("Создание объекта " +
                              "банковского счета");
            BankAccount ba = new BankAccount();
            ba.InitBankAccount();
            // Обращение к балансу при помощи метода Deposit()
            // вполне корректно; Deposit() имеет право доступа ко
            // всем членам-данным
            ba.Deposit(10);

            // Непосредственное обращение к члену-данным вызывает
            // ошибку компиляции
            Console.WriteLine("Здесь вы получите " +
                              "ошибку компиляции");
            ba.dBalance += 10;

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }

    // BankAccount - определение класса, представляющего
    // простейший банковский счет
    public class BankAccount
```

```

private static int nNextAccountNumber = 1000;
private int nAccountNumber;

// хранение баланса в виде одной переменной типа double
private double dBalance;

// Init - инициализация банковского счета с нулевым
// балансом и использованием очередного глобального
// номера
public void InitBankAccount()
{
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0.0;

// GetBalance - получение текущего баланса
public double GetBalance()
{
    return dBalance;

// Номер счета
public int GetAccountNumber()
{
    return nAccountNumber;
}

public void SetAccountNumber(int nAccountNumber)
{
    this.nAccountNumber = nAccountNumber;

// Deposit - позволен любой положительный вклад
public void Deposit(double dAmount)
{
    if (dAmount > 0.0)
    {
        dBalance += dAmount;

// Withdraw - вы можете снять со счета любую сумму, не
// превышающую баланс; функция возвращает реально снятую
// сумму
public double Withdraw(double dWithdrawal)
{
    if (dBalance <= dWithdrawal)
    {
        dWithdrawal = dBalance;

        dBalance -= dWithdrawal;
        return dWithdrawal;

```

```

{
private static int nNextAccountNumber = 1000;
private int nAccountNumber;

// хранение баланса в виде одной переменной типа double
private double dBalance;

// Init - инициализация банковского счета с нулевым
// балансом и использованием очередного глобального
// номера
public void InitBankAccount()
{
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0.0;
}

// GetBalance - получение текущего баланса
public double GetBalance()

    return dBalance;

// Номер счета
public int GetAccountNumber()

    return nAccountNumber;

public void SetAccountNumber(int nAccountNumber)

    this.nAccountNumber = nAccountNumber;

// Deposit - позволен любой положительный вклад
public void Deposit(double dAmount)
{
    if (dAmount > 0.0)
    {
        dBalance += dAmount;
    }
}

// Withdraw - вы можете снять со счета любую сумму, не
// превышающую баланс; функция возвращает реально снятую
// сумму
public double Withdraw(double dWithdrawal)
{
    if (dBalance <= dWithdrawal)
    {
        dWithdrawal = dBalance;
    }

    dBalance -= dWithdrawal;
    return dWithdrawal;
}
}

```

```
// GetString - возвращает информацию о состоянии счета в
// -виде строки
public string GetString()
{
    string s = String.Format("#{0} = {1:C}",
                               GetAccountNumber(),
                               GetBalance());
    return s;
}
```



В этом коде выражение `dBalance -= dWithdrawal` означает то же, что и `dBalance = dBalance - dWithdrawal`. Обычно программисты на C# стараются использовать наиболее короткую запись из возможных.

Объявляя член как `public`, вы делаете его доступным для любого кода вашей программы.

Класс `BankAccount` предоставляет метод `InitBankAccount()` для инициализации членов класса, метод `Deposit()` — для обработки вкладов на счет и метод `Withdraw()` — для снятия денег со счета. Методы `Deposit()` и `Withdraw()` даже печивают выполнение некоторых рудиментарных правил — "нельзя вкладывать отрицательные суммы" и "нельзя снимать больше, чем есть на счету". Однако в открытой! системе, где член-данные `dBalance` доступен для внешних методов (под **внешними** подразумеваются методы "в пределах той же программы, но внешние по отношению! к классу"), эти правила могут быть нарушены кем угодно. Особенно существенной проблемой! это может оказаться при разработке больших проектов группами программистов! Это может стать проблемой и для одного человека, поскольку ему свойственно ошибаться! Хорошо спроектированный код с правилами, выполнение которых проверяет компилятор, значительно снижает количество источников возможных ошибок.

Перед тем как идти дальше, обратите внимание, что приведенная демонстрационная программа не будет компилироваться — при такой попытке вы получите сообщение о том, что обращение к члену `DoubleBankAccount.BankAccount.dBalance` невозможно:

```
'DoubleBankAccount.BankAccount.dBalance' is inaccessible
due to its protection level.
```

Трудно сказать, зачем компилятор заставили выводить такие скучные сообщения вместо короткого "нелезь к `private`", но суть именно в этом. Выражение `ba.dBalance += 10;` оказывается некорректным именно по этой причине — в силу объявления `dBalance` как `private` этот член недоступен виртуальной функции `Main()`, расположенной вне класса `BankAccount`. Замена данного выражения на `ba.Deposit(10)` решает возникшую проблему — метод `BankAccount.Deposit()` объявлен как `public`, а потому доступен для функции `Main()`.



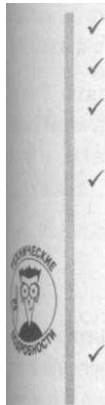
Тип доступа по умолчанию — `private`, так что если вы забыли или совершенно пропустили модификатор для некоторого члена — это аналогично тому, как если бы вы описали его как `private`. Однако настоятельно рекомендуется всегда использовать это ключевое слово явно во избежание любых недоразумений. Хороший программист всегда явно указывает свои намерения, что является еще одним методом снижения количества возможных ошибок.

Прочие уровни безопасности



В этом разделе используются определенные знания о наследовании и пространствах имен, которые будут рассмотрены в более поздних главах книги. Вы можете сейчас пропустить настоящий раздел и вернуться к нему позже, получив необходимые знания.

C# предоставляет следующие уровни безопасности.



- ✓ Члены, объявленные как `public`, доступны любому классу программы.
 - ✓ Члены, объявленные как `private`, доступны только из текущего класса.
 - ✓ Члены, объявленные как `protected`, доступны только из текущего класса и всех его подклассов.
 - ✓ Члены, объявленные как `internal`, доступны для любого класса в том же модуле программы.
- Модулем** в C# называется отдельно компилируемая часть кода, представляющая собой выполняемую `.EXE`-программу либо библиотеку `.DLL`. Одно пространство имен может распространяться на несколько модулей.
- ✓ Члены, объявленные как `internal protected`, доступны для текущего класса и всех его подклассов в том же модуле программы.

Скрытие членов путем объявления их как `private` обеспечивает максимальную степень безопасности. Однако зачастую такая высокая степень и не нужна. В конце концов, члены подклассов и так зависят от членов базового класса, так что ключевое слово `protected` предоставляет достаточно удобный уровень безопасности.

Зачем нужно управление доступом

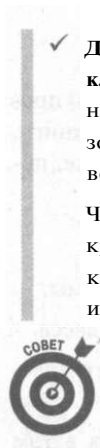
I Объявление внутренних членов класса как `public` — не лучшая мысль как минимум по следующим причинам.



✓ **Объявляя члены-данные `public`, вы не в состоянии просто определить, когда и как они модифицируются.** Зачем беспокоиться и создавать методы `Deposit()` и `Withdraw()` с проверками корректности? И вообще, зачем создавать любые методы — ведь любой метод любого класса может модифицировать данные счета в любой момент. Но если другая функция может обращаться к этим данным, то она практически обязательно это сделает.

Ваша программа `BankAccount` может проработать длительное время, прежде чем вы заметите, что баланс одного из счетов — отрицателен. Метод `Withdraw()` призван оградить от подобной ситуации, но в описанном случае непосредственный доступ к балансу, минуя метод `Withdrawn`, имеют и другие функции. Вычислить, какие именно функции и при каких условиях поступают так некорректно — задача не из легких.

✓ **Доступ ко всем членам-данным класса делает его интерфейс слишком сложным.** Как программист, использующий класс `BankAccount`, вы не хотите знать о том, что делается внутри него. Вам достаточно знаний о том, как положить деньги на счет и снять их с него.



✓ **Доступ ко всем членам-данным класса приводит к "растеканию" права класса.** Например, класс `BankAccount` не позволяет балансу стать отрицательным ни при каких условиях. Это — бизнес-правило, которое должно быть локализовано в методе `Withdraw()`. В противном случае вам придется добавлять шиветствующую проверку в весь код, в котором осуществляется изменение баланса. Что произойдет, когда банк решит изменить правила, и часть клиентов с хорошей кредитной историей получит право на небольшой отрицательный баланс в течение короткого времени? Вам придется долго рыскать по всей программе и вносить изменения во все места, где выполняется непосредственное обращение к балансу.

Не делайте классы и методы более доступными, чем это необходимо. Эту параноидальную боязнь хакеров — это просто поможет вам снизить количество ошибок в коде. По возможности используйте модификатор `private`, а затем при необходимости поднимайте его до `protected`, `internal`, `internal protected` или `public`.

Методы доступа

Если вы более внимательно посмотрите на класс `BankAccount`, то увидите несколько других методов. Один из них, `GetString()`, возвращает строковую версию ссч для вывода ее на экран посредством функции `Console.WriteLine()`. Дело в том, что вывод содержимого объекта `BankAccount` может быть затруднен, если это содержимое недоступно. К тому же, следуя принципу "отдайте кесарю кесарево", класс должен иметь право сам решать, как он будет представлен при выводе.

Кроме того, имеется один метод для получения значения — `GetBalance()` и набор методов для получения и установки значения — `GetAccountNumber()` и `SetAccountNumber()`. Вы можете удивиться — зачем так волноваться из-за того, что член `dBalance` был объявлен как `private`, и при этом предоставлять метод `GetBalance()`? На самом деле для этого имеются достаточно веские основания.

✓ **`GetBalance()` не дает возможности изменять член `dBalance` — он только возвращает его значение.** Тем самым значение баланса делается доступным только для чтения. Используя аналогию с настоящим банком, вы можете просмотреть состояние своего счета в любой момент, но не можете снять с него деньги иначе, чем с применением процедур, предусмотренных для этого банком.

✓ **Метод `GetBalance()` скрывает внутренний формат класса от внешних методов.** Метод `GetBalance()` может в процессе работы выполнять некоторые вычисления, обращаться к базе данных банка — словом, выполнять какие-то действия, чтобы получить состояние счета. Внешние функции ничего об этом не знают и не должны знать. Продолжая аналогию, вы интересуетесь состоянием счета, но не знаете, как, где и в каком именно виде хранятся ваши деньги.

И наконец, метод `GetBalance()` предоставляет механизм для внесения внутренних изменений в класс `BankAccount`, абсолютно не затрагивая при этом его пользователя! Если от нацбанка придет распоряжение хранить деньги как-то иначе, это никак не должно сказаться на вашем способе обращения с вашим счетом (по крайней мере так должно быть в цивилизованном обществе).

Пример управления доступом



Приведенная далее демонстрационная программа DoubleBankAccount указывает потенциальные изъяны программы BankAccount.

```
// DoubleBankAccount - создание банковского счета с
// использованием переменной типа double для хранения
// баланса счета (она объявлена как private, чтобы скрыть
// баланс от внешнего мира)
using System;

namespace DoubleBankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Открытие банковского счета
            Console.WriteLine("Создание объекта " +
                              "банковского счета");
            BankAccount ba = new BankAccount();
            ba.InitBankAccount();
            // Вклад на счет
            double dDeposit = 123.454;
            Console.WriteLine("Вклад {0:C}", dDeposit);
            ba.Deposit(dDeposit);

            // Баланс счета
            Console.WriteLine("Счет = {0}", ba.GetString());

            // Вот где имеется неприятность
            double dAddition = 0.002;
            Console.WriteLine("Вклад {0:C}", dAddition);
            ba.Deposit(dAddition);

            // Результат
            Console.WriteLine("В результате счет = {0}",
                              ba.GetString());

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}

// BankAccount - определение класса, представляющего
// простейший банковский счет
public class BankAccount
{
    private static int nNextAccountNumber = 1000;
    private int nAccountNumber;
```

```

// хранение баланса в виде одной переменной типа double
private double dBalance;

// Init - инициализация банковского счета с нулевым
// балансом и использованием очередного глобального
// номера
public void InitBankAccount()
{
    nAccountNumber = ++nNextAccountNumber;
    dBalance = 0.0;
}

// GetBalance - получение текущего баланса
public double GetBalance()
{
    return dBalance;
}

// AccountNumber
public int GetAccountNumber()
{
    return nAccountNumber;
}

public void SetAccountNumber(int nAccountNumber)
{
    this.nAccountNumber = nAccountNumber;
}

// Deposit - позволен любой положительный вклад
public void Deposit(double dAmount)
{
    if (dAmount > 0.0)
    {
        dBalance += dAmount;
    }
}

// Withdraw - вы можете снять со счета любую сумму, не
// превышающую баланс; функция возвращает реально снятую
// сумму
public double Withdraw(double dWithdrawal)
{
    if (dBalance <= dWithdrawal)
    {
        dWithdrawal = dBalance;
    }
    dBalance -= dWithdrawal;
    return dWithdrawal;
}

// GetString - возвращает информацию о состоянии счета в
// виде строки
public string GetString()

```

```

{
    string s = String.Format("#{0} = {1:C}",
                              GetAccountNumber(),
                              GetBalance());

    return s;
}
}

```

Функция `Main()` создает банковский счет и вносит на него сумму 123.454 — т.е. сумму с дробным количеством копеек. Затем функция `Main()` вносит на счет еще одну долю копейки и выводит баланс счета.

Вывод программы выглядит следующим образом:

Создание объекта банковского счета

Вклад \$123 .45

Счет = #1001 = \$123 .45

Вклад \$0.00

В результате счет = #1001 = \$123.46

Нажмите <Enter> для завершения программы...

Пользователь начинает жаловаться на некорректные расчеты. Лично я ничего не имею против округления счета до ближайшей сотни сверху, но ведь не все клиенты такие покладистые... В общем, что греха таить — в программе действительно имеется ошибка.



Проблема, конечно, в том, что 123.454 выводится как 123.45. Чтобы избежать проблем, банк принимает решение округлять вклады и снятия до ближайшей копейки. Простейший путь осуществить это — конвертировать счета в `decimal` и использовать метод `Decimal.Round()`, как это сделано в демонстрационной программе `DecimalBankAccount`.

```

// DecimalBankAccount - создание банковского счета с
// использованием переменной типа decimal для хранения
// баланса счета
using System;

namespace DecimalBankAccount
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Открытие банковского счета
            Console.WriteLine("Создание объекта " +
                              "банковского счета");

            BankAccount ba = new BankAccount();
            ba.InitBankAccount();
            // Вклад на счет
            double dDeposit = 123.454;
            Console.WriteLine("Вклад {0:C}", dDeposit);
            ba.Deposit(dDeposit);

            // Баланс счета

```

```

        Console.WriteLine("Счет = {0}", ba.GetString());

        // Добавляем очень малую величину
        double dAddition = 0.002;
        Console.WriteLine("Вклад {0:C}", dAddition);
        ba.Deposit(dAddition);

        // Результат
        Console.WriteLine("В результате счет = {0}",
            ba.GetString());

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");
        Console.Read();
    }
}

// BankAccount - определение класса, представляющего
// простейший банковский счет
public class BankAccount
{
    private static int nNextAccountNumber = 1000;
    private int nAccountNumber;

    // хранение баланса в виде одной переменной типа decimal
    private decimal mBalance;

    // Init - инициализация банковского счета с нулевым
    // балансом и использованием очередного глобального
    // номера
    public void InitBankAccount()
    {
        nAccountNumber = ++nNextAccountNumber;
        mBalance = 0;
    }

    // GetBalance - получение текущего баланса
    public double GetBalance()
    {
        return (double)mBalance;
    }

    // AccountNumber
    public int GetAccountNumber()
    {
        return nAccountNumber;
    }

    public void SetAccountNumber(int nAccountNumber)
    {
        this.nAccountNumber = nAccountNumber;
    }
}

```

```

// Deposit - позволен любой положительный вклад
public void Deposit(double dAmount)
{
    if (dAmount > 0.0)
    {
        // Округление к ближайшей копейке перед внесением
        // вклада
        decimal mTemp = (decimal)dAmount;
        mTemp = Decimal.Round(mTemp, 2);

        mBalance += mTemp;
    }
}

// Withdraw - вы можете снять со счета любую сумму, не
// превышающую баланс; функция возвращает реально снятую
// сумму
public decimal Withdraw(decimal dWithdrawal)
{
    if (mBalance <= dWithdrawal)
    {
        dWithdrawal = mBalance;
    }
    mBalance -= dWithdrawal;
    return dWithdrawal;
}

// GetString - возвращает информацию о состоянии счета в
// виде строки
public string GetString()

    string s = String.Format("#{0} = {1:C}",
                               GetAccountNumber(),
                               GetBalance());

    return s;
}

```

Внутреннее представление поменялось на использование значений типа `decimal`, который в любом случае более подходит для работы с банковским счетом, чем тип `double`. Метод `Deposit()` теперь применяет функцию `Decimal.Round()` для округления вкладываемой суммы до ближайшей копейки. Вывод программы оказывается таким, как и ожидалось:

```

Создание объекта банковского счета
Вклад $123.45
Счет = #1001 = $123.45
Вклад $0.00
В результате счет = #1001 = $123.45
Нажмите <Enter> для завершения программы...

```

Выводы

Вы можете сказать, что нужно было с самого начала писать программу BankAccount с использованием decimal, и, пожалуй, с вами можно согласиться. Но в этом. Могут быть разные приложения и ситуации. Главное, что класс BankAccount оказался в состоянии решить проблему так, что не пришлось вносить никаких изменений в использующую его программу (обратите внимание, что открытый интерфейс не изменился: метод Balance() так и возвращает значение типа double).



Повторюсь еще раз: приложение, использующее класс BankAccount должно изменяться при изменении класса.

В данном случае единственной функцией, которую бы пришлось изменить при любом средственном обращении к балансу, является функция Main(), но в реальной программе могут иметься десятки таких функций, и они могут оказаться в не меньшем количестве модулей. В анализируемом примере ни одна из этих функций не требует внесения изменений, но если бы они непосредственно обращались ко внутренним членам класса было бы решительно невозможно.



Внесение внутренних изменений в класс требует определенного тестирования использующего класс кода несмотря на то, что в него не вносятся никакие модификации.

Определение свойств класса

Методы GetX() и SetX(), продемонстрированные в программе BankAccount называются **функциями доступа** (access functions). Хотя их использование теоретически является хорошей привычкой, на практике это зачастую приводит к грустным результатам. Судите сами — чтобы увеличить на 1 член nAccountNumber, требуется следующий код:

```
SetAccountNumber(GetAccountNumber() + 1);
```

C# имеет конструкцию, называемую **свойством** и делающую использование функции доступа существенно более простым. Приведенный далее фрагмент кода определяет свойство AccountNumber для чтения и записи:

```
public int AccountNumber { get; set; } // Скобки не нужны

get { return nAccountNumber; } // Фигурные скобки и точка с запятой
set { nAccountNumber = value; } // value - ключевое слово
```

Раздел get реализуется при чтении свойства, а set — при записи. В приведенном далее фрагменте исходного текста свойство Balance является свойством только для чтения, так как здесь определен только раздел get:

```
public double Balance
{
    get
    {
```

```
return (double)mBalance;
```

Использование свойств выглядит следующим образом:

```
BankAccount ba = new BankAccount ();  
// Записываем свойство AccountNumber  
ba.AccountNumber = 1001;  
// Считываем оба свойства  
Console.WriteLine ("#{ 0} = {1:C}", ba.AccountNumber,  
ba.Balance);
```

Свойства `AccountNumber` и `Balance` очень похожи на открытые члены-данные как внешне, так и в использовании. Однако свойства позволяют классу защитить свои внутренние члены (так, член `mBalance` остается при этом `private`). Обратите внимание, что `Balance` выполняет приведение типа — точно так же может производиться любое количество вычислений. Свойства вовсе не обязательно должны представлять собой одну строку кода.



По соглашению имена свойств начинаются с прописной буквы. Обратите также внимание, что свойства не имеют скобок: следует писать просто `Balance`, а не `Balance()`.



Свойства совсем не обязательно неэффективны. Компилятор C# может оптимизировать простую функцию доступа так, что она будет генерировать не больше машинных команд, чем непосредственное обращение к члену. Это важно не только для прикладных программ, но и для самого C#. Библиотека C# широко использует свойства, и то же должны делать и вы — даже для обращения к членам-данным класса из методов этого же класса.

Статические свойства

Статические члены-данные могут быть доступны через статические свойства, как показано в следующем простейшем примере:

```
public class BankAccount  
  
    private static int nNextAccountNumber = 1000;  
    public static int NextAccountNumber  
  
    get {return nNextAccountNumber;}
```

Свойство `NextAccountNumber` доступно посредством указания имени его класса, так как оно не является свойством конкретного объекта.

```
// Считываем свойство NextAccountNumber  
int nValue = BankAccount.NextAccountNumber;
```

Дополнительные действия свойств

Операция `get` может выполнять дополнительную работу помимо простого получения значения, связанного со свойством. Взгляните на следующий код:


```
public static int AccountNumber
{
    // Получение значения переменной и увеличение ее значения,
    // чтобы в следующий раз получить уже новое ее значение
    get{return ++nNextAccountNumber;}
}
```

Это свойство увеличивает статический член класса перед тем, как вернуть результат. Однако это не слишком умная идея, ведь пользователь ничего не знает о такой особенности и не подозревает, что происходит что-то помимо чтения значения. Увеличение переменной в данном случае представляет собой *побочное действие*.



Подобно функциям доступа, которые они имитируют, свойства не должны менять состояния класса иначе чем через установку значения соответствующего члена данных. В общем случае и свойства, и методы должны избегать побочных действий, так как это может привести к трудноуловимым ошибкам. Изменяйте класс настолько явно и непосредственно, насколько это возможно,

Конструирование объектов посредством конструкторов

Управление доступом — это только половина проблемы. Рождение объекта — один из самых важных этапов в его жизни. Класс, конечно, может предоставить метод до инициализации вновь созданного объекта, но беда в том, что приложение может попросту забыть его вызвать. В таком случае члены-данные класса окажутся заполнены "мусором", и корректной работы от такого объекта ждать не придется.

C# решает эту проблему путем вызова инициализирующей функции автоматически. Например, в строке

```
MyObject mo = new MyObject();
```

не только выделяется память для объекта, но и выполняется его инициализация посредством вызова специальной инициализирующей функции.



Не путайте термины **класс** и **объект**. Dog — это класс, но собака Scooter-I — это объект класса Dog.

Конструкторы, предоставляемые C#

C# хорошо умеет отслеживать инициализацию переменных и не позволяет исполнять неинициализированные переменные. Например, представленный далее код приведет к генерации ошибки компиляции:

```
public static void Main(string[] args)
{
    int n;
    double d;
    double dCalculatedValue = n + d;
}
```

C# отслеживает тот факт, что ни `n`, ни `d` не имеют присвоенного значения и не могут использоваться в выражении. Компиляция этой микропрограммы приводит к генерации следующих ошибок:

```
Use of unassigned local variable 'n'
Use of unassigned local variable 'd'
```

Однако C# предоставляет конструктор по умолчанию для объектов классов, который инициализирует члены-данные значением `0` для встроенных переменных, `false` — для логических и `null` — для ссылок. Рассмотрим следующую простую демонстрационную программу:

```
using System;
namespace Test
(
    public class Program
    {
        public static void Main(string[] args)
        {
            // Сначала создаем объект
            MyObject localObject = new MyObject();
            Console.WriteLine("localObject.n = {0}", localObject.n);
            if (localObject.nextObject == null)
            {
                Console.WriteLine("localObject.nextObject = null");
            }
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
    public class MyObject
    {
        internal int n;
        internal MyObject nextObject;
    }
}
```

Эта программа определяет класс `MyObject`, который содержит переменную `n` типа `int` и ссылку на объект `nextObject`, позволяющую создавать **связанные списки** объектов. Функция `Main()` создает объект класса `MyObject` и выводит начальное содержание его членов. Вывод этой программы имеет вид

```
localObject.n = 0
localObject.nextObject = null
Нажмите <Enter> для завершения программы...
```

C# при создании объекта выполняет небольшой код по инициализации объекта и его членов. Если бы не этот код, члены-данные `localObject.n` и `localObject.nextObject` содержали бы какие-то случайные значения, попросту говоря — "мусор".

[Сод, инициализирующий значения при создании, называется **конструктором**. Он "конструирует" класс в смысле инициализации его членов.

Конструктор по умолчанию

C# гарантирует, что объект начинает существование в определенном состоянии: полным нулем. Однако для многих классов (пожалуй, для подавляющего большинства) такое нулевое состояние не является корректным. Рассмотрим класс `BankAccount`, о котором уже шла речь ранее в этой главе,

```
public class BankAccount
{
    int nAccountNumber;
    double dBalance;
    //... другие члены
}
```

Хотя нулевое начальное значение баланса вполне корректно, нулевое значение номера счета определенно корректным не является.

Поэтому класс `BankAccount` включает метод `InitBankAccount()`, инициализирующий объект. Однако такой подход перекладывает слишком большую ответственность на прикладную программу, использующую данный класс. Если вдруг приложение забудет вызвать метод `InitBankAccount()`, то прочие методы банковского счета могут оказаться неработоспособными, хотя при этом и не будут содержать никаких ошибок. Класс не должен полагаться на внешние функции наподобие метода `InitBankAccount()`, которые должны обеспечивать корректное состояние его объектов.

Для решения данной проблемы класс предоставляет специальную функцию, автоматически вызываемую C# при создании объекта — **конструктор класса**. C# требует, чтобы конструктор носил то же имя, что и имя самого класса, так что конструктор класса `BankAccount` имеет следующий вид:

```
public void Main(string[] args)
{
    BankAccount ba = new BankAccount();
}

public class BankAccount
{
    // Номера банковских счетов начинаются с 1000 и
    // назначаются последовательно в возрастающем порядке
    static int nNextAccountNumber = 1000;
    // Для каждого счета поддерживаются его номер и баланс
    int nAccountNumber;
    double dBalance;
    // Конструктор BankAccount - обратите внимание на его имя
    public BankAccount() // Требуются круглые скобки, могут
                        // иметься аргументы, возвращаемый
                        // тип отсутствует

    {
        nAccountNumber = ++nNextAccountNumber;
        dBalance = 0.0;
    }
    //... прочие члены . . .
}
```

Содержимое конструктора `BankAccount` то же, что и у первоначального метода `InitBankAccount()`. Однако конструктор имеет некоторые особенности:

- ✓ он всегда имеет то же имя, что и сам класс;
- ✓ он не имеет возвращаемого типа, даже типа `void`;
- ✓ функция `Main()` не должна вызывать никаких дополнительных функций для инициализации объекта при его создании.

Создание объектов



Теперь посмотрим на конструкторы в деле. Для этого рассмотрим программу `DemonstrateDefaultConstructor`.

```
// DemonstrateDefaultConstructor - демонстрация работы
// конструкторов по умолчанию; создает класс с конструктором
// и рассматриваем несколько сценариев
using System;

namespace DemonstrateDefaultConstructor
{
    // MyObject - создание класса с "многословным"
    // конструктором и внутренним объектом
    public class MyObject
    {
        // Этот член-данные является свойством класса
        static MyOtherObject staticObj = new MyOtherObject();

        // Этот член-данные является свойством объекта
        MyOtherObject dynamicObj;

        // Конструктор (с обильным выводом на экран)
        public MyObject()
        {
            Console.WriteLine("Начало конструктора MyObject");
            Console.WriteLine(" (Статические члены-данные " +
                "конструируются до вызова этого " +
                "конструктора)11");
            Console.WriteLine("Теперь динамически создаем " +
                "нестатический член-данные:");
            dynamicObj = new MyOtherObject();
            Console.WriteLine("Завершение конструктора MyObject");
        }
    }

    // MyOtherObject - у этого класса тоже многословный
    // конструктор, но внутренние члены-данные отсутствуют
    public class MyOtherObject

    {
        public MyOtherObject ()
        {
            Console.WriteLine("Конструирование MyOtherObject");
        }
    }
}
```

```

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Начало функции Main()");
        Console.WriteLine("Создание локального объекта " +
                           "MyObject в Main():");
        MyObject localObject = new MyObject();
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");

        Console.Read();
    }
}
1 }
}

```

Выполнение данной программы приводит к следующему выводу на экран:

```

Начало функции Main()
Создание локального объекта MyObject в Main() :
Конструирование MyOtherObject
Начало конструктора MyObject
(Статические члены-данные конструируются до вызова
этого конструктора)
Теперь динамически создаем нестатический член-данные:
Конструирование MyOtherObject
Завершение конструктора MyObject
Нажмите <Enter> для завершения программы...

```

Вот реконструкция происходящего при запуске программы.

1. Программа начинает работу, и функция `Main()` выводит начальное сообщение о предстоящем создании локального объекта `MyObject`.
2. Функция `Main()` создает объект `localObject` типа `MyObject`.
3. `MyObject` содержит статический член `staticObj` класса `MyOtherObject`. В статические члены-данные создаются до первого запуска конструктора `MyObject`. В этом случае C# присваивает переменной `staticObj` ссылку на вновь созданный объект перед тем, как передать управление конструктору `MyObject`.
4. Конструктор `MyObject` получает управление. Он выводит начальное сообщение и напоминает, что статический член уже сконструирован до того, как начал работу конструктор `MyObject`.
5. После объявления о своих намерениях по динамическому созданию нестатического члена конструктор `MyObject` создает объект класса `MyOtherObject` использованием оператора `new`, что сопровождается выводом второго сообщения о создании `MyOtherObject` на экран.
6. Управление возвращается конструктору `MyObject`, который, в свою очередь, возвращает управление функции `Main()`.
7. Программа выполнена.

Выполнение конструктора в отладчике

Для того чтобы выполнить рассматриваемую программу в отладчике, произведите следующие действия.

1. Соберите программу с помощью команды меню **Build** о **Build DemonstrateDefaultConstructor**.
2. Перед тем как приступить к выполнению программы в отладчике, установите точку останова на вызове `Console.WriteLine` в конструкторе `MyOtherObject`.



Для установки точки останова щелкните на сером поле с левой стороны окна напротив строки, в которой хотите разместить точку останова. На рис. 11.1 показано окно отладки с точкой останова, о чем свидетельствует красная пиктограмма на серой полосе.

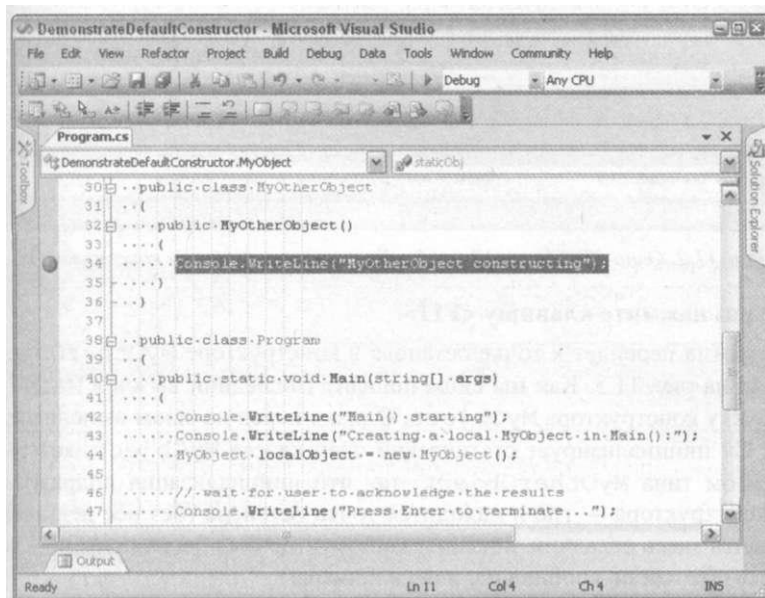


Рис. 11.1. Красная пиктограмма на серой полосе свидетельствует о наличии точки останова

3. Воспользуйтесь командой меню **Debug** ^ **Step Into** (или нажмите клавишу <F11>).

Ваши меню, полосы инструментов и окна должны немного измениться, а открывающая фигурная скобка функции `Main()` — оказаться выделенной желтым цветом фона.

4. Нажмите клавишу <F11> еще три раза и установите курсор мыши над переменной `localObject` (без щелчка).

Вы находитесь перед вызовом конструктора `MyObject`. Ваш экран должен выглядеть примерно так, как на рис. 11.2. На рисунке видно, что в настоящий мо-

мент объект `localObject` под курсором имеет значение `null`. То же показывает и окно **Locals**.

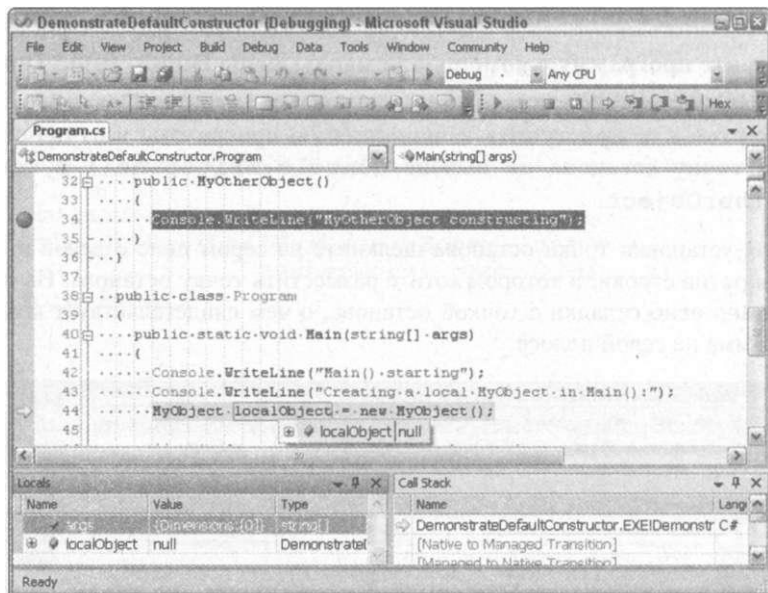


Рис. 11.2. Окно отладчика Visual Studio перед выполнением конструктора

5. Еще раз нажмите клавишу <F11>.

Программа перейдет к точке останова в конструкторе `MyOtherObject`, как показано на рис. 11.3. Как мы сюда попали? Последний вызов в `Main()` приводит к запуску конструктора `MyObject`. Однако перед началом выполнения конструктора C# инициализирует статический член класса `MyObject`, который является объектом типа `MyOtherObject`, так что инициализация подразумевает вызов его конструктора — где и находится точка останова (без нее нельзя было бы остановить здесь отладчик, хотя сам конструктор был бы выполнен — вы бы могли судить об этом по сообщению в окне консоли).

6. Дважды нажмите клавишу <F11>, после чего вы остановитесь на строке! статическим членом `staticObj`, как показано на рис. 11.4.

Это означает, что конструктор этого объекта завершил свою работу.

7. Продолжайте нажимать клавишу <F11> для пошагового выполнения программы.

При первом нажатии клавиши <F11> вы остановитесь в начале конструктора `MyObject`. Обратите внимание, что вы еще раз попадете в конструктор `OtherObject`, но на этот раз, когда конструктор `MyObject` будет создавать статический член `dynamicObj`.

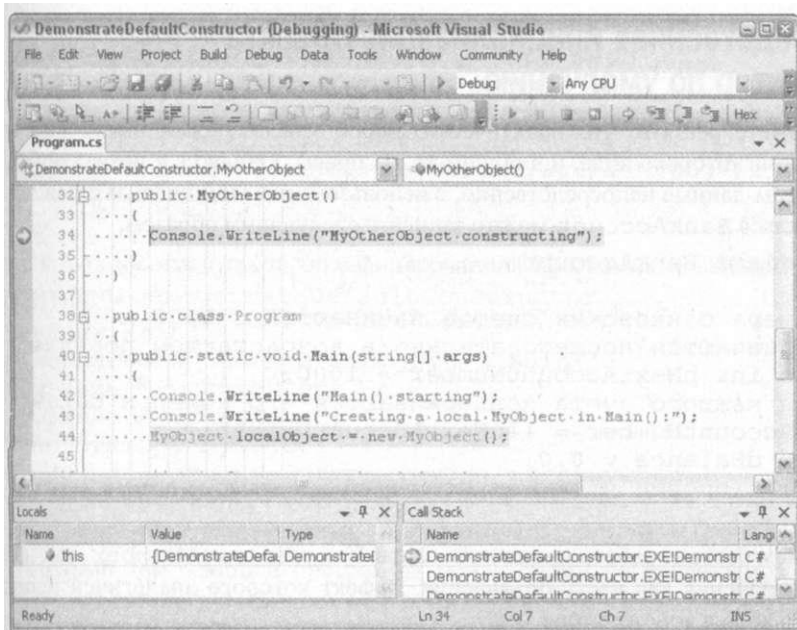


Рис. 11.3. Перед вызовом конструктора *MyObject* управление передается конструктору *MyOtherObj* ет

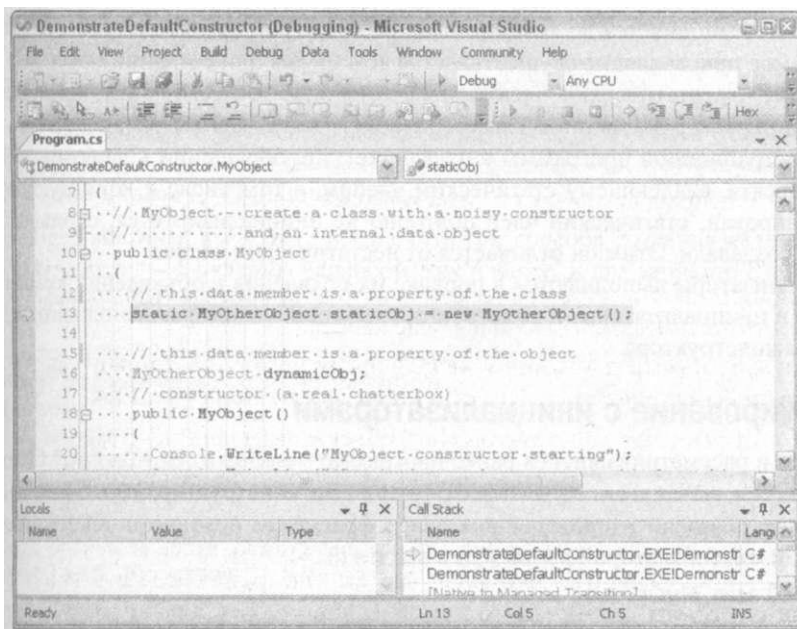


Рис. 11.4. После выполнения конструктора *MyOtherObject* вы возвращаетесь в точку его вызова

Непосредственная инициализация объекта - конструктор по умолчанию

Вы можете решить, что практически любой класс должен иметь конструктор! умолчанию некоторого вида, и в общем-то вы правы. Однако С# позволяет инициализировать члены-данные непосредственно, с использованием инициализаторов.

Итак, класс `BankAccount` можно записать следующим образом:

```
public class BankAccount
{
    // Номера банковских счетов начинаются с 1000 и
    // назначаются последовательно в возрастающем порядке
    static int nNextAccountNumber = 1000;
    // Для каждого счета поддерживаются его номер и баланс
    int nAccountNumber = ++nNextAccountNumber;
    double dBalance = 0.0;
    // ... прочие члены . . .
}
```

Вот в чем состоит работа инициализаторов. Как `nAccountNumber`, так и `dBalance` получают значения как часть объявления, эффект которого аналогичен использованному указанному коду в конструкторе.

Надо очень четко представлять себе картину происходящего. Вы можете решить, что это выражение присваивает значение `0.0` переменной `dBalance` непосредственно. О ведь `dBalance` существует только как часть некоторого объекта. Таким образом, присваивание не выполняется до тех пор, пока не будет создан объект `BankAccount`. Рассматриваемое присваивание осуществляется всякий раз при создании объекта.

Заметим, что статический член-данные `nNextAccountNumber` инициализируется при самом первом обращении к классу `BankAccount` (как вы убедились при выполнении демонстрационной программы в отладчике), т.е. обращении к любому свойству или методу объекта, владеющему статическим членом, в том числе и конструктору. Будет ли инициализирован, статический член повторно не инициализируется, сколько бы объектов вы не создавали. Этим он отличается от нестатических членов.

Инициализаторы выполняются в порядке их появления в объявлении класса. Если встречается и инициализаторы, и конструктор, то инициализаторы выполняются до выведения тела конструктора.

Конструирование с инициализаторами

Давайте в рассматривавшейся ранее программе `DemonstrateDefaultConstructor` перенесем вызов `new MyOtherObject()` из конструктора `MyObject` в объявление так, как показано в приведенном далее фрагменте исходного текста полужирным шрифтом, и изменим второй вызов `WriteLine()`.

```
public class MyObject
{
    // Этот член является свойством класса
    static MyOtherObject staticObj = new MyOtherObject();
    // Этот член является свойством объекта
    MyOtherObject dynamicObj = new MyOtherObjectO;
    public MyObject()
```

```

Console.WriteLine ("Начало конструктора MyObject");
Console.WriteLine (" (Статические члены " +
    "инициализированы до конструктора)");
// Ранее здесь создавался dynamicObj
Console.WriteLine ("Завершение конструктора MyObject");

```

Сравните вывод на экран такой модифицированной программы с выводом на экран исходной программы `DemonstrateDefaultConstructor`.

```

Начало функции Main()
Создание локального объекта MyObject в Main() :
Конструирование MyOtherObject
Конструирование MyOtherObject
Начало конструктора MyObject
(Статические члены инициализированы до конструктора)
Завершение конструктора MyObject
Нажмите <Enter> для завершения программы...

```



Полный текст данной программы можно найти на прилагаемом компакт-диске в каталоге `DemonstrateConstructorWithInitializer`.

Перегрузка конструкторов

Конструкторы можно перегружать так же, как и прочие методы.



Перегрузка функции обозначает определение двух функций с одним и тем же именем, но с разными типами аргументов (подробно этот вопрос рассматривается в главе 7, "Функции функций").

Предположим, вы хотите обеспечить три способа создания объекта `BankAccount` — с нулевым балансом, как и ранее, и два варианта с некоторыми начальными значениями.

`BankAccountWithMultipleConstructors` - банковский счет с разными вариантами конструкторов

```

Using System;

```

```

    Namespace BankAccountWithMultipleConstructors

```

```

Using System;

```

```

public class Program

```

```

{
    public static void Main(string[] args)
    {
        // Создание банковских счетов с корректными начальными
        // значениями
        BankAccount bal = new BankAccount();
        Console.WriteLine(bal.GetString());
    }
}

```

Классы

```

BankAccount ba2 = new BankAccount(100);
Console.WriteLine (ba2 . GetString ());

BankAccount ba3 = new BankAccount(1234, 200);
Console.WriteLine (ba3.GetString());

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");

Console.Read();

```

```

// BankAccount - простейший банковский счет
public class BankAccount
{
    // Первый номер счета — 1000; номера счетов
    // назначается последовательно
    static int nNextAccountNumber = 1000;

    // Номер счета и его баланс
    int nAccountNumber;
    double dBalance;

    // Несколько конструкторов - в зависимости от ваших
    // потребностей
    public BankAccount() // Автоматического конструктора нет
    {
        nAccountNumber = ++nNextAccountNumber;
        dBalance = 0.0;
    }

    public BankAccount(double dInitialBalance)
    {
        // Повторение части кода из конструктора по умолчанию
        nAccountNumber = ++nNextAccountNumber;

        // Теперь — код, специфичный для данного конструктора
        // Начинаем с переданного баланса (если он
        // положительный)
        if (dInitialBalance < 0)
        {
            dInitialBalance = 0;
        }
        dBalance = dInitialBalance;
    }

    public BankAccount(int nInitialAccountNumber,
        double dInitialBalance)
    {
        // Игнорируем отрицательный номер счета
        if (nInitialAccountNumber <= 0)
        {

```

```

        nInitialAccountNumber = ++nNextAccountNumber;
    }
    nAccountNumber = nInitialAccountNumber;

    // Начинаем с переданного баланса (если он
    // положительный)
    if (dInitialBalance < 0)
    {
        dInitialBalance = 0;
    }
    dBalance = dInitialBalance;
}

public string GetString ()
{
    return String.Format("#{0} = {1:N}",
                           nAccountNumber, dBalance);
}

```



Как только вы определили собственный конструктор — не имеет значения, какого именно типа, C# больше не создает конструктор по умолчанию для вашего класса. Поэтому нужно самим определить конструктор без параметров в этой демонстрационной программе.

В приведенной выше демонстрационной программе `BankAccountWithMultipleConstructors` имеются три конструктора:

- ✓ первый назначает номер счета и нулевой баланс;
- ✓ второй назначает номер счета и инициализирует баланс переданным положительным значением (отрицательные значения игнорируются);
- ✓ третий позволяет пользователю самостоятельно определить номер счета и положительный начальный баланс.

Функция `Main ()` создает различные банковские счета с использованием каждого из их конструкторов и выводит информацию о созданных объектах. Вывод этой программы на экран имеет следующий вид:

```

#1001 = 0.00
#1002 = 100.00
#1234 = 2000.0

```

Нажмите <Enter> для завершения программы...



В реальных классах требуется выполнять более строгую проверку входных параметров конструктора, чтобы гарантировать их корректность.

Конструкторы различаются между собой по тем же правилам, что и перегруженные функции. Первый объект, конструируемый в функции `Main ()`, `bal`, создается без аргументов, так что для него вызывается первый конструктор `BankAccount ()`, не получающий аргументов (он все еще именуется конструктором по умолчанию, хотя и не создается C# автоматически). Соответственно, этот счет получает номер по умолчанию и нулевой ба-

```

// BankAccount - банковский счет
public class BankAccount
(
    // Первый номер счета — 1000; номера счетов
    // назначается последовательно
    static int nNextAccountNumber = 1000;

    // Номер счета и его баланс
    int nAccountNumber;
    double dBalance;

    // Размещаем весь инициализирующий код в отдельной
    // функции, вызываемой из конструкторов
    public BankAccount() // Автоматического конструктора нет
    {
        Init(++nNextAccountNumber, 0.0);
    }

    public BankAccount(double dInitialBalance)
    {
        Init(++nNextAccountNumber, dInitialBalance);
    }

    // Конструктор с наибольшим количеством аргументов
    // выполняет всю работу
    public BankAccount(int nInitialAccountNumber,
                        double dInitialBalance)
    {
        // На самом деле тут надо проверить, чтобы значение
        // nInitialAccountNumber (а) не совпадало с уже
        // назначенными номерами счетов и (б) было не меньше
        // 1000
        Init(nInitialAccountNumber, dInitialBalance);
    }

    private void Init(int nInitialAccountNumber,
                      double dInitialBalance)
    {
        nAccountNumber = nInitialAccountNumber;

        // Используем переданный баланс (если он положителен)
        if (dInitialBalance < 0)
        {
            dInitialBalance = 0;
        }
        dBalance = dInitialBalance;
    }

    public string GetStringO

        return String.Format("#{0} = {1:N}",
                               nAccountNumber, dBalance);
}
}

```

Здесь метод `Init()` выполняет всю работу, связанную с конструированием. Оди по ряду причин такой подход недостаточно "кошерный" — не в последнюю очередь! за того, что при этом вызывается метод объекта, который еще не полностью построй это очень опасная игра!



К счастью, такой подход не является необходимым. Один конструктор« жет обращаться к другому с использованием ключевого слова `this` а дующим образом:

```
// BankAccountConstructorsAndThis - банковский счет с
// несколькими конструкторами
using System;

namespace BankAccountConstructorsAndFunction
{
    using System;

    public class Program
    {
        public static void Main(string[] args)
        {
            // Создание банковского счета с корректными начальными
            // значениями
            BankAccount bal = new BankAccount();
            Console.WriteLine(bal.GetString());

            BankAccount ba2 = new BankAccount(100);
            Console.WriteLine(ba2.GetString());

            BankAccount ba3 = new BankAccount(1234, 200);
            Console.WriteLine(ba3.GetString());

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы. ..").;
            Console.Read();
        }
    }

    // BankAccount - банковский счет
    public class BankAccount
    {
        // Первый номер счета — 1000; номера счетов
        // назначается последовательно
        static int nNextAccountNumber = 1000;

        // Номер счета и его баланс
        int nAccountNumber;
        double dBalance;

        // Вызываем конструктор с наибольшим количеством
        // аргументов, передавая значения по умолчанию для
        // отсутствующих параметров
    }
}
```

```

public BankAccount() : this(0, 0) {}

public BankAccount(double dInitialBalance)
    :this(0, dInitialBalance) {}

// Конструктор с наибольшим количеством аргументов
// выполняет всю работу
public BankAccount(int nInitialAccountNumber,
                    double dInitialBalance)
{
    // Игнорируем отрицательные номера счетов; нулевое
    // значение означает, что мы хотим использовать
    // очередное свободное значение номера счета
    if (nInitialAccountNumber <= 0)
    {
        nInitialAccountNumber = ++nNextAccountNumber;
    }
    nAccountNumber = nInitialAccountNumber;

    // Используем переданный баланс (если он положителен)
    if (dInitialBalance < 0)
    {
        dInitialBalance = 0;
    }
    dBalance = dInitialBalance;
}

public string GetString()
{
    return String.Format("#{0} = {1:N}",
                          nAccountNumber, dBalance);
}

```

В этой версии класса `BankAccount` имеются те же три варианта конструкторов, что и в предыдущих демонстрационных программах. Однако вместо повторения некоторых проверок в каждом конструкторе более простые конструкторы вызывают наиболее сложный и гибкий конструктор с использованием значений по умолчанию, а в нем и выполняются все необходимые проверки. Наличие функции `Init()` становится ненужным.

Создание объекта с использованием конструктора по умолчанию включает вызов конструктора `BankAccount()`:

```
BankAccount bal = new BankAccount(); // Параметров нет
```

Конструктор `BankAccount()` тут же передает управление конструктору `BankAccount(int, double)`, передавая ему значения по умолчанию 0 и 0.0:

```
public BankAccount() : this(0, 0) {}
```

Поскольку тело конструктора пустое, весь конструктор можно смело записывать в одну строку.

После выполнения основного конструктора с двумя параметрами управление возвращается конструктору по умолчанию, тело которого в данном случае совершенно пустое.

Создание банковского счета с ненулевым балансом и номером счета по умолчанию осуществляется следующим образом:

```
public BankAccount(double d) : this(0, d) {}
```

Фокусы с объектами

Невозможно создать объект без конструктора какого-либо вида. Если вы определите собственный конструктор, C# будет работать только с ним. Объединяя эти два факта, можно создать класс, который может быть инстанцирован только локально.

Например, создать объект `BankAccount`, если конструктор объявлен как `internal` (показан полужирным шрифтом в приведенном далее исходном тексте), могут те методы, определенные в том же модуле, что и `BankAccount`.

```
// BankAccount - моделирование простейшего банковского счета'
public class BankAccount
```

```
    // Номера банковских счетов начинаются с 1000 и
    // назначаются последовательно в возрастающем порядке
    static int nNextAccountNumber = 1000;
    // Для каждого счета поддерживаются его номер и баланс
    int nAccountNumber;
    double dBalance;
```

```
    internal BankAccount() // Конструктор
    {
        nAccountNumber = ++nNextAccountNumber;
        dBalance = 0;
    }
```

```
    public string GetStringO
    {
        return String.Format("#{0} = {1:N}",
                               nAccountNumber, dBalance);
    }
}
```



Глава 12

Наследование

В этой главе...

Определение одного класса посредством другого, более фундаментального

Разница между "является" и "содержит"

Изменение класса объекта

Построение статических членов и членов классов

Включение конструкторов в иерархию наследования

Вызов конструктора базового класса



Объектно-ориентированное программирование основано на трех принципах: управления доступом (инкапсуляция), наследования других классов и возможности соответствующего отклика (полиморфизм).

Наследование — распространенная концепция. Я — человек (не считая раннего утра, когда я только-только просыпаюсь). Я наследую ряд свойств класса Human (человек), такие как зависимость от воздуха, еды, умение разговаривать и т.п. Класс Human наследует потребность в воздухе, воде и еде от класса Mammal (млекопитающее), а тот, в свою очередь, — от класса Animal (животное).

Возможность передачи свойств очень важна. Она позволяет экономно описывать вещи и концепции. Например, на вопрос ребенка: "Что такое утка?" — ему можно ответить: "Это птица, которая крикает". Независимо от того, что вы подумали о таком ответе, он содержит значительное количество информации. Ребенок знает, что такое птица, а теперь он знает, что все то же можно сказать и об утке, а кроме того, у утки имеется дополнительное свойство "кряканье".

Объектно-ориентированные языки программирования выражают отношение наследования, позволяя одному классу наследовать другой, что, в свою очередь, дает возможность объектно-ориентированным языкам генерировать модели, более близкие к реальности, чем модели, генерируемые языками, объектно-ориентированное программирование не поддерживающими.

Наследование класса



В приведенной далее демонстрационной программе InheritanceExample класс Subclass наследован от класса BaseClass.

```
// InheritanceExample - простейшая демонстрация наследования
using System;
```

```

namespace InheritanceExample
{
    public class BaseClass
    {
        public int nDataMember;

        public void SomeMethodO
        {
            Console.WriteLine("SomeMethod()");
        }
    }

    public class Subclass : BaseClass
    {
        public void SomeOtherMethod()
        {
            Console.WriteLine("SomeOtherMethod()");
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            // Создание объекта базового класса
            Console.WriteLine("Работа с объектом базового класса:") *
            BaseClass be = new BaseClass O;
            be.nDataMember = 1;
            be.SomeMethod();
            // Создание объекта подкласса
            Console.WriteLine("Работа с объектом подкласса:");
            Subclass sc = new Subclass O;
            sc.nDataMember = 2;
            sc.SomeMethod();
            sc.SomeOtherMethod();

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");

            Console.Read();
        }
    }
}

```

Класс `BaseClass` определен как имеющий член-данные и простой метод `SomeMethod()`. Функция `Main()` создает объект `be` базового класса `BaseClass` и работает с ним.

Класс `Subclass` наследуется от класса `BaseClass` путем размещения и класса `BaseClass` после двоеточия в определении класса. `Subclass` получает члены класса `BaseClass` в качестве собственных, а также члены, которые могут быть в него добавлены. Функция `Main()` демонстрирует, что `Subclass` имеет член-д

nDataMember и член-функцию SomeMethod (), унаследованные от класса BaseClass, также новый метод SomeOtherMethod (), которого нет у базового класса.

Вывод программы на экран выглядит так, как от него и ожидалось (когда все прошло для без сбоев, это всегда приятный сюрприз):

```
Работа с объектом базового класса:
SomeMethod ()
работа с объектом подкласса:
SomeMethod ()
SomeOtherMethod ()
Нажмите <Enter> для завершения программы...
```

Это потрясающе

Люди составляют обширные системы, чтобы было проще разбираться в том, что их окружает. Тузик является частным случаем собаки, которая относится к собакообразным, входящим в состав млекопитающих, и т.д. Так легче познавать мир.

Если привести другой пример, можно сказать, что студент является человеком (точнее, его частным случаем). После этих слов сразу становится известно довольно много о студентах (об американских, естественно). Известно, что они имеют номера социального страхования, слишком много смотрят телевизор и постоянно мечтают о сексе. Но известно все это потому, что эти свойства присущи всем людям.

В С# говорится, что класс Student наследует класс Person. Кроме того, Person является *базовым классом* для класса Student. Наконец, можно сказать, что Student ЯВЛЯЕТСЯ Person (использование прописных букв — общепринятый метод отражения уникального типа связи). Эта терминология применяется в С++ и других объектно-ориентированных языках программирования.

Заметьте, что хотя Student и ЯВЛЯЕТСЯ Person, обратное не верно. Person не ЯВЛЯЕТСЯ Student (такое выражение следует трактовать в общем смысле, поскольку конкретный человек, конечно же, может оказаться студентом). Существует много людей, являющихся членами класса Person, но не членами класса Student. Кроме того, класс Student имеет средний балл, а Person — нет.

Свойство наследования транзитивно. Например, если определить новый класс GraduateStudent как подкласс класса Student, то он тоже будет наследником Person. Это значит, что будет выполняться следующее: если GraduateStudent ЯВЛЯЕТСЯ Student и Student ЯВЛЯЕТСЯ Person, то GraduateStudent ЯВЛЯЕТСЯ Person.

Зачем нужно наследование

Наследование выполняет ряд важных функций. Вы можете решить, что главная из них — уменьшить количество ударов по клавишам в процессе ввода программы. И это тоже — вам не надо заново вводить все свойства Person при описании класса Student.

Однако более важна возможность *повторного использования* (reuse). Нет нужды начинать каждый новый проект с нуля, если можно воспользоваться готовыми программными компонентами.

Сравним разработку программного обеспечения с другими областями человеческой деятельности. Многие ли производители автомобилей начинают проектировать **новую** модель с разработки для этого новых шурупов, болтов и гаек? И даже если это так, что вы скажете о разработке новых молотков, отверток и прочего инструментария? Конечно же нет — по возможности при проектировании и сборке новой модели **максимально** используются детали и части старой — не только болты и гайки, но и **крупные** узлы, такие как компрессоры или даже двигатели.

Наследование позволяет настроить уже имеющиеся программные компоненты. Старые классы могут быть адаптированы для применения в новых программах без внесения в них кардинальных изменений. Существующий класс наследуется — с расширением его возможностей — новым подклассом, который содержит все необходимые добавления и изменения. Если базовый класс написан кем-то иным, у вас может просто не быть возможности вносить в него изменения, и наследование оказывается единственным **способом** его использования.

Данная возможность тесно связана с третьим преимуществом применения наследования. Представим ситуацию, когда вы наследуете базовый класс. Позже выясняется, что в нем имеется ошибка, которую нужно исправить. Если вы модифицировали класс для его повторного использования, вы должны вручную внести изменения и протестировать каждое приложение в отдельности. При наследовании класса без внесения изменений в общем случае исправляете только базовый класс, не затрагивая сами приложения.

Однако главное преимущество наследования в том, что оно описывает реальный мир таким, каков он есть.

Более сложный пример наследования



Банк поддерживает несколько типов счетов. Один из них — депозитный счет — обладает всеми свойствами простого банковского счета плюс возможность накопления процентов. Такое отношение на языке C# моделируется в приведенной далее демонстрационной программе. Версия этой программы на компакт-диске включает некоторые изменения из следующего раздела, так что она несколько отличается от приведенного здесь листинга.

```
// SimpleSavingsAccount - реализация счета SavingsAccount
// как разновидности BankAccount; здесь не используются
// виртуальные методы (о них будет сказано в главе 13)
using System;
namespace SimpleSavingsAccount
{
    // BankAccount - модель банковского счета, который имеет
    // номер и хранит текущий баланс
    public class BankAccount // the base class
    {
        // Номера счетов начинаются с 1000 и образуют
        // возрастающую последовательность
        public static int nNextAccountNumber = 1000;
        // Номер счета и баланс для каждого объекта свои
        public int nAccountNumber;
        public decimal mBalance;
        // Init - Инициализация счета очередным свободным
        // номером и конкретным начальным балансом (по умолчанию
```

```

// ноль)
public void InitBankAccount()
{
    InitBankAccount(0);
}
public void InitBankAccount(decimal mInitialBalance)
{
    nAccountNumber = ++nNextAccountNumber;
    mBalance = mInitialBalance;
}
// Свойство Balance
public decimal Balance
{
    get { return mBalance; }
}
// Deposit - Позволен любой положительный вклад
public void Deposit(decimal mAmount)
{
    if (mAmount > 0)
    {
        mBalance += mAmount;
    }
}
// Withdraw - можно снять не более того, что имеется на
// счету; метод возвращает снятую сумму
public decimal Withdraw(decimal mWithdrawal)
{
    if (Balance <= mWithdrawal) // используется свойство
    {                             // Balance
        mWithdrawal = Balance;
    }
    mBalance -= mWithdrawal;
    return mWithdrawal;
}
// ToString - строка с информацией о состоянии счета
public string ToBankAccountString()
{
    return String.Format("{0} - {1:C}",
        nAccountNumber, Balance);
}

// SavingsAccount - банковский счет с накоплением
// процентов
public class SavingsAccount : BankAccount // Подкласс
{
    public decimal mInterestRate;
    // InitSavingsAccount - использует процентную ставку,
    // выражаемую числом от 0 до 100
    public void InitSavingsAccount(decimal mInterestRate)
    {
        InitSavingsAccount(0, mInterestRate);
    }
    public void InitSavingsAccount(decimal mInitial,
        decimal mInterestRate)

```

```

    {
        InitBankAccount(mInitial);
        this.mInterestRate = mInterestRate / 100;
    }
    // AccumulateInterest - вызывается однократно в конце
    // периода начисления процентов
    public void AccumulateInterest()
    {
        mBalance = Balance + (decimal) (Balance*mInterestRate);i
    }
    // ToString - строка с информацией о состоянии счета
    public string ToSavingsAccountString()

        return String.Format("{0} ({1}%)"(
                                ToBankAccountStringO ,
                                mInterestRate * 100);

    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Создание банковского счета и вывод на экран
        BankAccount ba = new BankAccount();
        ba.InitBankAccount(1000);
        ba.Deposit(100);
        Console.WriteLine("Счет {0}",
                           ba.ToBankAccountString());
        // Теперь — депозитный счет
        SavingsAccount sa = new SavingsAccount();
        sa.InitSavingsAccount(100, 12.5M);
        sa.AccumulateInterest();
        Console.WriteLine("Счет {0}",
                           sa.ToSavingsAccountString());
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
        Console.Read();
    }
}
}

```

Класс `BankAccount` ничем не отличается от того, каким он был в других главах книги. Он начинается с перегруженной функции инициализации `InitBankAccount` одной для произвольного начального значения баланса, другой — для нулевого балам. Обратите внимание, что здесь не использованы конструкторы, которые были в поем ней рассматривавшейся версии этого класса в главе 11, "Классы". Позже вы поймете! почему сделан такой шаг назад.

Свойство `Balance` позволяет другим читать значение баланса, запрещая при этом! изменять. Метод `Deposit` принимает любой положительный вклад, а метод `Withdraw` предоставляет возможность снять со счета любую сумму (в пределах наличного баланс. Метод `ToBankAccountString()` создает строку с описанием состояния счета.

Класс `SavingsAccount` наследует все, что можно, от класса `BankAccount`. К этому он добавляет процентную ставку и возможность накопления процентов.

Функция `Main()` делает минимально возможную работу. Она создает счет `BankAccount`, выводит информацию о нем, создает счет `SavingsAccount`, один раз начисляет проценты и выводит результат. Полностью вывод программы выглядит следующим образом:

Счет 1001 - \$200.00

Счет 1002 - \$112.50 (12.500%),

Нажмите <Enter> для завершения программы...



Обратите внимание, что метод `InitSavingsAccount()` вызывает метод `InitBankAccount()`, который инициализирует члены-данные банковского счета. Метод `InitSavingsAccount()` мог бы делать это непосредственно, однако лучше позволить классу `BankAccount` самостоятельно инициализировать свои члены. Каждый класс должен отвечать за себя сам.

ЯВЛЯЕТСЯ или СОДЕРЖИТ

Отношения между `SavingsAccount` и `BankAccount` представляют собой фундаментальное отношение ЯВЛЯЕТСЯ, которое присуще наследованию. Чуть позже будет рассмотрено альтернативное отношение СОДЕРЖИТ.

Отношение ЯВЛЯЕТСЯ

Отношение ЯВЛЯЕТСЯ (IS_A) между `SavingsAccount` и `BankAccount` можно ремонстрировать путем следующего изменения в классе `Program` демонстрационной программы `SimpleSavingsAccount` из предыдущего раздела,

```
public class Program
{
    // Добавим:
    // DirectDeposit - автоматический вклад на счет
    public static void DirectDeposit(BankAccount ba,
                                     decimal mPay)
    {
        ba.Deposit(mPay);
    }
    public static void Main(string[] args)
    {
        // Создание банковского счета и вывод на экран
        BankAccount ba = new BankAccount();
        ba.InitBankAccount(100);
        DirectDeposit(ba, 100);
        Console.WriteLine("Счет {0}",
                           ba.ToBankAccountString());
        // Теперь - депозитный счет
        SavingsAccount sa = new SavingsAccount();
        sa.InitSavingsAccount(100, 12.5M);
        DirectDeposit(sa, 100);
        sa.AccumulateInterest();
    }
}
```

```

        Console.WriteLine("Счет {0}",
                           sa.ToSavingsAccountString());
// Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");

        Console.Read();
    }
}

```

Почти ничего не изменилось. Единственное реальное отличие заключается в том, J теперь все вклады делаются при помощи локальной функции `DirectDeposit()`, которая не является частью класса `BankAccount`. Аргументами этой функции являлись банковский счет и величина вклада.

Обратите внимание (начинается интересное!), что функция `Main()` может перед функцией `DirectDeposit()` как обычный банковский счет, так и депозитный, поскольку `SavingsAccount` ЯВЛЯЕТСЯ `BankAccount` и, таким образом, имеем права и привилегии последнего. Поскольку `SavingsAccount` ЯВЛЯЕТСЯ `BankAccount`, вы можете присвоить `SavingsAccount` переменной типа `BankAccount` и использовать его в качестве аргумента `BankAccount`.

Доступ к BankAccount через содержание

Класс `SavingsAccount` может получить доступ к членам `BankAccount` и другим способом, как показано в приведенном далее фрагменте кода (ключевая строка здесь делена полужирным шрифтом).

```

// SavingsAccount - банковский счет с накоплением процентов
public class SavingsAccount_ // Обратите внимание на
                             // подчеркивание: это не класс
{                             // SavingsAccount.
    public BankAccount bankAccount; // Содержит BankAccount
    public decimal mInterestRate;
    // InitSavingsAccount - ввод процентной ставки как
    // величины от 0 до 100
    public void InitSavingsAccount(BankAccount bankAccount,
                                   decimal mInterestRate)

    {
        this.bankAccount = bankAccount;
        this.mInterestRate = mInterestRate / 100;

        // AccumulateInterest - выполняется однократно при
        // начислении процентов
        public void AccumulateInterest()

        {
            bankAccount.mBalance = bankAccount.Balance
                + (bankAccount.Balance * mInterestRate);

            // Deposit - разрешен любой положительный вклад
            public void Deposit(decimal mAmount)

            {
                // Делегирование содержащемуся объекту BankAccount
                bankAccount.Deposit(mAmount);
            }
        }
    }
}

```



```

//Withdraw - можно снять не более того, что имеется на
// счету; метод возвращает снятую сумму
public double Withdraw(decimal mWithdrawal)

return bankAccount.Withdraw(mWithdrawal) ;

```

В этом случае класс `SavingsAccount_` содержит член-данные `bankAccount` вместо наследования от `BankAccount`). Объект `bankAccount` включает номер счета, баланс, необходимые для функционирования `SavingsAccount_`. Класс `SavingsAccount_` содержит данные, специфичные для депозитного счета и *делегует* при необходимости запросы к содержащемуся в нем объекту `BankAccount` (т.е. когда классу `SavingsAccount_` нужен, например, баланс, он запрашивает его у содержащегося в объекте `BankAccount`).

В этом случае речь идет о том, что `SavingsAccount_` СОДЕРЖИТ `BankAccount`.

Отношение СОДЕРЖИТ

Отношение СОДЕРЖИТ фундаментально отличается от отношения ЯВЛЯЕТСЯ. Это ~~отноше~~ кажется не столь существенным в следующем фрагменте исходного текста:

```

// Создание нового депозитного счета
BankAccount ba = new BankAccount ()
// Особая версия SavingsAccount:
SavingsAccount_ sa = new SavingsAccount_ () ;
sa.InitSavingsAccount(ba, 5);
// Вкладываем 100 на счет
sa.Deposit(100);
// Подсчитываем проценты
sa.AccumulateInterest();

```

Проблема в том, что теперь `SavingsAccount_` не может использоваться как `BankAccount`, поскольку не является его наследником. Он теперь *содержит* `BankAccount`, а это далеко не одно и то же. Например, следующий код компилироваться не будет:

```

// DirectDeposit - автоматический вклад на счет
void DirectDeposit(BankAccount ba, int nPay)
{
    ba.Deposit(nPay);
}

void SomeFunctionO
{
    // Этот код не скомпилируется
    SavingsAccount_ sa = new SavingsAccount_ () ;
    DirectDeposit(sa, 100);
    // .. продолжение ...
}

```

Функция `DirectDeposit()` не может принять `SavingsAccount_` вместо `BankAccount`. Между этими классами нет такого очевидного отношения, как в случае наследования.

Когда использовать отношение ЯВЛЯЕТСЯ, а когда — СОДЕРЖИТ

Различие между отношениями ЯВЛЯЕТСЯ и СОДЕРЖИТ гораздо глубже, чем просто предмет программного соглашения. Эти отношения проистекают из отношений в реальном мире.

Например, "Запорожец" ЯВЛЯЕТСЯ автомобилем. Автомобиль СОДЕРЖИТ мотор. Если ваш знакомый скажет, чтобы вы заехали за ним на автомобиле, и вы приедете "Запорожце", ему будет не на что пожаловаться — вы приехали на автомобиле. Но если вы притащите к нему двигатель от "Запорожца", у него будут все основания обидеться на глупую шутку.

Класс `Zarogzhets` должен расширять класс `Car` не только для того, чтобы получить доступ к методам `Car`, но и чтобы выразить фундаментальные отношения между этими классами.

К сожалению, начинающий программист может унаследовать `Car` от `Motor`, как простейший способ получения доступа к членам `Motor`, которые нужны классу `Car` для управления. Например, `Car` может унаследовать у класса `Motor` метод `Go()`. Однако этот пример вскрывает одну из проблем, возникающих при таком подходе. Несмотря на то что "поехал" звучит одинаково и в машине, и даже в ракете, "поехали" по отношению к машине совсем то, что "поехали" по отношению к мотору. Для того чтобы поехала машина, надо обязательно завести мотор, но это далеко не одно и то же — ведь для того чтобы поехала машина, и еще отпустить тормоз, переключиться на первую передачу, отпустить сцепление и т.д.

Словом, автомобиль просто не является видом мотора, и этого достаточно.



Элегантность программного обеспечения — это не просто эстетический фактор. Она способствует пониманию кода, повышает его надежность, облегчает поддержку, снижает количество возможных ошибок.

Специалисты в области объектно-ориентированного программирования для простоты дизайна рекомендуют отдавать предпочтение отношению СОДЕРЖИТ.

Поддержка наследования в C#

C# имеет ряд возможностей, разработанных для поддержки наследования.

Изменение класса

Программа может изменить класс объекта. Вы уже встречались с этим в одном из примеров. `SomeFunction()` может передать объект `SavingsAccount` методу, который ожидает объект `BankAccount`.

Это преобразование можно сделать явным следующим образом:

```
BankAccount ba, -
SavingsAccount sa = new SavingsAccount();
```

```

// Верно:
ba = sa; // Неявное преобразование к
// базовому классу разрешено
ba = (BankAccount)sa; // Но явное преобразование
// предпочтительнее
// Ошибка:
sa = ba; // Неявное преобразование к
// подклассу запрещено
sa = (SavingsAccount)ba; // Допустимо

```

В первой строке объект `SavingsAccount` сохраняется в переменной типа `BankAccount`. C# выполняет необходимое преобразование за вас. Во второй строке явным образом использован оператор приведения типа.

Последние две строки преобразуют объект типа `BankAccount` в `SavingsAccount`.



Отношение ЯВЛЯЕТСЯ не рефлексивно. То есть, несмотря на то что "Запорожец" является автомобилем, автомобиль — не обязательно "Запорожец". Аналогично, `BankAccount` — не обязательно `SavingsAccount`, так что неявное преобразование в этом направлении запрещено. Последняя строка разрешена, поскольку в ней программист явно указывает, что он берет на себя ответственность за выполнение данного преобразования.

Неверное преобразование времени выполнения

В общем случае приведение объекта от типа `BankAccount` к типу `SavingsAccount` — достаточно опасная операция. Рассмотрим следующий пример:

```

public static void ProcessAmount(BankAccount bankAccount)
{
    // Вносим на счет большую сумму
    bankAccount.Deposit(10000.00M);
    // Если объект — SavingsAccount, добавляем проценты
    SavingsAccount sa = (SavingsAccount)bankAccount;
    sa.AccumulateInterest();
}

public static void TestCastO
{
    SavingsAccount sa = new SavingsAccount();
    ProcessAmount(sa);
    BankAccount ba = new BankAccount();
    ProcessAmount(ba);
}

```

Функция `ProcessAmount()` выполняет несколько операций, включая вызов метода `AccumulateInterest()`. Приведение `ba` к типу `SavingsAccount` необходимо, поскольку `ba` объявлено как `BankAccount`. Программа корректно компилируется, так как все преобразования типов выполнены явно.

Все нормально работает при первом вызове `ProcessAmount()` из `Test()`. Объект `SavingsAccount` передается методу `ProcessAmount()`. Преобразование типа из `BankAccount` в `SavingsAccount` не вызывает проблем, поскольку объект `ba` изначально был объектом типа `SavingsAccount`.

Однако со вторым вызовом `ProcessAccount()` не все так гладко. Преобразование к типу `SavingsAccount` не может быть разрешено. Объект `ba` не имеет метода `AccumulateInterest()`.



Некорректное преобразование типов генерирует ошибку в процессе выполнения программы (так называемую *ошибку времени выполнения* (run-time error). Ошибки времени выполнения гораздо сложнее найти и исправить, чем ошибки времени компиляции. Что еще более неприятно, такая ошибка может произойти не с вами, а с другим пользователем программы. Обычно особого восторга у пользователей такие ошибки не вызывают.

Ключевые слова `is` и `as`

Функция `ProcessAccount()` работала бы корректно, если бы могла убедиться, что переданный ей объект действительно имеет тип `SavingsAccount`, перед тем, как вы поднять преобразование. C# предоставляет для этого два ключевых слова — `is` и `as`.

Использование оператора `is`

Оператор `is` получает объект в качестве левого аргумента и тип — в качестве правого. Оператор возвращает значение `true`, если тип времени выполнения объекта слева совместим с типом справа. Этот оператор можно использовать для проверки корректности преобразования перед его выполнением.

Предыдущий пример можно модифицировать с применением оператора `is`, что позволит избежать ошибки времени выполнения.

```
public static void ProcessAmount(BankAccount bankAccount)
{
    // Вносим на счет большую сумму
    bankAccount.Deposit(10000.00M);
    // Если объект — SavingsAccount...
    if (bankAccount is SavingsAccount)
    {
        // ... добавляем проценты (преобразование типов
        // гарантированно работает)
        SavingsAccount savingsAccount =
            (SavingsAccount)bankAccount;
        savingsAccount.AccumulateInterest();
    }
    // В противном случае преобразование не выполняется.
    // Однако почему BankAccount — не то, что вы ожидали?
    // Возможно, это какая-то ошибочная ситуация?
}

public static void TestCastO
{
    SavingsAccount sa = new SavingsAccount();
    ProcessAmount(sa);
    BankAccount ba = new BankAccount();
    ProcessAmount(ba);
}
```

Добавление оператора `is` дает гарантию, что преобразование будет выполнено, только если объект `bankAccount` в действительности имеет тип `SavingsAccount`. При

первом вызове функции `ProcessAmount()` оператор `is` вернет значение `true`, но при втором вызове, когда в функцию будет передан объект `BankAccount`, оператор `is` вернет `false`, что позволит избежать некорректного преобразования типов. Такая версия программы не генерирует ошибку времени выполнения.



С одной стороны, я настоятельно рекомендую вам защищать все выполняемые преобразования оператором `is` во избежание возможных ошибок времени выполнения. С другой стороны, я рекомендую избегать приведения типов вообще.

Класс `object`

Рассмотрим следующие связанные классы:

```
public class MyBaseClass {}
public class MySubClass : MyBaseClass {}
```

Соотношение между этими двумя классами позволяет программисту сделать следующий тест времени выполнения:

```
public class Test
{
    public static void
        GenericFunction(MyBaseClass mc)

        // Если объект действительно является подклассом

        MySubClass msc = mc as MyBaseClass;
        if(msc != null)
        {
            // ... то и обрабатываем его как подкласс
            // . . . продолжение . . .
        }
}
```

В этом случае функция `GenericFunction()` в состоянии различить подклассы класса `MyBaseClass` с помощью оператора `as`.

Но как различить два не связанных между собой класса с использованием того же оператора `as`? Оказывается, C# производит все классы от одного общего предка — базового класса `object`. Таким образом, любой класс, который явно не наследует другой класс, наследует класс `object`. А значит, два следующих выражения объявляют классы с одним и тем же базовым классом `object`:

```
class MyClass1 : object {}
class MyClass2 {}
```

Общий базовый класс `object` позволяет написать следующую обобщенную функцию:

```
public class Test
{
    public static void
        GenericFunction(object o)
    {
        MyClass1 mcl = o as MyClass1;
    }
}
```

```

        if(mcl != null)
        {
            // Используем объект mcl, полученный преобразованием
            // . . .
        }
    }
}

```

Функция `GenericFunction()` может быть вызвана для объекта любого типа. Выражаясь поэтически, ключевое слово `as` извлекает жемчужину `MyClass1` из устрицы `object`.

Использование оператора `as`

Оператор `as` работает несколько иначе, чем оператор `is`. Вместо возврата значения типа `bool` он преобразует объект слева от себя к типу справа, но при этом возвращение `null`, если такое преобразование некорректно— вместо генерации ошибки времени выполнения при использовании обычного преобразования. Так что вы всегда должны проверять, не равен ли результат работы оператора `as` ссылке `null`:

```

SavingsAccount SavingsAccount =
    bankAccount as SavingsAccount;
if(SavingsAccount != null)
{
    // Продолжаем работу с использованием SavingsAccount
}
// В противном случае мы не можем использовать этот объект и
// должны сгенерировать сообщение об ошибке самостоятельно

```

Какой из операторов предпочесть? Вообще говоря, оператор `as` более эффективен. Он сразу выполняет преобразование, в то время как оператор `is` требует двух этапов проверки с его использованием с последующим преобразованием типа.



К сожалению, `as` не работает с переменными типов-значений, так что вы можете применять его с такими типами, как `int`, `long`, `double` и подобный. В этом случае предпочтительней использовать оператор `is`.

Наследование и конструктор

Программа `InheritanceExample`, с которой вы встречались ранее в этой главе применяет функции `Init...()` для инициализации объектов `BankAccount` и `SavingsAccount` и приведения их в корректное состояние. Оснащение этих классов конструкторами, определенно, правильное решение, хотя и со своими сложностями.

Вызов конструктора по умолчанию базового класса



Когда создается подкласс, всякий раз вызывается конструктор по умолчанию базового класса. Конструктор подкласса автоматически вызывает конструктор базового класса, что видно на примере приведенной далее демонстрационной программы.

```
//InheritingAConstructor - демонстрация автоматического
// вызова конструктора по умолчанию базового класса
using System;
namespace InheritingAConstructor
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Создание объекта BaseClass");
            BaseClass be = new BaseClass();
            Console.WriteLine("\nСоздание объекта Subclass");
            Subclass sc = new Subclass();

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");

            Console.Read();
        }
    }
}
```

```
public class BaseClass
{
    public BaseClass()
    {
        Console.WriteLine("Конструктор BaseClass");
    }
}

public class Subclass : BaseClass
{
    public Subclass()
    {
        Console.WriteLine("Конструктор Subclass");
    }
}
```

Конструкторы `BaseClass` и `Subclass` не делают ничего, кроме вывода строки на экран. Создание объекта `BaseClass` приводит к вызову конструктора по умолчанию `BaseClass`. Создание объекта `Subclass` приводит к вызову конструктора по умолчанию `BaseClass` перед тем, как вызывается собственно конструктор `BaseClass`.

Это ясно видно из вывода рассмотренной демонстрационной программы на экран.

```
Создание объекта BaseClass
Конструктор BaseClass
Создание объекта Subclass
Конструктор BaseClass
Конструктор Subclass
Нажмите <Enter> для завершения программы...
```

Иерархия наследуемых классов весьма напоминает этажи здания. Каждый класс, построенный на основе другого класса, представляет собой новый, верхний этаж. То же относится и к конструкторам классов: прежде чем будет вызван конструктор верхнего эта-

жа для его построения, надо построить нижний этаж. Очевидна и причина этого: каждый класс сам отвечает за себя, а значит, подкласс не должен отвечать за инициализацию членов базового класса. `BaseClass` должен получить возможность сконструировать свои члены до того, как члены `Subclass` смогут к ним обратиться. Лошадь нужной ставить перед телегой...

Передача аргументов конструктору базового класса

Подкласс вызывает конструктор по умолчанию базового класса, если только не указано иное, например, из конструктора подкласса, не являющегося конструктором по умолчанию. Вот немного исправленная демонстрационная программа, иллюстрирующая сказанное:

```
using System;
namespace Example
{
    public class Program
    {
        public static void Main(string[] args)

            Console.WriteLine("Вызов Subclass()");
            Subclass sc1 = new Subclass(),
            Console.WriteLine("\nВызов Subclass(int)");
            Subclass sc2 = new Subclass(0);
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");

            Console.Read();

        public class BaseClass
        {
            Console.WriteLine("Конструктор BaseClass (по умолчанию)");

            public BaseClass(int i)
            {
                Console.WriteLine("Конструктор BaseClass (int)");
            }
        }
        public class Subclass : BaseClass
        {
            public Subclass()
            {
                Console.WriteLine("Конструктор Subclass (по умолчанию)")

            public Subclass(int i)
            {
                Console.WriteLine("Конструктор Subclass (int)");
            }
        }
    }
}
```


Выполнение программы приводит к следующему выводу на экран:

```
Вызов Subclass ()
Конструктор BaseClass (по умолчанию)
Конструктор Subclass (по умолчанию)
Вызов Subclass (int)
Конструктор BaseClass (по умолчанию)
Конструктор Subclass (int)
Нажмите <Enter> для завершения программы. . .
```

Данная демонстрационная программа сперва создает объект по умолчанию. Как и ожидалось, C# выполняет конструктор по умолчанию Subclass, который сначала передает управление конструктору по умолчанию BaseClass. Затем программа создает объект, передавая целочисленный аргумент. Как и предполагалось, теперь C# вызывает конструктор Subclass (int).

Этот конструктор, в свою очередь, вызывает конструктор по умолчанию BaseClass, как и в предыдущем примере, поскольку никакие данные базовому классу не передаются.

Указание конкретного конструктора базового класса

Конструктор подкласса может вызвать определенный конструктор базового класса с использованием ключевого слова base.



Эта возможность аналогична способу, которым один конструктор может вызвать другой конструктор того же класса с применением ключевого слова this (см. главу 11, "Классы").

Рассмотрим, например, следующую демонстрационную программу:

```
// InvokeBaseConstructor - демонстрация того, как подкласс
// может вызвать конструктор базового класса по своему
// выбору с использованием ключевого слова base
using System;

namespace InvokeBaseConstructor
{
    public class BaseClass
    {
        public BaseClass()
        {
            Console.WriteLine("Конструктор BaseClass (по умолчанию)");
        }
        public BaseClass(int i)
        {
            Console.WriteLine("Конструктор BaseClass({0})", i);
        }
    }

    public class Subclass : BaseClass
    {
        public Subclass()
        {
            Console.WriteLine("Конструктор Subclass (по умолчанию)");
        }
    }
}
```

```

    }
    public Subclass(int i1, int i2) : base(i1)
    {
        Console.WriteLine("Конструктор Subclass({0}, {1})", i1,
i2) ;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Вызов Subclass()");
        Subclass sc1 = new Subclass();
        Console.WriteLine("\nВызов Subclass(1, 2)");
        Subclass sc2 = new Subclass(1, 2);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");
        Console.Read();
    }
}
}

```

Вывод программы выглядит следующим образом:

```

Вызов Subclass()
Конструктор BaseClass (по умолчанию)
Конструктор Subclass (по умолчанию)
Вызов Subclass(1, 2)
Конструктор BaseClass(1)
Конструктор Subclass(1, 2)
Нажмите <Enter> для завершения программы...

```

Эта версия демонстрационной программы начинается так же, как и предыдущие при! меры — с создания объекта Subclass с применением конструкторов по умолчанию как для класса Subclass, так и для BaseClass.

Второй объект создается при помощи выражения new Subclass(1,2). С# вызы- вает конструктор Subclass(int, int), в котором используется ключевое слово base для передачи одного из значений конструктору BaseClass(int). По всей видимости, Subclass передает первый аргумент для обработки базовому классу, а со вторым рабо- тает самостоятельно.

Обновленный класс BankAccount



Демонстрационная программа ConstructorSavingsAccount, имею- щаяся на прилагаемом компакт-диске, представляет собой обновленную! версию демонстрационной программы SimpleBankAccount. В этой вер- сии конструктор SavingsAccount может передавать информацию конст-1 руктору BankAccount. Здесь приведены только функция Main() и ука- занные конструкторы.

```

// ConstructorSavingsAccount - реализует SavingsAccount как
// вир, BankAccount; не использует виртуальные методы, но
// корректно реализует конструкторы
using System;
namespace ConstructorSavingsAccount
{
    // BankAccount - модель банковского счета с номером счета
    // (назначаемым при создании) и балансом
    public class BankAccount
    {
        // Номера счетов начинаются с 1000 и последовательно
        // увеличиваются
        public static int nNextAccountNumber = 1000;
        // Номер счета и баланс для каждого объекта
        public int nAccountNumber;
        public decimal mBalance;
        // Конструкторы
        public BankAccount(): this(0)
        {
        }
        public BankAccount(decimal mInitialBalance)
        {
            nAccountNumber = ++nNextAccountNumber;
            mBalance = mInitialBalance;

            // . . . Остальные методы . . .
        }
        // SavingsAccount - банковский счет с начислением
        // процентов
        public class SavingsAccount : BankAccount
        {
            public decimal mInterestRate;
            // Конструкторы. Процентная ставка выражается числом от
            // 0 до 100
            public SavingsAccount(decimal mInterestRate)
            : this(mInterestRate, 0)
            {
            }
            public SavingsAccount(decimal mInterestRate,
                                   decimal mInitial)
            : base(mInitial)
            {
                this.mInterestRate = mInterestRate / 100;

                // . . . Остальные методы . . .
            }
        }
        public class Program
        {
            // DirectDeposit - автоматический внос денег на счет
            public static void DirectDeposit (BankAccount ba,
                                                decimal mPay)
            {
                ba.mBalance += mPay;
            }
        }
    }
}

```

```

    }
    public static void Main(string[] args)
    {
        // Создание банковского счета и вывода информации о
        // нем
        BankAccount ba = new BankAccount(100);
        DirectDeposit(ba, 100);
        Console.WriteLine("Счет {0}",
                           ba.ToBankAccountString());
        // То же для счета с накоплением процентов
        SavingsAccount sa = new SavingsAccount(12.5M);
        DirectDeposit(sa, 100);
        sa.AccumulateInterest();
        Console.WriteLine("Счет {0}",
                           sa.ToSavingsAccountString() );
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");

        Console.Read();
    }
}
}

```

Класс `BankAccount` определяет два конструктора: один, который получает начальное значение баланса, и конструктор по умолчанию, не получающий никаких аргументов. Чтобы избежать дублирования кода конструкторов, конструктор по умолчанию вызывает конструктор с передаваемым начальным значением баланса посредством этого слова `this`.

Класс `SavingsAccount` также предоставляет в распоряжение программиста! конструктора. Конструктор `SavingsAccount`, принимающий в качестве аргумента» личину процентной ставки, вызывает конструктор `SavingsAccount`, принимающий в качестве аргументов величину процентной ставки и начальное значение баланса, передавая в качестве последнего 0. В свою очередь, этот конструктор наиболее объективно передает начальное значение баланса соответствующему конструктору `BaseClass` это отражено на диаграмме на рис. 12.1).

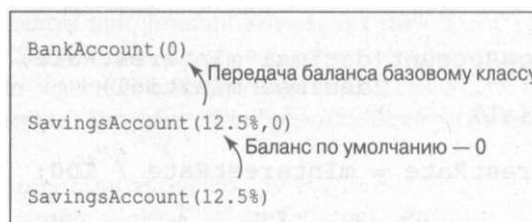


Рис. 12.1. Передача параметров в цепочке вызовов конструкторов

Программа модифицирована таким образом, чтобы избежать вызова внутренних функций `Init...()`, заменив их автоматически вызываемыми конструкторами. Вывод этой демонстрационной программы ничем не отличается от вывода ее предшественницы.

Деструктор

C# предоставляет также метод, обратный конструктору и именуемый *деструктором*. Деструктор имеет то же имя, что и имя класса, но предваренное символом тильды (~). Например, метод `~BaseClass()` является деструктором класса `BaseClass()`.

C# вызывает деструктор, когда перестает использовать объект. Деструктор по умолчанию — единственный, который может иметь класс, поскольку деструктор не вызывается явно. Кроме того, деструктор всегда виртуален (о виртуальных методах будет сказано в главе 13, "Полиморфизм").

При использовании наследования деструкторы вызываются в порядке, обратном порядку вызова конструкторов. Таким образом, деструктор подкласса вызывается перед деструктором базового класса.



Сборка мусора и деструкторы C#

Деструктор в C# менее полезен, чем в ряде других объектно-ориентированных языков программирования, таких как C++, поскольку в C# используется *недетерминистическая деструкция*. Этот термин и его важность требуют определенных пояснений.

Память для объекта выделяется из кучи при выполнении команды `new`, например, `new Subclass()`. Блок памяти остается зарезервированным до тех пор, пока имеется хотя одна корректная ссылка на эту память. Вы можете иметь несколько переменных, ссылающихся на один и тот же объект.

О памяти говорят, что она "недостижима", когда из области видимости выходит последняя ссылка на нее. Другими словами, никто не в состоянии обратиться к блоку памяти после утраты последней ссылки на нее.

Когда блок памяти становится недостижимым, C# не предпринимает никаких специфических действий. В фоновом режиме выполняется низкоприоритетный системный процесс, который проводит поиск недостижимых блоков памяти. Такой "сборщик мусора" запускается, когда в работе программы наступает затишье — чтобы не повлиять отрицательно на ее производительность. Когда сборщик мусора находит недостижимый блок памяти, он возвращает его в кучу.

Обычно сборщик мусора незаметно работает в фоновом режиме и получает управление только на короткие периоды времени, когда начинает чувствоваться нехватка памяти.

Деструкторы C#, такие как `~BaseClass()`, являются недетерминистическими, поскольку не вызываются до тех пор, пока объект не будет подобран сборщиком мусора, а это может случиться через продолжительное время после того, как объект перестанет использоваться. Может даже возникнуть ситуация, когда программа завершится до того, как будет выполнена очередная сборка мусора, и в этом случае деструктор не будет вызван вообще. *Недетерминистический* означает, что вы не можете предсказать, когда объект будет уничтожен сборщиком мусора. Может пройти немало времени до того, как объект будет подобран сборщиком мусора и будет вызван деструктор этого объекта.

Основной вывод — программист на C# не может полагаться на автоматический вызов деструктора, как в таком языке, как C++, так что деструкторы в C# используются крайне редко. В C# имеются другие способы вернуть системе захваченные ресурсы, которые больше не нужны, — с применением метода `Dispose()`, изучение которых, увы, выходит за рамки настоящей книги.

Глава 13

Полиморфизм

В этой главе...

- > Скрывать или перекрывать методы базового класса?
- > Реально ли создание абстрактных классов?
- > Объявление абстрактного метода
- > Создание новой иерархии поверх существующей
- > Защита класса от наследования



Наследование позволяет одному классу "приспособить" члены другого класса. Таким образом, можно создать класс `SavingsAccount`, который наследует члены-данные и методы от базового класса `BankAccount`. Однако этого недостаточно для имитации объектов реального мира.



Вернитесь к главе 12, "Наследование", если вам требуется освежить свои знания о наследовании.

Микроволновая печь представляет собой определенный тип печи, но не из-за внешнего вида, а потому что она выполняет те же функции, что и любая печь. Она может выполнять и ряд дополнительных функций, но как минимум она должна реализовать базовую функцию печи — готовить закуски. При этом вас не должно беспокоить, что у нее внутри, кто ее сделал и как продавец сумел-таки всучить ее вашей жене по такой цене на распродаже... Хотя нет, именно последнее наверняка беспокоит вас больше всего...

Для обычного потребителя отличия микроволновой печи от обычной не так важны — лишь бы они обе могли готовить любимые блюда, но если взглянуть на это со точки зрения печи, то эти отличия становятся крайне существенны, поскольку внутреннее устройство печей совершенно различно. Мощь наследования заключается в том факте, что подкласс не обязан наследовать каждый метод базового класса в том виде, в котором он написан. Подкласс может наследовать суть метода базового класса при полном отличии его реализации.

Перегрузка унаследованного метода

Как описывалось в главе 7, "Функции функций", две или большее число функций могут иметь одинаковые имена — лишь бы отличались количества и/или типы их аргументов.

Простейший случай перегрузки функции



Две функции с одинаковым именем называются *перегруженными*.

Аргументы функции становятся частью ее расширенного имени (используемого в interne), как показано в следующем фрагменте исходного текста:

```
public class MyClass
{
    public static void AFunctionO
    } // Некоторые действия

    public static void AFunction(int)
    } // Некоторые другие действия

    public static void AFunction(double d)
    } // Некоторые действия, отличные от первых двух

    public static void Main(string[] args)
    {
        AFunction();
        AFunction(1);
        AFunction(2.0);
    }
}
```

С# в состоянии различать эти методы по их аргументам. Каждый из вызовов в функции Main () обращается к своей функции.



Возвращаемый тип не является частью расширенного имени функции, так что вы не можете иметь две функции, отличающиеся только типами возвращаемой значения.

Различные классы, различные методы

Не удивительно, что класс, которому принадлежит функция, также становится частью ее расширенного имени. Рассмотрим следующий фрагмент исходного текста:

```
public class MyClass
{
    public static void AFunctionO;
    public void AMethod();
}

public class UrClass
{
    public static void AFunctionO;
    public void AMethod();
}

public class Program
```



```

public static void Main(string[] args)
{
    UrClass.AFunction(); // Вызов статической функции
    // Вызов функции-члена MyClass.AMethod()
    MyClass mcObject = new MyClass();
    mcObject.AMethod();
}

```

Имя класса является частью расширенного имени функции, так что для C# очевидно, какую именно функцию `AFunction()` или метод `AMethod()` вызывать в каждом конкретном случае.

Сокрытие метода базового класса

Итак, метод одного класса может перегружать другой метод того же класса, если использует другие аргументы. Метод также может перегружать метод базового класса. Перегрузка метода базового класса известна как *сокрытие* метода.

Предположим, ваш банк проводит новую политику, в соответствии с которой снятие депозитного счета отличается от других типов снятия со счета. Предположим для конкретности, что каждое снятие со счета обходится вкладчику в сумму 1.50.

При использовании функционального подхода вы можете реализовать эту политику посредством переменной-флага в классе, который бы указывал, принадлежит ли объекту типу `SavingsAccount` или `BankAccount`. В этом случае метод снятия со счета должен проверять значение флага, чтобы выяснить, следует ли снимать дополнительные .50, как показано в следующем фрагменте исходного текста,

```

public class BankAccount
{
    private decimal mBalance;
    private bool isSavingsAccount;
    // Начальный баланс и флаг, указывающий, является ли счет
    // депозитным или нет
    public BankAccount(decimal mInitialBalance,
                        bool isSavingsAccount)
    {
        mBalance = mInitialBalance;
        this.isSavingsAccount = isSavingsAccount;
    }
    public decimal Withdraw(decimal mAmount)
    {
        // Если счет депозитный . . .
        if (isSavingsAccount)
        {
            // ...снимаем лишние 1.50
            mBalance -= 1.50M;
        }
        // Далее обычный код снятия со счета
        if (mAmountToWithdraw > mBalance)
        {
            mAmountToWithdraw = mBalance;
        }
    }
}

```

```

        mBalance -= mAmountToWithdraw;
        return mAmountToWithdraw;
    }
}

class MyClass
{
    public void SomeFunction()
    {
        // Создаем депозитный счет
        BankAccount ba = new BankAccount(0, true);
    }
}

```

Ваша функция должна указывать, какой именно счет создается, путем передачи дополнительного аргумента конструктору BankAccount. Конструктор сохраняет файл затем используемый в методе Withdraw () Для снятия дополнительной суммы 1.50.



Объектно-ориентированный подход скрывает метод Withdraw () базом го класса BankAccount новым методом с тем же именем в классе SavingsAccount, как показано в приведенной далее демонстрационной программе.

```

// HidingWithdrawal - сокрытие метода базового класса
// методом подкласса с тем же именем
using System;

```

```

namespace HidingWithdrawal
{
    // BankAccount - базовый банковский счет
    public class BankAccount
    {
        protected decimal mBalance;

        public BankAccount(decimal mInitialBalance)
        {
            mBalance = mInitialBalance;
        }

        public decimal Balance
        {
            get { return mBalance; }
        }

        public decimal Withdraw(decimal mAmount)
        {
            decimal mAmountToWithdraw = mAmount;
            if (mAmountToWithdraw > Balance)
            {
                mAmountToWithdraw = Balance;
            }
            mBalance -= mAmountToWithdraw;
            return mAmountToWithdraw;
        }
    }
}

```

```

// SavingsAccount - банковский счет с начислением
// процентов
public class SavingsAccount : BankAccount
{
    public decimal mInterestRate;
    // SavingsAccount - процентная ставка передается как
    // число от 0 до 100
    public SavingsAccount(decimal mInitialBalance,
                           decimal mInterestRate)
        : base(mInitialBalance)
    {
        this.mInterestRate = mInterestRate / 100;
    }

    // AccumulateInterest - начисление процентов
    public void AccumulateInterest()
    {
        mBalance = Balance + (Balance * mInterestRate);
    }

    // Withdraw - со счета можно снять любую сумму, не
    // превышающую баланс; функция возвращает снятую сумму
    public decimal Withdraw(decimal mWithdrawal)
    {
        // Дополнительное снятие 1.50
        base.Withdraw(1.5M);

        // Теперь снимаем со счета как обычно
        return base.Withdraw(mWithdrawal);
    }
}

public class Program
{
    public static void MakeAWithdrawal(BankAccount ba,
                                        decimal mAmount)
    {
        ba.Withdraw(mAmount);
    }

    public static void Main(string[] args)
    {
        BankAccount ba;
        SavingsAccount sa;

        // Создаем банковский счет, снимаем 100, выводим
        // результат
        ba = new BankAccount(200M);
        ba.Withdraw(100M);

        // Делаем то же с депозитным счетом
        sa = new SavingsAccount(200M, 12);
        sa.Withdraw(100M);
    }
}

```

```
// Выводим состояния счетов
Console.WriteLine("Баланс BankAccount равен {0:C}",
    ba.Balance);
Console.WriteLine("Баланс SavingsAccount равен {0:C}",
    sa.Balance);

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
```



В этом случае функция `Main()` создает объект `BankAccount` с начальным балансом 200 и снимает с него 100. Затем те же действия выполняются с объектом `SavingsAccount`. Когда функция `Main()` снимает деньги со счета базового класса, метод `BankAccount.Withdraw()` снимает только указанную сумму (но не более суммы на счету). Когда же функция `Main()` снимает деньги с депозитного счета, метод `SavingsAccount.Withdraw()` снимает дополнительную сумму, равную 1.50.



Обратите внимание, что метод `SavingsAccount.Withdraw()` использует метод базового класса `BankAccount.Withdraw()`, а не работает непосредственно с балансом. Если это возможно — пусть базовый класс сам работает своими членами-данными.

Чем сокрытие лучше проверки флага

На первый взгляд добавление флага в метод `BankAccount.Withdraw()` представляется более простым решением, чем предложенный вариант с сокрытием метода базового класса. В конце концов, использование флага потребовало добавления всего лишь четырех строк.

Однако простое решение порождает массу проблем. Первая заключается в том, что класс `BankAccount` не должен беспокоиться о деталях работы `SavingsAccount`. Говоря формально, это нарушает принцип инкапсуляции. Базовый класс не должен ничего знать о своих потомках.

Все сказанное приводит ко второй, более сложной проблеме. Предположим, банки по очереди вводят новые счета — например, `CheckingAccount`, `CDAccount`, `TBillAccount`. У каждого из них — свои правила снятия денег со счета и каждый использует свой собственный флаг. После трех-четырех добавлений новых типов счетов старый метод `BankAccount.Withdraw()` начинает выглядеть слишком сложным. Каждый новый вид счета приводит, ко все большим изменениям этого метода.

Такое решение совершенно не подходит. Классы должны отвечать сами за себя.

Случайное сокрытие метода базового класса

Метод базового класса может оказаться скрытым случайно. Пусть, например, имеется метод `Vehicle.TakeOff()`, который начинает движение транспортного средства. Пусть же кто-то может расширить класс `Vehicle`, создав класс `Airplane`. Понятно, что метод `TakeOff()` этого класса совершенно не тот же, что у класса `Vehicle`. Очевидно, что это случай ложной тождественности — два метода не имеют ничего общего, кроме имени.

К счастью, C# в состоянии обнаружить такую проблему.

C# генерирует зловещего вида предупреждение при компиляции рассматривавшейся ранее демонстрационной программы `HidingWithdrawal`. Из всего длинного текста предупреждения интерес представляет только небольшая его часть, а именно:

```
1.. SavingsAccount.Withdraw(decimal) ' hides inherited member
   '...BankAccount.Withdraw(decimal)'. Use the new
   keyword if hiding was intended.
```

C# пытается сообщить, что вы написали метод подкласса, который имеет то же имя, что и метод базового класса. Действительно ли вы хотите именно этого?



Это всего лишь предупреждение. Вы можете и не реагировать на него, но все же крайне желательно ознакомиться со всеми предупреждениями, выводимыми компилятором, и избавиться от них. Предупреждение почти всегда говорит о какой-то мелочи, которая может перерасти в крупные неприятности, если вовремя о ней не позаботиться.



Неплохо дать указание компилятору C# рассматривать **все предупреждения** как, ошибки, по крайней мере на этапе отладки. Для этого следует воспользоваться командой меню **Projects Properties** и прокрутить панель **Build** страницы свойств проекта до раздела **Errors and Warnings**. Установите значение параметра **Warning Level** равным 4, наивысшей возможной величине. Кроме того, в подразделе **Treat Warnings as Errors** выберите флаг **ALL**. При этом при работе над программой вы будете вынуждены устранять все предупреждения так же, как устраняете реальные ошибки. Даже если вы не будете заставлять компилятор считать предупреждения ошибками, все равно тщательно просматривайте весь список предупреждений после каждой сборки программы.

Описатель `new`, упомянутый в предупреждении и показанный в приведенном далее фрагменте исходного текста, говорит компилятору C# о том, что сокрытие метода преднамеренное (тем самым предупреждение устраняется).

```
// Теперь с Withdraw() никаких проблем
new public decimal Withdraw(decimal mWithdrawal)
{
    // ... Никаких иных изменений не требуется ...
}
```



Такое использование ключевого слова `new` не имеет ничего общего с его применением для создания объекта.

Вызов методов базового класса

Вернемся к методу `SavingsAccount.Withdraw()` из демонстрационной программы `HidingWithdrawal`, рассматривавшейся ранее в этой главе. Вызов `BankAccount.Withdraw()` из этого нового метода осуществляется при помощи ключевого слова `base`.

Приведенная далее версия функции без ключевого слова `base` работать не будет:

```
new public decimal Withdraw(decimal mWithdrawal)
{
    decimal mAmountWithdrawn = Withdraw(mWithdrawal);
```

```

    mAmountWithdrawn += Withdraw(1.5);
    return mAmountWithdrawn;
}

```

В этом случае возникает та же проблема, что и в следующем фрагменте:

```

void fn()
{
    fn(); // Вызов функцией самой себя
}

```

Вызов `fn()` из `fn()` приводит к *рекурсивному* вызову функцией самой себя. Аналогично, такой вызов `Withdraw()`, как показано в фрагменте выше, приводит к ~~вызову~~ функцией самой себя, пока программа в конечном счете не завершится аварийно.

Требуется указать C#, что в методе `SavingsAccount.Withdraw()` следует *звать* метод `BankAccount.Withdraw()`. Один из вариантов решения поставленной задачи состоит в преобразовании указателя `this` в указатель на объект `BankAccount` перед выполнением вызова:

```

// Withdraw - эта версия обращается к сокрытому методу
// базового класса посредством явного преобразования this
new public decimal Withdraw(decimal mWithdrawal)
{
    // Преобразование указателя this в объект класса
    // BankAccount
    BankAccount ba = (BankAccount)this;
    // Вызов Withdraw() с использованием объекта BankAccount
    decimal mAmountWithdrawn = ba.Withdraw(mWithdrawal);
    mAmountWithdrawn += ba.Withdraw(1.5);
    return mAmountWithdrawn;
}

```

Данное решение вполне работоспособно: вызов `ba.Withdraw()` вызывает метод класса `BankAccount`. Однако в будущем изменение программы может привести к изменению иерархии классов, что `SavingsAccount` не будет непосредственным потомком `BankAccount`. Подобная модификация приведет к неверной работе функции, найти причину которой будет нелегко.

Необходим способ пояснить C#, что требуется вызвать функцию `Withdraw()` из класса, являющегося непосредственным предшественником текущего — причем без *явной* именования этого класса. Для этой цели в C# служит ключевое слово `base`.



Это то же ключевое слово `base`, которое конструктор использует для передачи аргумента конструктору базового класса.

Ключевое слово C# `base` в показанном далее фрагменте кода представляет собой то же, что и `this`, но приведение к базовому классу выполняется независимо от того, кап именно класс является таковым.

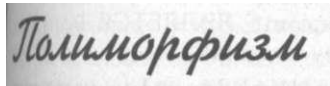
```

// Withdraw - можно снимать любую сумму в пределах баланса;
// возвращает снятую со счета сумму
new public decimal Withdraw(decimal mWithdrawal)
{
    // Снятие дополнительной суммы 1.50
    base.Withdraw(1.5M);
}

```

```
// Снятие со счета с оставшейся суммой
return base . Withdraw (mWithdrawal) ;
}
```

Вызов `base . Withdraw ()` приводит к вызову метода `BankAccount . Withdraw ()`; и самым проблема, связанная с рекурсией, снимается. Кроме того, данное решение работает и при изменении иерархии наследования.



Можно перегрузить метод базового класса методом в подклассе. Это и замечательно, одновременно очень опасно.

Проведем мысленный эксперимент: когда должно приниматься решение о том, какой из методов — `BankAccount . Withdraw ()` или `SavingsAccount . Withdraw ()` — будет вызван: во время компиляции или во время выполнения программы?



Для того чтобы понять, в чем здесь отличие, будет немного изменена рассматривавшаяся ранее программа `HidingWithdrawal` (здесь приведена только та часть, в которую внесены изменения).

```
// HidingWithdrawalPolymorphically - сокрытие метода
// Withdraw () базового класса методом с тем же именем в
// подклассе
public class Program
{
    public static void MakeAWithdrawal(BankAccount ba,
                                       decimal mAmount)
    {
        ba.Withdraw(mAmount);
    }
}

public static void Main(string[] args)
{
    BankAccount ba;
    SavingsAccount sa;
    ba = new BankAccount(200M);
    MakeAWithdrawal(ba, 100M);
    sa = new SavingsAccount(200M, 12);
    MakeAWithdrawal(sa, 100M);
    // Выводим состояния счетов
    Console.WriteLine("Баланс BankAccount равен {0:C}",
                      ba.Balance);
    Console.WriteLine("Баланс SavingsAccount равен {0:C}",
                      sa.Balance);
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}
```

Вывод этой демонстрационной программы на экран может вас удивить (а может и не удивить — в зависимости от того, чего именно вы ожидали):

```
Баланс BankAccount равен $100.00
Баланс SavingsAccount равен $100.00
Нажмите <Enter> для завершения программы...
```

В этот раз, вместо снятия со счета в функции `Main()`, программа передает объект счета функции `MakeAWithdrawal()`.

Первый вопрос очевиден: как функция `MakeAWithdrawal()` может принимать объект `SavingsAccount`, если она ожидает в качестве аргумента объект `BankAccount`? Ответ не менее ясен: потому что `SavingsAccount` ЯВЛЯЕТСЯ `BankAccount` (см. главу 12, "Наследование").

Второй вопрос не так очевиден. Когда функции `MakeAWithdrawal()` передает! объект `BankAccount`, она вызывает `BankAccount.Withdraw()` — это понятно. Не когда передается объект типа `SavingsAccount`, вызывается тот же метод. Должен ли в этом случае вызываться метод `Withdraw()` подкласса?

С одной стороны, поскольку объект `ba` принадлежит типу `BankAccount`, вызов `ba.Withdraw()` должен вызывать метод `BankAccount.Withdraw()`. С другой стороны, хотя объект `ba` и объявлен как `BankAccount`, фактически он представляет собой объект `SavingsAccount`, так что должен быть вызван метод `SavingsAccount.Withdraw()`. Оба аргумента достаточно логичны.

В данном случае C# принимает как более весомый первый аргумент. Это более безопасный выбор — работать с объявленным типом — поскольку он устраняет все недоразумения. Объект объявлен как `BankAccount`, и так тому и быть.

Что неверно в стратегии использования объявленного типа

В ряде случаев вам не требуется работа с объявленным типом. На самом деле необходимо, чтобы вызов базировался на *реальном типе*, т.е. на типе времени исполнения, а не на объявленном типе. Например, вам нужно, чтобы выполнялись действия со счетом типа `SavingsAccount`, который хранится в переменной типа `BankAccount`. Такая возможность принятия решения во время выполнения программы называется *полиморфизмом*, или *поздним связыванием* (late binding). Стратегия использования объявленного типа называется *ранним связыванием* (early binding), в противоположность позднему.



Термин *полиморфизм* происходит из греческого языка: *поли* означает много, а *морф* — форма, или действие.

Полиморфизм и позднее связывание вообще-то не одно и то же. Однако их отличие весьма тонкое. *Полиморфизм* означает возможность принятия решения о том, какой метод должен быть вызван в процессе выполнения программы. *Позднее связывание* — способ реализации полиморфизма языком программирования.

Полиморфизм является ключевой составляющей частью объектно-ориентированного программирования. Он настолько важен, что языки, его не поддерживающие, не имеют права называться объектно-ориентированными.



Языки программирования, поддерживающие классы, но не поддерживающие полиморфизм, называются *объектно-основанными языками* (object-based languages). Примером такого языка может служить язык Ada.

Без полиморфизма в наследовании мало толку. Позвольте привести наглядный пример, иллюстрирующий данный тезис. Предположим, вы написали мощную программу, использующую класс... ну, скажем, `Student`. После нескольких месяцев проектирования, кодирования и тестирования вы наконец-то вынесли ее на суд восхищенных пользователей (начинающих даже поговаривать, что совершенно напрасно не существует Нобелевской премии в области программирования).

Проходит время, и ваш шеф требует, чтобы вы добавили в программу возможность работы с аспирантами, которые, конечно, похожи на студентов, но все же немного отличаются от них (сами аспиранты считают, что они отличаются во всем). Предположим, что формулы для вычисления оплаты за обучение для студентов и аспирантов совершенно различны. Вашему боссу это безразлично, но в программе имеется масса вызовов функции `CalcTuition()`, являющейся предназначенной для таких расчетов. Вот пример одного из таких вызовов:

```
void SomeFunction(Student s)
{
    // . . . Какие-то действия . . .
    s.CalcTuition();
    //... продолжение ...
}
```

Если бы C# не поддерживал позднее связывание, то вам бы пришлось редактировать функцию `SomeFunction()`, чтобы проверять в ней, является ли переданный объект `s` переменной типа `Student` или `GraduateStudent`. Программа должна была бы вызывать `Student.CalcTuition()` в случае, когда переменная `s` принадлежала бы классу `Student`, и `GraduateStudent.CalcTuition()` в случае класса `GraduateStudent`.

Это было бы не так страшно, если бы не две вещи.

- ✓ Это только одна функция. А теперь представьте, что `CalcTuition()` вызывается в сотнях мест...
- ✓ Предположим, что `CalcTuition()` — не единственное отличие между двумя классами. Шансы, что вы найдете все места в программе, требующие изменений, резко снижаются...

При поддержке полиморфизма вы просто позволяете C# самостоятельно решить, какой метод должен быть вызван.

Использование `is` для полиморфного доступа к скрытому методу

Каким образом сделать программу полиморфной? Один из подходов для решения этой задачи в C# состоит в использовании ключевого слова `is` (о котором рассказывается в главе 12, "Наследование"). Выражение `ba is SavingsAccount` возвращает значение `true` или `false` в зависимости от класса объекта во время выполнения программы. Объявленный тип может быть `BankAccount`, но с какого типа объектом приходится иметь дело в реальности? В приведенном далее фрагменте исходного текста `is` используется для обращения к функции `Withdraw()` класса `SavingsAccount`.

```
public class Program

{
    public static void MakeAWithdrawal(BankAccount ba,
                                       decimal mAmount)
```

```

{
    if ba is SavingsAccount
    {
        SavingsAccount sa = (SavingsAccount)ba,
        sa.Withdraw(mAmount);
    } else
    {
        ba.Withdraw(mAmount);
    }
}

```

Теперь, когда `Main()` передает функции объект типа `SavingsAccount`, функции `MakeAWithdrawal()` проверяет тип времени выполнения объекта `ba` и вызывает функцию `SavingsAccount.Withdraw()`.



Программист может выполнить вызов одной строкой:

```
((SavingsAccount)ba).Withdraw(mAmount);
```

Об этом упоминается только потому, что имеется масса программ, в которых вызовы осуществляются именно таким образом. Однако данный подход приводит к менее удобочитаемому исходному тексту, соответственно — к большому количеству ошибок в программе.

Подход с использованием `is` вполне работоспособен, но это далеко не лучшая идея. Применение `is` требует от функции `MakeAWithdrawal()` осведомленности о всех возможных типах счетов, которые имеются (и могут появиться в дальнейшем) в банке. Это накладывает на функцию `MakeAWithdrawal()` слишком большую ответственность. Да, сейчас ваше приложение обходится двумя классами, но завтра от вас могут потребовать реализовать новый вид счета, например, `CheckingAccount`, и вы будете вынуждены перерывать всю программу в поисках мест, в которые надо внести добавления связанные с проверкой типа аргумента функции в процессе выполнения программы.

Объявление метода виртуальным

В качестве автора функции `MakeAWithdrawal()` вы бы, конечно, не хотели делать ее осведомленной о всех возможных типах счетов. Хотелось бы предоставить это программисту, использующему функцию `MakeAWithdrawal()`, т.е. заставить `C#` саму, принимать решение о том, какой метод должен быть вызван,

основываясь на информации о типе объекта времени выполнения программы.

Можно заставить `C#` самостоятельно принимать решение о версии `Withdraw()` которую следует вызвать. Для этого необходимо пометить функцию базового класса при помощи ключевого слова `virtual`, а каждую версию функции в подклассах — ключевым словом `override`.



Предыдущая демонстрационная программа переписана с использованием полиморфизма, при этом в каждый из методов `Withdraw()` добавлен вод строки, чтобы было точно видно, какого класса метод вызывается в или ином случае.

```
// PolymorphicInheritance - полиморфное сокрытие метода
// базового класса
```

```

using System;

namespace PolymorphicInheritance
{
    // BankAccount - простейший банковский счет
    public class BankAccount
    {
        protected decimal mBalance;

        public BankAccount(decimal mInitialBalance)
        {
            mBalance = mInitialBalance;
        }

        public decimal Balance
        {
            get { return mBalance; }
        }

        public virtual decimal Withdraw(decimal mAmount)
        {
            Console.WriteLine("BankAccount.Withdraw() с ${0}...",
                               mAmount);
            decimal mAmountToWithdraw = mAmount;
            if (mAmountToWithdraw > Balance)
            {
                mAmountToWithdraw = Balance;
            }
            mBalance -= mAmountToWithdraw;
            return mAmountToWithdraw;
        }
    }

    // SavingsAccount - банковский счет с начислением
    // процентов
    public class SavingsAccount : BankAccount
    {
        public decimal mInterestRate;

        // SavingsAccount - процентная ставка указывается как
        // число от 0 до 100
        public SavingsAccount(decimal mInitialBalance,
                               decimal mInterestRate)
            : base(mInitialBalance)
        {
            this.mInterestRate = mInterestRate / 100;
        }

        // AccumulateInterest - начисление процентов
        public void AccumulateInterest()
        {
            mBalance = Balance + (Balance * mInterestRate);
        }

        // Withdraw - снятие со счета произвольной суммы, не

```

```

// превышающей имеющейся на счету; возвращает снятую
// сумму
override public decimal Withdraw(decimal mWithdrawal)
{
    Console.WriteLine("SavingsAccount.Withdraw()...");
    Console.WriteLine("Вызов функции Withdraw базового " +
        "класса дважды. ...");

    // Снятие 1.50
    base.Withdraw(1.5M);

    // Снятие в пределах оставшейся суммы
    return base.Withdraw(mWithdrawal);
}
}

public class Program
{
    public static void MakeAWithdrawal(BankAccount ba,
        decimal mAmount)
    {
        ba.Withdraw(mAmount);
    }

    public static void Main(string[] args)

        BankAccount ba;
        SavingsAccount sa;

        // Вывод баланса
        Console.WriteLine("MakeAWithdrawal(ba, ... )");
        ba = new BankAccount(200M) ;;
        MakeAWithdrawal(ba, 100M);
        Console.WriteLine("Баланс BankAccount равен {0:C}",
            ba.Balance);
        Console.WriteLine("MakeAWithdrawal(sa, ... )");
        sa = new SavingsAccount(200M, 12);
        MakeAWithdrawal(sa, 100M);
        Console.WriteLine("Баланс SavingsAccount равен {0:C}",
            sa.Balance);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");

        Console.Read();
    }
}
}

```

Вывод программы имеет следующий вид:

```

MakeAWithdrawal(ba, ... )
BankAccount.Withdraw() с $100...
Баланс BankAccount равен $100.00
MakeAWithdrawal(sa, ... )
SavingsAccount.Withdraw()...

```

```

Вызов функции Withdraw базового класса дважды...
BankAccount.Withdraw() с $1.5...
BankAccount.Withdraw() с $100...
Баланс SavingsAccount равен $98.50
Нажмите <Enter> для завершения программы...

```

Метод `Withdraw()` помечен в базовом классе `BankAccount` как `virtual`, в то время как в подклассе он помечен как `override`. Метод `MakeAWithdrawal()` остается без изменений, и вывод при его вызове различен из-за того, что разрешение вызова `ba.Withdraw()` осуществляется на основании типа `ba` во время выполнения программы.



Для полного понимания того, как это работает, желательно пошагово пройти программу в отладчике Visual Studio 2005. Для этого соберите программу как обычно, а затем нажимайте клавишу <F11> для пошагового ее выполнения. Это достаточно впечатляющее зрелище, когда один и тот же вызов приводит в разные моменты к двум разным методам.



Будьте экономны при объявлении методов виртуальными. Все имеет свою цену, так что используйте ключевое слово `virtual` только при необходимости.

Абстракционизм в C#

Утка — вид птицы. Так же как воробей или колибри. Любая птица представляет какой-то подвид птиц. Но обратная сторона медали в том, что нет птицы, которая была бы птицей вообще. С точки зрения программирования это означает, что все объекты `Bird` являются экземплярами каких-то подклассов `Bird`, но не имеется ни одного экземпляра класса `Bird`. Так что же такое птица? Это всегда какой-то конкретный вид — пингвин, курица или, к примеру, страус.

Различные типы птиц имеют множество общих свойств (в противном случае они бы не были птицами), но нет двух типов, у которых бы общими были все свойства. Если бы такие типы были, они были бы одинаковыми типами, ничем не отличающимися друг от друга.

Разложение классов

Люди систематизируют объекты, выделяя их общие черты. Чтобы увидеть, как это работает, рассмотрим два класса — `HighSchool` и `University`, показанные на рис. 13.1. Здесь для описания классов использован Унифицированный Язык Моделирования (Unified Modeling Language, UML), графический язык, описывающий классы и их взаимоотношения друг с другом.



Помните — машина ЯВЛЯЕТСЯ транспортным средством, но СОДЕРЖИТ мотор.

[†] Как видно на рис. 13.1, у школы и университета много общих свойств. И у школы, и у университета имеется открытый метод `Enroll()` для добавления объекта `Student` (зачисления в учебное заведение). Оба класса имеют закрытый член `numStudents`, в котором хранится число учащихся. Еще одно общее свойство — взаимоотношения учащихся и учебных заведений: в учебном заведении может быть много учащихся, в то время

как один учащийся учится одновременно только в одном учебном заведении. Само собой, имеется масса других свойств учебных заведений, но для данного рассмотрения ограничимся перечисленным.

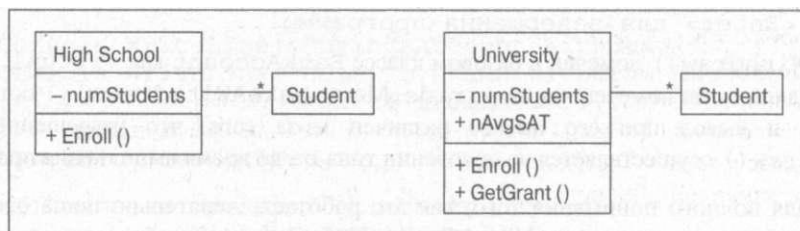


Рис. 13.1. UML-описание классов HighSchool и University

В дополнение к свойствам школы университет содержит метод GetGrant () и член-данные nAvgSAT.

Рис. 13.1 корректно отображает ситуацию, но большая часть информации дублируется. Уменьшить дублирование можно, если позволить классу University унаследовать более простой класс HighSchool, как показано на рис. 13.2.

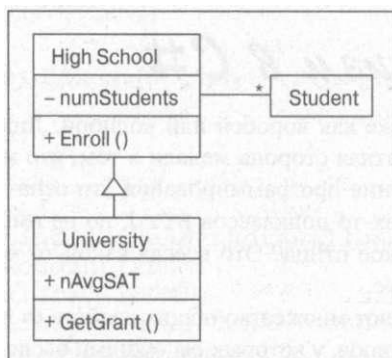


Рис. 13.2. Наследование класса HighSchool упрощает класс University, но привносит определенные проблемы

Класс HighSchool остается неизменным, но класс University при этом проще описать. Можно сказать, что University — это класс HighSchool с членом nAvgSAT и методом GetGrant (). Однако такое решение имеет одну фундаментальную проблему — университет вовсе не школа со специальными свойствами.

Вы можете сказать: "Ну и что? Главное, что наследование работает и экономит наши усилия". Да, конечно, это так, но сказанное выше — не просто стилистическая тривиальность. Такое неверное представление может ввести в заблуждение программиста как сейчас, так и в будущем. В один прекрасный день ему, незнакомому с вашими фокусами, придется читать и разбираться в ваших исходных текстах, и такое неверное представление может привести к неправильному пониманию программы.

Кроме того, неверное представление может привести к реальным проблемам. Предположим, что в школе решили выбирать лучшего ученика, и для этого програм-

мист просто добавляет в класс `HighSchool` метод `NameFavorite()`, указывающий имя такого ученика.

И вот — проблема. В университете не намерены определять лучшего студента, но метод `NameFavorite()` оказывается унаследованным. Это может показаться небольшой проблемой — в конце концов, этот метод в классе `University` можно просто игнорировать.

Да, один лишний метод не делает погоды, но это еще один кирпич в стене непонимания. Постепенно лишние члены-данные и методы накапливаются, и наступает момент, когда ваш класс уже не в состоянии вынести такой багаж. Несчастный программист уже не понимает, какие методы "реальны", а какие — нет.



UML Lite

Унифицированный Язык Моделирования (Unified Modeling Language, UML) представляет собой выразительный язык, способный ясно определять взаимоотношения объектов в программе. Одно из достоинств UML заключается в том, что вы можете не зависеть от конкретного языка программирования.

Ниже перечислены основные свойства UML.

- ✓ Классы представлены прямоугольниками, разделенными по вертикали на три части. Имя класса указывается в верхней части прямоугольника.
- ✓ Члены-данные класса находятся в средней части, а методы — в нижней. Можно опустить среднюю или нижнюю часть прямоугольника, если в классе нет членов-данных или методов.
- ✓ Члены со знаком плюс (+) перед именем являются открытыми, со знаком минус (-) — закрытыми. В UML отсутствует специальный знак для защищенных членов, но некоторые программисты используют для обозначения таких членов символ #. Закрытые члены доступны только для других членов того же класса; открытые члены доступны всем классам.
- ✓ Метка `{abstract}` после имени указывает абстрактный класс или метод. На самом деле UML использует для этого иное обозначение, но так мне кажется проще. Ведь мы имеем дело с упрощенной версией — UML Lite.
- ✓ Стрелка между двумя классами представляет отношение между ними. Число над линией означает мощность — сколько элементов может быть с каждого конца стрелки. Звездочка (*) означает *произвольное число*. Если число опущено, по умолчанию предполагается значение 1. Таким образом, на рис. 13.1 видно, что один университет может иметь сколько угодно студентов — они связаны отношением один-ко-многим.
- ✓ Линия с большой открытой или треугольной стрелкой на конце выражает отношение ЯВЛЯЕТСЯ (наследование). Стрелка указывает в иерархии классов на базовый класс. Другие типы взаимоотношений включают отношение СОДЕРЖИТ, которое указывается линией с закрашенным ромбиком со стороны владельца.

Имеются и другие проблемы. При наследовании, показанном на рис. 13.2, как видно из схемы, классы `University` и `HighSchool` имеют одну и ту же процедуру зачисления. Как бы странно это ни звучало, будем считать, что это так и есть. Программа разработана, упакована и отправлена потребителям.

Несколькими месяцами позже министерство просвещения решает изменить правила зачисления в школы, что, в свою очередь, приводит к изменению процедуры зачисления и в университеты, что, конечно же, неверно.

Как избежать указанной проблемы? Понятно, что ее корень — в отношениях класса Университет не является школой. Отношение СОДЕРЖИТ также не будет работать — ну же! университет содержит школу или школа — университет? Конечно же, нет. Решение заключается в том, что и школа, и университет — это специальные типы учебных заведений.

На рис. 13.3 показано более корректное решение. Новый класс `School` содержит общие свойства двух типов учебных заведений, включая отношения с объектами `student`. Более того, класс `School` даже имеет метод `Enroll()`, хотя он и абстрактный! поскольку и `University`, и `HighSchool` реализуют его по-разному.

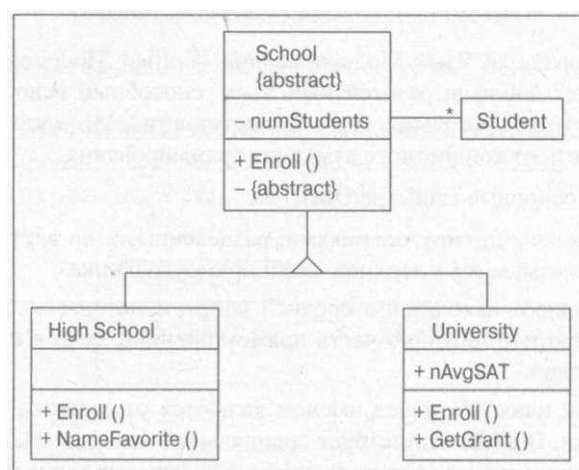


Рис. 13.3. Классы `University` и `HighSchool` должны иметь общий базовый класс `School`

Теперь классы `University` и `HighSchool` наследуют общий базовый класс. Каждый из них содержит свои уникальные члены: `HighSchool` — `NameFavorite()`, а `University` — `GetGrant()`. Кроме того, оба класса перекрывают метод `Enroll()`, описывающий правила зачисления учащихся в разные учебные заведения. По сути, здесь выделено общее путем создания базового класса из двух схожих классов, которые после этого стали подклассами.

Введение класса `School` имеет как минимум два больших преимущества.

- ✓ **Это соответствует реальности.** Университет является учебным заведением, но не школой. Соответствие действительности — важное, но не главное преимущество.
- ✓ **Это изолирует один класс от изменений или дополнений в другой класс.** Если потребуется, например, внести добавления в класс `University`, то его новые методы никак не повлияют на класс `HighSchool`.

Процесс выделения общих свойств из схожих классов называется *разложением* классов (*factoring*). Это важное свойство объектно-ориентированных языков программирования как по описанным выше причинам, так и с точки зрения снижения избыточности.



Разложение корректно только в том случае, когда отношения наследования соответствуют действительности. Можно выделять общие свойства классов `Mouse` и `Joystick`, поскольку оба они представляют собой указательные устройства, но делать то же для классов `Mouse` и `Display` будет ошибкой.

Разложение обычно приводит к нескольким уровням абстракции. Например, программа, охватывающая более широкий круг школ, может иметь структуру классов, показанную на рис. 13.4.

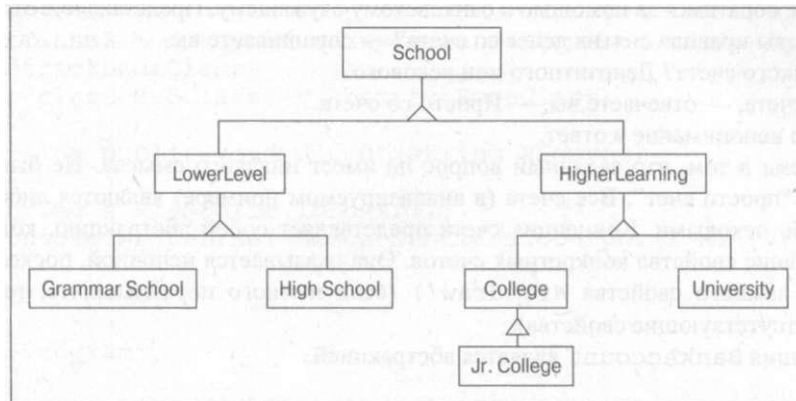


Рис. 13.4. Разложение классов обычно дает дополнительные уровни в иерархии наследования

Как видите, внесено два новых класса между `University` и `School`: `HigherLearning` и `LowerLevel`. Например, новый класс `HigherLearning` делится на классы `College` и `University`. Такая многослойная иерархия — обычное и даже желательное явление при разложении, соответствующем реальному миру.

Заметим, однако, что никакой теории разложения классов не существует. Так, разложение на рис. 13.4 можно считать вполне корректным, но если программа в большей степени связана с вопросами администрирования учебных заведений местными властями, то более естественной будет иерархия классов, представленная на рис. 13.5.

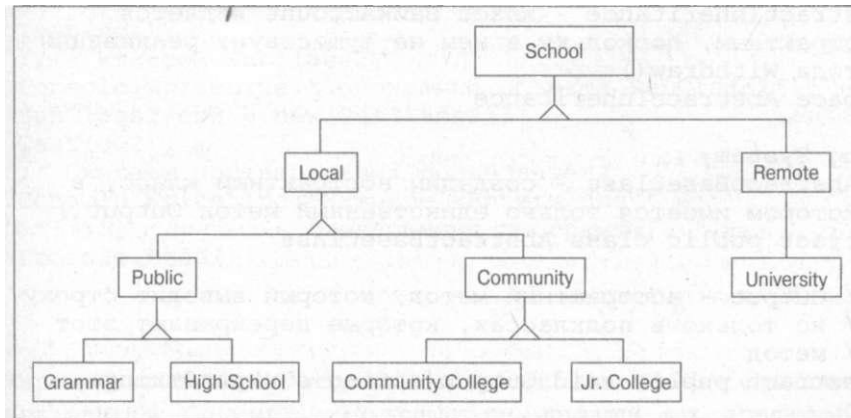


Рис. 13.5. Разложение классов зависит от решаемой задачи

Голая концепция, выражаемая абстрактным классом

Вернемся в очередной раз к классу `BankAccount`.

Большинство методов этого класса не вызывают проблем, поскольку оба типа банковских счетов одинаково их реализуют. Однако правила снятия со счета оказываются различными, так что вы должны реализовать `SavingsAccount.Withdraw()` не так, как `CheckingAccount.Withdraw()`. Но как вы предполагаете реализовать `BankAccount.Withdraw()`?

Давайте обратимся за помощью к банковскому служащему. Представляете этот диалог?

— Каковы правила снятия денег со счета? — спрашиваете вы.

— С какого счета? Депозитного или чекового?

— Со счета, — отвечаете вы. — Просто со счета.

Полное непонимание в ответ.

Проблема в том, что заданный вопрос не имеет никакого смысла. Не бывает такой вещи, как "просто счет". Все счета (в анализируемом примере) являются либо депозитными, либо чековыми. Концепция счета представляет собой абстракцию, которая объединяет общие свойства конкретных счетов. Она оказывается неполной, поскольку в ней недостает важного свойства `Withdraw()` (если немного поразмышлять, то найдутся и другие отсутствующие свойства).

Концепция `BankAccount` является абстракцией.

Как использовать абстрактные классы

Абстрактные классы используются для описания абстрактных концепций.

Абстрактный класс — это класс с одним или несколькими абстрактными методами. Наверное, это не слишком прояснило ситуацию? Тогда вот дополнительное пояснение: абстрактный метод — это метод, описанный при помощи ключевого слова `abstract`. Ничуть не легче? Тогда следующее добьет вас окончательно: абстрактный метод не имеет реализации.



Теперь рассмотрим урезанную демонстрационную программу.

```
// AbstractInheritance - класс BankAccount является
// абстрактным, поскольку в нем не существует реализации
// метода Withdraw()
namespace AbstractInheritance
{
    using System;
    // AbstractBaseClass - создадим абстрактный класс, в
    // котором имеется только единственный метод Output()
    abstract public class AbstractBaseClass
    {
        // Output - абстрактный метод, который выводит строку,
        // но только в подклассах, которые перекрывают этот
        // метод
        abstract public void Output(string sOutputString);
    }
    // SubClass1 - первая конкретная реализация класса
```

```

// AbstractBaseClass
public class SubClass1 : AbstractBaseClass
{
    override public void Output(string sSource)
    {
        string s = sSource.ToUpper();
        Console.WriteLine("Вызов SubClass1.Output() из {0}",
                           s) ;
    }
}
// SubClass2 - еще одна конкретная реализация класса
// AbstractBaseClass
public class SubClass2 : AbstractBaseClass
{
    override public void Output(string sSource)
    {
        string s = sSource.ToLower();
        Console.WriteLine("Вызов SubClass2.Output() из {0}",
                           s) ;
    }
    public static void Test(AbstractBaseClass ba)
    {
        ba.Output("Test");
    }
}

public static void Main(string[] strings)
{
    // Нельзя создать объект класса AbstractBaseClass,
    // поскольку он — абстрактный. Если вы снимете
    // комментарий со следующей строки, то C# сгенерирует
    // сообщение об ошибке компиляции
    // AbstractBaseClass ba = new AbstractBaseClass();
    // Теперь повторим наш эксперимент с классом Subclass1
    Console.WriteLine("Создание объекта SubClass1");
    SubClass1 scl = new SubClass1();
    Test(scl) ;
    // и классом Subclass2
    Console.WriteLine("\nСоздание объекта SubClass2");
    SubClass2 sc2 = new SubClass2();
    Test(sc2) ;
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}

```

В программе сначала определяется класс `AbstractBaseClass` с единственным абстрактным методом `Output()`. Поскольку он объявлен как `abstract`, метод `Output()` не имеет реализации, т.е. тела метода.

Класс `AbstractBaseClass` наследуют два подкласса: `SubClass1` и `SubClass2`. Оба — конкретные классы, так как перекрывают метод `Output()` "настоящими" методами и не содержат собственных абстрактных методов.



Класс может быть объявлен как абстрактный независимо от наличия в нем абстрактных методов. Однако конкретным класс может быть тогда и только тогда, когда все абстрактные методы всех базовых классов выше него сокрыты (перекрыты) реальными методами.

Методы `Output()` двух рассматриваемых подклассов немного различны — один из них преобразует передаваемую ему строку в верхний регистр, другой — в нижний. Вывод программы демонстрирует полиморфную природу класса `AbstractBaseClass`.

```
Создание объекта SubClass1
Вызов SubClass1.Output() из TEST
Создание объекта SubClass2
Вызов SubClass2.Output() из test
Нажмите <Enter> для завершения программы...
```



Абстрактный метод автоматически является виртуальным, так что добавлять ключевое слово `virtual` к ключевому слову `abstract` не требуется.

Создание абстрактных объектов невозможно

Обратите внимание еще на одну вещь в рассматриваемой демонстрационной программе: нельзя создавать объект `AbstractBaseClass`, но аргумент функции `Test()` объявлен как объект класса `AbstractBaseClass` или одного из его подклассов. Это дополнение крайне важно. Объекты `SubClass1` и `SubClass2` могут быть переданы в функцию, поскольку оба являются конкретными подклассами `AbstractBaseClass`. Здесь использовано отношение ЯВЛЯЕТСЯ. Это очень мощная методика, позволяющая писать высоко обобщенные методы.

Создание иерархии классов



Для создания новой иерархии наследования можно также использовать ключевое слово `virtual`. Рассмотрим иерархию классов, показанную в приведенной далее демонстрационной программе `InheritanceTest`.

```
// InheritanceTest - пример использования ключевого слова
// virtual для создания новой иерархии классов
namespace InheritanceTest
{
    using System;

    public class Program
    {
        public static void Main(string[] strings)
        {
            Console.WriteLine("\nПередача BankAccount")
            BankAccount ba = new BankAccount();
        }
    }
}
```

```

Test1(ba) ;

Console.WriteLine("\nПередача SavingsAccount");
SavingsAccount sa = new SavingsAccount();
Test1(sa) ;
Test2(sa) ;

Console.WriteLine("\nПередача SpecialSaleAccount");
SpecialSaleAccount ssa = new SpecialSaleAccount();
Test1(ssa);
Test2(ssa) ;
Test3(ssa) ;

Console.WriteLine("\nПередача SaleSpecialCustomer");
SaleSpecialCustomer ssc = new SaleSpecialCustomer();
Test1(ssc);
Test2 (ssc)
Test3(ssc) ;
Test4(ssc) ;

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
                  "завершения программы...");
Console.Read();
}

public static void Test1(BankAccount account)
{
    Console.WriteLine("\tB Test(BankAccount)");
    account.Withdraw(100) ;
}

public static void Test2(SavingsAccount account)
{
    Console.WriteLine ( "\tB Test (SavingsAccount),") ;
    account.Withdraw(100);
}

public static void Test3(SpecialSaleAccount account)
{
    Console.WriteLine("\tB Test(SpecialSaleAccount)");
    account.Withdraw(100);
}

public static void Test4(SaleSpecialCustomer account)
{
    Console.WriteLine("\tB Test(SaleSpecialCustomer)");
    account.Withdraw(10 0);
}

// BankAccount - моделирует банковский счет с номером,
// присваиваемым при создании, и балансом
public class BankAccount
{

```

```

// Withdrawal - вы можете снять со счета любую сумму, не
// превышающую баланс. Возвращает реально снятую со
// счета сумму
virtual public void Withdraw(decimal dWithdraw)
{
    Console.WriteLine("\B\Bвызывает " +
                      "BankAccount.Withdraw()");
}

}

// SavingsAccount - банковский счет с начислением
// процентов
public class SavingsAccount : BankAccount
{
    override public void Withdraw(decimal mWithdrawal)
    {
        Console.WriteLine("\t\tBBI3biBaeT " +
                          "SavingsAccount.Withdraw()");
    }
}

// SpecialSaleAccount - счет используется только для
// продаж
public class SpecialSaleAccount : SavingsAccount
{
    new virtual public void Withdraw(decimal mWithdrawal)
    {
        Console.WriteLine("\t\tTBBI3BMAET " +
                          "SpecialSaleAccount.Withdraw()");
    }
}

// SaleSpecialCustomer - счет только для специальных
// покупателей
public class SaleSpecialCustomer : SpecialSaleAccount
{
    override public void Withdraw(decimal mWithdrawal)
    {
        Console.WriteLine("\t\tTBBI3BIBAET " +
                          "SaleSpecialCustomer.Withdraw()");
    }
}
}

```

Каждый из указанных классов расширяет наследуемый класс. Заметьте, однако, что метод `SpecialSaleAccount.Withdraw()` помечен как `virtual`, что разрывая цепь наследования в этой точке. При рассмотрении с точки зрения `BankAccount` классы `SpecialSaleAccount` и `SaleSpecialCustomer` выглядят в точности как `SavingsAccount`. И только при рассмотрении с точки зрения `SpecialSaleAccount` становятся доступны новые версии `Withdraw()`.

Все это показано в приведенной демонстрационной программе. Функция `Main()` вызывает ряд методов `Test()`, каждый из которых разработан для своего подкласса,

Каждая из версий метода `Test()` вызывает `Withdraw()` с точки зрения различного класса объекта.

Вывод программы имеет следующий вид:

```
Передача BankAccount
  в Test(BankAccount)
    вызывает BankAccount.Withdraw()
Передача SavingsAccount
  в Test(BankAccount)
    вызывает SavingsAccount.Withdraw()
  в Test(SavingsAccount)
    вызывает SavingsAccount.Withdraw()
Передача SpecialSaleAccount
  в Test(BankAccount)
    вызывает SavingsAccount.Withdraw()
  в Test(SavingsAccount)
    вызывает SavingsAccount.Withdraw()
  в Test(SpecialSaleAccount)
    вызывает SpecialSaleAccount.Withdraw()
Передача SaleSpecialCustomer
  в Test(BankAccount)
    вызывает SavingsAccount.Withdraw()
  в Test(SavingsAccount)
    вызывает SavingsAccount.Withdraw()
  в Test(SpecialSaleAccount)
    вызывает SaleSpecialCustomer.Withdraw()
  в Test(SaleSpecialCustomer)
    вызывает SaleSpecialCustomer.Withdraw()
Нажмите <Enter> для завершения программы...
```

Полужирным шрифтом выделены строки, представляющие особый интерес. Классы `BankAccount` и `SavingsAccount` работают в точности так, как и ожидалось. Однако при вызове `Test(SavingsAccount)` и `SpecialSaleAccount`, и `SaleSpecialCustomer` передаются как `SavingsAccount`. В то же время `SaleSpecialCustomer` при передаче в `Test(SpecialSaleAccount)` работает как наследник этого пасса, т.е. фактически создается новая иерархия наследования.

Создание новой иерархии

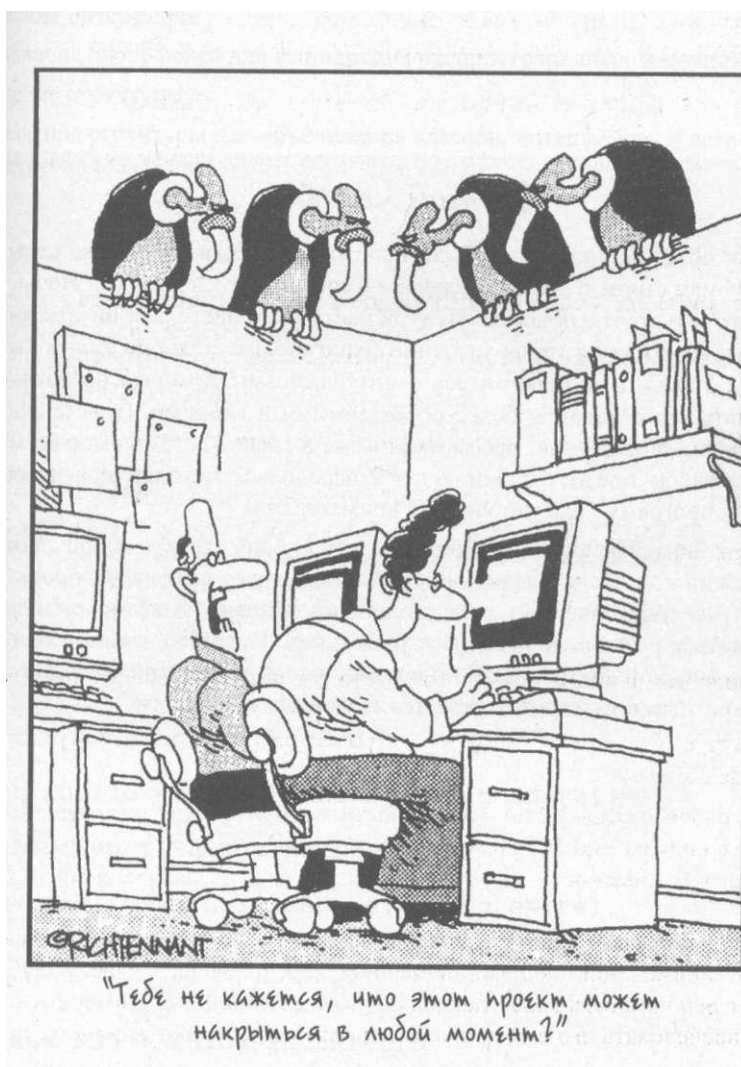
Зачем C# поддерживает создание новой иерархии наследования? Неужели обычного полиморфизма недостаточно?

C# формировался как "сетевой" язык в том смысле, что классы при работе программы—и даже подклассы—могут быть распределены по сети Интернет. То есть программа, которую вы пишете, может непосредственно использовать классы из стандартных хранилищ, расположенных на других компьютерах в Интернете.

Вы можете расширять класс, загруженный из Интернета. Перекрытие методов стандартной, протестированной иерархии классов может привести к непреднамеренным эффектам. Создание новой иерархии классов позволяет вашим программам пользоваться всеми преимуществами полиморфизма без опасности разрушения существующего кода.

Часть V

За базовыми классами



В этой части...

Ваши объекты до сих пор были простыми вещами наподобие целых чисел или строк, в крайнем случае — счетов `BankAccount`. Но в C# имеются и другие объекты. Из этой части вы узнаете, как писать собственные объекты типов-значений (работающие подобно типам `int` или `float`), и познакомитесь с интерфейсами, которые позволяют сделать ваши объекты более обобщенными и гибкими. Вместе с абстрактными классами, рассмотренными в главе 13, "Полиморфизм", интерфейсы предоставляют ключ к передовым методам проектирования программ. Так что читайте внимательно!

Однако интерфейсы — не единственный способ сделать обобщенный и гибкий код. Новые возможности C# позволяют создавать *обобщенные* (generic) объекты — например, контейнеры, в которых могут храниться различные данные других типов. Пока что это звучит для вас сплошной абстракцией, но к концу части абстракция наполнится конкретным содержанием, так что запаситесь терпением.

Глава 14

Интерфейсы и структуры

В этой главе...

Отношение МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК

Определение интерфейса

Использование интерфейса для выполнения распространенных операций

Определение структуры

Использование структуры для объединения классов, интерфейсов и встроенных типов в одну иерархию классов

К

лассе может содержать ссылку на другой класс. Это — простое отношение СОДЕРЖИТ. Один класс может расширять другой класс с помощью наследования. Б—отношение ЯВЛЯЕТСЯ. Интерфейсы С# реализуют еще одно, не менее важное отношение — МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК.

Что значит
МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК

Если вы хотите написать заметку, вы можете взять ручку и обрывок бумаги, можете использовать органайзер или сделать это посредством своего компьютера. Все эти объекты реализуют операцию "написать заметку" — TakeANote. Используя магию наследования, на языке С# это можно реализовать следующим образом:

```
abstract class ThingsThatRecord
{
    abstract public void TakeANote (string sNote) ;
}
public class Pen : ThingsThatRecord
{
    override public void TakeANote (string sNote)
    {
        // ... Написание заметки ручкой ...
    }
}
public class PDA : ThingsThatRecord
{
    override public void TakeANote (string sNote)
    {
        // ... при помощи органайзера ...
    }
}
```

```

}
public class Laptop : ThingsThatRecord
{
    override public void TakeANote(string sNote)
    {
        //... еще каким-то образом ...
    }
}

```



Если ключевое слово `abstract` вас смущает — обратитесь к главе "Полиморфизм", за пояснениями. Если вам непонятно, что такое наследование — перечитайте главу 12, "Наследование".

Решение с использованием наследования выглядит неплохо до тех пор, пока интерес вызывает только операция `TakeANote()`. Функция наподобие показанной далее `RecordTask()` может использовать метод `TakeANote()` для того, чтобы записать список необходимых покупок в зависимости от того, какое средство есть у вас под рукой:

```

void RecordTask(ThingsThatRecord things)
{
    // Этот абстрактный метод реализован во всех классах,
    // которые наследуют ThingsThatRecord
    things.TakeANote("Список покупок"), -
    // . . . и так далее . . .
}

```

Однако это решение сталкивается с двумя большими проблемами.

- ✓ **Первая проблема — фундаментальная.** Дело в том, что реально связать ручку органайзер и компьютер соотношением ЯВЛЯЕТСЯ невозможно. Знание того, как работает ручка, не дает никаких сведений о том, как записывают информацию компьютер или органайзер.
- ✓ **Вторая проблема чисто техническая.** Гораздо лучше описать `Laptop` как подкласс класса `Computer`. Хотя PDA также можно наследовать от того же ~~класса~~ `Computer`, этого нельзя сказать о классе `Pen`. Вы можете охарактеризовать ручку как некоторый тип `MechanicalWriteDevice` (механическое пишущее устройство) или `DeviceThatStainsYourShirt` (устройство, пачкающее ваши ~~штаны~~ `штаны`). Однако в C# класс не может быть наследован от двух разных классов одновременно — класс C# может быть вещью только одного сорта.

Вернемся к трем исходным классам. Единственное общее, что у них есть — то, что все они могут использоваться для записи чего-либо. Отношение МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК `Recordable` позволяет связать их пригодность некоторой цели без применения наследования.

Что такое интерфейс

Описание интерфейса выглядит очень похожим на описание класса без членов-данных, в котором все методы абстрактны. Описание интерфейса для "вещей, которые могут записывать", может выглядеть следующим образом:

```
interface IRecordable
```

```
{  
    void TakeANote(string sNote)  
}
```

Обратите внимание на ключевое слово `interface` там, где обычно стоит ключевое слово `class`. В фигурных скобках интерфейса приведен список абстрактных методов. Интерфейсы не содержат определения никаких членов-данных.

Метод `TakeANote()` записан без реализации. Ключевые слова `public` и `virtual` ни `abstract` не являются необходимыми. Все методы интерфейса открыты, а сам он не включается ни в какое обычное наследование — это интерфейс, а не класс.

Класс, который *реализует* интерфейс, должен предоставить реализацию для каждого элемента интерфейса. Метод, реализующий метод интерфейса, не использует ключевое слово `override` — это не похоже на перекрытие виртуальной функции.



По соглашению имена интерфейсов начинаются с буквы `I`. Кроме того, для них, как правило, используются прилагательные (в то время как для имен классов — существительные). Как обычно, это только соглашение — `C#` совершенно все равно, как именно вы назовете ваш интерфейс.

Далее приведено объявление, указывающее, что класс `PDA` реализует интерфейс `IRecordable`.

```
public class PDA : IRecordable  
{  
    public void TakeANote(string sNote)  
    {
```



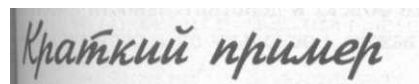
```
        // . . . Написание памятки . . .  
    }
```

Как видите, не существует отличий между синтаксисом объявления наследования базового класса `ThingsThatRecord` и объявлением о реализации интерфейса `IRecordable`.



В этом и заключается основная причина соглашения об именовании интерфейсов — чтобы сразу отличать их от классов.

Вывод из всего сказанного — интерфейс описывает возможности и свойства. Кроме того, он представляет собой *контракт*. Если вы согласны реализовать все методы, определенные в интерфейсе, вы получите все его возможности.



Класс реализует интерфейс, предоставляя определения всех методов интерфейса, как показано в приведенном далее фрагменте исходного текста,

```
public class Pen : IRecordable  
{  
    public void TakeANote(string sNote)  
    {  
        // . . . Запись ручкой . . .  
    }
```

```

    }
    public class PDA : ElectronicDevice, IRecordable
    {
        public void TakeANote(string sNote)
        {
            // . . . Использование органайзера . . .
        }
    }
    public class Laptop : Computer, IRecordable
    {
        public void TakeANote(string sNote)
        {
            // Запись при помощи компьютера
        }
    }

```

Каждый из этих трех классов наследует свой базовый класс, но реализует один и тот же интерфейс `IRecordable`, указывающий, что каждый из трех классов может использоваться для написания памятки с применением метода `TakeANote()`. Чтобы понять, почему может оказаться полезным, рассмотрим следующую функцию `RecordShoppingList()`

```

public class Program
{
    static public void RecordShoppingList(IRecordable
                                         recordingObject)
    {
        // Создание списка покупок
        string sList = GenerateShoppingList();
        // Запись списка
        recordingObject.TakeANote(sList) ;
    }
    public static void Main(string[] args)
    {
        PDA pda = new PDA();
        RecordShoppingList(pda);
    }
}

```

Данный фрагмент кода гласит, что функция `RecordShoppingList()` может принимать в качестве аргумента любой объект, реализующий метод `TakeANote()` — говоря человеческим языком, любой объект, который в состоянии записать памятку. Функция `RecordShoppingList()` не делает никаких предположений о том, какой в точности тип имеет `recordingObject`. Тот факт, что объект в действительности имеет `PDA` или `ElectronicDevice`, совершенно не важен, поскольку он в состоянии записать памятку.

Это чрезвычайно важное свойство, так как оно обеспечивает функции `RecordShoppingList()` высокую степень обобщенности, а следовательно, и повторное применение в других программах. Это более высокая степень общности, чем при использовании базового класса в качестве типа аргумента, поскольку интерфейс в качестве аргумента позволяет передавать практически произвольный объект, который может не иметь ничего общего с другими разрешенными для использования объектами, если не считать реализации интерфейса. Эти объекты могут быть никак не связаны общей иерархией классов.

Пример программы, использующей отношение МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК

Рассматриваемая далее программа `SortInterface` демонстрирует применение описанных сведений на практике.

Для лучшего понимания я должен разбить программу на несколько частей. Так я смогу более четко продемонстрировать применение отдельных принципов. Если у меня вообще есть принципы... :) Сейчас главная цель — обеспечить понимание того, как работает данная программа.

Создание собственного интерфейса

Интерфейс `IDisplayable` удовлетворяется любым классом, который содержит метод `GetString()` (и, само собой, объявляет, что он реализует `IDisplayable`). `GetString()` возвращает объект типа `string`, который может быть выведен на экран (использованием `WriteLine()`):

```
//IDisplayable - объект, реализующий метод GetStringO
interface IDisplayable
{
    // Возвращает собственное описание
    string GetStringO ;
}
```

Приведенный далее класс `Student` реализует интерфейс `IDisplayable`:

```
class Student : IDisplayable

private string sName;
private double dGrade = 0.0;
// Методы доступа только для чтения
public string Name
{
    get { return sName; }
}

public double Grade
{
    get { return dGrade; }
}

// GetString - возвращает строковое представление
// информации о студенте
public string GetString() // implements the interface
{
    string sPadName = Name.PadRight(9);
    string s = String.Format("{{0}} : {{1:N0}}",
                             sPadName, Grade);
    return s;
}
```

Вызов `PadRight()` гарантирует, что поле имени будет иметь ширину не менее 9 символов (справа от имени при необходимости будет добавлено необходимое количество пробелов). Это делает вывод на экран более привлекательным (данный вопрос рассматривался в главе 9, "Работа со строками в C#"). `{1:N0}` гласит: выводит час с запятыми (или точками — в зависимости от региональных настроек) через каждые 3 цифры. `O` означает — округлить дробную часть.

С использованием приведенного объявления можно написать следующий фрагмент исходного текста (полностью программа будет приведена позже):

```
// DisplayArray - вывод массива объектов, которые реализуют
// интерфейс IDisplayable
public static void DisplayArray(IDisplayable[] displayables)

{
    int length = displayables.Length;
    for(int index = 0; index < length; index++)
    {
        IDisplayable displayable = displayables[index];
        Console.WriteLine("{0}", displayable.GetString())
    }
}
```

Приведенный метод `DisplayArray()` может вывести информацию о массиве любого типа, лишь бы его элементы определяли метод `GetString()`. Вот пример вывода описанной функции:

```
Homer      : 0
Marge      : 85
Bart       : 50
Lisa       : 100
Maggie     : 30
```

Предопределенные интерфейсы

Аналогично можно использовать интерфейсы из стандартной библиотеки C#. Например, C# определяет интерфейс `Comparable` следующим образом:

```
interface Comparable
{
    // Сравнивает текущий объект с объектом 'o'; возвращает 1,
    // если текущий объект больше, -1, если меньше, и 0 в
    // противном случае
    int CompareTo(object o);
}
```

Класс реализует интерфейс `Comparable` путем реализации метода `CompareTo`. Например, `String` реализует этот метод путем сравнения двух строк. Если строки идентичны, метод возвращает 0. Если строки различны, метод возвращает либо 1, либо -1 в зависимости от того, какая из строк "больше".

Как ни странно, но отношение сравнения можно задать и для объектов типа `Student` — например, по их успеваемости.

Реализация метода `CompareTo()` приводит к тому, что объекты могут быть отсортированы. Если один студент "больше" другого, их можно упорядочить от "меньше" к "большему". На самом деле в классе `Array` уже реализован соответствующий метод: `Array.Sort(Comparable[] objects);`

Этот метод сортирует массив объектов, которые реализуют интерфейс `Comparable`. Не имеет значения, к какому классу в действительности принадлежат объекты — ~~пример~~, это могут быть объекты `Student`. Класс `Array` может сортировать следующую версию `Student`:

```
//Student - описание студента с использованием имени и
//успеваемости
class Student : Comparable

private double dGrade;
// Методы доступа только для чтения
public double Grade
{
    get { return dGrade; }
}

// CompareTo - сравнение двух студентов; студент с лучшей
// успеваемостью "больше"
public int CompareTo (object rightObject)
{
    Student leftStudent = this;
    Student rightStudent = (Student) rightObject;
    // Возвращаем 16 -1 или 0 в зависимости от выполнения
    // критерия сортировки
    if (rightStudent.Grade < leftStudent.Grade)
    {
        return -1;
    }
    if (rightStudent.Grade > leftStudent.Grade)
    {
        return 1;
    }
    return 0;
}
```

Сортировка массива объектов `Student` сводится к единственному вызову:

```
id MyFunction (Student [] students)
```

```
// Сортировка массива объектов Comparable
Array.Sort(students) ;
```

Ваше дело — обеспечить компаратор; `Array` сделает все остальное сам.

Сборка воедино



И вот наступил долгожданный момент: полная программа `SortInterface`, использующая описанные ранее возможности.

```
[SortInterface - демонстрационная программа SortInterface
иллюстрирует концепцию интерфейса
using System;
```

```

namespace SortInterface
{
    // IDisplayable - Объект, который может представить
    // информацию о себе в строковом формате
    interface IDisplayable
    {
        // GetString - возврат строки, представляющей информации
        // об объекте
        string GetString();
    }

    class Program
    {
        public static void Main(string[] args)
        {
            // Сортировка студентов по успеваемости...
            Console.WriteLine("Сортировка списка студентов");
            // Получаем несортированный список студентов
            Student[] students = Student.CreateStudentList();

            // Используем интерфейс IComparable для сортировки
            // массива
            IComparable[] comparableObjects =
                (IComparable[])students;
            Array.Sort(comparableObjects);

            // Теперь интерфейс IDisplayable выводит результат
            IDisplayable[] displayableObjects =
                (IDisplayable[])students;
            DisplayArray(displayableObjects);

            // Теперь отсортируем массив птиц по имени с
            // использованием той же процедуры, хотя классы Bird
            // и Student не имеют общего базового класса
            Console.WriteLine("\nСортировка списка птиц");
            Bird[] birds = Bird.CreateBirdList();

            // Обратите внимание на отсутствие необходимости
            // явного преобразования типа объектов...
            Array.Sort(birds);
            DisplayArray(birds);

            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                "завершения программы...");
            Console.Read();
        }

        // DisplayArray - вывод массива объектов, реализующих
        // интерфейс IDisplayable
        public static void

```

```

        DisplayArray (IDisplayable [] displayables)
    {
        int length = displayables.Length;
        for(int index = 0; index < length; index++)
        {
            IDisplayable displayable = displayables[index];
            Console.WriteLine("{0}", displayable.GetString());
        }
    }

//-----Students - сортировка по успеваемости-----
// Student - описание студента с использованием имени и
// успеваемости
class Student : IComparable, IDisplayable
(
    private string sName;
    private double dGrade = 0.0;

    // Конструктор - инициализация нового объекта
    public Student(string sName, double dGrade)
    {
        this.sName = sName;
        this.dGrade = dGrade;
    }

    // CreateStudentList - для простоты просто создаем
    // фиксированный список студентов
    static string[] sNames =
        {"Homer", "Marge", "Bart", "Lisa", "Maggie"};
    static double[] dGrades = {0, 85, 50, 100, 30};
    public static Student[] CreateStudentList ()
    {
        Student[] sArray = new Student[sNames.Length];
        for (int i = 0; i < sNames.Length; i++)
        {
            sArray[i] = new Student(sNames[i], dGrades[i]);
        }
    }
    return sArray;

    // Методы доступа только для чтения
    public string Name
    {
        get { return sName; }
    }

    public double Grade
    {
        get { return dGrade; }
    }

    //Реализация интерфейса IComparable:
    // CompareTo - сравнение двух объектов (в нашем случае —

```

```

// объектов типа Student) и выяснение того, что из них
// должен идти раньше в отсортированном списке
public int CompareTo(object rightObject)
{
    // Сравнение текущего Student (назовем его левым) и
    // другого (назовем его правым) и генерация ошибки,
    // если эти объекты — не Student
    Student leftStudent = this;
    if (!(rightObject is Student))
    {
        Console.WriteLine("Компаратору передан не Student")
        return 0;
    }
    Student rightStudent = (Student)rightObject;
    // Генерируем -1, 0 или 1 на основании критерия
    // сортировки
    if (rightStudent.Grade < leftStudent.Grade)
    {
        return -1;
    }
    if (rightStudent.Grade > leftStudent.Grade)
    {
        return 1;
    }

    return 0;
}

// Реализация интерфейса IDisplayable:
// GetString - возвращает строковое представление
// информации о студенте
public string GetStringO
{
    string sPadName = Name.PadRight(Y);
    string s = String.Format("{0}: {1:N0}",
                             sPadName, Grade)

    return s;
}
}

//-----Birds - сортировка птиц по именам-----
// Bird - just an array of bird names
class Bird : IComparable, IDisplayable
{
    private string sName;

    // Конструктор - инициализация объекта Bird
    public Bird(string sName)
    {
        this.sName = sName;
    }

    // CreateBirdList - возвращает список птиц; для простоты
    // используем фиксированный список

```

```

static string [] sBirdNames =
    { "Oriole", "Hawk", "Robin", "Cardinal",
      "Bluejay", "Finch", "Sparrow" };
public static Bird[] CreateBirdList ()
{
    Bird[] birds = new Bird[sBirdNames.Length];
    for(int i = 0; i < birds.Length; i++)
    {
        birds[i] = new Bird(sBirdNames[i]);
    }
    return birds;

    // Методы доступа только для чтения
    public string Name
    {
        get { return sName; }
    }

    // Реализация интерфейса IComparable:
    // CompareTo - сравнение имен птиц; используется
    // встроенный метод сравнения класса String
    public int CompareTo(object rightObject)
    {
        // Сравнение текущего Bird (назовем его левым) и
        // другого (назовем его правым)
        Bird leftBird = this;
        Bird rightBird = (Bird)rightObject;

        return String.Compare(leftBird.Name, rightBird.Name);
    }

    // Реализация интерфейса IDisplayable:
    // GetString - возвращает строку с именем птицы
    public string GetString()
    {
        return Name;
    }

```

Класс `Student` (примерно в середине листинга) реализует интерфейсы `IComparable` и `IDisplayable`, как описано ранее. Метод `CompareTo` сравнивает студентов по успеваемости, что приводит к соответствующей сортировке их списка. Метод `GetString()` возвращает имя и успеваемость студента.

Прочие методы класса `Student` включают свойства только для чтения `Name` и `Grade`, простой конструктор и метод `CreateStudentList()`. Последний метод просто возвращает фиксированный список студентов.

Класс `Bird` внизу листинга также реализует интерфейсы `IComparable` и `IDisplayable`. Он реализует метод `CompareTo()`, который сравнивает названия птиц посредством встроенного метода сравнения класса `String`. Таким образом, в результате сортировки получается список птиц в алфавитном порядке. Метод `GetName()` просто возвращает название птицы.

Теперь можно вернуться к функции `Main()`. Метод `CreateStudentList()` используется для получения несортированного списка, который сохраняется в массиве `students`.



Обычно для имен коллекций объектов, таких как массивы, используются существительные.

Массив студентов сперва преобразуется в массив `comparableObjects`. Это отличается от массивов, использованных ранее в этой книге (в основном в главе 6 "Объединение данных — классы и массивы"). Эти массивы были массивами объектов определенного класса, наподобие массива объектов `Student`, в то время как `comparableObjects` представляет собой массив объектов, реализующих интерфейс `Comparable` безотносительно к классу, которому принадлежат объекты. Использование интерфейса в качестве типа элементов массива, типа параметра или возвращаемого значения — мощный метод повышения гибкости программы.

Массив `comparableObjects` передается встроенному методу `Array.Sort()`, который и сортирует студентов по их успеваемости.

Затем отсортированный массив объектов типа `Student` передается локально определенному методу `DisplayArray()`, итеративно проходящему по всем элементам массива объектов, которые реализуют метод `GetString()`. Для выяснения количества элементов массива используется свойство `Array.Length`. Затем для каждого объекта вызывается метод `GetString()`, и его результат выводится на дисплей с помощью функции `WriteLine()`.

Далее программа сортирует и выводит список птиц. Несомненно, вы согласитесь, что между птицами и студентами нет ничего общего. Однако класс `Bird` реализует интерфейс `Comparable` путем сравнения названий птиц и интерфейс `IDisplayable` путем возврата названий птиц.

Обратите внимание, что функция `Main()` не выполняет преобразования типа массива птиц — в этом нет необходимости. Это аналогично следующему фрагменту исходного текста:

```
class BaseClass {}
class Subclass : BaseClass {}
class Program
{
    public static void SomeFunction(BaseClass bc) {}
    public static void AnotherFunction()
    {
        Subclass sc = new Subclass();
        SomeFunction(sc);
    }
}
```

Здесь объект класса `Subclass` может быть передан как объект `BaseClass`, поскольку `Subclass` ЯВЛЯЕТСЯ `BaseClass`.

Аналогично, массив объектов `Bird` может быть передан методу, ожидающему массив объектов `Comparable`, поскольку класс `Bird` реализует интерфейс `Comparable`. Следующий вызов — `DisplayArray()` — также получает массив `birds` без преобразования типа, поскольку класс `Birds` реализует интерфейс `IDisplayable`.

Вывод программы выглядит следующим образом:

Часть V. За базовыми классами

```

Сортировка списка студентов
Lisa      : 100
Marge     : 85
Bart      : 50
Maggie    : 30
Homer     : 0

```

```

Сортировка списка птиц
Blue jay
Cardinal
! Finch
Hawk
Oriole
Robin
Sparrow
Нажмите <Enter> для завершения программы. . . "

```

И студенты, и птицы отсортированы, каждый список — в соответствии со своим критерием сортировки.

Наследование интерфейса

Интерфейс может "наследовать" методы другого интерфейса. Я использую кавычки, поскольку это не истинное наследование, независимо от того, как оно выглядит. В приведенном далее фрагменте кода перечислены *базовые интерфейсы*, что очень напоминает указание базового класса.

```

// ICompare - интерфейс, который может как сравнивать
// объекты, так и выводить информацию о своем значении
public interface ICompare : IComparable
{
    // GetValue - возвращает собственное целое значение
    int GetValue();
}

```

Интерфейс `ICompare` наследует требования по реализации `CompareTo()` от интерфейса `IComparable`. К этому добавляется требование о реализации метода `GetValue()`. Объект `ICompare` может использоваться в качестве объекта `IComparable`, поскольку по определению он реализует его требования. Однако это не полное наследование в объектно-ориентированном C#-смысле этого слова. Здесь невозможен никакой полиморфизм. Кроме того, здесь неприменимы обычные отношения между конструкторами.

Наследование интерфейсов будет проиллюстрировано в демонстрационной программе `AbstractInterface` в следующем разделе.

Абстрактный интерфейс



Для реализации интерфейса класс должен реализовать все его методы. Однако класс может реализовать метод интерфейса как абстрактный метод (само собой, такой класс является абстрактным), как, например, в приведенной далее демонстрационной программе.

```

// AbstractInterface - демонстрирует реализацию интерфейса
// абстрактным классом
using System;

namespace AbstractInterface
{
    // ICompare - интерфейс, который может как сравнивать
    // объекты, так и выводить собственное значение
    public interface ICompare : IComparable

        // GetValue - возвращает собственное значение как int
        int GetValue();
    }

    // BaseClass - реализует интерфейс ICompare, реализуя
    // конкретный метод GetValue() и абстрактный метод
    // CompareTo()
    abstract public class BaseClass : ICompare
    {
        int nValue;

        public BaseClass(int nInitialValue)
        {
            nValue = nInitialValue;
        }

        // Реализация интерфейса ICompare:
        // сначала конкретный метод
        public int GetValue()
        {
            return nValue;
        }

        // а затем реализует интерфейс ICompare при помощи
        // абстрактного метода
        abstract public int CompareTo(object rightObject);

        // Subclass - Завершает базовый класс путем перекрытия
        // абстрактного метода CompareTo()
        public class Subclass: BaseClass
        {
            ^ // Передаем значение, полученное конструктором,
            // конструктору базового класса
            public Subclass(int nInitialValue) : base(nInitialValue)
            {
            }

            // CompareTo - реализует интерфейс IComparable;
            // возвращает указание, больше ли один объект подкласса
            // другого или нет
            override public int CompareTo(object rightObject)

```



```

    BaseClass be = (BaseClass)rightObject;
    return GetValue().CompareTo(be.GetValue());
}

```

```

public class Program

{
    public static void Main(string[] strings)

    {
        Subclass scl = new Subclass(10) ;
        Subclass sc2 = new Subclass(20) ;

        MyFunc(scl, sc2) ,-

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
        Console.Read();
    }

    // MyFunc - использует метод, предоставленный
    // интерфейсом ICompare для вывода значений двух
    // объектов и указания, какой из них больше
    public static void MyFunc(ICompare ic1, ICompare ic2)
    {
        Console.WriteLine("Значение ic1 равно {0}, " +
                           "а ic2 - {1}",
                           ic1.GetValue(), ic2.GetValue());

        string s;
        switch (ic1.CompareTo(ic2))
        {
            case 0 :
                s = "равен";
                break;
            case -1 :
                s = "меньше";
                break;
            case 1 :
                s = "больше";

            default:
                s = "непонятно";
                break;
        }
        Console.WriteLine (
            "Объект ic1 {0} объекта ic2". s) ;
    }
}

```

Интерфейс `ICompare` описывает класс, который может сравнивать два объекта и получать их значения. `ICompare` наследует требование реализации `CompareTo()` от интерфейса `IComparable`, а кроме того, добавляет собственный метод `GetValue()`, который возвращает целочисленное значение объекта.



Несмотря на то что метод может вернуть значение объекта как `int`, `GetValue()` ничего не говорит о внутреннем устройстве класса. Генерация™ целого значения может потребовать проведения сложных вычислений.

Класс `BaseClass` реализует интерфейс `ICompare` — конкретный метод `GetValue` возвращает значение члена `nValue`. Однако метод `CompareTo()`, требуемый интерфейса `ICompare`, объявлен как абстрактный.



Объявление класса `abstract` означает, что в нем отсутствует реализация одной или нескольких свойств — в рассматриваемом случае метода `CompareTo()`. Реализация этого метода отложена до подкласса данного абстрактного класса.

`Subclass` реализует метод `CompareTo()`, что необходимо для того, чтобы этот класс был конкретным.



Заметим, что `Subclass` автоматически реализует интерфейс `ICompare` не смотря на то, что это не сказано явно. `BaseClass` обещает реализовать методы `ICompare`, а `Subclass` ЯВЛЯЕТСЯ `BaseClass`. Наследуя эти методы, `SubClass` автоматически удовлетворяет требованиям по реализации `ICompare`.

Функция `Main()` создает два объекта класса `Subclass` с различными значениям а затем передает их функции `MyFunc()`. Метод `MyFunc()` ожидает получения двух объектов с интерфейсом `ICompare`. Функция `MyFunc()` использует метод `CompareTo()` для принятия решения о том, какой из объектов больше, а затем применяет метод `GetValue()` для вывода "значений" этих двух объектов.

Вывод этой программы достаточно короток:

Значение `ic1` равно 10, а `ic2` — 20

Объект `ic1` меньше объекта `ic2`

Нажмите <Enter> для завершения программы...



Из главы 15, "Обобщенное программирование", вы узнаете, как можно писать обобщенные интерфейсы.

Структуры C# и их отличия от классов

C# обладает определенной дихотомией при объявлении переменных. Вы объявляете и инициализируете переменные типов-значений, таких как `int` или `double`, следующим образом:

```
int p; // Объявление
p = 1; // Инициализация
```

Однако ссылки на объекты вы объявляете и инициализируете совершенно иначе:

```
public class MyClass
{
    public int n;
}

MyClass mc;           // Объявление
mc = new MyClass ();  // Инициализация
```



Переменная *mc* имеет *ссылочный тип*, поскольку она ссылается на потенциально удаленную память. Встроенные переменные типа *int* или *double* известны как *переменные типов-значений*.

Если вы посмотрите на *n* или *mc* более пристально, то увидите, что единственное реальное отличие состоит в том, что *C#* выделяет память для переменных типов-значений автоматически, в то время как для объектов классов вы должны выделять память явно. Интересно, нельзя ли объединить их посредством некоторой Единой Теории Классов?

Структуры *C#*

C# определяет третий тип переменных, именуемый *структурой*, который представляет собой мост через пропасть между ссылочными типами и типами-значениями. Синтаксис объявления структуры выглядит почти так же, как и у класса:

```
public struct MyStruct
{
    public int n;
    public double d;
}

public class MyClass
{
    public int n;
    public double d;
}
```

Обращение к объекту структуры осуществляется так же, как и к объекту класса, но выделение памяти — как для типа-значения, что демонстрируется следующим кодом:

```
// Объявление и доступ к простому типу-значению
int n;
n = 1;

// Объявление структуры похоже на объявление простого int
MyStruct ms; // Автоматическое выделение памяти
ms.n = 3;    // Доступ к членам выполняется так же, как и
ins.d = 3.0; // для объекта класса

// Память для объекта класса должна быть выделена в
// отдельной области при помощи оператора new
MyClass mc = new MyClass();
mc.n = 2;
mc.d = 2.0;
```

Объект *struct* хранится в памяти так же, как и переменные встроенных типов. Переменная *ms* не является ссылкой на некоторый блок внешней памяти, выделенный в отдельной области. Объект *ms* занимает память в той же области, что и переменная *n*, как показано на рис. 14.1.

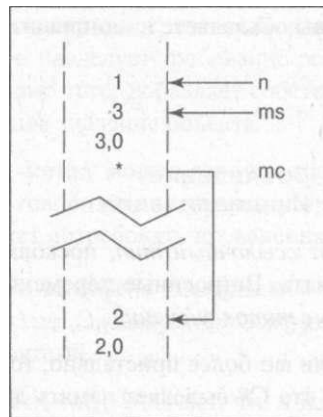


Рис. 14.1. Структурная переменная *ms* "живет" в той же области памяти, что и переменная *n*, в то время как память для объекта *mc* выделяется в отдельной области памяти

Отличие между ссылками и типами-значениями еще более очевидно в следующее примере. Создание массива из 100 ссылочных объектов требует от программы 101 **выз**в оператора `new` (один для массива и по одному для каждого объекта):

```
MyClass[] mc = new MyClass[100];
for(int i = 0; i < ms.Length; i++)
{
    mc[i] = new MyClass();
}
mc[0].n = 0;
```

Массив также приводит к определенным накладным расходам, как в смысле **времеу**б работы, так и используемой памяти.

- ✓ Каждый элемент в массиве *mc* должен быть достаточно большим, чтобы **содер**жать ссылку на объект.
- ✓ Каждый объект *MyClass* имеет невидимые накладные расходы памяти до и **после** единственного члена-данных *n*.
- ✓ При рассмотрении времени работы программы следует учесть, что она **должна** 100 раз выделить область памяти, а обращение к элементу массива требует двух обращений к ссылкам — ссылке на массив и ссылке в соответствующем элементе массива.

Память для объекта структуры выделяется сразу, как для части массива:

```
// Объявление массива int
int[] integers = new int[100]; // Выделение памяти
integers[0] = 0;
// Объявление массива структур такое же простое
MyStruct[] ms = new MyStruct[100]; // Выделение памяти
ms[0].n = 0;
```

В этом случае выделяется один большой блок памяти.

Конструктор структуры

Интересно, что структура может быть инициализирована с использованием синтаксиса, применяемого для инициализации класса,

```
public struct MyStruct
{
    public int n;
    public double d;
}
```

```
MyStruct ms = new MyStruct(); // Использование new
```

Несмотря на внешний вид, такая инициализация не приводит к выделению памяти в отдельной области — она всего лишь обнуляет поля `n` и `d`.

Вы можете создать собственный конструктор, выполняющий какие-то другие действия. Рассмотрим следующий исходный текст:

```
public struct Test
(
    ; private int n;
    public Test(int n)
    {
        this.n = n;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Test test = new Test(10);
    }
}
```

Несмотря на свой внешний вид, объявление `test = new Test(10);` не выделяет память, а только инициализирует уже выделенную память для типов-значений. Обратите внимание на использование круглых скобок, в отличие от массива, где скобки — квадратные.

Методы структур

Структура может иметь члены экземпляра, включая методы и свойства. Она может иметь и статические члены. Статические члены структуры могут иметь инициализаторы, но нестатические члены (члены экземпляра) их иметь не могут. Обычно объект структуры передается функции по значению, но может быть передан и по ссылке, если это специальным образом указывается в вызове функции с помощью ключевого слова `ref`. Структура не может наследовать класс (отличный от `Object`, который будет описан позже в данной главе) и не может наследоваться другими классами. Структура может реализовывать интерфейс.



Отличия между статическими членами и членами экземпляра описаны в главе 8, "Методы класса". В главе 7, "Функции функций", вы найдете информацию о передаче функции аргументов по значению и по ссылке. Наследование рассматривается в главе 12, "Наследование".



Все классы (и структуры) наследуют один тип `Object`, сказано об этом явно или нет. Вы можете перекрывать методы `Object`. С практической точки зрения единственный метод, который вы можете захотеть перекрыть — это метод `ToString()`, позволяющий объекту создавать строковое представление информации о самом себе. Если вы не реализуете собственный метод `ToString()`, то метод по умолчанию класса `Object` вернет полное имя класса, например `MyNamespace.MyClass`. Обычно в этом мало толку.

Пример применения структуры



Следующая демонстрационная программа иллюстрирует различные возможности структур:

```
// StructureExample - демонстрация различных свойств
// структурного объекта
using System;
using System.Collections;

namespace StructureExample
{
    public interface IDisplayable
    {
        string ToString();
    }

    // Структура может реализовывать интерфейс
    public struct Test : IDisplayable
    {
        // Структура может иметь члены как объекта, так и класса
        // (статические); статические члены могут иметь
        // инициализаторы
        private int n;
        private static double d = 20.0;

        // Для инициализации членов-данных структуры может
        // использоваться конструктор
        public Test(int n)
        {
            this.n = n;
        }

        // Структура может иметь свойства как объекта, так и
        // класса (статические)
        public int N
        {
            get { return n; }
            set { n = value; }
        }

        public static double D
        {

```

```

    get { return d; }
    set { d = value; }
}

// Структура может иметь методы
public void ChangeMethod(int nNewValue,
                        double dNewValue)
{
    n = nNewValue;
    d = dNewValue;
}

// ToString - перекрытие метода ToStringO и реализация
// интерфейса IDisplayable
override public string ToString()
{
    return string.Format("{0:N}, {1:N})\ n, d);
}

public class Program
{
    public static void Main(string[] args)
    {
        // Создание объекта Test
        Test test = new Test(10);
        Console.WriteLine("Начальное значение test");
        OutputFunction(test);

        // Попытка модифицировать объект, передавая его в
        // качестве аргумента
        ChangeValueFunction(test, 100, 200.0);
        Console.WriteLine("Значение test после вызова" +
                          " ChangeValueFunction(100, 200.0)");
        OutputFunction(test);

        // Попытка модифицировать объект, передавая его в
        // качестве аргумента
        ChangeReferenceFunction(ref test, 100, 200.0);
        Console.WriteLine("Значение test после вызова" +
                          " ChangeReferenceFunction(100, 200.0)");
        OutputFunction(test);

        // Метод может модифицировать объект
        test.ChangeMethod(1000, 2000.0);
        Console.WriteLine("Значение test после вызова" +
                          " ChangeMethod(1000, 2000.0)");
        OutputFunction(test);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                          "завершения программы...");
    }
}

```

```

        Console.Read();
    }

    // ChangeValueFunction - передача структуры по значению j
    public static void ChangeValueFunction(Test t,
                                           int newValue, double dNewValue)
    {
        t.N = newValue,-
        Test.D = dNewValue;
    }

    // ChangeReferenceFunction - передача структуры по
    // ссылке
    public static void ChangeReferenceFunction(ref Test t,
                                              int newValue, double dNewValue)
    {
        t.N = newValue;
        Test.D = dNewValue;
    }

    // OutputFunction - вывод информации об объекте, который
    // реализует метод ToString()
    public static void OutputFunction(IDisplayable id)
    {
        Console.WriteLine("id = {0}", id.ToString());
    }
}
}

```

Программа `StructureExample` сначала определяет интерфейс `IDisplayable`, а затем простую структуру `Test`, которая реализует этот интерфейс. `Test` также определяет два члена — член экземпляра `p` и статический член `d`. Статический инициализатор устанавливает член `d` равным 20; инициализатор для члена `p` не разрешен.

Структура `Test` определяет конструктор, свойство экземпляра `N` и статическое свойство `D`.

`Test` также определяет собственный метод `ChangeMethod()` и перекрывает метод `ToString()`. Предоставлением метода `ToString()` структура `Test` реализует интерфейс `IDisplayable`.

Функция `Main()` создает объект `test` вне локальной памяти и использует конструктор для инициализации выделенной ему памяти. Затем `Main()` вызывает функцию `OutputFunction()` для вывода объекта.

Далее функция `Main()` вызывает функцию `ChangeValueFunction()`, передавая ей `test` с двумя числовыми константами. Функция `ChangeValueFunction()` присваивает эти значения членам `p` и `d` структуры `Test`. После возврата из функции `ChangeValueFunction()` вызывается функция `OutputFunction()`, которая позволяет убедиться, что значение `d` изменилось, а значение `p` — нет.

Вызов `ChangeValueFunction()` получает объект `test` по значению, так что объект `t` в теле этой функции представляет собой копию исходного объекта `test`, а не сам объект. Таким образом, присваивание `t.N` изменяет локальную копию и никак не влияет на объект `test` функции `Main()`. Однако все объекты структуры `Test` совместно ис-

пользуют статический член `d`, так что присваивание `Test.D` изменяет `d` для всех объектов, включая `test`.

Следующей вызывается функция `ChangeReferenceFunction()`, которая выглядит в точности как и функция `ChangeValueFunction()`, за исключением использования в списке аргументов ключевого слова `ref`. Это ключевое слово обеспечивает передачу объекта `test` функции по ссылке, так что объект `t` в функции является ссылкой на исходный объект `test`, а не вновь созданной копией.

Последним вызовом в `Main()` является вызов метода `ChangeMethod()`, всегда использующий передачу текущего объекта по ссылке, поэтому изменения, сделанные при вызове этого метода, сохраняются при возврате в `Main()`.

Вывод программы имеет следующий вид:

```
Начальное значение test
id = (10.00, 20.00)
Значение test после вызова ChangeValueFunction(100, 200.0)
id = (10.00, 200.00)
Значение test после вызова ChangeReferenceFunction(100, 200.0)
id = (100.00, 200.00)
Значение test после вызова ChangeMethod(1000, 2000.0)
id = (1,000.00, 2,000.00)
Нажмите <Enter> для завершения программы...
```

Унификация системы типов

Структуры и классы имеют одну общую черту: и те и другие порождены из класса `Object`, указано ли это явно или нет. Этот факт унифицирует различные типы переменных в одну всеобъемлющую иерархию классов.

Предопределенные типы структур

Схожесть структур и простых типов-значений не только внешняя. В действительности *простые типы-значения являются структурами*. Например, `int` — просто другое имя структуры `Int32`, `double` — другое имя структуры `Double` и так далее. В табл. 14.1 приведен полный список типов и соответствующих имен структур.

Таблица 14.1. Имена структур для встроенных типов-значений

Имя типа	Имя структуры
<code>bool</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>sbyte</code>	<code>SByte</code>
<code>char</code>	<code>Char</code>
<code>decimal</code>	<code>Decimal</code>
<code>double</code>	<code>Double</code>
<code>float</code>	<code>Single</code>
<code>int</code>	<code>Int32</code>
<code>uint</code>	<code>UInt32</code>

Имя типа	Имя структуры
long	Int64
ulong	UInt64
object	Object
short	Int16
ushort	UInt16



Тип `string` является ссылочным, а не типом-значением, поэтому для него не существует соответствующей структуры. Вместо этого тип `string` соответствует классу `String`. Тип `string` в C# особенный и обладает рядом свойств, присущих структурам. Более детально он рассматривается в главе 9, "Работа со строками в C#".

Унификация системы типов с помощью структур



Тип `int` — другое имя для `Int32`. Поскольку все структуры порождены от класса `Object`, `int` также не должен быть исключением. Это приводит к очень интересным результатам, которые демонстрирует приведенная ниже программа.

```
// TypeUnification - демонстрация того, что int и Int32 в
// действительности одно и то же, и что они порождены от
// класса Object
using System;

namespace TypeUnification
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Создаем int и инициализируем его нулем
            int i = new int(); // Да, так можно делать
            // Присваиваем ему значение 1 и выводим при помощи
            // интерфейса IFormattable, который реализует Int32
            i = 1;
            OutputFunction(i);

            // Константа 2 также реализует IFormattable
            OutputFunction(2);

            // В действительности вы можете даже вызвать метод
            // константы
            Console.WriteLine("Непосредственный вывод = {0}",
                              3.ToString());

            // Это может быть полезным, например, для выбора
            // целого значения из списка:
```

```

Console.WriteLine("\nВыбираем из списка целые");
object[] objects = new object[5];
objects[0] = "это строка";
objects[1] = 2;
objects[2] = new Program();
objects[3] = 4;
objects[4] = 5.5;
for(int index = 0; index < objects.Length; index++)
{
    if (objects[index] is int)
    {
        int n = (int)objects[index];
        Console.WriteLine("Элемент {0} — {1}", index, n);
    }
}
// Унификация типов позволяет выводить типы-значения и
// ссылки, не различая их
Console.WriteLine("\nВывод всех объектов из списка");
int nCount = 0;
foreach(object o in objects)
{
    Console.WriteLine("Objects[{0}] - <{1}>",
        nCount++, o.ToString());
}

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}

// OutputFunction - вывод произвольного объекта,
// реализующего ToString()
public static void OutputFunction(IFormattable id)

    Console.WriteLine("Значение из OutputFunction = {0}",
        id.ToString());
}

// ToString - простая строковая функция
override public string ToString()
{
    return "TypeUnification Program";
}
}

```

Функция Main() начинается с создания объекта i типа int. В ней используется конструктор по умолчанию Int32 O (можно просто написать int O), который инициализирует переменную i нулевым значением. Далее программа присваивает переменной i значение 1. Это несколько отличается от формата, используемого при создании структуры.

Функция `Main()` передает переменную `i` функции `OutputFunction()`, которая явлена как принимающая объект, реализующий интерфейс `IFormattable`. Интерфейсом `IFormattable` похож на интерфейс `IDisplayable`, который определялся в других программах. Единственным методом интерфейса `IFormattable` является метод `ToString()`.

Функция `OutputFunction()` выводит объект `IFormattable` с использованием строки, возвращаемой его методом `ToString()`. У переменной `Int32`, которая реализует этот метод, не возникает никаких проблем. Но самое интересное, что никаких проблем не возникает и при вызове `OutputFunction(2)`. Поскольку константа `2` имеет тип `Int32`, она также реализует интерфейс `IFormattable`. И наконец, функция `Main()` вызывает `3.ToString()` непосредственно. Вывод этой части функции `Main()` имеет следующий вид:

```
Значение из OutputFunction = 1
Значение из OutputFunction = 2
Непосредственный вывод = 3
```

Далее функция `Main()` объявляет массив объектов типа `Object` и сохраняет в первом элементе массива `string`, во втором — `int`, в третьем — экземпляр класса `Program` и т.д. Все это можно сделать, так как `String`, `Int32` и `Program` порождены от `Object`. И так массив внутри класса `Program` хранит экземпляр `Program` — правда, это интересно?



Затем программа проходит по всем элементам массива. Функция `Main()` может выбрать из него все целые числа, запрашивая каждый объект, ЯВЛЯЕТСЯ ли он `Int32` с помощью ключевого слова `is`. Вывод этой части программы выглядит следующим образом:

```
Выбираем из списка целые
Элемент 1 — 2
Элемент 3 — 4
```

Программа завершается демонстрацией использования того факта, что все подклассы `Object` — т.е. все классы — реализуют метод `ToString()`. Таким образом, чтобы вывести все члены массива объектов, вам не надо беспокоиться об их типах. Функция `Main()` просто вновь проходит по массиву, вызывая метод `ToString()` для каждого его элемента. В результате программа выводит на экран следующую информацию:

```
Вывод всех объектов из списка
Objects[0] - от строка>
Objects[1] - <2>
Objects[2] - <TypeUnification Program>
Objects[3] - <4>
Objects[4] - <5.5>
Нажмите <Enter> для завершения программы...
```

Для класса `Program` реализован тривиальный метод `ToString()`, просто чтобы показать, как это работает.



Свойство `ToString()`, несомненно, поясняет магию функции `Console.WriteLine()`. Честно говоря, даже не смотря исходный текст, можно поспорить, что `Write()` принимает аргументы как принадлежащие типу `Object`. Затем она просто вызывает метод `ToString()` для получения выводимой строки, которая подставляется вместо соответствующего элемента форматирования `{p}` в первой строке.

Упаковка типов-значений

Что действительно делает ссылочные типы и типы-значения наподобие `int`, `bool`, `char` и любой структуры гражданами C# первого сорта — так это технология, называемая *упаковкой* (boxing). Во многих ситуациях компилятор временно конвертирует объекты типов-значений в ссылочные объекты. Упаковка означает перемещение части данных типа-значения в объект ссылочного типа в куче. Вот пример, в котором выполняется упаковка:

```
int i = 999;           // Простой int (тип-значение)
object o = i;          // Помещаем i в ссылочную упаковку
int j = (int)o;        // Получаем 999 из упаковки
```

Все, что было упаковано, рано или поздно потребует распаковки, которая влечет за собой приведение типа. В демонстрационной программе `TypeUnification` каждое присваивание `object` требовало упаковки, а обратное преобразование переменной `object` — распаковки.

Обе операции требуют определенного времени. Упаковка до 20 раз продолжительнее обычного присваивания, а распаковка — до 4 раз. Кроме того, упаковка требует дополнительной памяти для размещения объекта в куче, так что большое количество упаковок может снизить производительность вашей программы. Упаковка во многих ситуациях выполняется автоматически, включая такие ситуации, как передача аргумента, возврат значения из функции, присваивание, работа с массивами `object []`, вызовы `WriteLine()` и многое другое. По возможности избегайте упаковки — например, вызовами `ToString()` для значений, передаваемых `WriteLine()`, избегая работы с массивами `object` и используя новые обобщенные коллекции, рассматривающиеся в главе 15, "Обобщенное программирование".

Глава 15

Обобщенное программирование

В этой главе...

- Коллекционирование: преимущества и проблемы
- Экономия времени и кода с помощью обобщенных коллекций
- Написание собственных обобщенных классов, методов и интерфейсов



⁶ # предоставляет массу специализированных альтернатив массивам, о которых речь шла в главе 6, "Объединение данных — классы и массивы". В этой главе будут рассмотрены списки, стеки, очереди и другие "массивоподобные" классы коллекций, такие как универсальный `ArrayList`, который может использоваться для решения множества программистских задач. В отличие от массивов, эти коллекции не являются безопасными с точки зрения типов и могут вызвать определенные накладные расходы.

Однако можно сохранить массу времени и усилий, если воспользоваться обобщенной версией. *Обобщенные классы*⁶ (generics) — новая возможность C#, появившаяся в версии 2.0. Обобщенные классы представляют собой классы, методы и интерфейсы, в которых поля типов остаются незаполненными. Чтобы понять, о чем идет речь, рассмотрим конкретный пример. Так, класс `List<T>` определяет обобщенный список, очень похожий на `ArrayList`. Когда вы используете этот список для создания (инстанцирования) собственного списка, например чисел типа `int`, вы заменяете параметр типа `T` конкретным типом `int`:

```
List<int> myList = new List<int> (); // Список чисел типа int
```

Универсальность такого списка состоит в том, что вы можете инстанцировать `List<T>` для *любого единого* типа данных — `string`, `Student`, `BankAccount` — и при этом получить такую же безопасность типов, как у массива, причем без лишних затрат. Это — супермассив.

Обобщенные классы в C# могут быть встроенными, такими как `List<T>`, и пользовательскими, т.е. написанными вами. После чтения этой главы вы научитесь писать собственные обобщенные классы не хуже встроенных.

⁶ Здесь следует сделать небольшое пояснение. Дело в том, что это новая для C# возможность, а потому русскоязычная терминология еще не устоялась. В C++, в котором обобщенное программирование было реализовано существенно раньше, обобщенные, или универсальные классы, называются шаблонами. В C# такие классы именуются `generic class`, или просто `generic`. В русскоязычной литературе встречаются такие переводы, как обобщенные классы, универсальные классы и даже термин "джереник". В данной книге будет использоваться термин "обобщенные классы". — *Примеч. ред.*

Необобщенные коллекции

Чтобы понять, что такое обобщенные классы и что в них хорошего, давайте начнем рассмотрения обычных классов-коллекций.



Массивы обеспечивают быстрый и эффективный доступ к произвольным элементам. Но зачастую массивы не удовлетворяют вашим требованиям из-за их недостатков.

- ✓ Программа должна объявить размер массива при его создании. В отличие от Visual Basic, C# не позволяет изменять размер массива после его определения! делать, если вы не знаете заранее, массив какого размера вам потребуется?
- ✓ Вставка или удаление элемента из середины массива весьма неэффективна, 1 должны сдвинуть все элементы, чтобы освободить память.

Для решения этих проблем C# предоставляет ряд необобщенных коллекций в **System.Collections.Generic** в альтернатив массивам. Каждая из коллекций имеет свои сильные и слабые стороны.

Необобщенные коллекции

C# предоставляет ряд хорошо спроектированных альтернатив массивам. В табл. 15.1 описаны несколько наиболее полезных необобщенных коллекций. Вне сомнения, **по** характеристикам вы всегда сможете выбрать подходящий класс для решения стояи перед вами задачи (но не спешите — еще немного, и вы познакомитесь с обобщенными классами).

Таблица 15.1. Необобщенные классы коллекций

Класс	Характеристика
ArrayList	Автоматически растущий при необходимости массив. Этот наиболее часто используемый класс обладает всеми преимуществами массивов, но лишен их недостатков — хотя, конечно же, не является идеальным решением. В отличие от массивов, все рассматриваемые здесь коллекции способны при необходимости увеличиваться
LinkedList	В C# отсутствует связанный список, однако в одной из дополнительных глав вы узнаете, как разработать его самостоятельно. Но, скорее всего, после этих упражнений вы предпочтете воспользоваться новым обобщенным классом <code>LinkedList</code> . В нем быстрее, чем в массиве, выполняется вставка новых элементов, но доступ к определенному элементу в нем медленнее, чем в массиве или <code>ArrayList</code>
Queue	Это структура данных, построенная по принципу "первым вошел, первым вышел". Элементы добавляются к очереди с одного конца, и удаляются из очереди с другого. Вы не можете ни вставить, ни удалить элемент из середины очереди
Stack	Стандартная аналогия стопке пластинок. Для добавления элемента вы должны положить его на вершину стека, для удаления — снять с вершины. Работает по принципу "последний вошел, первым вышел". Вы не можете ни вставить, ни удалить элемент из середины стека
Словарь	Коллекция объектов, предназначенная для быстрого поиска. Вы можете быстро найти элемент по его ключу так же, как находите пояснение слова в словаре. Необобщенный "словарный" класс в C# называется <code>Hashtable</code>

Использование необобщенных коллекций



Коллекции легче применять, чем массивы. Для этого нужноinstancировать объект коллекции, добавить в него элементы и итеративно работать с ними (для этого лучше всего воспользоваться циклом `foreach`). Приведенная дальше демонстрационная программа иллюстрирует эту последовательность действий.

```
// NongenericCollections    демонстрация использования
// классов коллекций
using System;
using System.Collections;
namespace NongenericCollections
(
    public class Program
    {
        // Демонстрация ArrayList, Stack, Queue и Hashtable
        public static void Main(string[] args)
        {
            // ArrayList
            //
            // Инстанцирование ArrayList (вы можете указать
            // начальный размер, но можете этого и не делать)
            ArrayList aListWithSpecifiedSize = new ArrayList(1000);
            ArrayList aList = new ArrayList(); // размер по
                                                // умолчанию (16)
            aList.Add("one"); // Добавление в конец списка
            aList.Add("two"); // В списке - "one","two"
            aList.Add("three"); // В списке - "one","two","three"
            Console.WriteLine("{0} items in the ArrayList:",
                              aList.Count);
            // Цикл с использованием foreach
            foreach(string s in aList)
            {
                // Выводим строку и ее индекс в ArrayList
                Console.WriteLine(s + " в ({0})", aList.IndexOf(s));
            }
            //
            // Stack
            //
            // Инстанцируем стек
            Stack stack = new Stack();
            // Вносим элементы в стек и снимаем с него один
            // элемент
            stack.Push("one");
            stack.Push("two"); // "two","one"
            stack.Push("three"); // "three","two","one"
            Console.WriteLine("{0} элементов в стеке: ",
                              stack.Count);
            foreach (string s in stack)
            {
                Console.WriteLine(s) ;
            }
        }
    }
}
```



```

        string sval = (string)stack.Pop(); // "two","one"
        Console.WriteLine("Снят элемент:");
        Console.WriteLine(sval);
        Console.WriteLine("Вершина стека: {0}", stack.Peek());
        //
        // Queue
        //
        // Инстанцирование очереди
        Queue queue = new Queue();
        // Постановка в очередь нескольких элементов
        queue.Enqueue("one");
        queue.Enqueue("two");
        queue.Enqueue("three"); // "one","two","three"
        Console.WriteLine("{0} элементов в очереди:",
                           queue.Count);
        foreach (string s in queue)
        {
            Console.WriteLine(s);
        }
        Console.WriteLine("Вывод из очереди: {0}",
                           queue.Dequeue());
        Console.WriteLine("Голова очереди: {0}",
                           queue.Peek());
        //
        // Hashtable
        //
        // Инстанцирование Hashtable (словарь)
        Hashtable table = new Hashtable();
        Student student1 = new Student("Randy");
        Student student2 = new Student("Chuck");
        // Добавляем объект Student, ключом является его имя
        table.Add(student1.Name, student1);
        table.Add(student2.Name, student2);
        // Порядок неизвестен
        Console.WriteLine("{0} элементов в словаре:",
                           table.Count);
        // Элементы, возвращаемые из словаря, имеют тип
        // DictionaryEntry
        foreach(DictionaryEntry de in table)
        {
            // Приведение свойства DictionaryEntry.Value к типу
            // Student
            Student stu = (Student)de.Value;
            Console.WriteLine(stu.Name);
        }
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
    } Console.Read();
}
public class Student
{

```

```

private string name;
public Student(string name)
{
    this.name = name;
}
public string Name
{
    get { return name; }
}

```

Подробно останавливаться на том, как работают использованные в демонстрационной программе классы-коллекции, нет необходимости, так как сейчас вы познакомитесь с более интересными обобщенными классами. Получить информацию о необобщенных классах-коллекциях можно в разделе "System.Collections namespace" справочной системы.

Обобщенные классы

Теперь, при доступности в C# обобщенных классов, вы вряд ли захотите использовать описанные в предыдущем разделе необобщенные классы. Обобщенные классы предпочтительнее по двум причинам: безопасность и производительность.

Обобщенные классы безопасны



Когда вы объявляете массив, вы должны указать точный тип данных, которые могут в нем храниться. Если это `int` — то массив не может хранить ничего, кроме `int` или других числовых типов, которые C# в состоянии неявно преобразовать в `int`. Если вы попытаетесь поместить в массив данные неверного типа, то получите от компилятора сообщение об ошибке. Таким образом компилятор обеспечивает *безопасность типов*, т.е. вы обнаруживаете и исправляете проблему еще до того, как она проявится. Гораздо лучше получить сообщение об ошибке от компилятора, чем в процессе работы программы.



Необобщенные коллекции небезопасны. В C# переменная любого типа ЯВЛЯЕТСЯ `Object`, поскольку класс `Object` является базовым классом для всех других типов, как типов-значений, так и типов-ссылок (см. раздел об унификации типов в главе 14, "Интерфейсы и структуры"). Однако когда вы сохраняете типы-значения (числа, `bool`, `struct`) в коллекции, они должны быть *упакованы* при помещении в нее и *распакованы* при извлечении из нее (см. окончание главы 14, "Интерфейсы и структуры").

Первое следствие небезопасности необобщенных классов заключается в том, что вам требуется приведение типов (как показано в следующем фрагменте исходного текста) для получения исходного объекта из `ArrayList`, так как этот тип скрыт при упаковке.

```

ArrayList aList = new ArrayList();
// Добавляем пять-шесть элементов, а затем...
string myString = (string)aList[4]; // преобразуем в string

```



Второе следствие в том, что в `ArrayList` *одновременно* могут храниться *объекты разных типов*. То есть вы можете написать, например, такой исходный текст:

```
ArrayList aList = new ArrayListO;
aList.Add("a string"); // string -- OK
aList.Add(3);          // int    -- OK
aList.Add(aStudent);   // Student -- OK
```

Однако если вы поместите в `ArrayList` (или другую необобщенную *коллекции* объекты разных несовместимых типов, то как вы потом сможете узнать тип, *наприм*р третьего элемента? Если это `Student`, а вы попытаетесь преобразовать его в `string` то получите ошибку времени выполнения программы.



Для безопасности следует производить проверку с использованием *оператор* `is` (рассматривавшегося в главе 12, "Наследование") или альтернативного я ратора `as` следующим образом:

```
if(aList[i] is Student) // Объект - Student?
{
    Student aStudent = (Student)aList[i]; // Да
}

// или ...

Student aStudent = aList[i] as Student; // Получаем Student
if(aStudent != null) // Невозможно, "as"
{ // возвращает null
    // Можно работать с aStudent
}
```

Избавиться от лишней работы можно посредством обобщенных классов, ные коллекции работают как и массивы: вы определяете один и только один тип, кото| рый может храниться в коллекции при ее объявлении.

Обобщенные классы эффективны

Полиморфизм позволяет типу `Object` хранить любой другой тип. Однако за удобство приходится платить упаковкой и распаковкой типов-значений при размете] их в необобщенных коллекциях.

Упаковка не так уж снижает эффективность, если ваша коллекция мала. Но если ва перемещаете тысячи или даже миллионы целых чисел типа `int` в необобщенной *кн* лекции, это может занять примерно в 20 раз больше времени (и потребовать дополни тельной памяти) по сравнению с хранением объектов ссылочного типа. Упаковка *так* может привести к некоторым трудно находимым ошибкам. Обобщенные коллекции» знакомы с проблемами, связанными с упаковкой и распаковкой.

Использование обобщенных коллекций

Теперь, когда вы знаете, почему обобщенные коллекции предпочтительнее *набб* обобщенных, пришло время познакомиться с тем, как они используются. В табл. 15.2 *прд*

ставлен частичный список обобщенных классов коллекций (в третьем столбце таблицы указаны их необобщенные эквиваленты).

Таблица 15.2.		
Класс	Описание	Аналог
<code>List<T></code>	Динамический массив	<code>ArrayList</code>
<code>LinkedList<T></code>	Связанный список	<code>LinkedList</code> из дополнительной главы
<code>Queue<T></code>	Список "первым вошел, первым вышел"	<code>Queue</code>
<code>Stack<T></code>	Список "последним вошел, первым вышел"	<code>Stack</code>
<code>Dictionary<T></code>	Коллекция пар ключ/значение	<code>Hashtable</code>

Помимо указанных классов, имеются и другие, а также несколько соответствующих интерфейсов для большинства из них, таких как `ICollection<T>` или `IList<T>`. За более подробной информацией о них обратитесь к разделу "System.Collections.Generic namespace" справочной системы.

Понятие `<T>`

В этой странно выглядящей записи `<T>` обозначает место, куда будет помещен некий реальный тип. Чтобы вызвать к жизни этот символический объект, его инстанцируют путем указания реального типа:

```
list<int> intList =  
    new List<int>(); // Инстанцирование для int
```

Например, в следующем разделе `List<T>` будет инстанцирован для типов `int`, `string` и `Student`. Кстати говоря, `T` отнюдь не священная корова, и вместо него можно использовать все что угодно — например `<dummy>` или `<myType>`. Обычно для параметра типа применяются буквы `T`, `U`, `V` и т.д.

Использование `List<T>`



Если `ArrayList` — один из наиболее часто используемых необобщенных классов коллекций, то его обобщенный двойник — `List<T>`. Его применение проиллюстрировано в демонстрационной программе `GenericCollections`. (Для того чтобы компилировать эту программу, вы должны закомментировать строки, приводящие [ошибкам] времени компиляции.) Полный текст программы можно найти на прилагаемом компакт-диске.

```
// GenericCollections - демонстрация обобщенных коллекций  
using System;  
using System.Collections;  
using System.Collections.Generic;  
namespace GenericCollections  
{  
    public class Program  
    {  
        //
```

```

public static void Main(string[] args)
{
    // Объявление ArrayList для сравнения
    ArrayList aList = new ArrayList();
    // List<T>: обратите внимание на угловые скобки и
    // параметр типа T
    List<string> sList =
        new List<string>(); // Инстанцирование для string
    sList.Add("one");
    sList.Add(3);           // Ошибка компиляции!
    sList.Add(
        new Student("du Bois")); // Ошибка компиляции!
    // Инстанцирование для int
    List<int> intList = new List<int>();
    intList.Add(3);        // Никакой упаковки
    intList.Add(4);
    Console.WriteLine("Вывод intList:");
    foreach(int i in intList) // Цикл foreach работает для
                               // всех коллекций
    {
        // Обратите внимание: приведения типа нет
        Console.WriteLine("int i = " + i.ToString());
    }
    // Инстанцирование для Student
    List<Student> studentList = new List<Student>();
    Student student1 = new Student("Vigil11");
    Student student2 = new Student("Finch");
    studentList.Add(student1);
    studentList.Add(student2);
    Student[] students = new Student[]
    { new Student("Mox"), new Student("Fox") };
    studentList.AddRange(students); // Добавляем весь
                                     // массив в List
    Console.WriteLine("Студентов в studentList = {0}",
        studentList.Count);
    // Поиск при помощи IndexOf()
    Console.WriteLine("Student2 в " +
        studentList.IndexOf(student2));
    string name = studentList[3].Name; // Обращение при
                                         // помощи индекса
    if(studentList.Contains(student1)) // Поиск Contains()
    {
        Console.WriteLine(student1.Name + " есть в списке");
    }
    studentList.Sort(); // Считаем, что Student реализует
                        // интерфейс IComparable
    studentList.Insert(3, new Student("Ross"));
    studentList.RemoveAt(3); // Удаляем элемент
    // name определено выше
    Console.WriteLine("Удаляем {0}", name);
    Student[] moreStudents =
        studentList.ToArray(); // Преобразуем список в массив

```

```
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы. ..");
Console.Read();
```

```
public class Student : IComparable
// См. полную версию программы на прилагаемом
// компакт-диске
```

В приведенном листинге имеется три инстанцирования `List<T>`: для `int`, `string` в `Student`. В программе также продемонстрировано следующее:

- ✓ безопасность типов, позволяющая избежать добавления данных неверного типа;
- ✓ возможность применения для коллекции `List<T>` цикла `foreach`, как и для любой другой коллекции;
- ✓ добавление объектов, как по одному, так и сразу целым массивом;
- ✓ сортировка списка (в предположении, что элементы реализуют интерфейс `IComparable`);
- ✓ вставка нового элемента между имеющимися;
- ✓ получение количества элементов в списке;
- ✓ проверка, содержится ли в списке конкретный объект;
- ✓ удаление элемента из списка;
- ✓ копирование элементов из списка в массив.

Это только небольшой пример использования методов `List<T>`. У других обобщений коллекций имеются свои наборы методов, однако все они схожи в применении.



Главное улучшение заключается в том, что компилятор предупреждает добавление в коллекцию данных типа, отличного от того, для которого она инстанцирована.

Создание собственного обобщенного класса

Помимо встроенных обобщенных классов коллекций, C# позволяет написать собственные обобщенные классы — как коллекции, так и другие типы классов. **Главное**, что вы имеете возможность создать обобщенные версии классов, которые проектированы *вами*.

Определение обобщенного класса переполнено записями `<T>`. Когда вы инстанцируете такой класс, вы указываете тип, который заменит `T` так же, как и в случае усмотренных обобщенных коллекций. Посмотрите, насколько схожи приведенные ниже объявления:

```
LinkedList<int> aList = new LinkedList<int> ();
MyClass<int> aClass = new MyClass<int> ();
```

Оба являются инстанцированиями классов: одно — встроенного, второе — польвательского. Не каждый класс имеет смысл делать обобщенным, но далее в главе будет рассмотрен пример класса, который следует сделать именно таковым.



Классы, которые логически могут делать одни и те же вещи с данными разных типов — наилучшие кандидаты в обобщенные классы. Наиболее типичным примером являются коллекции, способные хранить различные данные. Если в какой-то момент у вас появляется мысль: "А ведь мне придется написать версию этого класса еще и для объектов `Student`", — вероятно, ваш класс стоит сделать обобщенным.

Чтобы показать, как пишутся собственные обобщенные классы, будет разработан обобщенный класс для очереди специального вида, а именно очереди с приоритетами.

Очередь с приоритетами

Представим себе почтовую контору наподобие FedEx. В нее поступает постоянный поток пакетов, которые надо доставить получателям. Однако пакеты неравны по возможности: некоторые из них следует доставить немедленно (для них уже ведутся разработка телепортаторов), другие можно доставить авиапочтой, а третьи могут быть доставлены наземным транспортом.

Однако в контору пакеты приходят в произвольном порядке, так что при поступлении очередного пакета его нужно поставить в очередь на доставку. Слово прозвучало — необходима очередь, но очередь необычная. Вновь прибывшие пакеты становятся в очередь на доставку, но часть из них имеет более высокий приоритет и должна ставиться если и не в самое начало очереди, то уж точно не в ее конец.

Попробуем сформулировать правила такой очереди. Итак, имеются входящие пакеты с высоким, средним и низким приоритетом. Ниже описан порядок их обработки.

- ✓ **Пакеты с высоким приоритетом** помещаются в начало очереди, но после других пакетов с высоким приоритетом, уже присутствующих в ней.
- ✓ **Пакеты со средним приоритетом** ставятся в начало очереди, но после пакетов с высоким приоритетом и других пакетов со средним приоритетом, уже присутствующих в ней.
- ✓ **Пакеты с низким приоритетом** ставятся в конец очереди.

C# предоставляет встроенный обобщенный класс очереди, но он не подходит для создания очереди с приоритетами. Таким образом, нужно написать собственный класс очереди, но как это сделать? Распространенный подход заключается в разработке класса-оболочки (wrapper class) для нескольких очередей:

```
class Wrapper // Или PriorityQueue!  
{  
    Queue queueHigh = new Queue ();  
    Queue queueMedium = new Queue ();  
    Queue queueLow = new Queue ();  
    // Методы для работы с этими очередями...
```

Оболочка инкапсулирует три обычные очереди (которые могут быть обобщенными)» управляет внесением пакетов в эти очереди и получением их из очередей. Стандартный интерфейс класса `Queue`, реализованного в C#, содержит два ключевых метода:



✓ Enqueue () — для помещения объектов в конец очереди;

✓ Dequeue () — для извлечения объектов из очереди.



Оболочки представляют собой классы (или функции), которые инкапсулируют сложную работу. Оболочки могут иметь интерфейс, существенно отличающийся от интерфейса(ов) использованных в нем элементов. Однако в данном случае интерфейс оболочки совпадает с интерфейсом обычной очереди. Класс реализует метод Enqueue (), который получает пакет и его приоритет, и на основании приоритета принимает решение о том, в какую из внутренних очередей его поместить. Он также реализует метод Dequeue (), который находит пакет с наивысшим приоритетом в своих внутренних очередях и извлекает его из очереди. Дадим рассматриваемому классу-оболочке формальное имя PriorityQueue.



Вот исходный текст этого класса:

```
// PriorityQueue - демонстрация использования объектов
// низкоуровневой очереди для реализации высокоуровневой
// обобщенной очереди, в которой объекты хранятся с учетом
// их приоритета
using System;
using System.Collections.Generic;
namespace PriorityQueue
(
    class Program
    {
        //Main - заполняем очередь с приоритетами пакетами,
        // затем извлекаем из очереди их случайное количество
        static void Main(string[] args)
        {
            Console.WriteLine("Создание очереди с приоритетами:");
            PriorityQueue<Package> pq =
                new PriorityQueue<Package>();
            Console.WriteLine("Добавляем случайное количество" +
                " (0 - 20) случайных пакетов" +
                " в очередь:");

            Package pack;
            PackageFactory fact = new PackageFactory();
            // Нам нужно случайное число, меньшее 20
            Random rand = new Random();
            // Случайное число в диапазоне 0 - 20
            int numToCreate = rand.Next(20);
            Console.WriteLine("^Создание {0} пакетов: ",
                numToCreate);
            for (int i = 0; i < numToCreate; i++)
            {
                Console.WriteLine("\t^Непакет^ и добавление " +
                    "случайного пакета {0}", i);
                pack = fact.CreatePackage();
                Console.WriteLine(" с приоритетом {0}",
                    pack.Priority);
            }
        }
    }
}
```



```

        pq.Enqueue(pack);
    }
    Console.WriteLine("Что получилось:");
    int nTotal = pq.Count;
    Console.WriteLine("Получено пакетов: {0}", nTotal);
    Console.WriteLine("Извлекаем случайное количество" +
        " пакетов: 0-20: ");
    int numToRemove = rand.Next(20);
    Console.WriteLine("^Извлекаем {0} пакетов",
        numToRemove);
    for (int i = 0; i < numToRemove; i++)
    {
        pack = pq.Dequeue();
        if (pack != null)
        {
            Console.WriteLine("\b\bДоставка пакета " +
                "с приоритетом {0}",
                pack.Priority);
        }
    }
    // Сколько пакетов "доставлено"
    Console.WriteLine("Доставлено {0} пакетов",
        nTotal - pq.Count);
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
        "завершения программы...");
    Console.Read();
}

// Priority - вместо числовых приоритетов наподобие 1, 2,
// 3, ... используем приоритеты с именами
enum Priority // Об enum мы поговорим позже
{
    Low, Medium, High
}

// IPrioritizable - определяем пользовательский интерфейс:
// классы, которые могут быть добавлены в PriorityQueue,
// должны реализовывать этот интерфейс
interface IPrioritizable
{
    // Пример свойства в интерфейсе
    Priority Priority { get; }
}

// PriorityQueue - обобщенный класс очереди с приоритетами;
// типы данных, добавляемых в очередь, обязаны
// реализовывать интерфейс IPrioritizable
class PriorityQueue<T>
    where T : IPrioritizable // <-- этот момент мы
    // обсудим позже
{
    // Queues - три внутренние (обобщенные!) очереди
    private Queue<T> queueHigh = new Queue<T>();
    private Queue<T> queueMedium = new Queue<T>();
    private Queue<T> queueLow = new Queue<T>();

```

```

//Enqueue - Добавляет T в очередь в соответствии с
// приоритетом
public void Enqueue(T item)
{
    switch (item.Priority) // Требуется реализации
    {                       // IPrioritizable
    case Priority.High:
        queueHigh.Enqueue(item);
        break;
    case Priority.Low:
        queueLow.Enqueue(item);
        break;
    case Priority.Medium:
        queueMedium.Enqueue(item);
        break;
    default:
        throw new
            ArgumentOutOfRangeException(
                item.Priority.ToString(),
                "Неверный приоритет в PriorityQueue.Enqueue");
    }
}

//Dequeue - извлечение T из очереди с наивысшим
// приоритетом
public T Dequeue()
{
    // Просматриваем очередь с наивысшим приоритетом
    Queue<T> queueTop = TopQueue();
    // Очередь не пуста
    if (queueTop != null && queueTop.Count > 0)
        return queueTop.Dequeue(); // Возвращаем первый
                                    // элемент
    }
    // Если все очереди пусты, возвращаем null (здесь
    // можно сгенерировать исключение)
    return default(T); // Что это — мы рассмотрим позже

//TopQueue - непустая очередь с наивысшим приоритетом
private Queue<T> TopQueue()
{
    if (queueHigh.Count > 0) // Очередь с высоким
        return queueHigh;  // приоритетом пуста?
    if (queueMedium.Count > 0) // Очередь со средним
        return queueMedium; // приоритетом пуста?
    if (queueLow.Count > 0) // Очередь с низким
        return queueLow;    // приоритетом пуста?
    :- return queueLow;     // Все очереди пусты

//IsEmpty - Проверка, пуста ли очередь
public bool IsEmpty()
{
    // true, если все очереди пусты
    return (queueHigh.Count == 0) &

```

```

        (queueMedium.Count == 0) &
        (queueLow.Count == 0);

    } //Count - Сколько всего элементов во всех очередях?
    public int Count // Реализуем как свойство только
    {
        // для чтения
        get { return queueHigh.Count +
                queueMedium.Count +
                queueLow.Count; }
    }
}
//Package - пример класса, который может быть размещен в
// очереди с приоритетами
class Package : IPrioritizable
{
    private Priority priority;
    // Конструктор
    public Package(Priority priority)
    {
        this.priority = priority;
    }
    //Priority - возвращает приоритет пакета; только для
    // чтения

    public Priority Priority
    {
        get { return priority; }
    }
    //А также методы ToAddress, FromAddress, Insurance,
    //и другие...
}
// Класс PackageFactory опущен - см. полную версию на
// прилагаемом компакт-диске
}

```

Демонстрационная программа `PriorityQueue` несколько длиннее прочих демонстрационных программ в книге, поэтому внимательно рассмотрите каждую ее часть.

Распаковка пакета

Класс `Package` преднамеренно очень прост и написан исключительно для данной демонстрационной программы. Основное в нем — часть с приоритетом, хотя реальный класс `Package`, несомненно, должен содержать массу других членов. Все, что требуется классу `Package` для участия в данном пакете — это член-данные для хранения приоритета, конструктор для создания пакета с определенным приоритетом и метод (реализованный здесь как свойство только для чтения) для возврата значения приоритета.

Требуют пояснения два аспекта класса `Package`: тип приоритета и интерфейс `IPrioritizable`, реализуемый данным классом.

Определение возможных приоритетов

Приоритеты представляют собой перечислимый тип (enum) под названием `Priority`. Он выглядит следующим образом:

```
//Priority - вместо числовых приоритетов наподобие 1, 2,
// 3, ... используем приоритеты с именами
enum Priority
{
    Low, Medium, High
}
```

Реализация интерфейса IPrioritizable

Любой объект, поступающий в `PriorityQueue`, должен знать собственный приоритет (общий принцип объектно-ориентированного программирования гласит, что каждый объект отвечает сам за себя).



Можно просто неформально убедиться, что класс `Package` имеет член для получения его приоритета, но лучше заставить компилятор проверять это требование, т.е. то, что у любого объекта, помещаемого в `PriorityQueue`, имеется этот член.

Один из способов обеспечить это состоит в требовании, чтобы все объекты реализовывали интерфейс `IPrioritizable`:

```
// IPrioritizable - определяем пользовательский интерфейс:
// классы, которые могут быть добавлены в PriorityQueue,
// должны реализовывать этот интерфейс
interface IPrioritizable
```

```
    Priority Priority { get; }
```



Запись `{get;}` определяет, как должно быть описано свойство в объявлении интерфейса. Обратите внимание, что тело функции доступа `get` отсутствует, но интерфейс указывает, что свойство `Priority` — только для чтения и возвращает значение перечислимого типа `Priority`.

Класс `Package` реализует интерфейс путем предоставления реализации свойства `Priority`:

```
public Priority Priority
{
    get { return priority; }
```

Функция Main()

Перед тем как приступить к исследованию класса `PriorityQueue`, стоит посмотреть, как он применяется на практике. Вот исходный текст функции `Main()`:

```
//Main - заполняем очередь с приоритетами пакетами,
// затем извлекаем из очереди их случайное количество
static void Main(string[] args)
{
    Console.WriteLine("Создание очереди с приоритетами:");
    PriorityQueue<Package> pq =
        new PriorityQueue<Package>();
    Console.WriteLine("Добавляем случайное количество" +
```

```

        *      " (0 - 20) случайных пакетов" +
        "в очередь:");
Package pack;
PackageFactory fact = new PackageFactory();
// Нам нужно случайное число, меньшее 20
Random rand = new Random();
// Случайное число в диапазоне 0 - 20
int numToCreate = rand.Next(20);
Console.WriteLine ("^Создание {0} пакетов: ",
                    numToCreate);
for (int i = 0; i < numToCreate; i++)
{
    Console.WriteLine("\t\tТекущее создание и добавление " +
                      "случайного пакета {0}", i);
    pack = fact.CreatePackage();
    Console.WriteLine(" с приоритетом {0}",
                      pack.Priority);
    pq.Enqueue(pack);
}
Console.WriteLine("Что получилось: ");
int nTotal = pq.Count;
Console.WriteLine("Получено пакетов: {0}", nTotal);
Console.WriteLine("Извлекаем случайное количество" +
                  " пакетов: 0-20: ");
int numToRemove = rand.Next(20);
Console.WriteLine ("^Извлекаем {0} пакетов",
                    numToRemove);
for (int i = 0; i < numToRemove; i++)
{
    pack = pq.Dequeue();
    if (pack != null)
    {
        Console.WriteLine("\t\tДоставка пакета
                          "с приоритетом {0}",
                          pack.Priority);
    }
}
// Сколько пакетов "доставлено"
Console.WriteLine("Доставлено {0} пакетов",
                  nTotal - pq.Count);
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
                  "завершения программы...");
} Console.Read();

```

Итак, что же происходит в функции Main () ?

1. Инстанцируется объект PriorityQueue для типа Package.
2. Создается объект PackageFactory, работа которого состоит в формировании новых пакетов со случайно выбранными приоритетами. (Фабрика— это класс или метод, который создает для вас объекты.)

3. Для генерации случайного числа используется класс `Random` из библиотеки `.NET`, а затем вызывается `PackageFactory` для создания соответствующего количества новых объектов `Package` со случайными приоритетами.
4. Выполняется добавление созданных пакетов в `PriorityQueue` с помощью вызова `pg.Enqueue(pack)`.
5. Выводится число созданных пакетов, после чего некоторое случайное их количество извлекается из `PriorityQueue`.
6. Функция завершается выводом количества извлеченных из `PriorityQueue` пакетов.

Написание обобщенного кода

Как же написать собственный обобщенный класс со всеми этими `<T>`? Выглядит это, конечно, устрашающе, но все не так уж и страшно.



Простейший путь написания обобщенного класса состоит в создании сначала его необобщенной версии, а затем расстановки в ней всех этих `<T>`. Так, например, вы можете написать класс `PriorityQueue` для объектов `Package`, протестировать его, а затем "обобщить".

Вот небольшая часть необобщенного класса `PriorityQueue` для иллюстрации сказанного:

```
public class PriorityQueue
{
    //Queues - три внутренние (обобщенные!) очереди
    private Queue<Package> queueHigh = new Queue<Package>();
    private Queue<Package> queueMedium = new Queue<Package>();
    private Queue<Package> queueLow = new Queue<Package>();
    //Enqueue - на основании приоритета Package добавляем его
    // в соответствующую очередь
    public void Enqueue(Package item)
    {
        switch(item.Priority) // Package имеет это свойство
        {
            case Priority.High:
                queueHigh.Enqueue(item);
                break;
            case Priority.Low:
                queueLow.Enqueue(item);
                break;
            case Priority.Medium:
                queueMedium.Enqueue(item);
                break;
        }
    }
    // и так далее ...
}
```

Написание необобщенного класса упрощает тестирование его логики. Затем, после тестирования и исправления всех ошибок, вы можете сделать контекстную замену `Package` на `<T>` (конечно, все не так прямолинейно, но и не очень отличается от сказанного).

Обобщенная очередь с приоритетами

Теперь пришло время разобраться с основным классом, из-за которого все и затеялось — с обобщенным классом `PriorityQueue`.

Внутренние очереди

Класс `PriorityQueue` — оболочка, за которой скрываются три обычных объекта `Queue<T>`, по одному для каждого уровня приоритета. Вот первая часть исходного текста `PriorityQueue`, в которой показаны эти три внутренние очереди:

```
//PriorityQueue - обобщенный класс очереди с приоритетами;
// типы данных, добавляемых в очередь, обязаны
// реализовывать интерфейс IPrioritizable
class PriorityQueue<T>
    where T : IPrioritizable
{
    //Queues - три внутренние (обобщенные!) очереди
    private Queue<T> queueHigh = new Queue<T>();
    private Queue<T> queueMedium = new Queue<T>();
    private Queue<T> queueLow = new Queue<T>();
    // Все остальное мы вот-вот рассмотрим...
```

В данных строках объявляются три закрытых члена-данных типа `Queue<T>`, инициализируемые путем создания соответствующих объектов `Queue<T>`.

Метод Enqueue()

`Enqueue()` добавляет элемент типа `T` в `PriorityQueue`. Работа состоит в том, чтобы выяснить приоритет элемента и поместить его в соответствующую приоритету очередь. В первой строке метод получает приоритет элемента и использует конструкцию `switch` для определения целевой очереди исходя из полученного значения. Например, получив элемент с приоритетом `Priority.High`, метод `Enqueue()` помещает его в очередь `queueHigh`. Вот исходный текст метода `PriorityQueue.Enqueue()`:

```
// Добавляет элемент T в очередь на основании значения его
// приоритета
public void Enqueue(T item)
{
    switch (item.Priority) // Требуется реализации
    {                       // IPrioritizable
    case Priority.High:
        queueHigh.Enqueue(item);
        break;
    case Priority.Low:
        queueLow.Enqueue(item);
        break;
    case Priority.Medium:
        queueMedium.Enqueue(item);
        break;
    default:
        throw new
            ArgumentOutOfRangeException(
                item.Priority.ToString(),
```

"Неверный приоритет в PriorityQueue.Enqueue");



Метод Dequeue()

Работа метода Dequeue () немного более хитрая. Он должен найти непустую очередь элементов с наивысшим приоритетом и выбрать из нее первый элемент. Первую часть своей работы — поиск непустой очереди элементов с наивысшим приоритетом — Dequeue () делегирует закрытому методу TopQueue (), который будет описан ниже. Затем метод Dequeue () вызывает метод Dequeue () найденной очереди для извлечения из нее объекта, который и возвращает. Вот исходный текст метода Dequeue ():

```
//Dequeue - извлечение T из очереди с наивысшим
// приоритетом
public T Dequeue ()

    // Просматриваем очередь с наивысшим приоритетом
    Queue<T> queueTop = TopQueue();
    // Очередь не пуста
    if (queueTop != null && queueTop.Count > 0)
    {
        return queueTop.Dequeue(); // Возвращаем первый
                                   // элемент
    }
    // Если все очереди пусты, возвращаем null (здесь
    // можно сгенерировать исключение)
    return default(T);
}
```

Единственная сложность состоит в том, как поступить, если все внутренние очереди пусты, т.е. по сути пуста очередь PriorityQueue в целом? Что следует вернуть в этом случае? Представленный метод Dequeue () в этом случае возвращает значение null. Таким образом, клиент — код, вызывающий PriorityQueue.Dequeue () — должен проверять, не вернул ли метод Dequeue () значение null. Где именно возвращается значение null? В default (T), в конце исходного текста метода. О выражении default (T) речь пойдет чуть позже.

Вспомогательный метод TopQueue()

Метод Dequeue () использует вспомогательный метод TopQueue () для того, чтобы найти непустую внутреннюю очередь с наивысшим приоритетом. Метод TopQueue () начинает с очереди queueHigh и проверяет ее свойство Count. Если оно больше 0, очередь содержит элементы, так что метод TopQueue () возвращает ссылку на эту внутреннюю очередь (тип возвращаемого значения метода TopQueue () — Queue<T>). Если же очередь queueHigh пуста, метод TopQueue () повторяет свои действия с очередями queueMedium и queueLow.

Что происходит, если все внутренние очереди пусты? В этом случае метод TopQueue () мог бы вернуть значение null, но более полезным будет возврат одной из пустых очередей. Когда после этого метод Dequeue () вызовет метод Dequeue () возвращенной очереди, тот вернет значение null. Вот как выглядит исходный текст метода TopQueue ():


```
//TopQueue - непустая очередь с наивысшим приоритетом
private Queue<T> TopQueue()
{
    if (queueHigh.Count > 0) // Очередь с высоким
        return queueHigh; // приоритетом пуста?
    if (queueMedium.Count > 0) // Очередь со средним
        return queueMedium; // приоритетом пуста?
    if (queueLow.Count > 0) // Очередь с низким
        return queueLow; // приоритетом пуста?
    return queueLow; // Все очереди пусты
}
```

Остальные члены PriorityQueue

Полезно знать, пуста ли очередь PriorityQueue или нет, и если нет, то сколько элементов в ней содержится (каждый объект отвечает сам за себя!). Вернитесь к листингу демонстрационной программы и рассмотрите исходный текст метода IsEmpty () и свойства Count класса PriorityQueue.

Незавершенные дела



PriorityQueue все еще нуждается в небольшой доработке.

- ✓ Сам по себе класс PriorityQueue не защищен от попыток инстанцирования для типов, например, int, string или Student, т.е. типов, не имеющих приоритетов. Вы должны наложить ограничения на класс с тем, чтобы он мог быть инстанцирован только для типов, реализующих интерфейс IPrioritizable. Попытки инстанцировать PriorityQueue для классов, не реализующих IPrioritizable, должны приводить к ошибкам времени компиляции.
- ✓ Метод Dequeue () класса PriorityQueue возвращает значение null вместо реального объекта. Однако обобщенные типы наподобие <T> не имеют естественного значения null по умолчанию, как, например, int или string. Эта часть метода Dequeue () также требует обобщения.

Добавление ограничений

Класс PriorityQueue должен быть способен запросить у помещаемого в очередь объекта о его приоритете. Для этого все классы, объекты которых могут быть размещены в PriorityQueue, должны реализовывать интерфейс IPrioritizable, как это делает класс Package. Класс Package указывает интерфейс IPrioritizable в заголовке своего объявления:

```
class Package : IPrioritizable
```

после чего реализует свойство Priority интерфейса IPrioritizable.



Компилятор в любом случае сообщит об ошибке, если один из методов обобщенного класса вызовет метод, отсутствующий у типа, для которого инстанцируется обобщенный класс. Однако лучше использовать явные *ограничения*. Поскольку вы можете инстанцировать обобщенный класс буквально для любого типа, должен быть способ указать компилятору, какие типы допустимы, а какие — нет.



Вы добавляете ограничение путем указания интерфейса `IPrioritizable` в заголовке `PriorityQueue`:

```
class PriorityQueue<T> where T: IPrioritizable
```

Обратите внимание на выделенную полужирным шрифтом конструкцию, начинающуюся со слова `where`. Это *принудитель* (enforcer), который указывает, что тип `T` обязан реализовывать интерфейс `IPrioritizable`, т.е. как бы говорит компилятору: "Убедись, подставляя конкретный тип вместо `T`, что он реализует интерфейс `IPrioritizable`, а иначе просто сообщи об ошибке".



Вы указываете ограничения, перечисляя в конструкции `where` *одно или несколько* имен:

- ✓ имя базового класса, от которого должен быть порожден класс `T` (или должен быть этим классом);
- ✓ имя интерфейса, который должен быть реализован классом `T`, как было показано в предыдущем примере.

Дополнительные варианты ограничений включают ключевые слова `struct`, `class` и `new()`. С `new()` вы встретитесь чуть позже в этой главе, а об ограничениях `struct` и `class` можно прочесть в разделе "Generics, constraints" справочной системы.

Эти варианты ограничений повышают гибкость в описании поведения обобщенного класса. Вот пример гипотетического обобщенного класса, объявленного с несколькими ограничениями на `T`:

```
class MyClass<T> : where T: class, IPrioritizable, new()
{
}
```

Здесь тип `T` должен быть классом, а не типом-значением; он должен реализовать интерфейс `IPrioritizable` и содержать конструктор без параметров. Достаточно



А если у вас есть два обобщенных параметра и оба должны иметь ограничения? (Да, да — вы можете использовать несколько обобщенных параметров одновременно!) Вот как можно записать две конструкции `where`:

```
class MyClass<T, U> : where T: IPrioritizable, where U: new()
```

Определение значения `null` для типа `T`

Как уже упоминалось ранее, у каждого типа есть свое значение по умолчанию, означающее "ничто" для данного типа. Для `int` и других типов чисел это `0` (или `0.0`). Для `string` — пустая строка `""`. Для `bool` это `false`, а для всех ссылочных типов, таких как `Package`, это `null`.

Однако поскольку обобщенный класс наподобие `PriorityQueue` может быть инстанцирован практически для любого типа данных, `C#` не в состоянии предсказать, каким *должно* быть правильное значение `null` в исходном тексте обобщенного класса. Например, в методе `Dequeue()` класса `PriorityQueue` вы можете оказаться именно в такой ситуации: вы вызываете `Dequeue()`, но очередь пуста и пакетов нет. Что вы должны вернуть, что бы могло означать "ничего"? Поскольку `Package` — класс, следует вернуть значение `null`. Это сообщит вызывающей функции, что ничего вернуть не удалось (вызывающая функция, само собой, должна проверять, не вернулось ли значение `null`).



Компилятор не может придать смысл ключевому слову `null` в исходном тексте обобщенного класса, поскольку обобщенный класс может быть инстанцирован для любых типов данных. Вот почему в исходном тексте метода `Dequeue()` используется следующая конструкция:

```
return default(T); // Значение null для типа T
```

Эта строка указывает компилятору, что нужно посмотреть, что собой представляет тип `T` и вернуть верное значение `null` для этого типа. В случае `Package`, который в качестве класса представляет собой ссылочный тип, верным возвращаемым значением будет `null`. Однако для некоторых других `T` это значение может быть иным, и компилятор сможет верно определить, что именно следует вернуть.



Если вы думаете, что обобщенный класс `PriorityQueue` достаточно гибок, то на прилагаемом компакт-диске вы можете найти еще более новую версию обобщенного класса `PriorityQueue` и познакомиться с некоторыми принципами объектно-ориентированного дизайна, обратившись к демонстрационной программе `ProgrammingToAnInterface`.

Обобщенные методы

Часто методы в обобщенных классах также должны быть обобщенными. Вы уже видели это в примере в предыдущем разделе. Метод `Dequeue()` в классе `PriorityQueue` имеет возвращаемый тип `T`. В этом разделе рассказывается, как можно использовать обобщенные методы как в обобщенных, так и в необобщенных классах.

Обобщенными могут быть даже методы в обычных необобщенных классах. Например, приведенный далее код показывает обобщенный метод `Swap()`, разработанный для обмена двух своих аргументов. Первый аргумент получает значение второго аргумента и наоборот (такой обмен проиллюстрирован на рис. 6.2 в главе 6, "Объединение данных — классы и массивы"). Чтобы увидеть, что он работает, объявите два параметра `Swap()` с применением ключевого слова `ref`, чтобы аргументы типов-значений также можно было передавать по ссылке, и посмотрите на результат работы метода (об использовании ключевого слова `ref` рассказывается в главе 7, "Функции функций").



Перед вами исходный текст демонстрационной программы, в которой объявляется и используется метод `Swap()`.

```
// GenericMethod - метод, который может работать с данными
// разных типов
using System;
namespace GenericMethod
{
    class Program
    {
        // Main - проверка двух версий обобщенного метода; один
        // работает с классом на том же уровне, что и функция
        // Main(); второй находится в обобщенном классе
        static void Main(string[] args)
        {
```

```

// Проверка обобщенного метода из необобщенного класса
Console.WriteLine("Обобщенный метод из " +
    "необобщенного класса:\n");
Console.WriteLine("\tПроверка для аргументов типа int");
int nOne = 1;
int nTwo = 2;
Console.WriteLine("\t\tПеред: nOne = {0}, nTwo = {1}",
    nOne, nTwo);
// Инстанцирование для int
Swap<int>(ref nOne, ref nTwo);
Console.WriteLine("\t\tПосле: nOne = {0}, nTwo = {1}",
    nOne, nTwo);
Console.WriteLine("\tПроверка для аргументов " +
    "типа string");
string sOne = "one";
string sTwo = "two";
Console.WriteLine("\t\tПеред: sOne = {0}, sTwo = {1}",
    sOne, sTwo);
// Инстанцирование для string
Swap<string>(ref sOne, ref sTwo);
Console.WriteLine("\t\tПосле: sOne = {0}, sTwo = {1}",
    sOne, sTwo);
Console.WriteLine("\tОбобщенный метод в " +
    "обобщенном классе");
Console.WriteLine("^Проверка для аргументов типа int");
Console.WriteLine("\t GenericClass.Swap с " +
    "аргументами типа int");

nOne = 1;
nTwo = 2;
GenericClass<int> intClass = new GenericClass<int>();
Console.WriteLine("\t\tПеред: nOne = {0}, nTwo = {1}",
    nOne, nTwo);
intClass.Swap(ref nOne, ref nTwo);
Console.WriteLine("\t\tПосле: nOne = {0}, nTwo = {1}",
    nOne, nTwo);
Console.WriteLine("\tПроверка для аргументов " +
    "типа string");
Console.WriteLine("\t GenericClass.Swap с " +
    "аргументами типа string");

sOne = "one";
sTwo = "two";
GenericClass<string> strClass =
    new GenericClass<string>();
Console.WriteLine("До: sOne = {0}, sTwo = {1}",
    sOne, sTwo);
strClass.Swap(ref sOne, ref sTwo);
Console.WriteLine("После: sOne = {0}, sTwo = {1}",
    sOne, sTwo);
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();

```

```

    } // end Main
    //static Swap - обобщенный метод в необобщенном классе
    public static void Swap<T>(ref T leftside,
                               ref T rightSide)

    {
        T temp;
        temp = leftside;
        leftside = rightSide;
        rightSide = temp;
    }
} // end Program
//GenericClass - обобщенный класс с собственным методом
// Swap
class GenericClass<T>
{
    //Swap '- метод обобщенный, поскольку принимает параметры
    // типа T; обратите внимание, что мы не можем
    // использовать Swap<T>, иначе получим предупреждение
    // компилятора о дублировании параметра класса
    public void Swap(ref T leftside, ref T rightSide)
    {
        T temp;
        temp = leftside;
        leftside = rightSide,-
        rightSide = temp;
    }
}

```

Обобщенные методы в необобщенных классах

Первая версия Swap () в предыдущем примере (всего в нем рассмотрены две версии) — статическая функция класса Program, объявленная следующим образом:

```
public static void Swap<T>(ref T leftside, ref T rightSide)
```

Объявление обобщенного метода схоже с объявлением обобщенного класса— за именем метода следуют обобщенные параметры наподобие <T>. После этого можно использовать T в методе в качестве произвольного типа, включая параметры метода и возвращаемый тип.

В примере функция Main () дважды вызывает статическую функцию Swap (), сперва инстанцируя ее для int, а затем для string (в листинге эти места выделены полужирным шрифтом). Вот как выглядят вызовы этих методов (все верно, инстанцирование выполняется при вызове метода):

```

Swap<int>(ref  nOne,
          ref nTwo);      // Инстанцирование для int

Swap<string>(ref  sOne,
            ref sTwo);    // Инстанцирование для string

```

При инстанцировании `Swap ()` для `int` вы можете использовать `int` в качестве типа аргументов в вызове. Аналогично, при инстанцировании `Swap ()` для `string` можно применить аргументы типа `string`.

Обобщенный метод `Swap ()` в обобщенном классе несколько отличается от описанного метода. Эти отличия рассматриваются в следующем подразделе.

Обобщенные методы в обобщенных классах

В предыдущем примере имеется обобщенный класс `GenericClass`, в котором содержится обобщенный метод `Swap ()`, объявленный следующим образом (здесь показан и заголовок класса, чтобы было понятно, откуда берется тип `T`):

```
class GenericClass<T> // Параметр <T> используется ниже в
                    // функции Swap()
```

```
public void Swap(ref T leftSide, ref T rightSide) ...
```

Основное отличие между методом `Swap ()` и его статическим аналогом в классе `Program` заключается в том, что обобщенную параметризацию предоставляет класс `GenericClass`, так что метод `Swap ()` в ней не нуждается (и более того, не может ее использовать). В этой версии `Swap ()` вы не найдете выражения `<T>`, а сам `T` применяется только в самом методе `Swap ()`.

Кроме этого отличия, версии `Swap ()` практически идентичны. Вот несколько вызовов метода `Swap ()` в функции `Main ()`:

```
GenericClass<int> intClass =           // Создание объекта
    new GenericClass<int>();           // для int
```

```
intClass.Swap (ref nOne, ref nTwo); // Вызов его SwapO
```

```
GenericClass<string> strClass =       // Создание объекта
    new GenericClass<string>();       // для string
```

```
StrClass.Swap (ref sOne, ref sTwo); // Вызов его SwapO
```

В данном случае для типа `int` или `string` инстанцируется сам класс. Тип `T` может использоваться в методе `Swap ()` точно так же, как и в его версии в необобщенном классе.

Ограничения для обобщенного метода

Вам могут понадобиться ограничения для обобщенного метода, с тем чтобы он мог принимать только определенные виды типов, отвечающих некоторым требованиям — как-то было сделано для класса `PriorityQueue` ранее в настоящей главе. В этом случае вы должны объявить метод примерно таким образом:

```
static void Sort<T>(T[] tArray) where T: IComparable<T>
{ ... }
```

Например, если методу необходимо сравнение параметров типа `T`, то лучше потребовать от `T` реализации интерфейса `IComparable` и указать это ограничение в описании обобщенного метода.

Обобщенные интерфейсы

Вы уже встречались с обобщенными классами и методами, и вполне логично ~~здесь~~ вопросом — могут ли быть обобщенные *интерфейсы*! (Необобщенные интерфейсы рассматривались в главе 14, "Интерфейсы и структуры".)

В этом разделе вы познакомитесь с примером, объединяющим обобщенные ~~классы~~ методы и интерфейсы.

Обобщенные и необобщенные интерфейсы

Давайте сравним обобщенные и необобщенные интерфейсы.

```
// Необобщенный                // Обобщенный
interface IDisplayable          interface ICertifiable<T>
{
    void Display();              {
                                void Certify(T criteria);
                                }
}
```

Вот как выглядит шаблон использования интерфейса. Сначала его надо объявить, и показано в приведенном фрагменте. Затем он должен быть реализован в вашем *исходе* тексте некоторым классом следующим образом:

```
// Необобщенный
class MyClass : IDisplayable ...

// Обобщенный
class MyClass : ICertifiable<MyCriteria> ...

После этого следует завершить реализацию интерфейса в вашем классе:

// Необобщенный
class MyClass : IDisplayable
{
    public void Display()
    {
        ...
    }
}

// Обобщенный
class MyClass : ICertifiable<MyCriteria> // Инстанцирование
{
    public void Certify(MyCriteria criteria)
    {
        ...
    }
}
```



Обратите внимание, что когда вы реализуете обобщенный интерфейс в классе, вы инстанцируете его обобщенную часть с использованием имени реального] типа, такого как MyCriteria.

Теперь вы можете видеть, почему интерфейс является обобщенным: он требует ~~зме~~ны <T>, используемого в качестве типа параметра или возвращаемого типа в одном или нескольких методах интерфейса. Другими словами, как и в случае обобщенного *класса*, вы указываете замещаемый тип для применения в обобщенных методах. Возможно, эти методы нужны для работы с различными типами данных, как и в случае класса коллекции List<T> или метода Swap<T> (T item1, T item2).

Часть V. За базовыми классами

Обобщенные интерфейсы — новинка, и я еще не рассматривал здесь все способы их использования. Основное их применение — в обобщенных коллекциях. Использование обобщенной версии распространенного интерфейса C#, такого как `ICollection<T>`, поможет избежать упаковки/распаковки типов-значений. Обобщенные интерфейсы могут также помочь реализовать такие вещи, как функции сортировки для применения коллекциями. (О том, что такое упаковка/распаковка, было рассказано в главе 14, "Интерфейсы и структуры".)

Приведенный далее пример достаточно абстрактный, поэтому он будет создаваться постепенно.

Использование (необобщенной) фабрики классов



Ранее в главе уже использовалась фабрика классов — хотя само название "фабрика" вводится только сейчас — для генерации бесконечного потока объектов `Package` со случайными приоритетами. Вот как выглядит этот класс:

```
// PackageFactory является частью демонстрационной программы
// PriorityQueue, находящейся на прилагаемом компакт-диске
// PackageFactory - нам нужен класс, который знает, как
// создаются новые пакеты нужного нам типа по требованию;
// такой класс называется фабрикой
class PackageFactory

    Random rand = new Random(); // Генератор случайных чисел
    //CreatePackage - этот метод фабрики выбирает случайный
    // приоритет, а затем создает пакет с этим приоритетом
    public Package CreatePackage()
    {
        // Возвращает случайно выбранный приоритет пакета. Нам
        // нужны значения 0, 1 или 2 (меньшие 3)
        int nRand = rand.Next(3);
        // Используется для генерации нового пакета
        // Приведение к перечислению несколько громоздко, но
        // зато перечисления удобны при использовании
        // конструкции switch
        return new Package((Priority)nRand);
    }
}
```



Класс `PackageFactory` имеет один член-данные и один метод (фабрику легко реализовать и не как класс, а как метод — например, метод класса `Program`). Когда вы инстанцируете объект `PackageFactory`, он создает объект класса `Random` и сохраняет его в члене `rand`. `Random` — библиотечный класс C#, предназначенный для генерации случайных чисел. (Взгляните также на демонстрационную программу `PackageFactoryWithIterator` на прилагаемом компакт-диске.)

Использование `PackageFactory`

Для генерации объектов `Package` со случайными приоритетами вызывается метод объекта фабрики `CreatePackage()`, как показано в следующем фрагменте исходного текста:


```
PackageFactory fact = new PackageFactory();
IPrioritizable pack = fact.CreatePackage(); // Обратите
// внимание на интерфейс
```

CreatePackage() использует генератор случайных чисел для генерации случайного числа от 0 до 2 включительно, и применяет сгенерированное число в качестве приоритета нового объекта Package, возвращаемого этим методом (и сохраняемого в переменной типа Package, а еще лучше — IPrioritizable).

Еще немного о фабриках



Фабрики очень удобны для генерации большого количества тестовых данных! (фабрика не обязательно использует генератор случайных чисел — он потребовался для конкретной демонстрационной программы PriorityQueue).

Фабрики усовершенствуют программу, изолируя создание объектов. Каждый раз при упоминании имени определенного класса в вашем исходном тексте! создаете *зависимость* (dependency) от этого класса. Чем больше таких *звиз* мостей, тем больше степень связности классов, тем "теснее" они *связаны* друг с другом. Программистам давно известно, что следует избегать тесного *свнл* вания. (Один из многих методов *развязки* (decoupling) заключается в *примени* нии фабрик посредством интерфейсов, как, например, IPrioritizable! а не с использованием конкретных классов наподобие Package.) Программисты постоянно создают объекты непосредственно, с применением операторов new, и это нормальная практика. Однако использование фабрик может сделать код менее тесно связанным, а следовательно, более гибким.

Построение обобщенной фабрики



А если бы у вас был класс, который мог бы создавать любые необходимые вам объекты? Эту интересную концепцию достаточно легко *спрограммировать*, как видно из демонстрационной программы GenericInterfacea прилагаемом компакт-диске.

```
// GenericInterface - использование обобщенного интерфейса
// для реализации обобщенной фабрики
using System;
using System.Collections.Generic;
namespace GenericInterface
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Создание фабрики для " +
                              "производства Blob без параметров");
            GenericFactory<Blob> blobFact =
                new GenericFactory<Blob>();
            Console.WriteLine("Создание фабрики для " +
                              "производства Students, " +
                              "параметризованных строкой");
            GenericFactory1<Student, string> stuFact =
                new GenericFactory1<Student, string>();
        }
    }
}
```

```

// См. дополнительные классы на прилагаемом
// компакт-диске
// Готовим место для хранения объектов
List<Blob> bList = new List<Blob>();
Student[] students = new Student[10];
Console.WriteLine("Создание и сохранение объектов:");
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("^Создание Blob - " +
        "вызов конструктора без " +
        "параметров.");
    Blob b = blobFact.Create();
    b.name = "blob" + i.ToString();
    bList.Add(b);
    Console.WriteLine("^Создание Student с " +
        "установкой имени - " +
        "вызов конструктора с одним " +
        "параметром.");
    string sName = "student" + i.ToString();
    students[i] = stuFact.Create(sName);
    // ... см. полный текст на прилагаемом компакт-диске
}
// Вывод результатов.
foreach(Blob b in bList)
{
    Console.WriteLine(b.ToString());
}
foreach(Student s in students)
{ Console.WriteLine(s.ToString())

// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для '
    "завершения программы.

Console.Read();
}

// Классы данных: Student, Blob (см. также компакт-диск)
// Blob - простой класс с конструктором по умолчанию (без
// параметров, предоставляемым C#). Для экономии места код
// опущен (см. компакт-диск)
// Student - класс с конструктором по умолчанию и
// конструктором с одним параметром
class Student : ISettable<string> // Обобщенный интерфейс
{
    // Часть кода опущена, см. компакт-диск
    public Student() {} //Вы должны обеспечить наличие
        // этого конструктора
    public Student(string name) // конструктор с одним
    { // параметром
        this.name = name;
    }
    // Реализация ISettable
    public void SetParameter(string name)

```

```

    {
        this.name = name;
    }
    // ToString() - на компакт-диске
}
// См. также исходный текст на компакт-диске

// Интерфейсы ISettable, используемые фабриками
interface ISettable<U>
{
    void SetParameter(U u);
}
interface ISettable2<U, V>
{
    void SetParameter(U u, V v);
}
// Фабрика для объектов с конструктором без параметров не
// требует реализации ISettable
class GenericFactory<T> where T : new()
{
    public T Create()
    {
        return new T();
    }
}
// Фабрика для создания объектов с конструктором с одним
// параметром
class GenericFactory1<T, U> where T : ISettable<U>, new()
{
    // Создает новый объект типа T с параметром U и
    // возвращает T
    public T Create(U u)
    {
        T t = new T();
        t.SetParameter(u); // T должен реализовать ISettable,
        // так что у него есть метод SetParameter()
        return t;
    }
}
// На прилагаемом компакт-диске есть код фабрики для
// создания объектов, конструктор которых требует два
// параметра
}

Демонстрационная программа GenericInterface на самом деле создает два
обобщенных класса.

```

- ✓ Фабрику для создания объектов, которые имеют только конструктор по умолчанию, без параметров.
- ✓ Фабрику для создания объектов, имеющих конструктор с одним параметром. Данный подход легко расширить для работы с объектами, конструкторы которых содержат

произвольное число параметров. На прилагаемом компакт-диске имеется исходный текст фабрики для создания объектов, имеющих конструктор с двумя параметрами.

Обобщенное создание объектов

Создание объектов, конструктор которых не имеет параметров, очень простое, так как при этом отсутствуют аргументы, о которых вам пришлось бы беспокоиться (за исключением `<T>` — параметра типа):

```
// Метод Create() класса GenericFactory<T>
public T Create ()
{
    return new T(); // Вызов конструктора без параметров
}
```

Обобщенный интерфейс вступает в игру из-за пределов обобщенных ограничений, рассматривавшихся ранее в этой части. Вот несколько способов наложения ограничений на параметр типа в заголовках обобщенных классов:

```
// T должен быть классом MyBase или его подклассом
class MyClass<T>: where T: MyBase
// T должен реализовывать IMylInterface
class MyClass<T>: where T: IMylInterface
// T может быть только ссылочным типом
class MyClass<T>: where T: class
// T может быть только типом-значением
class MyClass<T>: where T: struct
// T должен иметь конструктор без параметров
class MyClass<T>: where T: new()
```

Именно это последнее ограничение и устанавливает пределы вашим возможностям по написанию мощных обобщенных фабрик. Оно требует, чтобы тип `T` имел конструктор по умолчанию, т.е. конструктор, у которого нет параметров. Тип `T` может иметь и другие конструкторы, но один из них обязательно должен быть конструктором по умолчанию — напишите ли вы его сами или он будет сгенерирован `C#`.



Ограничение `new ()` представляет собой требование для каждого обобщенного класса или метода, которые хотят *создавать объекты* типа `T`. Однако у этого ограничения нет версий наподобие `new (U)` или `new (U, V)` для конструкторов с параметрами.

Обобщенный интерфейс также может иметь ограничения:

```
interface ISettable<T> : where T: new() ...
```

Поиск обобщенного решения

Вопрос в том, как поступить в случае, когда конструктору надо передавать параметры. Фабрика объектов, конструктор которых требует одного параметра, должна использовать в методе `Create ()` код наподобие приведенного:

```
public T Create(U u) // u — аргумент, который мы хотим
                    // передать конструктору
{
    T t = new T(u); // Но new T() не может принимать аргументы
    ...           // И что теперь?...
```

В результате следующий подход оказывается неработоспособным:

```
T t = new T(u); // Не работает
           // или
T t = new T(U); // Не работает
```

Итак, как же поступить, чтобы передать аргумент и новому объекту?

Обход конструктора по умолчанию

Только что описанная проблема решается путем использования обобщенного интерфейса. Чтобы позволить фабрике передавать параметры новым объектам, необходимо наложить определенные ограничения на производимые объекты: они должны реализовывать интерфейс наподобие `ISettable<T>`:

```
interface ISettable<U>
{
    void SetParameter(U u);
}
```

Тогда вы можете объявить фабрику для объектов с одним параметром следующим образом:

```
// T — создаваемый тип; U — тип параметра конструктора
class GenericFactory<T, U> where T: ISettable<U>, new() '{ }
```

Любой тип `T`, производимый такой фабрикой, должен реализовывать интерфейс `ISettable<U>` с `U` замененным реальным типом, таким как `string`.

Использование найденного решения

Если вы хотите производить объекты `Student` посредством такой фабрики, а конструктор `Student` требует один параметр, скажем, `string`, для передачи имени студента, то класс `Student` должен реализовать интерфейс `ISettable<string>`:

```
// Инстанцирование интерфейса для типа string
class Student: ISettable<string>
{
    private string name;
    public Student() {} // Требуется наличие
                        // конструктора по умолчанию
    public Student(string name) // Конструктор с одним
    {                             // параметром
        SetParameter(name);
    }
    public void SetParameter(string name) // Реализация
    {                                     // ISettable<string>
        this.name = name;
    }
    //.. Прочие методы и члены-данные класса Student
}
```



При этом вы можете создавать класс `Student` и так, как это делали ранее, с помощью оператора `new`:

```
students[0] = new Student("Juan Valdez");
```



Однако для использования фабрики вам нужен объект `GenericFactory1<T, U>`:

```
// Инстанцируем с T = Student, U = string
GenericFactory1<Student, string> fact1 =
    new GenericFactory1<Student, string>();
```

Вам также необходим метод `Create ()` этой фабрики для получения объектов с установленным именем:

```
// Использование строкового аргумента
students [1] = fact1.Create ("Richard Corey");
```

Вот что происходит внутри метода `GenericFactory1<T, U>.Create ()`:

```
public T Create (U u)
{
    T t = new T(); // Как и ранее, параметры конструктору
                  // не передаются
    t.SetParameter(u); // Используется метод, предоставленный
                      // при реализации ISettable
    return t;
}
```

Поскольку `Create ()` может создавать только объекты без параметров, вы используете метод `SetParameter ()` для передачи параметра и объекту `t`. После этого можно вернуть объект `Student` с установленным членом имени — такой же, как если бы для его создания был вызван конструктор с одним параметром. Вы знаете, что тип `T` имеет метод `SetParameter ()` из-за ограничений, накладываемых на класс `Student`: `ISettable<string>`. Этот интерфейс гарантирует, что класс `Student` имеет метод `SetParameter ()`.

Обсуждение решения

Насколько хорошее решение получено? Да, оно не является лучшим из тех, которые вы видели в своей практике. По сути, это обходной путь, но он приводит нас туда, куда нужно!

Но что делать, если конструктор объекта требует двух параметров? трех или четырех? Увы, `ISettable<U>` годится только для конструктора с одним параметром. В случае двух параметров вы должны добавить интерфейс `ISettable2<U, V>`. Для трех — интерфейс `ISettable3<U, V, W>` и т.д. Кроме того, для каждого из этих типов потребуется своя фабрика. Хорошая новость только в том, что крайне редко встречаются конструкторы более чем с пятью-шестью параметрами. Это и определяет, сколько интерфейсов `ISettable<U, V, ...>` и фабрик вам понадобится.



Конечно, при желании класс может реализовывать как интерфейс `ISettable<U>`, так и `ISettable2<U, V>`. В этом случае вам потребуется реализовать как метод `SetParameter (U u)`, так и метод `SetParameter (U u, V v)`. (Это — перегрузка, поскольку два метода `SetParameter ()` имеют разные сигнатуры.)

Часть VI

Великолепные десятки



В этой части...

Какая книга из серии *Для чайников* была бы полна без этой части? C# отлично умеет искать ошибки в ваших программах — вы, наверное, и сами это заметили. Однако сообщения об ошибках часто напоминают военные шифровки — это тоже наверняка бросилось вам в глаза. В главе 16, "Десять наиболее распространенных ошибок компиляции", будут рассмотрены десять наиболее часто встречающихся ошибок в программах C#, а также будет рассказано, что они означают и как с ними бороться.

Многие читатели интересуются местом C# в семье объектно-ориентированных языков программирования и его связью с наиболее распространенным объектно-ориентированным языком — C++. В главе 17, "Десять основных отличий C# и C++", вкратце описаны отличия этих двух языков, включая различия между обобщенными классами C# и шаблонами C++.

Глава 16

Десять наиболее распространенных ошибок компиляции

В этой главе...

- > The name 'memberName' does not exist in the class or namespace 'className'
- > Cannot implicitly convert type 'x' into 'y'
- > 'className.memberName' is inaccessible due to its protection level
- > Use of unassigned local variable 'n'
- } Unable to copy the file 'programName.exe' to 'programName.exe.' The process cannot...
- > 'subclassName.methodName' hides inherited member 'baseclassName.methodName'. Use the new keyword if hiding was intended.
- > 'subclassName' : cannot inherit from sealed class 'baseclassName'
- > 'className' does not implement interface member 'methodName'
- > 'methodName' : not all code paths return a value
- > j expected

С

очень строго подходит к программам и буквально с лупой выискивает в них мельчайшие ошибки. В этой главе будут рассмотрены 10 наиболее распространенных сообщений об ошибках. Но перед тем как приступить к этому, следует сделать несколько замечаний. С# достаточно многословен, и при работе над книгой мне попадались ошибки, сообщения о которых не помещались на одной странице, так что я обрезаю некоторые сообщения до одной-двух первых строк. Кроме того, в сообщениях об ошибке С# часто вставляет имена переменных, методов или классов, с которыми эти ошибки связаны. Вместо конкретных имен здесь я использую такие имена, как `variableName`, `memberName` или `className`. Наконец, С# не просто выводит имя класса — он предпочитает выводить его полностью, с указанием пространства имен, что тоже никак не приводит к сокращению сообщения.

*The name 'memberName' does not exist
in the class or namespace 'className'*

Это сообщение об ошибке может свидетельствовать о том, что вы забыли объявить переменную, как в следующем примере:

```
for(index = 0; index < 10; index++)
{
    // . . . Какие-то действия . . .
}
```

Переменная `index` нигде не определена (см. главу 3, "Объявление переменных-значений", о том, как правильно объявлять переменные). Приведенный исходный теш должен быть переписан следующим образом:

```
for(int index = 0; index < 10; index++)
{
    И . . . Какие-то действия . . .
}
```

То же применимо и к членам класса (см. главу 6, "Объединение данных — классы и массивы").

Не менее вероятно неверное написание имени переменной. Приведенный ~~далее~~ фрагмент исходного текста — хорошая иллюстрация такой ошибки,

```
class Student
{
    public string sStudentName;
    public int nID;
}
class MyClass
{
    static public void MyFunction(Student s)
    {
        Console.WriteLine("Имя = " + s.sStudentName);
        Console.WriteLine("Id = " + s.nld);
    }
}
```

Здесь проблема заключается в том, что `MyFunctionO` обращается к члену `nld`, в то время как настоящее имя члена — `nID`. Хотя это очень похожие имена, C# не считает их одинаковыми. Программист написал `nld`, но никакого `nld` не существует, и C# честно пытается об этом сообщить. (В данном случае сообщение об ошибке немного отличается: `'class.memberName' does not contain a definition for 'variableName'`. Более подробно об этом вопросе рассказывается в главе 3, "Объявление переменных-значений".)

Менее распространена, но все же попадает в десятку самых распространенных, ошибка, связанная с объявлением переменной в другой области видимости,

```
class MyClass
{
    static public void AverageInput()
    {
        int nSum = 0;
        int nCount = 0;
        while(true)
        {
            // Считываем число
            string s = Console.ReadLine();
            int n = Int32.Parse(s);
            // Выход, если введено отрицательное число
        }
    }
}
```

```

        if (n < 0)
        {
            break;
        }
        // Накопление вводимых чисел
        nSum += n;
        nCount++;
    }
    // Вывод результата
    Console.WriteLine("Сумма равна" + nSum);
    Console.WriteLine("Среднее равно " + nSum / nCount);
    // Здесь генерируется сообщение об ошибке
    Console.WriteLine("Завершающее значение — " + s);

```

Последняя строка этой функции некорректна. Дело в том, что переменная `s` ограничена областью видимости, в которой определена, и вне цикла `while` она не видна (см. главу 5, "Управление потоком выполнения").

Cannot implicitly convert type 'x' into 'y'

Эта ошибка обычно указывает на попытку использования двух переменных различного типа в одном выражении, например:

```

int nAge = 10;
// Генерация сообщения об ошибке
int nFactoredAge = 2.0 * nAge;

```

Проблема заключается в том, что `2.0` — переменная типа `double`. Целое значение `nAge` умножается на число с плавающей точкой `2.0`, что в результате дает значение типа `double`. C# не в состоянии автоматически сохранить значение типа `double` в переменной `nFactoredAge` типа `int`, потому что при этом может оказаться потерянной информации — скорее всего, дробная часть числа с плавающей точкой.

Некоторые преобразования не настолько очевидны, как в следующем примере:

```

class MyClass
{
    static public float FloatTimes2(float f)
    {
        // Генерируется сообщение об ошибке
        float fResult = 2.0 * f;
        return fResult;
    }
}

```

Вы можете решить, что здесь все в порядке, так как везде используется тип `float`. Я делю в том, что `2.0` имеет тип не `float`, а `double`, `double`, умноженный на `float`, не `double`. C# не может сохранить значение типа `double` в переменной типа `float` из-за возможной потери информации, в данном случае — количества знаков результата, но приводит к снижению точности (см. главу 3, "Объявление переменных-значений").

Неявное преобразование легко запутывает неопытного читателя. В приведенном фрагменте исходного текста функция `FloatTimes2()` работает вполне корректно:

```
class MyClass
{
    static public float FloatTimes2(float f)
    {
        // Все отлично работает
        float fResult = 2 * f;
        return fResult;
    }
}
```

Константа 2 имеет тип `int`. `int`, умноженный на `float`, дает `float`, который вполне может быть сохранен в переменной `fResult` типа `float`.

Такое же сообщение об ошибке может возникать и при выполнении операций не "натуральными" типами. Например, нельзя сложить две переменные типа `char`, не C# может при необходимости конвертировать переменную `char` в значение `int`. приводит к следующему:

```
class MyClass
{
    static public void SomeFunction()
    {
        char c1 = 'a';
        char c2 = 'b';
        // Я не знаю, что это должно означать, но все равно это
        // неверно — хотя и не по той причине, о которой вы
        // думаете
        char c3 = c1 + c2;
    }
}
```

Сложение двух символов не имеет смысла, но C# все равно попытается это сделать. Поскольку сложение для типа `char` не определено, он преобразует `c1` и `c2` в значение типа `int` и выполнит их сложение (технически `char` представляет собой интегральный тип). К сожалению, результат этого сложения преобразовать обратно в `char` не удастся без помощи со стороны программиста (см. главу 3, "Объявление переменных значений").

Большинство (хотя и не все) преобразований без проблем выполняется при их **явном** указании. Так, следующая функция компилируется без каких-либо нареканий:

```
class MyClass
{
    static public float FloatTimes2(float f)
    {
        // Здесь использовано явное преобразование
        float fResult = (float)(2.0 * f);
        return fResult;
    }
}
```

Результат умножения `2.0*f` имеет тип `double`, как и ранее, однако программист явно указал, что он хочет преобразовать полученное значение к типу `float`, даже **если** это приведет к потере информации (см. главу 3, "Объявление переменных-значений").

Второй подход к проблеме может состоять в том, чтобы явно указать необходимые тип константы:

```

class MyClass
{
    static public float FloatTimes2(float f)
    {
        // Здесь 2.OF — константа типа float
        float fResult = 2.OF * f;
        return fResult;
    }
}

```

В этой версии функции использована константа 2.0 типа float, а не double, как принято по умолчанию, float, умноженный на float, дает float.

'className.memberName' is inaccessible due to its protection level

Данная ошибка указывает на попытку функции обратиться к члену, на обращение к которому она не имеет прав. Например, метод в одном классе может пытаться обратиться к закрытому члену другого класса (см. главу 11, "Классы"), как показано в приведенном фрагменте исходного текста,

```

public class MyClass
{
    public void SomeFunction()
    {
        YourClass uc = new YourClass();
        // MyClass не имеет доступа к закрытому члену
        uc.nPrivateMember = 1;
    }
}

public class YourClass
{
    private int nPrivateMember = 0;
}

```

Обычно такая ошибка не столь очевидна. Зачастую оказывается просто забытым дескриптор члена или всего класса, а по умолчанию член класса является закрытым. Так, nPrivateMember остается закрытым в следующем фрагменте исходного текста:

```

class MyClass // Доступ к классу по умолчанию - internal
(
    public void SomeFunction()
    {
        YourClass uc = new YourClass();
        // MyClass не имеет доступа к закрытому члену
        uc.nPrivateMember = 1;
    }
}

public class YourClass
(
    int nPrivateMember = 0; // Этот член - закрытый!
)

```

Кроме того, несмотря на то что функция `SomeFunction()` объявлена как `public` к ней нельзя обратиться из классов других модулей, поскольку класс `MyClass` сам себе является внутренним.

Мораль этой истории — всегда указывайте уровень защиты ваших классов и их членов. Кроме того, не объявляйте открытые члены во внутренних классах — это том сбивает с толку.

Use of unassigned local variable 'n'

Данное сообщение указывает на то, что вы объявили переменную, но не присвоили ей никакого начального значения. Обычно это простой недосмотр, но такая ситуация может возникнуть, когда вы передаете переменную в функцию как `out`-аргумент, как показано в приведенном далее фрагменте исходного текста,

```
public class MyClass
{
    public void SomeFunction()
    {
        int n;
        // Все в порядке, так как C# только возвращает значение
        // в n; в функцию значение этой переменной не передается
        SomeOtherFunction(out n);
    }
    public void SomeOtherFunction(out int n)
    {
        n = 1;
    }
}
```

В данном случае переменной `n` в функции `SomeFunction()` не присваивается ни какого значения, поскольку это выполняется в функции `SomeOtherFunction()`. Функция `SomeOtherFunction()` игнорирует значение `out`-аргумента, как если бы его не существовало вовсе. (В главе 3, "Объявление переменных-значений", рассказывается о переменных, а ключевое слово `out` рассмотрено в главе 7, "Функции функций").

Unable to copy the file 'programName.exe' to 'programName.exe'. The process cannot...

Обычно это сообщение повторяется несколько раз. И почти всегда означает, что вы забыли завершить выполнение программы перед ее пересборкой. Другими словами, вы сделали следующее.

1. Успешно собрали программу.
2. Когда вы запустили программу с помощью команды `Debug^Start without Debugging`, то получили сообщение Нажмите <Enter> для завершения программы. . . , но по какой-то причине не сделали этого (таким образом, вы-

полнение программы продолжается), а вы переключились в Visual Studio 2005 и продолжили редактирование файла.

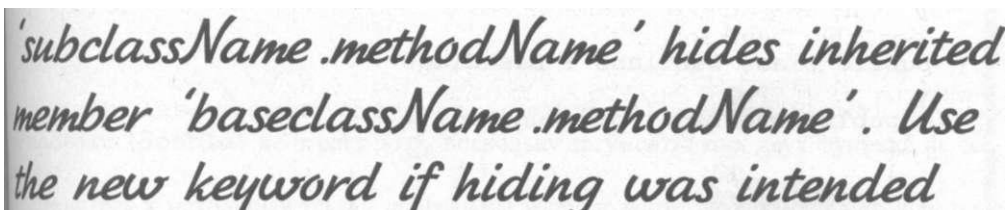
Примечание: если вы запустили программу посредством команды **Debug^Start Debugging** и забыли ее завершить, то Visual Studio 2005 запросит вас о том, не следует ли завершить выполнение программы.

3. Вы попытались собрать программу заново, с новыми обновлениями. В этот момент вы и получаете рассматриваемое сообщение об ошибке в окне **Error List**.

Выполнимый (.EXE) файл блокируется Windows до тех пор, пока программа не прекратит работу. Visual Studio не в состоянии перезаписать заблокированный старый .EXE-файл новой версией без полного завершения программы.

Чтобы исправить ситуацию, переключитесь на ваше активное приложение и завершите его работу. Если это одно из консольных приложений из данной книги, просто нажмите клавишу <Enter>. Вы также можете завершить программу из Visual Studio 2005 посредством команды меню **Debug^Stop Debugging**. После того как старое приложение прекратит работу, соберите приложение заново.

Если вы не можете избавиться от ошибки таким способом, выйдите из Visual Studio 2005 и перегрузите компьютер. Если не сработает и это — ну, тогда не знаю, чем вам можно помочь...



'subclassName.methodName' hides inherited member 'baseclassName.methodName'. Use the new keyword if hiding was intended

Посредством этого сообщения C# пытается информировать о том, что вы перегружаете метод базового класса без его перекрытия (см. детальное описание в главе 13, "Полиморфизм"). Давайте рассмотрим следующий пример:

```
public class BaseClass
{
    public void Function()
}

public class Subclass : BaseClass
{
    public void Function() // here's the overload
    {
    }
}

public class MyClass
{
    public void Test()
    {
        Subclass sb = new Subclass();
        sb.Function();
    }
}
```

Функция `Test()` не может обратиться к методу `BaseClass.Function()` из объекта подкласса `sb`, поскольку он скрыт методом `Subclass.Function()`. Вы ~~намер~~ вались сделать одно из двух перечисленных действий.

- ✓ Хотели скрыть метод базового класса. В этом случае добавьте ключевое слово `new` в определение `Subclass`, как показано в следующем фрагменте исходного текста:

```
public class Subclass : BaseClass
{
    new public void Function()
    {
    }
}
```

- ✓ Намеревались полиморфно наследовать базовый класс. В этом случае вы долж| объявить два класса следующим образом:

```
public class BaseClass
{
    public virtual void Function()
    {
    }
}

public class Subclass : BaseClass
{
    public override void Function()
    {
    }
}
```

См. детальное описание проблемы в главе 13, "Полиморфизм".



Это не ошибка, а всего лишь предупреждение.

'subclassName' : cannot inherit from sealed class 'baseclassName'

Это сообщение говорит о том, что класс, который вы хотите наследовать, опечатан, поэтому вы не можете ни наследовать его, ни изменить любое из его свойств. Обычно опечатываются только библиотечные классы. Обойти опечатывание невозможно, но вы можете попытаться воспользоваться классом с помощью отношения `СОДЕРЖИТ` (см. главу 13, "Полиморфизм").

'className' does not implement interface member 'methodName'

Реализация интерфейса представляет собой обещание предоставить определение для всех его методов. Данное сообщение свидетельствует о том, что вы нарушили это обещание и не реализовали указанный метод. Такое могло произойти по ряду причин.

- ✓ Ваша собака просто сожрала вашу работу. Конечно, это шутка — вы просто забыли о необходимости реализовать указанный метод. В следующий раз будьте внимательнее.
- ✓ Вы ошиблись при написании имени метода или дали ему неверные аргументы.

Рассмотрим следующий пример:

```
interface Me
{
    void aFunction(float f);
}

public class MyClass : Me
{
    public void aFunction(double d)
```

Класс `MyClass` не реализует функцию интерфейса `aFunction(float)`. Функция `aFunction(double)` не играет роли, поскольку аргументы этих двух функций не совпадают.

Вернитесь к исходному тексту программы и продолжите реализацию методов, пока интерфейс не будет реализован полностью (см. главу 14, "Интерфейсы и структуры").



Неполная реализация интерфейса — практически то же самое, что и попытка создать конкретный класс из абстрактного, не перекрывая при этом все его абстрактные методы.

'methodName' : not all code paths return a value

Этим сообщением C# ставит вас в известность, что ваш метод, который объявлен как возвращающий значение, на одном или нескольких путях выполнения не возвращает ни-что. Это может случиться по двум причинам.

- ✓ У вас имеется конструкция `if`, которая содержит выход без возврата значения.
- ✓ Более вероятно, что вы вычислили значение, но не возвращаете его.

Обе возможности иллюстрируются следующим исходным текстом:

```
public class MyClass
```

```

public string ConvertToString(int n)
{
    // Конвертируем int n в string s
    string s = n.ToString();
}

public string ConvertPositiveNumbers(int n)
{
    // Преобразуем только положительные числа
    if (n > 0)
    {
        string s = n.ToString();
        return s;
        Console.WriteLine("Аргумент {0} некорректен", n);
        // Требуется еще один оператор return
    }
}

```

Функция `ConvertToString()` вычисляет значение типа `string`, но не возвращает его. Все, что нужно для исправления этой ошибки — добавить в конце функции `return s;`.

Функция `ConvertPositiveNumbers()` возвращает `string`-версию аргумента типа `int`, если `n` положительно. Кроме того, функция совершенно корректно генерирует сообщение об ошибке, если `n` не положительно. Однако в этом случае функция ничего не возвращает. Здесь вы должны вернуть либо `null`, либо пустую строку `""` — что болы подходит для нужд вашего приложения (см. главу 7, "Функции функций").

} expected

Эта ошибка указывает, что `C#` ожидал закрывающую скобку в месте завершения исходного текста программы. Где-то по дороге вы забыли закрыть определение класса, функцию или блок. Вернитесь и внимательно просмотрите исходный текст программы, пока не найдете это место.

Такое сообщение об ошибке зачастую оказывается последним в серии практически бессмысленных сообщений. Не беспокойтесь о них до тех пор, пока не найдете место, где была потеряна закрывающая фигурная скобка. В поиске пар открывающихся и соответствующих закрывающихся скобок вам может помочь `Visual Studio 2005`.

Десять основных отличий C# и C++

В этой главе...

- > Отсутствие глобальных данных и функций
- > Все объекты размещаются вне кучи
- > Переменные-указатели запрещены
- > Обобщенные классы C# и шаблоны C++
-) Никаких включаемых файлов
- > Не конструирование, а инициализация
- > Корректное определение типов переменных
- > Нет множественного наследования
- > Проектирование хороших интерфейсов

Квалифицированная система типов



Язык C# в большей или меньшей степени основан на языке программирования C++. Это и не удивительно, если вспомнить, что Microsoft создала Visual C++, Iru из наиболее успешных сред программирования для Windows. Подавляющее большинство программ, с которыми приходится иметь дело, написаны на Visual C++.

Однако C# — не просто новые одежды на старом языке: в нем имеется масса улучшений, представляющих собой как новые возможности, так и замену старых возможностей C++ более мощными. В этой главе будет рассказано о десятке самых важных усовершенствований. Само собой, их гораздо больше, так что, в принципе, можно было бы написать и о двадцати (и более) главных улучшениях C# по сравнению с C++.



Вы могли прийти к C# разными путями — например, из Java или Visual Basic. C# даже более схож с Java, чем с C++ что также не удивительно, поскольку язык Java тоже появился как усовершенствование C++, ориентированное на использование в Интернете. Между C# и Java много синтаксических различий, но все равно они выглядят как близнецы. Если вы можете читать программы на одном из них — то сможете это делать и на другом.

Что касается Visual Basic — т.е. Visual Basic .NET, а не старого Visual Basic 6.0 — то и синтаксис совершенно иной, однако Visual Basic .NET основан на той же инфраструктуре .NET Framework, что и C#, так что он дает практически одинаковый с C# код CIL (Common Intermediate Language, общий промежуточный язык) и способен к взаимодействию с C#. Класс C# может наследовать класс Visual Basic и наоборот, а ваша программа может представлять комбинацию модулей C# и Visual Basic.

Отсутствие глобальных данных и функций

C++ позиционируется как объектно-ориентированный язык, и это так, в том смысле, что вы можете программировать в объектно-ориентированном стиле с использованием C++. Вы также можете отходить от объектно-ориентированного стиля программирования, размещая данные и функции в некотором глобальном пространстве, открытом для всех элементов программы и всех программистов с клавиатурой...

C# заставляет программиста быть преданным объектно-ориентированной присяге: все функции и все члены-данные должны объединяться в классы. Вы хотите обратиться к функции или данным? Вы должны делать это с согласия автора класса — и никаких исключений здесь быть не может.

Все объекты размещаются вне кучи

C/C++ позволяет выделять память несколькими способами, каждый из которых имеет свои недостатки.

- ✓ **Глобальные объекты существуют на протяжении всего времени жизни программы.** Программа может легко создать множество указателей на один и тот же глобальный объект. При изменении посредством одного из них изменения отражаются во всех. Указатель представляет собой переменную, содержащую адрес некоторого удаленного блока памяти. Технически ссылки C# являются указателями.
- ✓ **Стековые объекты уникальны для отдельных функций (и это хорошо), но они освобождаются при выходе из функции.** Любой указатель на освобожденный объект становится недействительным. Это было бы нормально, если бы никто не работал с этими указателями; однако ничто не мешает использовать указатель, даже если объекта, на который он указывает, уже нет. Стек C++ — это область памяти, отличная от кучи, и это действительно стек.
- ✓ **Объекты в куче создаются по мере необходимости.** Эти объекты уникальны в рамках одной нити выполнения программы.

Проблема заключается в том, что очень легко забыть, на какой тип памяти ссылается данный указатель. Объекты в куче следует освободить после того, как вы поработали с ними. Если забыть об этом, в программе образуется "утечка памяти", которая может привести к исчерпанию последней и неработоспособности программы. С другой стороны, если вы освободите блок в куче дважды или "освободите" блок глобальной или стековой памяти, ваша программа может вызвать проблемы, с которыми справятся только три веселые клавиши <Ctrl>, <Alt> и , и то не поодиночке, а все вместе...

C# решает эту проблему путем выделения памяти для всех объектов вне кучи. Кроме того, C# использует сборку мусора для ее возврата в кучу. При работе с C# вы можете забыть о синем экране смерти из-за пересылки неверного блока памяти в кучу.

Переменные-указатели запрещены

Введение указателей в С обеспечило успех этого языка программирования. Работа с указателями— очень мощная возможность. Старые программисты на машинных языках были несказанно рады, получив в свои руки такой инструмент. В С++ возможности работы с указателями унаследованы без изменений от С.

К сожалению, ни программист, ни программа не в состоянии отличить хороший указатель от плохого. Прочтите память по неинициализированному указателю— и ваша программа аварийно завершится, если, конечно, вам повезет... Если не повезет— она продолжит работу, сочтя случайный блок памяти корректным объектом...

Проблемы, связанные с указателями, трудно локализовать. Зачастую программа с некорректным указателем при каждом новом запуске ведет себя по-новому.

С# решает проблемы, связанные с указателями, очень просто — он попросту устраняет их из языка. Используемые вместо них ссылки безопасны с точки зрения типов и не могут быть применены так, чтобы это приводило к краху программы.

Обобщенные классы С# и шаблоны С++

Если вы сравните новые обобщенные возможности С# (см. главу 15, "Обобщенное программирование") с шаблонами С++, то обнаружите высокую степень схожести их (синтаксиса). Однако хотя оба средства имеют общее предназначение, такое сходство является чисто внешним.



Как обобщенные классы, так и шаблоны безопасны с точки зрения типов, но реализованы они совершенно по-разному. Шаблоны инстанцируются в процессе компиляции, в то время как инстанцирование обобщенных классов происходит во время выполнения программы. Это означает, что один и тот же шаблон в разных модулях дает в результате два различных типа, инстанцированных во время компиляции. Но один и тот же обобщенный класс в разных модулях дает только один тип, инстанцируемый во время выполнения. Это приводит к меньшему "раздутию" кода для обобщенных классов по сравнению с шаблонами.

Наибольшее различие между обобщенными классами и шаблонами состоит в том, что обобщенные классы работают с несколькими языками, включая Visual Basic, С++ и другие языки .NET, в том числе С#. Шаблоны же используются только (в рамках С++).

Что же лучше? Шаблоны более мощны — и более сложны, как и множество других идей в С++, но и больше подвержены ошибкам. Обобщенные классы проще в использовании и реже приводят к ошибкам в программах.

Конечно, это всего лишь некоторые тезисы бурной дискуссии. Существенно большую информацию можно найти в блоге Брендона Брея (Brandon Bray) по адресу weblogs.asp.net/branbray/archive/2003/11/19/51023.aspx.

Никаких включаемых файлов

C++ обеспечивает строгую проверку типов — и это хорошо. Он выполняет это заставляя объявлять функции и классы в так называемых *включаемых файлах*, которые затем используются модулями. Однако правильное перечисление в правильном порядке всех включаемых файлов для компиляции вашего модуля — задача не из простых.

C# избегает бессмысленной работы. Он ищет и находит определения всех классов. Если вы вызываете класс `Student`, C# находит определение этого класса, чтобы убедиться, что вы используете его корректно.

Не конструирование, а инициализация

Сначала казалось, что конструкторы приносят большую пользу. Наличие специальной функции, гарантирующей корректную настройку всех членов-данных... Отличная идея! Единственная проблема в том, что приходится добавлять в каждый написанный класс тривиальный конструктор по умолчанию. Рассмотрим следующий пример:

```
public class Account
{
    private double balance;
    private int numChecksProcessed;
    private CheckBook checkBook;
    public Account()
    {
        balance = 0.0;
        numChecksProcessed = 0;
        checkBook = new CheckBook();
    }
}
```

Почему же нельзя инициализировать члены-данные непосредственно и позволить языку программирования самому сгенерировать конструктор? C++ отвечает, почему; C# отвечает — почему нет? C# позволяет избавиться от ненужных конструкторов с помощью непосредственной инициализации:

```
public class Account
{
    private double balance = 0.0;
    private int numChecksProcessed = 0;
    private CheckBook checkBook = new CheckBook();
    // Больше это не надо делать в конструкторе
}
```

Более того, если все, что нужно — это соответствующая версия нуля для определенного типа, как в случае первых двух членов, C# примет необходимые меры автоматически, как минимум для членов-данных классов. Если вы хотите нечто, отличное от нуля, добавьте вашу собственную инициализацию к объявлению членов-данных. (Однако следует всегда инициализировать локальные переменные в функциях.)

Корректное определение типов переменных

C++ очень политкорректен. Он и шагу не ступит ни на одном компьютере без того, чтобы определить требования к диапазону значений и размеру конкретных типов. Он указывает, что `int` имеет такой-то размер, а `long` — больший. Все это приводит к появлению ошибок при переносе программ с одного типа процессора на другой.

C# не заботится о таких мелочах. Он прямо говорит — `int` имеет 32 бит, а `long` — 64 бит, и так должно быть.

Нет множественного наследования

C++ позволяет одному классу наследовать более чем один базовый класс. Например, класс `SleeperSofa` (диван-кровать) может наследовать классы `Bed` (кровать) и `Sofa` (диван). Наследование от двух классов звучит неплохо, и это и в самом деле бывает очень полезно. Проблема только в том, что множественное наследование может приводить к некоторым трудно обнаружимым ошибкам.

C# не рискует и снижает количество возможных ошибок, запрещая множественное наследование. Однако в C# имеется возможность, которая в ряде ситуаций может заменить множественное наследование, а именно — интерфейсы.

Проектирование хороших интерфейсов

Когда программисты продираются сквозь кошмар множественного наследования и 90% времени проводят в отладчике, зачастую выясняется, что второй базовый класс нужен только для того, чтобы *описать* подкласс. Например, обычный класс может наследовать абстрактный класс `Persistable` с абстрактными методами `read()` и `write()`. Это заставляет подкласс реализовать методы `read()` и `write()` и объявить всему миру, что эти методы доступны для использования.

После этого программисты осознают, что того же можно добиться существенно более легкими средствами — посредством интерфейса. Класс, который реализует интерфейс наподобие приведенного ниже, тоже обещает предоставить методы `read()` и `write()`:

```
interface IPersistable
{
    void read();
    void write();
}
```

Так вы избегаете опасностей множественного наследования C++ и получаете желаемый результат.

Унифицированная система типов

Класс C++ — очень хорошая возможность языка. Он позволяет данным и связанным с ними функциям быть объединенными в четкие пакеты, которые соответствуют челове-

ческим представлениям о реальном мире. Единственная проблема заключается в том, что любой язык программирования должен обеспечить еще и простейшие типы переменных для хранения, например, целых чисел или чисел с плавающей точкой. Это определяет необходимость системы преобразования типов. Объекты классов и переменные типов значений находятся по разные стороны баррикад, хотя и участвуют вместе в одних и тех же программах. Программист вынужден все время помнить о том, что это разные ~~вещи~~ по своей природе.

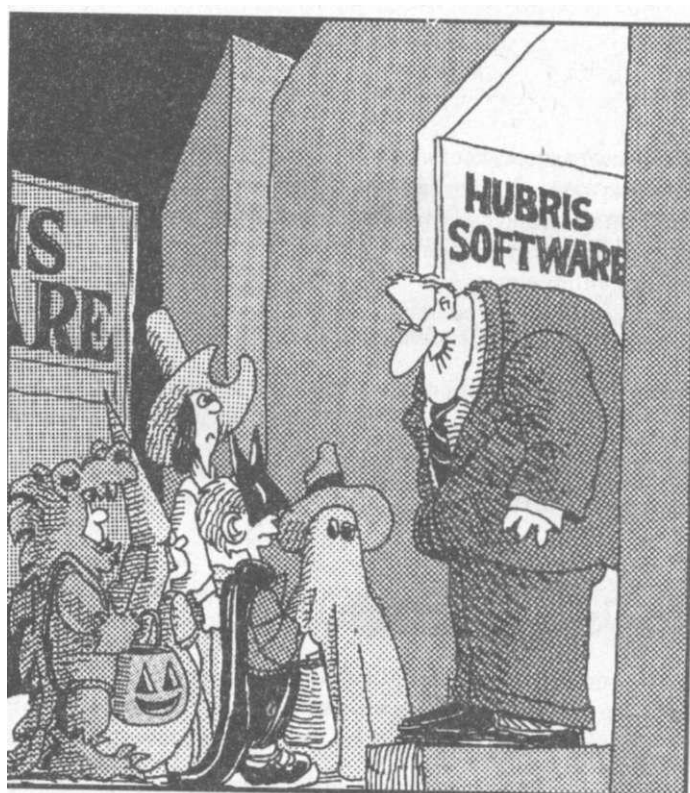
C# разрушает эту баррикаду, отделяющую типы-значения от классов. Для ~~каждом~~ типа-значения имеется соответствующий класс, именуемый структурой (вы можете так же писать и собственные структуры; см. главу 14, "Интерфейсы и структуры"). Эти структуры могут легко объединяться с классами, позволяя программисту писать ~~код~~ следующий текст наподобие следующего:

```
MyClass myObject = new MyClass O,-  
// Вывод "myObject" в строковом формате  
Console.WriteLine(myObj ect.ToString());  
int i = 5;  
// Вывод int в строковом формате  
Console.WriteLine(i.ToString());  
// Вывод константы 5 в строковом формате  
Console.WriteLine(5.ToString());
```

Можно вызвать один и тот же метод не только для переменной `int` и объекта ~~класса~~ `MyClass`, но даже для константы наподобие `5`. Такое вавилонское смешение типов-одна из мощных возможностей C#.

Часть VII

Дополнительные главы



"Ну и что, что мы не успели выпустить нашу программу "Хэллоцин" в срок? Зато к Рождеству будет готова более надежная версия этой программы!"

В этой части...

В оригинальном издании книги эти главы были помещены на прилагаемый к книге компакт-диск. В русском издании книги их решено опубликовать в виде отдельной, дополнительной части.

Глава 18

Эти исключительные исключения

В этой главе...

- > Обработка ошибок с помощью кодов ошибки
- У Использование механизма исключений вместо кодов ошибки
- } Создание собственного класса исключения
- } Перекрытие ключевых методов в классе исключения

Вне сомнения, трудно смириться с тем, что иногда метод (или функция) не делает то, для чего он предназначался. Это раздражает программистов ничуть не меньше, чем пользователей их программ, тоже часто являющихся источником недоразумений. В книге встречалась программа, в которой пользователь должен был вводить целое число как строку. Такой метод можно написать так, что он будет просто игнорировать введенный пользователем мусор вместо реального числа, но хороший программист напишет функцию таким образом, чтобы она распознавала неверный ввод пользователя и докладывала об ошибке.

Здесь говорится об ошибках времени выполнения, а не времени компиляции, с которыми C# разберется сам при сборке вашей программы.

Механизм исключений представляет собой средство для сообщения о таких ошибках способом, который вызывающая функция может лучше понять и использовать для решения возникшей проблемы.

Старый способ обработки ошибок

Промолчать о том, что произошла ошибка времени выполнения — это всегда худшее решение, применимое только в том случае, если вы не намерены отлаживать программу и вас не интересует результат ее работы...



В приведенной далее демонстрационной программе `FactorialWithError` показано, что может произойти, если не выявить ошибку. Эта программа вычисляет и выводит значение факториала для ряда значений.

Факториал числа N равен $N*(N-1)*(N-2)*... *1$. Например, факториал 4 равен $4*3*2*1 = 24$. Функция вычисления факториала работает только для положительных целых чисел. Это банальный программистский пример для иллюстрации ситуации, когда требуется обработка ошибок.

```
// FactorialWithError - пример функции вычисления
// факториала, в которой отсутствует проверка ошибок
```

```

using System;
namespace FactorialWithError
{
    // MyMathFunctions - набор созданных мною математических
    // функций
    public class MyMathFunctions
    {
        // Factorial - возвращает факториал переданного
        // аргумента
        public static double Factorial(double dValue)
        {
            // Начинаем со значения аккумулятора, равного 1
            double dFactorial = 1.0;
            // Цикл со счетчиком nValue, уменьшающимся до 1, с
            // умножением на каждой итерации значения аккумулятора
            // на величину счетчика
            do
            {
                dFactorial *= dValue;
                dValue -= 1.0;
            } while(dValue > 1);
            // Возвращаем вычисленное значение
            return dFactorial;
        }
    }
    public class Program
    {
        public static void Main(string[] args)
        {
            // Вызов функции вычисления факториала в
            // цикле от 6 до -6
            for (int i = 6; i > -6; I--)
            {
                // Вывод результата на каждой итерации
                Console.WriteLine("i = {0}, факториал = {1}",
                                i, MyMathFunctions.Factorial(i));
            }
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}

```

Функция `Factorial()` начинается с инициализации переменной-аккумулятора значением 1. Затем функция входит в цикл, в котором на каждой итерации выполняется умножение на последовательно уменьшающееся значение счетчика `nValue`, пока `nValue` не достигнет 1. Накопленное в аккумуляторе значение возвращается вызывающей функции.

Алгоритм `Factorial()` выглядит корректно — пока вы не начнете вызывать эту функцию. Функция `Main()` также содержит цикл, в котором вычисляются значения факториала для ряда убывающих значений. Однако вместо того чтобы остановиться на значе-

В результате на экране получается следующее:

```
i = 6, факториал = 72 0
i = 5, факториал = 12 0
i = 4, факториал = 24
i = 3, факториал = 6
i = 2, факториал = 2
i = 1, факториал = 1
i = 0, факториал = 0
i = -1, факториал = -1
i = -2, факториал = -2
i = -3, факториал = -3
i = -4, факториал = -4
i = -5, факториал = -5
```

Нажмите <Enter> для завершения программы...

Как видите, часть результатов не имеет смысла. Во-первых, значение факториала не может быть отрицательным. Во-вторых, обратите внимание, что отрицательные значения растут совсем не так, как положительные. Понятно, что здесь присутствует ошибка.



Если попытаться изменить цикл внутри `Factorial()` и записать его как `do{ . . . }while (dValue != 0)`, то программу при передаче отрицательного значения просто ждет крах. Поэтому никогда не пишите такой оператор сравнений — `while (dValue != 0)`, поскольку ошибка приближения может в любом случае привести к неверному результату проверки на равенство 0.

В особенности при работе с числами с плавающей точкой избегайте условий наподобие `dValue != 0`, в которых требуется точное сравнение для выхода из цикла. Используйте менее строгое условие, как, например, `dValue > 1`. Небольшая ошибка приближения — такая как `dValue = 0.00001` — может привести к бесконечному циклу. Об ошибках приближения рассказывается в главе 3, "Объявление переменных-значений".

Возврат индикатора ошибки

Несмотря на свою простоту, функция `Factorial()` требует проверки ошибочной ситуации: факториал отрицательного числа не определен. Функция `Factorial()` должна включать проверку этого условия.

Но что должна делать функция `Factorial()`, столкнувшись с ошибкой? Лучшее, что она может сделать в такой ситуации — это сообщить об ошибке вызывающей функции в надежде на то, что источник ошибки знает, почему она произошла и как с ней справиться.

Классический способ указать на происшедшую ошибку в функции — это вернуть значение, которое функция не в состоянии вернуть при безошибочной работе. Например, значение факториала не может быть отрицательным. Таким образом, факториал может возвращать значение -1, если ему передается отрицательный аргумент, -2 для нецелого аргумента и так далее — для каждой ошибки некоторое соответствующее ей число. Такие числа называются кодами ошибки. Вызывающая функция может проверить, не вернула ли вызываемая функция отрицательное значение, и если да — то вызывающая функция будет знать о том, что произошла ошибка. Значение возвращаемого кода ошибки позволяет определить ее природу.



Указанные изменения внесены в код демонстрационной программы Facto-

```
// FactorialErrorReturn - создание функции вычисления
// факториала, которая возвращает код ошибки, если что-то
// идет не так
using System;
namespace FactorialErrorReturn
{
    // MyMathFunctions - набор созданных мною математических
    // функций
    public class MyMathFunctions
    {
        // Следующие коды ошибок представляют некорректные
        // значения
        public const int NEGATIVE_NUMBER    = -1;
        public const int NON_INTEGER_VALUE = -2;
        // Factorial - возвращает факториал переданного
        // аргумента
        public static double Factorial(double dValue)
        {
            // Проверка: отрицательные значения запрещены
            if (dValue < 0)
            {
                return NEGATIVE_NUMBER;
            }
            // Проверка: передано ли целое значение аргумента
            int nValue = (int)dValue;
            if (nValue != dValue)
            {
                return NON_INTEGER_VALUE;
            }
            // Тесты пройдены, начинаем со значения аккумулятора,
            // равного 1
            double dFactorial = 1.0;
            // Цикл со счетчиком nValue, уменьшающимся до 1, с
            // умножением на каждой итерации значения аккумулятора
            // на величину счетчика
            do
            {
                dFactorial *= dValue;
                dValue -= 1.0;
            } while(dValue > 1);
            // Возвращаем вычисленное значение
            return dFactorial;
        }
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Вызов функции вычисления факториала в
        // цикле от 6 до -6
    }
}
```

```

for (int i = 6; i > -6; i--)
{
    double dFactorial = MyMathFunctions.Factorial(i);
    if (dFactorial == MyMathFunctions.NEGATIVE_NUMBER)
    {
        Console.WriteLine
            ("Factorial() получила отрицательный параметр");
        break;
    }
    if (dFactorial == MyMathFunctions.NON_INTEGER_VALUE)
    {
        Console.WriteLine
            ("Factorial() получила нецелый параметр");
        break;
    }
    // Вывод результата на каждой итерации
    Console.WriteLine("i = {0}, факториал = {1}",
        i, MyMathFunctions.Factorial(i));
}
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read ();
}
}

```



Теперь перед началом вычислений функция `Factorial()` выполняет ряд проверок. Первая проверка — не отрицателен ли переданный функции аргумент. Обратите внимание, что значение 0 разрешено, поскольку приводит к разумному результату⁷. Если проверка не пройдена, функция тут же возвращает код ошибки. Затем выполняется второй тест, проверяющий, равен ли переданный аргумент своей целочисленной версии. Если да — дробная часть аргумента равна 0.

Функция `Main()` проверяет результат, возвращаемый функцией `Factorial()`, на предмет обнаружения ошибок. Однако значения наподобие -1 и -2 мало информативны для программиста, так что класс `MyMathFunctions` определяет пару целочисленных констант. Константа `NEGATIVE_NUMBER` равна -1, а `NON_INTEGER_VALUE` — 2. Это ничего не меняет, но делает программу, в особенности функцию `Main()`, существенно более удобочитаемой.



Обычно по соглашению для имен констант используются строчные буквы, а слова в имени разделены символами подчеркивания.

Обращение к этим константам выполняется посредством имени класса, как `MyMathClass.NEGATIVE_NUMBER`. Константные переменные автоматически являются статическими, что делает их свойствами класса, разделяемыми всеми объектами. Другие варианты работы с константами описаны во врезке "Немного о константах".

⁷ Этот "разумный" результат некорректен, так как в математике принято, что факториал 0 равен 1. — *Примеч. ред.*



Немного о константах

Предпочтительный способ записи констант почти всегда использует следующий, более гибкий по сравнению с применением `const`, подход.

```
public static readonly int NEGATIVE_NUMBER = -1;
```

Значение `const` вычисляется во время компиляции и может быть инициализировано только числом или строкой. Статическая переменная только для чтения вычисляется во время выполнения программы и может быть инициализирована объектом любого вида. Используйте `const` только там, где производительность программы сверхкритична.

Еще один способ определения констант — в данном случае группы связанных констант — посредством ключевого слова `enum`, как описано в главе 15, "Обобщенное программирование". Типы ошибок для `MyMathClass` могут быть определены следующим образом:

```
enum MathErrors  
  
    NegativeNumber,  
    NonIntegerValue
```

Функция `Factorial()` может возвращать значение `MathErrors`, и вы можете проверить его в своей программе следующим образом (как можно часто увидеть в классах .NET Framework):

```
MathErrors meResult = MyMathFunctions.Factorial(6);  
if(meResult == MathErrors.NegativeNumber) ...
```

Теперь функция `Factorial()` сообщает об ошибках функции `Main()`, которая выводит соответствующее сообщение на экран и завершает на этом свою работу:

```
i = 6, факториал = 720  
i = 5, факториал = 120  
i = 4, факториал = 24  
i = 3, факториал = 6  
i = 2, факториал = 2  
i = 1, факториал = 1  
i = 0, факториал = 0  
Factorial() получила отрицательный параметр  
Нажмите <Enter> для завершения программы...
```

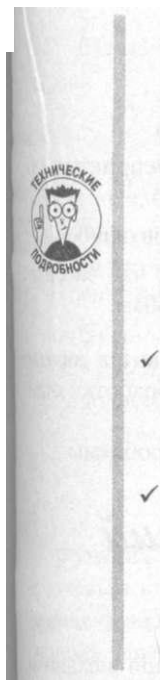
(Здесь я предпочел прекращать работу при обнаружении ошибки.) Указание о произошедшей ошибке посредством возвращаемого функцией значения повсеместно используется еще со времен FORTRAN. Зачем же менять этот механизм?

Чем плохи коды ошибок

Что же не так с кодами ошибок? Они были достаточно хороши даже для FORTRAN! Да, но в те времена компьютеры были ламповыми. Увы, но коды ошибок приводят к ряду проблем.



Этот метод основан на том факте, что у функции имеются значения, которые она не может вернуть при корректной работе. Однако существуют функции, у которых



любые возвращаемые значения корректны. Не всегда везет поработать с функцией, которая должна возвращать только положительные значения. Например, вы не можете получить логарифм отрицательного числа, но само значение логарифма может быть как положительным, так и отрицательным.

Вы можете предложить справиться с этой проблемой, возвращая код ошибки как значение функции, а необходимые данные — посредством аргумента, описанного как `out`, но такое решение менее интуитивно и теряет выразительную силу функции. Сперва познакомьтесь с исключениями, и вы убедитесь, что они предоставляют гораздо более красивый путь решения проблемы.

✓ В целом числе не удастся разместить большое количество информации. Так, рассматриваемая функция `Factorial 0` возвращает `-1`, если ее аргумент отрицателен. Локализовать ошибку было бы проще, если бы был известен сам аргумент, но в возвращаемом функцией типе для него просто нет места.

✓ Обработка ошибок является необязательной. Вы не получите никаких преимуществ от проверок в функции `Factorial ()`, если вызывающая функция не будет в свою очередь проверять возвращаемое значение. Конечно, руководитель группы может просто сказать: "Парни, или вы проверяете коды ошибок, или занимаете очередь на бирже труда", но совсем иное дело, когда проверку заставляет выполнить сам язык программирования.

Зачастую при проверке кода ошибки, возвращаемого функцией `Factorial ()` или любой другой функцией, практически вся вызывающая функция оказывается заполненной проверками всех возможных кодов ошибок от всех вызванных функций, при этом просто не остается ни сил, ни места сделать в функции хоть что-то полезное. Судите сами:

```
// Вызов SomeFuncO, проверка кода ошибки, его обработка и
// возврат из функции
errRtn = SomeFunc () ;
if (errRtn == SF_ERROR1)
(
    Console.WriteLine ("Ошибка типа 1 при вызове SomeFuncO");
    return MY_ERROR 1;
}
if (errRtn == SF_ERROR2)
{
    Console.WriteLine ("Ошибка типа 2 при вызове SomeFuncO");
    return My_ERROR 2;
}
// Вызов другой функции, проверка кода ошибки и так далее..
errRtn = SomeOtherFuncO;
if (errRtn == SOF_ERROR1)
{
    Console.WriteLine("Ошибка типа 1 при вызове " +
                      "SomeOtherFunc () ");
    return MY_ERROR 3;
}
if (errRtn == SOF_ERROR2)
{
    Console.WriteLine("Ошибка типа 2 при вызове " +
```

```

        "SomeOtherFunc()");
return MY_ERROR_4;
}

```

Такой механизм имеет ряд проблем.

- ✓ В нем очень много повторов. Дублирование кода обычно очень неприятно по-| пахивает...
- ✓ Он заставляет пользователя функции поддерживать проверку массы кодов ошибок
- ✓ Код обработки ошибок оказывается перемешан с обычным кодом, что затеняет основную работу программы и делает исходный текст неудобочитаемым.

Все эти проблемы кажутся мелкими в простых примерах, но все становится гораздо хуже с ростом сложности вызываемых функций. В конечном итоге код обработки ошибок не перехватывает все ошибки, которые могут возникнуть.

К счастью, описанный далее механизм исключений решает указанные проблемы.

Использование механизма исключений для сообщения об ошибках

В C# для перехвата и обработки ошибок используется совершенно иной механизм, называемый *исключениями*. Он основан на ключевых словах `try`, `catch`, `throw` и `finally`. Набросать схему его работы можно следующим образом. Функция пытается (`try`) пробраться через кусок кода. Если в нем обнаружена проблема, она бросает (`throw`) индикатор ошибки, который функции могут поймать (`catch`), и независимо от того, что именно произошло, в конце (`finally`) выполнить специальный блок кода, как показано в следующем наброске исходного текста:

```

public class MyClass
{
    public void SomeFunction()
    {
        // Настройка для перехвата ошибки
        try
        {
            // Вызов функции или выполнение каких-то иных
            // действий, которые могут генерировать исключение
            SomeOtherFunction();
            // . . . Какие-то иные действия . . .
        }
        catch(Exception e)
        {
            // Сюда управление передается в случае, когда в блоке
            // try сгенерировано исключение — в самом ли блоке, в
            // функции, которая в нем вызывается, в функции,
            // которая вызывается функцией, вызванной в try-блоке
            // и так далее — словом, где угодно. Объект Exception
            // описывает ошибку

```

' Далее будет использоваться выражение "генерирует исключение". — Примеч. ред.

```

    }
    finally
    {
        // Выполнение всех завершающих действий: закрытие
        // файлов, освобождение ресурсов и т.п. Этот блок
        // выполняется независимо от того, было ли
        // сгенерировано исключение.
    }
}

public void SomeOtherFunction()
{
    II... Ошибка произошла где-то в теле функции . . .
    II... Ж "пузырек" исключения "всплывает" вверх по
    // всей цепочке вызовов, пока не будет перехвачен в
    // блоке catch
    throw new Exception("Описание ошибки");
    II... Продолжение функции . . .

```

Функция `SomeFunction()` помещает некоторую часть кода в блок, помеченный ключевым словом `try`. Любая функция, вызываемая в этом блоке (или функция, вызываемая функцией, вызываемой в этом блоке — и так далее...), рассматривается как вызванная в данном `try`-блоке.

Непосредственно за блоком `try` следует ключевое слово `catch` с блоком, которому передается управление в случае, если где-то в `try`-блоке произошла ошибка. Аргумент `catch`-блока — объект класса `Exception` (или некоторого подкласса `Exception`).

Однако `catch`-блок не обязан иметь аргументы: пустой `catch` перехватывает все исключения, как и `catch (Exception)`:

```

catch

```

Если вам не нужен доступ к информации из объекта перехваченного исключения, вы можете указать в блоке только тип исключения:

```

catch (MyException)
{
    // Действия, которые не требуют обращения к объекту
    // исключения
}

```

Блок `finally` — если таковой имеется в вашем исходном тексте — выполняется даже в случае перехвата исключения, не говоря уже о том, что он выполняется при нормальной работе. Обычно он предназначается для "уборки" — закрытия открытых файлов, освобождения ресурсов и т.п.



В отличие от исключений C++, в которых аргументом `catch` может быть объект произвольного типа, исключения C# требуют, чтобы он был объектом класса `Exception` или производного от него.

Итак, где-то в дебрях кода на неизвестно каком уровне вызовов в функции `SomeOtherFunction()` случилась ошибка... Функция сообщает об этом, генерируя исклю-

чение в виде объекта Exception и передает его с помощью оператора throw вверх цепочке вызовов в первый же блок, который в состоянии перехватить его и обработать!



Иногда обработчик try/catch располагается в той же функции, в которой нерировано исключение. Первая же функция, владеющая достаточным количеством контекстуальной информации для выполнения действий по его обработке может перехватить и обработать его; если данная функция не в состоянии этого сделать, исключение передается дальше вверх по цепочке вызовов.

Пример



В демонстрационной программе FactorialException приведены ключевые элементы механизма исключений.

```
// FactorialException - создание функции вычисления
// факториала, которая сообщает о некорректном аргументе с
// использованием исключений
using System;
namespace FactorialException
{
    // MyMathFunctions - набор созданных мною математических
    // функций
    public class MyMathFunctions
    {
        // Factorial - возвращает факториал переданного
        // аргумента
        public static double Factorial(int nValue)
        {
            // Проверка: отрицательные значения запрещены
            if (nValue < 0)
            {
                // Сообщение об отрицательном аргументе
                string s = String.Format(
                    "Отрицательный аргумент в вызове Factorial {0}",
                    nValue);
                throw new Exception(s); // Генерация исключения...
            }
            // начинаем со значения аккумулятора,
            // равного 1
            double dFactorial = 1.0;
            // Цикл со счетчиком nValue, уменьшающимся до 1, с
            // умножением на каждой итерации значения аккумулятора
            // на величину счетчика
            do
            {
                dFactorial *= nValue,--
            } while(--nValue > 1);
            // Возвращаем вычисленное значение
        } return dFactorial;
    }
```

```

public class Program
{
    public static void Main(string[] args)
    {
        try // Исключения от функции Factorial() "всплывут" до
            // этого блока
        {
            // Вызов функции вычисления факториала в
            // цикле от 6 до -6
            for (int i = 6; i > -6; i--)
            {
                // Вычисление факториала
                double dFactorial = MyMathFunctions.Factorial(i);
                // Вывод результата на каждой итерации
                Console.WriteLine("i = {0}, факториал = {1}",
                    i, MyMathFunctions.Factorial(i));

            }

            catch(Exception e) // ... перехват исключения
            {
                Console.WriteLine("Ошибка:");
                Console.WriteLine(e.ToString());
            }
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                "завершения программы...");
            Console.Read();
        }
    }
}

```

Эта "исключительная" версия функции Main () практически полностью находится в try-блоке.



Всегда помещайте содержимое функции Main () в try-блок, поскольку функция Main () — начальная и конечная точка программы. Любое исключение, не перехваченное где-то в другом месте, будет передано функции Main (). Это последняя возможность перехватить исключение перед тем, как оно попадет прямо в Windows, где это сообщение об ошибке будет гораздо сложнее интерпретировать.

Блок catch в конце функции Main () перехватывает объект Exception и использует его метод ToString () для вывода информации об ошибке, содержащейся в этом объекте в виде строки.



Более консервативное свойство Exception.Message возвращает более удобочитаемую, но менее информативную информацию по сравнению с предоставляемой методом e.ToString ().

Эта версия функции Factorial () включает ту же проверку на отрицательность переданного аргумента, что и предыдущая (для экономии места в ней опущена проверка того, что аргумент — целое число). Если аргумент отрицателен, функция Factorial ()

форматирует сообщение об ошибке с описанием ситуации, включая само отрицательное значение, вызвавшее ошибку. Затем функция `Factorial()` вносит информацию в вновь создаваемый объект типа `Exception`, который передается с помощью механизма исключений вызывающей функции.

Вывод этой программы выглядит следующим образом:

```
i = 6, факториал = 720
i = 5, факториал = 120
i = 4, факториал = 24
i = 3, факториал = 6
i = 2, факториал = 2
i = 1, факториал = 1
i = 0, факториал = 0
Ошибка:
System.Exception: Отрицательный аргумент в вызове Factorial -1
at Factorial(Int32 nValue) in
  c:\c#programs\Factorial\Program.cs:line 21
at FactorialException.Program.Main(String[] args) in
  c:\c#programs\Factorial\Program.cs:line 49
Нажмите <Enter> для завершения программы...
```

В первых нескольких строках выводятся корректно вычисленные факториалы число от 6 до 0⁹. Попытка вычислить факториал -1 приводит к генерации исключения.

В первой строке сообщения об ошибке выводится информация, сгенерированная в функции `Factorial()`. Эта строка описывает природу ошибки, включая вызвавшее неприятности значение аргумента—1.

В оставшейся части вывода выполняется трассировка стека. В первой строке указывается, в какой функции сгенерировано исключение. В данном случае это было сделано в функции `Factorial(int)` — а именно в 21 строке исходного файла `Program.cs`. Функция `Factorial()` была вызвана из функции `Main(string[])` в строке 49 того же файла. На этом трассировка файла прекращается, поскольку функция `Main()` захватит блок, перехвативший и обработавший указанное исключение.



Трассировка стека доступна в одном из окон отладчика Visual Studio.

Вы должны согласиться, что это весьма впечатляюще. Сообщение об ошибке описывает случившееся и позволяет указать аргумент, приведший к ней. Трассировка стека полностью отслеживает, где именно и в результате какой последовательности вызовов произошла ошибка. При такой диагностике поиск ошибки и ее причины не должен составить никакого труда.



Получение подробной информации, такой как трассировка стека, удобно в процессе разработки и отладки, но при выходе программы к конечным пользователям вам наверняка захочется, чтобы она выдавала более простые и понятные сообщения. (При этом все равно останется возможность записи трассировки стека в журнальный файл, чтобы вы могли владеть всей необходимой информацией, когда пользователь обратится к вам за помощью.)

⁹ Еще раз напомним читателю, что в математике принято считать, что $0! = 1$. — *Примеч. ред.*

Создание собственного класса исключения

Стандартный класс исключения `Exception`, предоставляемый библиотекой `C#`, в состоянии предоставить вам достаточное количество информации. Вы можете запросить объект исключения о том, где он был сгенерирован, какая строка была передана ему генерирующей функцией. Однако в ряде случаев стандартного класса `Exception` бывает недостаточно. У вас может оказаться слишком много информации, чтобы разместить ее в одной строке. Например, функция приложения может захотеть передать вызвавший проблемы объект для последующего анализа. Изучение этого объекта может быть полезным вплоть до полного восстановления после произошедшей ошибки.

Локально определенный класс может наследовать класс `Exception` так же, как и любой другой класс. Однако пользовательский класс исключения должен наследовать не непосредственно класс `Exception`, а класс `ApplicationException`, являющийся подклассом `Exception`, как показано в следующем фрагменте исходного текста:

```
// CustomException - добавление ссылки на MyClass к
// стандартному классу исключения
public class CustomException : ApplicationException
{
    private MyClass myobject; // Хранит ссылку на вызвавший
                               // проблемы объект
    CustomException(string sMsg, MyClass mo) : base(sMsg)
    {
        myobject = mo;
    }
    // Предоставляет доступ к объекту, сохраненному в объекте
    // исключения
    public MyClass MyCustomObject{ get {return myobject;}}
}
```

Класс `CustomException` представляет собой самодельный класс для сообщения об ошибке в любой программе, работающей с классом `MyClass`. Этот подкласс класса `ApplicationException` содержит такую же строку, как и исходный класс, но добавляет к ней ссылку на объект `MyClass`, вызвавший проблемы. Это позволяет произвести детальное исследование случившегося.

В приведенном далее примере выполняется перехват исключения `CustomException` и используется информация об объекте `MyClass`:

```
public class Program
{
    public void SomeFunction()
    {
        try
        {
            // ... действия перед вызовом демонстрационной функции
            SomeOtherFunctionO ;
            // ... продолжение работы ...
        }
        catch(MyException me)
```

```

        // Здесь у вас имеется доступ к методам Exception и
        // ApplicationException
        string s = me.ToString();
        // Но у вас есть еще и доступ к методам, уникальным
        // для вашего класса исключения
        MyClass mo = me.MyCustomObject;
        // Например, вы можете запросить у объекта MyClass его
        // собственное описание
        string s = mo.GetDescription();
    }
}
public void SomeOtherFunctionO
{
    // Создание myobject
    MyClass myobject = new MyClassO;
    // ... сообщение об ошибке с участием myobject ..
    throw new MyException("Ошибка в объекте MyClass",
                           myobject);
    // ... Остальная часть функции ...
}
}

```

В этом фрагменте кода функция `SomeFunction()` вызывает функцию `SomeOtherFunction()` из охватывающего блока `try`. Функция `SomeOtherFunction()` создает и использует объект `myobject`. Где-то в функции `SomeOtherFunction()` программа проверки ошибок подготавливает исключение к генерации для сообщения о произошедшей ошибке. Вместо создания простого объекта типа `Exception` или `ApplicationException`, функция `SomeFunctionO` применяет разработанный ваш тип `MyException`, пригодный не только для передачи текстового сообщения об ошибке, но и ссылки на вызвавший ее объект.

Блок `catch` в функции `Main()` указывает, что он предназначен для перехвата объектов `MyException`. После того как такой объект перехвачен, код приложения в состоянии применить все методы `Exception`, как, например, метод `ToString()`. Однако в этом `catch`-блоке может использоваться и другая информация, к примеру, вызов методов объекта `MyClass`, ссылка на который передана в объекте исключения.

Использование нескольких *catch*-блоков

Фрагмент кода в предыдущем разделе продемонстрировал генерацию и перехват локально определенного объекта исключения `MyException`. Рассмотрим еще раз конструкцию `catch` из этого примера:

```

public void SomeFunction()
{
    try
    {
        SomeOtherFunctionO ;
    }
    catch(MyException me)
    {
    }
}

```


А если функция `SomeOtherFunction()` сгенерирует простое исключение `Exception` или исключение еще какого-то типа, отличного от `MyException`? Это будет напоминать ситуацию, когда футбольный вратарь ловит баскетбольный мяч — мяч, ловить который он не научен. К счастью, C# позволяет программе определить несколько блоков `catch`, каждый из которых предназначен для различного типа исключений.

Блоки `catch` должны в этом случае следовать один за одним, без разрывов, в порядке от наиболее специализированных классов ко все более общим. C# проверяет каждый `catch`-блок, последовательно сравнивая сгенерированный объект с аргументами `catch`-блоков, как показано в следующем фрагменте исходного текста:

```
public void SomeFunction ()
{
    try

        SomeOtherFunction0 ;

    catch(MyException me) // Наиболее специализированный тип
    {                       // исключения
        // Здесь перехватываются все объекты MyException
    } // Между этими catch-блоками могут находиться блоки с
        // другими типами исключений
    catch(Exception e)     // Наиболее общий тип исключения

        // Все остальные неперехваченные исключения
        // перехватываются в этом блоке
    }
```



Если функция `SomeOtherFunction()` сгенерирует объект `Exception`, он минует блок `catch (MyException)`, поскольку `Exception` не является типом `MyException`. Он будет перехвачен в следующем блоке — `catch (Exception)`.



Любой класс, наследующий `MyException`, **ЯВЛЯЕТСЯ** `MyException`:

```
class MySpecialException : MyException
{
    // ... что-то там ...
}
```

В этом случае блок для `MyException` перехватит и объект `MySpecialException`. (Наследование всех пользовательских исключений от одного базового пользовательского исключения — неплохая мысль. Само базовое исключение наследуйте от `ApplicationException`.)



Всегда располагайте `catch`-блоки от наиболее специализированного к наиболее общему. Никогда не размещайте более общий блок первым, как это сделано в приведенном фрагменте исходного текста:

```
public void SomeFunction ()
{
    try
    {
        SomeOtherFunction();
    }
}
```

```

catch(Exception me) // Самый общий блок - это неверно!
{
    // Все объекты MyException будут перехвачены здесь
}
catch(MyException e)
{
    // Сюда не доберется ни один объект - все они будут
    // перехвачены более общим блоком
}
}

```

Более общий блок отнимает объекты исключений у более специализированного блока. К счастью, компилятор в состоянии обнаружить такую ошибку и предупредить о ее наличии.

Как исключения протекают сквозь пальцы

Что, если С#, пройдя все catch-блоки, так и не найдет подходящего? Или в вызываемой функции вообще нет catch-блока? Что будет тогда?



Рассмотрим следующую простую цепочку вызовов функций:

```

// MyException - демонстрация того, как можно создать новый
// класс исключения и как функция может перехватывать только
// те исключения, которые может обработать
using System;
namespace MyException
{
    // Вводим некоторый тип MyClass
    public class MyClass{}
    // MyException - добавляем ссылку на MyClass к
    // стандартному классу исключения
    public class MyException : ApplicationException

        private MyClass myobject;

        public MyException(string sMsg, MyClass mo) : base(sMsg)

            myobject = mo,-
    }
    // Дает внешним классам доступ к объекту
    public MyClass MyCustomObject{ get {return myobject;}}
public class Program
{
    // f1 - перехватывает обобщенный объект исключения
    public void f1(bool bExceptionType)

        try
        {
            f2(bExceptionType);

        catch(Exception e)
        {

```

```

        Console.WriteLine("Перехват обобщенного " +
                           "исключения в fl 0");
        Console.WriteLine(e.Message);
    }
}
// f2 - - готов к перехвату MyException
public void f2(bool bExceptionType)
{
    try
    {
        f3(bExceptionType);
    }
    catch(MyException me)
    {
        Console.WriteLine("Перехват MyException в f2()");
        Console.WriteLine(me.Message);
    }
}
// f3 - - Не перехватывает никаких исключений
public void f3(bool bExceptionType)
{
    f4(bExceptionType);
}
// f4 - - генерация одного из двух типов исключений
public void f4(bool bExceptionType)
{
    // Работаем с некоторым локальным объектом
    MyClass mc = new MyClass 0;
    if(bExceptionType)
    {
        // Произошла ошибка – генерируем объект исключения с
        // объектом
        throw new MyException("Генерация MyException " +
                               "в f4()", mc);
    }
    throw new Exception("Обобщенное исключение в f4()");
}
public static void Main(string[] args)
{
    // Сначала генерируем обобщенное исключение...
    Console.WriteLine("Сначала генерируем " +
                      "обобщенное исключение");
    new Program0 .fl (false);
    // ... а теперь наше исключение
    Console.WriteLine("\nГенерируем исключение " +
                      "MyException");
    new Program0 .fl(true);
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}

```

Функция `Main()` создает объект `Program` и тут же использует его для вызова метода `f1()`, который, в свою очередь, вызывает метод `f2()`, который вызывает метод `f3()`, вызывающий метод `f4()`. Функция `f4()` выполняет сложную проверку ошибки которая выливается в генерацию либо исключения `MyException`, либо обобщенного исключения `Exception`, в зависимости от аргумента типа `bool`. Вначале сгенерированное исключение `Exception` передается в функцию `f3()`. Здесь `C#` не находит `catch`-блока, и управление передается вверх по цепочке в функцию `f2()`, которая перехватывает исключения `MyException` и его наследников. Этот тип исключения не соответствует обобщенному исключению `Exception`, и управление передается еще дальше вверх. Наконец, в функции `f1()` находится `catch`-блок, соответствующий сгенерированному исключению.

Второй вызов в функции `Main()` заставляет функцию `f4()` сгенерировать объект `MyException`, который перехватывается в функции `f2()`. Это исключение не пересылается функции `f1()`, поскольку оно перехватывается и обрабатывается функцией `f2()`.

(Может ли функция `Main()` в действительности создать объект класса, содержащий объект класса, в котором содержится `Main()` — т.е. класса `Program`? Конечно, почему бы и нет? См. последний раздел главы 14, "Интерфейсы и структуры".)

Вывод программы выглядит следующим образом:

```
Сначала генерируем обобщенное исключение
Перехват обобщенного исключения в f1()
Обобщенное исключение в f4()
```

```
'Генерируем исключение MyException
Перехват MyException в f2()
Генерация MyException в f4()
Нажмите <Enter> для завершения программы...
```

Функция наподобие `f3()`, не содержащая ни одного `catch`-блока, вовсе не редкость. Можно сказать даже больше — такие функции встречаются гораздо чаще, чем функции с `catch`-блоками. Функция не должна перехватывать исключения, если она не готова их обработать. Должна ли некоторая математическая функция `ComputeX()` в которой вызывается функция `Factorial()` как часть вычислений, перехватывать исключение, которое может быть сгенерировано функцией `Factorial()`? Функция `ComputeX()` может совершенно не представлять, откуда взялись неверные входные данные для функции `Factorial()` и что теперь делать. В этом случае функция `ComputeX()`, конечно же, не должна содержать `catch`-блока и перехватывать перехватываемые исключения.

Функция наподобие `f2()` перехватывает только один тип исключений. Она ожидает только один определенный тип ошибки, который умеет обрабатывать. Например `MyException` может быть исключением, определенным для выдающейся библиотеки классов гениального автора, написанной, понятное дело, мной, и так и называющейся — `BrilliantLibrary`. Функции, составляющие `BrilliantLibrary`, перехватывают и перехватывают только исключения `MyException`.

Однако функции `BrilliantLibrary` могут также вызывать функции обычной стандартной библиотеки `System`. Функции `BrilliantLibrary` могут не знать, как следует обрабатывать обобщенные исключения `System`, в особенности если они вызваны некорректными входными данными.



Если вы не знаете, что делать с исключением — лучше не делайте ничего: позвольте разобраться с ним вызывающей функции. Но будьте честны сами с собой: не позволяйте исключению уйти только потому, что вы слишком заняты, чтобы писать обработчик.

Регенерация исключения

В ряде случаев метод не в состоянии полностью обработать ошибку, но не хочет передавать исключение вызывающей функции, не вложив свои "пять копеек" в его обработку. В таком случае `catch`-блок может частично выполнить обработку исключения, а затем передать его дальше (вообще-то, не слишком привлекательная картина).

Рассмотрим, например, метод `F()`, который открывает файл при входе в метод с намерением закрыть его при выходе из метода. Где-то в середине работы `F()` вызывается `G()`. Исключение, сгенерированное в `G()`, может привести к тому, что у `F()` не будет шансов закрыть этот файл, который так и останется открытым до полного завершения программы. Идеальным решением было бы включение в `F()` `catch`-блока (или блока `finally`), который бы закрывал все открытые файлы. `F()` может передать исключение дальше после того, как закроет все необходимые файлы и выполнит прочие требуемые действия.

"Регенерировать" исключение можно двумя способами. Один из них состоит в генерации второго исключения с дополнительной (или, как минимум, той же) информацией следующим образом:

```
public void f1 ()
{
    try
    {
        f2 ();
    }
    // Перехват исключения...
    catch(MyException me)
    {
        // ... Частичная обработка исключения ...
        Console.WriteLine("Перехват MyException в f1()");
        // ... Генерация нового исключения для передачи его
        // вверх по цепочке вызовов
        throw new Exception(
            "Исключение, сгенерированное в f1()");
    }
}
```



Генерация нового объекта исключения позволяет классу переформулировать сообщение об ошибке, добавив в него дополнительную информацию. Генерация обобщенного объекта `Exception` вместо специализированного `MyException` обеспечивает гарантированный перехват этого исключения на уровнях выше `f1()`.

Генерация нового исключения имеет тот недостаток, что трассировка стека при этом начинается заново, с точки генерации нового исключения. Источник исходной ошибки оказывается потерянным, если только `finally` не предпримет специальных мер для его сохранения.

Второй путь состоит во включении в исходный текст команды `throw` без аргумента, что приводит к генерации того же объекта исключения, как показано в следующем фрагменте исходного текста:

```
public void f1()
{
    try
    {
        f2();
    }
    // Перехват исключения...
    catch(Exception e)
    {
        // ... Частичная обработка исключения ...
        Console.WriteLine("Перехват исключения в f1()");
        // ... исходное исключение продолжает свой путь по
        // цепочке вызовов
        throw;
    }
}
```

Повторная генерация того же объекта имеет свои преимущества и недостатки (ну почему они всегда идут рука об руку?). Регенерация дает возможность промежуточной функции перехватить исключение и освободить или закрыть используемые ресурсы, при этом позволяя объекту исключения донести информацию о месте происшествия до окончательного обработчика этой ошибки. Однако промежуточные функции не могут (или не должны) добавлять какую-либо информацию, модифицируя объект исключения перед его повторной генерацией.

Как реагировать на исключения

Какие у вас имеются варианты при написании `catch`-блоков? Как объяснялось ранее, вы можете выполнить одно из следующих трех действий:

- ✓ перехватить исключение;
- ✓ проигнорировать исключение;
- ✓ частично обработать исключение и повторно его сгенерировать (возможно, с добавлением новой информации) либо просто регенерировать его.

Но какой стратегии необходимо придерживаться при проектировании системы исключений?

- ✓ **В конечном итоге постарайтесь восстановиться после ошибки.** Скорректируйте входные данные, замените их верными, запросите корректные данные у пользователя — словом, решите вопрос каким-то образом, чтобы можно было продолжить выполнение программы, как будто ничего не случилось.

Если это возможно — используйте транзакционный подход. Откатите все изменения, которые были сделаны к моменту возникновения ошибки, восстановите файлы в исходное состояние и т.д. Главное правило — не испортить пользовательские данные. Всегда помогайте пользователю в восстановлении, насколько это возможно. Если даже ничего и не получится, то оставит о вас хорошее впечатление...

- ✓ Иногда, когда ошибка не фатальна, достаточно просто поставить пользователя в известность о происшедшем. Например, вы не смогли открыть указанный пользователем файл. Пользователь может быть раздражен и огорчен этим фактом, но, по крайней мере, он сможет продолжить свою работу.
- ✓ Иногда вы не можете сделать ничего. В этом случае вам остается развести руками, сообщить об этом пользователю и по возможности грациозно завершить работу программы.

Обычно это сводится к выводу красивого, соболезнующего, но информативного сообщения об ошибке и завершению программы. Если возможно, предложите пользователю варианты его действий по исправлению ситуации.

Перекрытие класса Exception

Следующий пользовательский класс может сохранить дополнительную информацию, что невозможно в процессе применения стандартных объектов Exception или ApplicationException:

```
// MyException - к стандартному классу исключения добавлена
// ссылка на MyClass
public class MyException : ApplicationException
{
    private MyClass myobject;
    MyException(string sMsg, MyClass mo) : base(sMsg)
    {
        myobject = mo;
    }
    // Позволяет внешним классам обращаться к сохраненному
    // в исключении классу
    public MyClass MyObject{ get {return myobject;}}
```

Вернемся вновь к библиотеке функций BrilliantLibrary. Эти функции знают, как заполнять и считывать новые члены класса MyException, тем самым предоставляя информацию, необходимую для отслеживания каждой ошибки. Проблема при таком подходе заключается в том, что только функции BrilliantLibrary могут получить все преимущества от использования новых членов MyException.



Перекрытие методов, имеющихся у классов Exception или ApplicationException, может предоставить функциям вне BrilliantLibrary доступ к новым данным. Рассмотрим класс исключения из следующей демонстрационной программы CustomException.

```
// CustomException - создание пользовательского исключения,
// которое выводит информацию в более дружелюбном формате
using System;
namespace CustomException
{
    public class CustomException : ApplicationException
    {
        private MathClass mathobject;
        private string sMessage;
```

```

public CustomException(string sMsg, MathClass mo)
{
    mathobject = mo;
    sMessage = sMsg;
}
override public string Message
{
    get{return String.Format("Сообщение <{0}>,
                                Объект {1}",
                                sMessage,
                                mathobject.ToString());}

override public string ToString()
{
    string s = Message;
    s += "\nИсключение сгенерировано в ";
    s += TargetSite.ToString(); // Информация о методе,
                                // сгенерировавшем исключение
    return s;
}
}
// MathClass - набор созданных мною математических функций
public class MathClass
{
    private int nValueOfObject;
    private string sObjectDescription;
    public MathClass(string sDescription, int nValue)
    {
        nValueOfObject = nValue;
        sObjectDescription = sDescription;
    }
    public int Value {get {return nValueOfObject;}}
    // Message - вывод сообщения со значением
    // присоединенного объекта MathClass
    public string Message
    {
        get
        {
            return String.Format("({0} = {1})",
                                sObjectDescription,
                                nValueOfObject);
        }
    }
}
// ToString - расширение нашего пользовательского
// свойства Message с использованием Message из базового
// класса исключения
override public string ToString()
{
    string s = Message + "\n";
    s += base.ToString();
    return s;
}
// Вычисление обратного значения 1/x
public double Inverse()

```



```

    {
        if (nValueOfObject == 0)
        {
            throw new CustomException("Нельзя делить на 0",
                                      this);
        }
        return 1.0 / (double)nValueOfObject;
    }
}

public class Program
{
    public static void Main(string[] args)

    try
    {
        // take the inverse of 0
        MathClass mathObject = new MathClass("Value", 0);
        Console.WriteLine("Обратное к d.Value равно {0}",
                          mathObject.Inverse());

    catch(Exception e)
    {
        Console.WriteLine(
            "\nНеизвестная фатальная ошибка:\n{0}",
            e.ToString());
    }
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}
}

```

Класс `CustomException` несложен. Он хранит сообщение и объект, как это делал класс `MyException` ранее. Однако вместо предоставления новых методов для обращения к этим элементам данных он перекрывает существующее свойство `Message`, которое возвращает сообщение об ошибке, содержащееся в исключении, и метод `ToString()`, возвращающий сообщение и трассировку стека.

Перекрытие этих функций означает, что даже функции, разработанные для перехвата обобщенного класса `Exception`, получают ограниченный доступ к новым членам-данным. Новый класс лучше обеспечить собственными методами для их данных и оставить нетронутыми методы базового класса.

Функция `Main()` демонстрационной программы начинается с создания объекта `MathClass` со значением 0, а затем пытается вычислить обратную к нему величину. Не знаю, как вам, а мне не приходилось видеть разумные результаты деления на 0, так что если моя функция вдруг сделает это, я отнесусь к происшедшему с явным недоверием.



На самом деле процессоры Intel возвращают значение `1.0/0.0`: бесконечность. Имеется ряд специальных значений с плавающей точкой, используемых вместо генерации исключений в языках, которые не поддерживают их. Эти специальные значения включают положительную и отрицательную бесконечности и положительное и отрицательное NaN (`Not_a_Number`, не число).

В нормальных условиях метод `Inverse ()` возвращает корректное значение. При передаче ему нуля он генерирует исключение `CustomException`, передавая ему фразу пояснения вместе с вызвавшим исключение объектом.

Функция `Main ()` перехватывает исключение и выводит короткое сообщение, поясняющее суть происшедшего. "Неизвестная фатальная ошибка", вероятно, означает, та программа "закрывает лавочку и уходит на отдых". Но функция `Main ()` дает исключению шанс пояснить, что же все-таки произошло, вызывая его метод `ToString ()`.

Визитка класса: метод `ToString ()`

Все классы наследуют один общий базовый класс с именем `Object`. Об этом уже говорилось в главе 14, "Интерфейсы и структуры". Здесь, однако, стоит упомянуть о методе `ToString ()` в составе этого класса. Метод предназначен для преобразования содержимого класса в строку. Идея заключается в том, что каждый класс должен переопределять метод `ToString ()`, чтобы осуществить вывод значащей информации. В первых главах был использован метод `GetString ()`, чтобы не касаться в них вопроса наследования; однако принцип остается тем же. Например, корректный метод `Student.ToString ()` может выводить имя и идентификатор студента.

Большинство функций — даже встроенных в библиотеку C# — применяют метод `ToString ()` для вывода объектов. Таким образом, переопределение `ToString ()` имеет очень полезное побочное действие, заключающееся в том, что каждый объект выводится в своем собственном формате, безотносительно к тому, кем именно он выведен.

Поскольку объект исключения в этом случае на самом деле принадлежит типу `CustomException`, управление передается `CustomException.ToString ()`.



Метод `Message ()` представляет собой виртуальный метод класса `Exception`, так что его можно переопределять, но пользовательское исключение должно наследовать его без переопределения.

Метод `Message ()` позволяет объекту `MathClass` выводить информацию о самом себе с использованием метода `ToString ()`. Метод `MathClass.ToString ()` возвращает строку, в которой содержится описание и значение объекта.



Не следует брать на себя больше того, что имеете. Используйте метод объекта `ToString ()` для создания строковой версии объекта, не пытайтесь влезть в сам объект и получить его значения. В общем случае нужно полагаться на открытый интерфейс — открытые члены, — а не на знания о внутреннем устройстве объекта. Оно инкапсулировано (по крайней мере должно быть инкапсулировано) и может измениться в новых версиях.

Вывод демонстрационной программы `CustomException` имеет следующий вид:

```
Неизвестная фатальная ошибка:
Сообщение <Нельзя делить на 0>, Объект (Value = 0)
CustomException.MathClass
Исключение сгенерировано в Double Inverse()
Нажмите <Enter> для завершения программы...
```

И последнее: сообщение "Неизвестная фатальная ошибка:" поступает от `Main()`. Строка "Сообщение <Нельзя делить на 0>, Объект <~~>" поступает от `CustomException`. Часть `Value = 0` предоставляет объект `MathClass`. Последняя строка, Исключение сгенерировано в `Double Inverse()`, принадлежит `CustomException`. Это нельзя назвать иначе, как исключительным сотрудничеством.

Глава 19

Работа с файлами и библиотеками

В этой главе...

- > Работа с несколькими исходными файлами в одной программе
- У Сборки и пространства имен
- > Библиотеки классов
- > Чтение и запись файлов данных

Доступ к файлам в С# может иметь два различных значения. Наиболее очевидное — это хранение и считывание данных с диска. О том, как осуществляется ввод-вывод данных с диска, вы узнаете из этой главы. Второе значение связано с тем, каким образом исходный текст С# группируется в исходные файлы.

Функции позволяют разделить длинную строку исходного текста на отдельные модули, которые можно легче сопровождать и поддерживать. Классы дают возможность группировать данные и функции для дальнейшего снижения сложности программы — программы достаточно сложны, а людям свойственно ошибаться, так что нужно пользоваться любой возможностью упрощения, которая может помочь избежать ошибок.

С# обеспечивает еще один уровень группировки: он позволяет сгруппировать подобные классы в отдельную библиотеку. Помимо написания собственных библиотек, вы можете использовать в ваших программах и чужие библиотеки. Такие программы содержат множество модулей, называемых сборками (assemblies). О них также будет рассказано в данной главе. Кроме того, описанное в главе 11, "Классы", управление доступом на самом деле несколько сложнее в связи с применением пространств имен — еще одного способа группирования похожих классов, которое заодно позволяет избежать дублирования имен в двух частях программы. В этой главе речь пойдет и о них.

Разделение одной программы на несколько исходных файлов

Программы в настоящей книге носят исключительно демонстрационный характер. Каждая из них длиной не более нескольких десятков строк и содержит не более пары классов. Программы же промышленного уровня со всеми "рюшечками" и "финтифлюшечками" могут состоять из сотен тысяч строк кода с сотнями классов.

Рассмотрим систему продажи авиабилетов. У вас должен быть один интерфейс для заказа билетов по телефону, другой — для тех, кто заказывает билет по Интернету, должна быть часть программы, отвечающая за управление базой данных билетов, дабы не продавать один и тот же билет несколько раз, еще одна часть должна следить за стой-

мостью билетов с учетом всех налогов и скидок, и так далее и тому подобное... Такая программа будет иметь огромный размер.

Размещение всех составляющих программу классов в одном исходном файле `Program.cs` быстро становится непрактичным. Оно даже более неприемлемо, чем разделение имущества, которого добилась моя бывшая жена, по следующим причинам.

- ✓ **У вас возникнут проблемы при поддержке классов.** Единый исходный файл очень трудно поддается пониманию. Гораздо проще разбить его на отдельные модули, например `ResAgentInterface.cs`, `GateAgentInterface.cs`, `ResAgent.cs`, `GateAgent.cs`, `Fare.cs` и `Aircraft.cs`.
- ✓ **Работа над большими программами обычно ведется группами программистов.** Два программиста не в состоянии редактировать одновременно один и тот же файл — каждому требуется его собственный исходный файл (или файлы). У вас может быть 20 или 30 программистов, одновременно работающих над одним большим проектом. Один файл ограничит работу каждого из 24 программистов над проектом всего одним часом в сутки, но стоит разбить программу на 24 файла, как становится возможным (хотя и сложным) заставить всех программистов трудиться круглые сутки. Разбейте программу так, чтобы каждый класс содержался в отдельном файле, и ваша группа заработает как слаженный оркестр.
- ✓ **Компиляция больших файлов занимает слишком много времени.** В результате босс начнет нервничать и выяснять, почему это вы так долго пьете кофе вместо того, чтобы стучать по клавишам?

Какой смысл перестраивать всю программу, когда кто-то из программистов изменил пару строк кода? Visual Studio 2005 может перекомпилировать только измененный файл и собрать программу из уже готовых объектных файлов.

По всем этим причинам программисты на C# предпочитают разделять программу на отдельные исходные файлы `.CS`, которые компилируются и собираются вместе в единый выполнимый `.EXE`-файл.



Файл проекта содержит инструкции о том, какие файлы входят в проект и как они должны быть скомбинированы друг с другом.

Можно объединить файлы проектов для генерации комбинаций программ, которые зависят от одних и тех же пользовательских классов. Например, вы можете захотеть объединить программу записи с соответствующей программой чтения. Тогда, если изменяется одна из них, вторая перестраивается автоматически. Один проект может описывать программу записи, второй — программу чтения. Набор файлов проектов известен под названием *решение* (solution). (Далее в главе будут рассматриваться две такие программы — `FileRead` и `FileWrite`, которые можно было бы объединить в одно решение, но это так и не было сделано.)



Программисты на Visual C# используют Visual Studio Solution Explorer для объединения нескольких исходных файлов C# в проекты в среде Visual Studio 2005. Solution Explorer будет описан в главе 21, "Использование интерфейса Visual Studio".

Разделение единой программы на сборки

В Visual Studio, а также в C#, Visual Basic .NET и прочих языках .NET один проект соответствует одному скомпилированному модулю — в .NET он носит имя *сборка*.

C# может создавать два основных типа сборок — выполнимые файлы (с расширением .EXE) и библиотеки классов (.DLL). Выполнимые файлы представляют собой программы сами по себе и используют код поддержки из библиотек. Во всей этой книге создавались исключительно выполнимые файлы. Что касается библиотек классов, то опять же все программы в книге их используют. Например, пространство имен *System* — место размещения таких классов, как *String*, *Console*, *Exception*, *Math* и *Object* — существует как набор библиотечных сборок. Каждой программе требуются классы *System*.



Библиотеки не являются самостоятельными выполнимыми программами.

Библиотека классов состоит из одного или нескольких классов, обычно работающих вместе тем или иным способом. Зачастую классы в библиотеках находятся в своем собственном *пространстве имен* (namespace). (О них речь пойдет в следующем разделе.) Вы можете построить библиотеку математических подпрограмм, библиотеку для работы со строками, библиотеку классов-фабрик и т.д.

Небольшие программы обычно состоят из одной сборки *programName.exe*. Однако часто создаются решения, состоящие из нескольких отдельных (но связанных) проектов, как упоминалось в предыдущем разделе. Каждый из них компилируется в отдельную сборку. В решении вы можете объединять и .EXE-, и .DLL-файлы, что является обычной практикой **ДЛЯ** больших программ. Когда **ВЫ** строите многопроектное решение, сборки работают совместно, обеспечивая функциональность приложения в целом.



Если решение содержит более одного .EXE-проекта, вы должны указать Visual Studio, какой проект является *начальным* (startup project). Именно он будет запускаться при выборе команды меню **Debug^Start Debugging** (F5) or **Debugs Start Without Debugging** (<Ctrl+F5>). Для указания начального проекта щелкните на нем правой кнопкой мыши в окне **Solution Explorer** и выберите в раскрывающемся меню команду **Set as Startup Project**. Имя начального проекта в окне **Solution Explorer** выделяется полужирным шрифтом. О **Solution Explorer** речь пойдет в главе 21, "Использование интерфейса Visual Studio".

Большие программы обычно разделяют свои компоненты на один выполнимый файл и несколько библиотек. Например, весь код, связанный с заказом билетов в рассматриваемом ранее приложении, может находиться в одной библиотеке, работа с Интернетом — в другой, а управление базами данных — в третьей. Когда такая программа устанавливается на компьютер пользователя, процесс инсталляции включает копирование ряда файлов в соответствующие места на диске компьютера, причем многие из них являются .DLL-файлами, или просто "DLL" на сленге программистов (DLL означает dynamic link library (динамически компокуемые библиотеки) — код, который загружается в память тогда, когда в нем возникает необходимость при запуске используемой программы).

Иногда бывает так, что все решение представляет собой не что иное, как библиотек классов, а не выполняемую программу. (Обычно при разработке такой библиотеки создается также сопутствующий .EXE-проект, именуемый *драйвером*, который предназначен для тестирования библиотеки в процессе разработки. Однако при выпуске готовой библиотеки для других программистов вы поставляете им только .DLL, но не .EXE, а также, надеюсь, документацию к этим классам!)

Как создавать собственные библиотеки классов, будет рассказано немного позже в этой главе.

Объединение исходных файлов в пространства имен

Пространства имен существуют для того, чтобы можно было поместить связанные классы в "одну корзину", и для снижения коллизий между именами, используемые в разных местах. Например, вы можете собрать все классы, связанные с математическими вычислениями, в одно пространство имен `MathRoutines`.

Можно (но вряд ли будет сделано на практике) разделить на несколько пространств имен один исходный файл. Гораздо более распространена ситуация, когда несколько файлов группируются в одно пространство имен. Например, файл `Point.cs` может содержать класс `Point`, а файл `ThreeDSpace.cs` — класс `ThreeDSpace`, описывающий свойства Евклидова пространства. Вы можете объединить `Point.cs`, `ThreeDSpace.cs` и другие исходные файлы C# в пространство имен `MathRoutines` (и, вероятно, в библиотечную сборку `MathRoutines`). Каждый файл будет помещать свой код в одно и то же пространство имен. (В действительности пространства имен составляют классы в этих исходных файлах, а не файлы сами по себе.)

Пространства имен служат для следующих целей.

- ✓ **Пространства имен помещают груши к грушам, а не к яблокам.** Как прикладной программист, вы можете не без оснований предполагать, что все классы, составляющие пространство имен `MathRoutines`, имеют отношение к математическим вычислениям. Так что поиск некоторой математической функции следует начать с просмотра классов, составляющих пространство имен `MathRoutines`.
- ✓ **Пространства имен позволяют избежать конфликта имен.** Например, библиотека для работы с файлами может содержать класс `Convert`, который преобразует представление файла одного типа к другому. В то же время библиотека перевода может содержать класс с точно таким же именем. Назначая этим двум множествам классов пространства имен `FileIO` и `TranslationLibrary`, вы устраняете проблему: класс `FileIO.Convert`, очевидно, отличается от класса `TranslationLibrary.Convert`.

Объявление пространств имен

Пространства имен объявляются с использованием ключевого слова `namespace`, за которым следует имя и блок в фигурных скобках. Классы в этом блоке являются частью пространства имен,

```
namespace MyStuff
{
```

```

class MyClass {}
class UrClass {}
}

```

В этом примере классы `MyClass` и `UrClass` являются частью пространства имен `MyStuff`.

Кроме классов, пространства имен могут содержать другие типы, такие как структуры и интерфейсы. Одно пространство имен может также содержать вложенные пространства имен с любой глубиной вложенности. У вас может быть пространство имен `Namespace2`, вложенное в `Namespace1`, как показано в следующем фрагменте исходного текста:

```

namespace Namespace1
{
    // Классы в Namespace1
    namespace Namespace2
    {
        // Классы в Namespace2
        public class Class2
        {
            public void AMethod() { }
        }
    }
}

```

Для вызова метода из `Class2` в `Namespace2` откуда-то извне пространства имен `Namespace1` применяется следующая запись:

```
Namespace1.Namespace2.Class2.AMethod();
```

Пространства имен неявно открыты, так что для них нельзя использовать спецификаторы доступа, даже спецификатор `public`.



Удобно добавлять к пространствам имен в ваших программах название вашей фирмы: `MyCompany.MathRoutines`. (Конечно, если вы работаете на фирме. Вы можете также использовать свое собственное имя. Я бы мог применять для своих программ что-то наподобие `CMSCo.MathRoutines` или `Sphar.MathRoutines`.) Добавление названия фирмы предупреждает коллизии имен в вашем коде при использовании двух библиотек сторонних производителей, у которых оказывается одно и то же базовое имя пространства имен, например, `MathRoutines`.

Такие "имена с точками" выглядят как вложенные пространства имен, но на самом деле это одно пространство имен, так что `System.Data` — это полное имя Пространства имен, а не имя пространства имен `Data`, вложенного в пространство имен `System`. Такое соглашение позволяет проще создавать несколько связанных пространств имен, таких как `System.10`, `System.Data` и `System.Text`.



Visual Studio Application Wizard помещает каждый формируемый им класс в пространство имен, имеющее такое же имя, как и создаваемый им каталог. Взгляните на любую программу в этой книге, созданную Application Wizard. Например, программа `AlignOutput` размещается в папке `AlignOutput`. Имя исходного файла — `Program.cs`, соответствующее имени класса по умолчанию. Имя пространства имен в `Program.cs` то же, что и имя папки: `AlignOutput`. (Можно изменить любое из этих имен, только делайте это осторожно и аккуратно. Изменение имени общего пространства имен проекта выполняется в окне **Properties** проекта.)

Если вы не помещаете ваши классы в пространство имен, C# сам поместит их в **глобальное** пространство имен. Однако лучше использовать собственные пространства имен.

Важность пространств имен



Самое важное в пространствах имен с практической точки зрения то, что они расширяют описание управления доступом, о котором говорилось в главе 11, "Классы" (где были введены такие ключевые слова, как `public`, `private`, `protected`, `internal` и `protected internal`). Пространства имен расширяют управление доступом с помощью дальнейшего ограничения на доступ к членам класса.

Реально пространства имен влияют не на доступность, а на видимость. По умолчанию классы и методы в пространстве имен `NamespaceA` невидимы классам в пространстве имен `NamespaceB`, независимо от их спецификаторов доступа. Но можно сделать классы и методы из пространства имен `NamespaceB` видимыми для пространства имен `NamespaceA`. Обращаться вы можете только к тому, что видимо для вас.

Видимы ли вам необходимые классы и методы?

Для того чтобы определить, может ли класс `Class1` в пространстве имен `NamespaceA` вызывать `NamespaceB.Class2.AMethod()`, рассмотрим следующие два элемента.

1. Видим ли класс `Class2` из пространства имен `NamespaceB` вызывающему классу `Class1`? Это вопрос видимости пространства имен, который будет вскоре рассмотрен.
2. Если ответ на первый вопрос — "да", то "достаточно ли открыты" `Class2` и его метод `AMethod()` классу `Class1` для доступа? "Достаточная открытость" определяется как наличие спецификаторов доступа нужной степени строгости с точки зрения вызывающего класса `Class1`. Это вопрос управления доступом, рассматривающийся в главе 11, "Классы".

Если `Class2` находится в сборке, отличной от сборки `Class1`, он должен быть открыт (`public`) для `Class1` для доступа к его членам. Если же это одна и та же сборка, `Class2` должен быть объявлен как минимум как `internal`. Классы могут быть только `public`, `protected`, `internal` или `private`.

Аналогично, метод класса `Class2` должен иметь как минимум определенный уровень доступа в каждой из этих ситуаций. Методы добавляют `protected internal` в список спецификаторов доступа класса. Детальнее об этом рассказывается в главе 11, "Классы".

Вы должны получить ответ "да" на оба поставленных вопроса, чтобы класс `Class1` мог вызывать методы `Class2`.

Остальной приведенный здесь материал посвящен вопросам использования и видимости пространств имен.

Как сделать видимыми классы и методы в другом пространстве имен

C# предоставляет два пути сделать элементы в пространстве имен `NamespaceB` видимыми в пространстве имен `NamespaceA`.

- ✓ Применяя *полностью квалифицированные имена* из пространства имен `NamespaceB` при использовании их в пространстве имен `NamespaceA`. Это приводит к коду наподобие приведенного, начинающемуся с имени пространства имен, к которому добавляется имя класса и имя метода:
`System.Console.WriteLine("my string");`
- ✓ Устраняя необходимость в полностью квалифицированных именах в пространстве имен `NamespaceA` посредством *директивы using* для пространства имен `NamespaceB`:
`using NamespaceB;`

Программы в этой книге используют последний способ — директиву `using`. Полностью квалифицированные имена и директивы `using` будут рассмотрены в двух следующих разделах.

Доступ к классам с использованием полностью квалифицированных имен

Пространство имен класса является составной частью его расширенного имени, что приводит к первому способу обеспечения видимости класса из одного пространства имен в другом. Рассмотрим следующий пример, в котором не имеется ни одной директивы `using` для упрощения обращения к классам в других пространствах имен:

```
namespace MathRoutines
{
    class Sort
    {
        public void SomeFunction(){}
    }
}

namespace Paint
{
    public class PaintColor
    {
        public PaintColor(int nRed, int nGreen, int nBlue) {}
        public void Paint() {}
        public static void StaticPaint() {}
    }
}

namespace MathRoutines
{
    public class Test
    {
        static public void Main(string[] args)
        {
            // Создание объекта типа Sort из того же пространства
            // имен, в котором мы находимся, и вызов некоторой
            // функции
            Sort obj = new Sort();
            obj.SomeFunction();
            // Создание объекта в другом пространстве имен —
            // обратите внимание, что пространство имен должно
            // быть явно включено в каждую ссылку на класс
        }
    }
}
```

```

        Paint.PaintColor black = new Paint.PaintColor(0, 0, 0);
        black.Paint();
        Paint.PaintColor.StaticPaint();
    }
}

```

В этом случае классы `Sort` и `Test` содержатся внутри одного и того же пространства имен `MathRoutines`, хотя и объявляются в разных местах файла. Это пространство имен разбито на две части (в данном случае в одном и том же файле).



В обычной ситуации `Sort` и `Test` оказались бы в различных исходных файлах `C#`, которые вы бы собрали в одну программу.

Функция `Test.Main()` может обращаться к классу `Sort` без указания его пространства имен, так как оба эти класса находятся в одном и том же пространстве имен. Однако `Main()` должна указывать пространство имен `Paint` при обращении к `PaintColor`, как это сделано в вызове `Paint.PaintColor.StaticPaint()`. Здесь использовано полностью квалифицированное имя.

Обратите внимание, что вам не требуется принимать специальных мер при обращении к `black.Paint()`, поскольку класс и пространство имен объекта `black` известны.

Директива using

Обращение к классу с использованием полностью квалифицированного имени быстро становится раздражающим. `C#` позволяет избежать излишнего раздражения с помощью ключевого слова `using`. Директива `using` добавляет указанное пространство имен в список пространств имен по умолчанию, в которых `C#` выполняет поиск при разрешении имени класса. Следующий пример компилируется без каких-либо замечаний:

```

namespace Paint
{
    public class PaintColor
    {
        public PaintColor(int nRed, int nGreen, int nBlue) {}
        public void Paint() {}
        public static void StaticPaint() {}
    }
}

namespace MathRoutines
{
    // Добавляем Paint к пространствам имен, в которых
    // выполняется автоматический поиск
    using Paint;
    public class Test
    {
        static public void Main(string[] args)
        {
            // Создаем объект в другом пространстве имен –
            // название пространства имен не требуется включать в
            // имя, поскольку само пространство имен было включено
            // полностью с использованием директивы "using"
            PaintColor black = new PaintColor(0, 0, 0);
        }
    }
}

```

```
black.Paint();
PaintColor.StaticPaint();
}
```

Директива `using` говорит компилятору: "Если ты не в состоянии найти определенный класс в текущем пространстве имен, посмотри еще и в этом пространстве имен, может, ты найдешь его там". Можно указать любое количество пространств имен, но все они должны быть указаны в самом начале программы (либо внутри, либо снаружи блока пространства имен), как описано ранее в разделе "Объявление пространств имен".



Все программы включают директиву `using System, -`. Эта команда дает программе автоматический доступ к функциям, включенным в системную библиотеку, таким как `WriteLine()`.

Использование полностью квалифицированных имен



Приведенная далее демонстрационная программа `NamespaceUse` иллюстрирует влияние пространств имен на видимость и использование директивы `using` и полностью квалифицированных имен, чтобы обеспечить видимость элемента.

```
// NamespaceUse - демонстрирует доступ к объектам с одним и
// тем же именем в разных пространствах имен
using System; // Все пространства имен нуждаются в этой
               // директиве для доступа к классам типа String
               // и Console
namespace MainCode
{
    using LibraryCode1; // Эта директива упрощает MainCode
    public class Class1
    {
        public void AMethod()
        {
            Console.WriteLine("MainCode.Class1.AMethod() ");
        }
    }
    // Функция Main():
    static void Main(string[] args)
    {
        // Создание экземпляра класса, содержащего функцию
        // Main, в данном пространстве имен
        Class1 c1 = new Class1(); // MainCode.Class1
        c1.AMethod();             // Никогда не вызывайте
                                // Main() самостоятельно!

        // Создание экземпляра LibraryCode1.Class1
        // Приведенный далее код создает объект
        // MainCode.Class1, а не тот, что вы хотели, так как
        // нет ни директивы using, ни полностью
        // квалифицированного имени
        Class1 c2 = new Class1(),
        c2.AMethod();
        // Однако полностью квалифицированное имя создает
        // объект требуемого класса. Имя следует
```

```

// квалифицировать даже при использовании директивы
// using, поскольку оба пространства имен содержат
// класс Class1
LibraryCode1.Class1 c3 = new LibraryCode1.Class1();
c3.AMethod(), -
// В то же время создание LibraryCode1.Class2 не
// требует полностью квалифицированного имени,
// поскольку имеется директива using при отсутствии
// коллизии имен; C# может без труда найти Class2
Class2 c4 = new Class2();
c4.AMethod();
// Создание экземпляра LibraryCode2.Class1 требует
// полностью квалифицированного имени, как из-за
// отсутствия директивы using для LibraryCode2, так и
// потому, что оба пространства имен имеют Class1
// Примечание: этот способ работает даже несмотря на
// то, что LibraryCode2.Class1 объявлен как internal,
// а не public, поскольку оба класса находятся в одной
// компилируемой сборке
LibraryCode2.Class1 c5 = new LibraryCode2.Class1();
c5.AMethod();
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
                  "завершения программы...");
Console.Read();
}

namespace LibraryCode1
{
    public class Class1 // Имя дублирует Class1 в другом
    { // пространстве имен
        public void AMethod() // Имя дублировано в другом
        { // пространстве имен
            Console.WriteLine("LibraryCode1.Class1.AMethod()");
        }
    }
    public class Class2 // Имя уникально, его нет в другом
    { // пространстве имен
        public void AMethod()
        {
            Console.WriteLine("LibraryCode1.Class2.AMethod()");
        }
    }
}

namespace LibraryCode2
{
    class Class1 // Нет ключевых слов, описывающих
    { // доступ: по умолчанию доступ –
        public void AMethod() // internal
        {
            Console.WriteLine("LibraryCode2.Class1.AMethod()");
        }
    }
}

```

Данная демонстрационная программа включает три пространства имен: `MainCode`, в которое входит один класс `Class1`, содержащий функцию `Main()` и один дополнительный метод `AMethod()`. Пространство имен `LibraryCode1` содержит два класса: `Class1` дублирует имя `Class1` из пространства имен `MainCode`, класс `Class2` уникален. Пространство имен `LibraryCode2` имеет один класс, также названный `Class1`, имя которого создавало бы коллизию с именем `Class1` в другом пространстве имен, если бы эти имена не были разделены и размещены каждое в своем пространстве имен. Каждый из этих классов имеет метод `AMethod()`.

Функция `Main()` из `MainCode.Class1` пытается создать и использовать `MainCode.Class1` (класс-владелец `MainO`), `LibraryCode1.Class1`, `LibraryCode1.Class2` и `LibraryCode2.Class1`. После создания объектов функция вызывает метод `AMethod()` каждого из них. Каждый метод идентифицирует свое местоположение. Вывод демонстрационной программы на экран выглядит следующим образом:

```
MainCode.Class1.AMethod()
MainCode.Class1.AMethod()
LibraryCode1.Class1.AMethod()
LibraryCode1.Class2.AMethod()
LibraryCode2.Class1.AMethod()
Нажмите <Enter> для завершения программы...
```

Без разделения на различные пространства имен компилятор не может позволить дублирования имен классов в `MainO`. При применении пространств имен исходный текст компилируется, но вы должны использовать либо директиву `using`, либо полностью квалифицированные имена для обращения к объектам в разных пространствах имен. Пространство имен `MainCode` содержит директиву `using` для пространства имен `LibraryCode1`, так что функция `MainO` из `MainCode.Class1` может обратиться к `LibraryCode1.Class2` без полностью квалифицированного имени благодаря наличию упомянутой директивы.

Попытка создать `LibraryCode1.Class1` без использования полностью квалифицированного имени приводит ко второй строке в выводе на экран в рассмотренном примере. Как видите, оказался создан объект класса `MainCode.Class1`, а не класса из пространства имен `LibraryCode1`. Без применения полностью квалифицированного имени компилятор находит `Class1` в пространстве имен `MainCode`. Остальные вызовы работают так, как от них и ожидалось.

Однако использование директивы `using` для пространства имен `LibraryCode1` сослужит функции `Main()` плохую службу при ее желании обратиться к `LibraryCode1.Class1`, так как имя этого класса дублировано в двух пространствах имен. Несмотря на наличие директивы `using`, единственным решением является применение полностью квалифицированного имени для доступа к `LibraryCode1.Class1`.



Последний вызов в `Main()` для создания и использования объекта `Library2.Class1` был бы неверен, если бы пространство имен `Library2` находилось в сборке, отличной от сборки `MainCode`. Причина заключается в том, что спецификаторы доступа у класса `Library2.Class1` не указаны, так что вместо того чтобы быть `public`, он является `internal`. Это поведение по умолчанию для классов без спецификаторов доступа (для методов по умолчанию используется `private`). Повторяясь еще раз — всегда явно указывайте уровень доступа к каждому классу и методу.

После того как вы используете полностью квалифицированные имена или директивы `using`, ваш класс сможет обратиться к чему угодно в другом пространстве имен, если конечно, это не запрещено ему правилами доступа — как, например, к закрытым членам.

Объединение классов в библиотеки

В главе 21, "Использование интерфейса Visual Studio", вы познакомитесь с тем, как создавать проект с несколькими `.CS`-файлами. Даже при разбиении проекта на несколько файлов (обычно по одному классу в файле) один проект равен одной скомпилированной сборке. Библиотеки классов располагаются в файлах с расширением `.DLL` и не являются выполняемыми программами сами по себе. Они служат для поддержки других программ, предоставляя им полезные классы и методы.



Простейшее определение проекта библиотеки классов — это классы, не содержащие функции `Main()`, что отличает библиотеку классов от выполняемой программы.

Visual Studio 2005 позволяет построить либо `.EXE`, либо `.DLL` (либо решение, содержащее их оба). В следующем разделе поясняются основы создания ваших собственных библиотек классов. Не беспокойтесь: рабочая лошадка `C#` вытянет за вас и этот груз.

Менее дорогие версии Visual Studio 2005, такие как Visual `C#` Express, не позволяют собирать библиотеки классов. Однако вам будет показан один трюк, позволяющий обойти данное ограничение.

Создание проекта библиотеки классов

Для формирования нового проекта библиотеки в полной версии Visual Studio 2005 выберите тип проекта **Class Library** в диалоговом окне **New Project**. Затем пропустите следующий раздел главы и переходите к созданию классов, которые будут составлять библиотеку.



Если вы используете Visual `C#` Express, процесс создания нового проекта библиотеки будет немного более сложным. Несмотря на снижение функциональности, которому Microsoft подвергла Visual `C#` Express, если выполнить описанные далее действия, все равно можно сформировать библиотеку классов.

1. При создании нового проекта создавайте **Console Application** (или **Windows Application**).

Если вы не уверены в том, как это делается, вернитесь к главам 1, "Создание вашей первой Windows-программы на `C#`", и 2, "Создание консольного приложения на `C#`". Обычно библиотека классов формируется под видом консольного приложения.

2. В **Solution Explorer** удалите файл `Program.cs`, закройте и сохраните решение.

Поскольку библиотека классов не может содержать функцию `Main()`, вы просто удаляете файл с ней.

3. Запустите Блокнот Windows или другой обычный текстовый редактор и откройте в нем файл `ProjectName.csproj` из вновь созданного вами проекта.

Он выглядит страшновато, но это всего лишь масса информации о программе, записанная с использованием языка XML.

4. Воспользуйтесь командой меню **Edit=>**Find** для того, чтобы найти строку:**

```
<OutputType>Exe</OutputType>
```

Если вы создали проект приложения Windows, а не консольного приложения, найдите строку

```
<OutputType>WinExe</OutputType>
```

Это примерно восьмая строка в файле.

5. Замените в найденной строке **Exe (или **WinExe**) на **Library**:**

```
<OutputType>Library</OutputType>
```

Вот и все!

6. Сохраните файл и заново откройте проект в Visual C# Express.

7. Добавляйте в проект новые классы и работайте. Только убедитесь, что вы не разместили где-то в проекте функцию `Main()`.

Когда вы соберете новый проект, то получите .DLL-файл, а не .EXE-файл.

Создание классов для библиотеки



После того как вы сформировали проект библиотеки классов, вы создаете классы, составляющие эту библиотеку. Приведенный далее пример `ClassLibrary` демонстрирует простую библиотеку классов, которую вы сможете увидеть в действии (в примере показан как исходный текст библиотеки, так и описываемого далее драйвера).

```
// ClassLibrary - простая библиотека классов и ее
// программа-драйвер
// Файл ClassLibrary.cs
using System;
namespace ClassLibrary
{
    public class MyLibrary
    {
        public void LibraryFunction1()
        {
            Console.WriteLine("Это LibraryFunction1()");
        }
        public int LibraryFunction2(int input)
        {
            Console.Write("Это LibraryFunction2(), " +
                          "возвращает {0}, input);", input);
            return input; // Возвращает аргумент
        }
    }
}

// Драйвер - в отдельном проекте
// Файл Program.cs
using System;
```



```

using ClassLibrary; // Вам надо использовать эту библиотеку
                    // в программе
namespace ClassLibraryDriver
{
    class Program
    {
        static void Main(string[] args)
        {
            // Создание объекта библиотеки и использование его
            // методов
            MyLibrary ml = new MyLibrary();
            ml.LibraryFunction1();
            // Вызов статической функции посредством класса
            int nResult = MyLibrary.LibraryFunction2(27);
            Console.WriteLine(nResult.ToString());
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.Read();
        }
    }
}

```

Вывод тестовой программы-драйвера, описанной в следующем разделе, выглядит следующим образом:

```

Это LibraryFunction1()
Это LibraryFunction2(), возвращает 27
27
Нажмите <Enter> для завершения программы...

```

Библиотеки зачастую предоставляют только статические функции. В этом случае не нужно инстанцировать библиотечный объект — можно просто вызвать функцию посредством класса.

Создание проекта драйвера

Сама по себе библиотека классов не делает ничего, так что вам нужна программа-драйвер, небольшая выполняемая программа, которая тестирует библиотеку в процессе разработки путем вызова ее методов. Для создания драйвера для проекта `ClassLibrary` выполните следующие действия.

1. Щелкните правой кнопкой мыши на имени решения в окне **Solution Explorer** проекта `ClassLibrary` и выберите **Add : New Project**.
Тем самым вы добавите проект в то же решение, в котором находится тестируемая библиотека классов.
2. В диалоговом окне **New Project** выберите **Console Application** и назовите его `ClassLibraryDriver` или как вам заблагорассудится.
3. Находясь в диалоговом окне **New Project**, укажите местоположение проекта, щелкнув на кнопке **Browse** рядом с полем **Location**. Перейдите в папку, в которой вы хотите хранить проект драйвера, и щелкните на кнопке **Open**.

Где именно вы расположите проект, зависит от того, как вы хотите организовать ваше решение. Вы можете поместить папку `ClassLibraryDriver` в той же общей папке, что и папку `ClassLibrary`, или вложить ее в папку `ClassLibrary`. Демонстрационная программа `ClassLibrary` в этом разделе придерживается первого подхода.

Выбор местоположения не зависит от того, что вы добавляете новый проект непосредственно в решение `ClassLibrary`. Папки этих двух проектов могут находиться в совершенно разных местах.

4. Щелкните на кнопке `OK` для того, чтобы закрыть диалоговое окно `New Project` и создать проект.

Мастер Visual Studio AppWizard создаст папку проекта вместе с файлами проекта. В окне **Solution Explorer** вы увидите два проекта: `ClassLibrary` и `ClassLibraryDriver`.

5. Щелкните правой кнопкой мыши на проекте `ClassLibraryDriver` и выберите в меню `Set as Startup Project`.

Тем самым вы указываете `C#`, где находится функция `Main()` для данного решения. Она должна находиться в сборке драйвера, но не в сборке библиотеки классов.

6. В файле `Program.cs` проекта `ClassLibraryDriver` добавьте в функцию `Main()` исходный текст наподобие приведенного:

```
MyLibrary myLib = new MyLibrary(); // Или что именно вы
                                   // хотите вызывать

// вызов библиотечной функции
myLib.LibraryFunction1();
// Вызов статической функции
int result = MyLibrary.LibraryFunction2();
```

Другими словами, напишите программу, которая использует классы и методы из библиотеки. Вы уже сталкивались с этим ранее как в данной главе, так и в других главах книги — например, когда вызывали метод `WriteLine()` класса `Console` из библиотеки `.NET Framework`. (`Console` находится в пространстве имен `System` в файле `mscorlib.dll`.)

Код примера библиотеки и драйвера приведен выше — см. листинг демонстрационной программы `ClassLibrary`.

7. Выберите команду меню `Projects Add Reference`.

8. В диалоговом окне `Add Reference` щелкните на вкладке `Projects`. Выберите ваш проект `ClassLibrary` и щелкните на кнопке `OK`.

Вы можете также добавить директиву `using` для пространства имен `ClassLibrary` в файл `Program.cs` проекта `ClassLibraryDriver`, чтобы сэкономить на набираемом тексте.

В любой программе, которую вы напишете в будущем, достаточно включить директиву `using` для пространства имен вашей библиотеки классов и добавить ссылку на `.DLL`-файл, содержащий библиотеку, чтобы иметь возможность использовать библиотечные классы в своей программе. Именно так программы в данной книге применяют классы из библиотеки `.NET Framework`.

Хранение данных в файлах

Консольные программы в настоящей книге в большинстве случаев получают входные данные с консоли и выводят результат работы на консоль. Однако стоит заметить, что вероятность встретить в реальном мире программу, не работающую с файлами, сопоставима с вероятностью встретить в академическом институте рекламу казино в Ницце.

Классы для работы с файлами определены в пространстве имен `System.IO`. Базовым классом для файлового ввода-вывода является класс `FileStream`. Для работы с файлом программист должен его открыть. Команда `open` подготавливает файл к работе и возвращает его дескриптор. Обычно дескриптор — это просто число, которое используется всякий раз при чтении из файла или записи в него.



Асинхронный ввод-вывод: есть ли что-то хуже ожидания?

Обычно программа ожидает завершения ее запроса на ввод-вывод и только затем продолжает выполнение. Вызовите метод `read()`, и в общем случае вы не получите управление назад до тех пор, пока данные из файла не будут считаны. Такой способ работы называется синхронным вводом-выводом.

Классы `C# System.IO` поддерживают также и асинхронный ввод-вывод. При использовании асинхронного ввода-вывода вызов `read()` тут же вернет управление программе, позволяя ей заниматься чем-то еще, пока ее запрос на чтение данных из файла выполняется в фоновом режиме. Программа может проверить флаг выполнения запроса, чтобы узнать, завершено ли его выполнение.

Это чем-то напоминает варианты приготовления гамбургеров. При синхронном изготовлении вы нарезаете мясо и жарите его, после чего нарезаете лук и выполняете все остальные действия по приготовлению гамбургера. При асинхронном приготовлении вы начинаете жарить мясо, и, поглядывая на него, тут же, не дожидаясь готовности мяса, начинаете резать лук и делать все остальное.

Асинхронный ввод-вывод может существенно повысить производительность программы, но при этом вносит дополнительный уровень сложности.

`C#` использует более интуитивный подход. Он связывает каждый файл с объектом класса `FileStream`. Конструктор класса `FileStream` открывает файл, а методы `FileStream` выполняют операции ввода-вывода.

`FileStream` — не единственный класс, который может осуществлять файловый ввод-вывод. Однако он предоставляет хорошую основу для работы с файлами, выполняя 90% всех ваших нужд по работе с ними. Это корневой класс, описываемый в данном разделе. Он достаточно хорош для `C#` и для вас.

`FileStream` — фундаментальный класс. Весь набор его действий — это открытие файла, чтение и запись блока байтов. К счастью, пространство имен `System.IO` содержит, помимо прочего, следующий набор классов, которые обернуты вокруг `FileStream` и предоставляют более простые и богатые возможности.



`BinaryReader/BinaryWriter` — пара потоковых классов, которые содержат методы для чтения и записи каждого из типов-значений: `ReadChar()`, `Write-`

`Char()`, `ReadByte()`, `WriteByte()` и так далее. Эти классы полезны для чтения и записи объекта в бинарном (не читаемом человеком) формате, в противоположность текстовому формату. Для работы с бинарными данными можно использовать массив или коллекцию элементов типа `Byte`.

- ✓ `TextReader/TextWriter`— пара классов для чтения символов (текста). Эти классы предоставляются в двух видах (наборах подклассов): `StringReader/StringWriter` и `StreamReader/StreamWriter`.
- ✓ `StringReader/StringWriter`— простые потоковые классы, которые ограничены чтением и записью строк. Они позволяют рассматривать строку как файл, предоставляя альтернативу доступу к символам строк с помощью записи с использованием квадратных скобок (`[]`), цикла `foreach` или методов класса `String` наподобие `Split()`, `Concatenate()` и `IndexOf()`. Вы считываете и записываете строки почти так же, как и файлы. Этот метод полезен для длинных файлов с сотнями или тысячами символов, которые вы хотите обработать вместе. Методы в этих классах аналогичны методам классов `StreamReader` и `StreamWriter`, описываемым далее.
- ✓ `StreamReader/StreamWriter`— более интеллектуальные классы чтения и записи текста. Например, класс `StreamWriter` имеет метод `WriteLine()`, очень похожий на метод класса `Console`. `StreamReader` имеет соответствующий метод `ReadLine()` и очень удобный метод `ReadToEnd()`, собирающий весь текстовый файл в одну группу и возвращающий считанные символы как строку `string` (которую вы можете затем использовать с классом `StringReader`, циклом `foreach` и тому подобным).

`TextReader/TextWriter` применяются как сами по себе, так и их более удобные подклассы, такие как `StreamReader/StreamWriter`.

В следующем разделе будут рассмотрены демонстрационные программы `FileWrite` и `FileRead`, которые иллюстрируют способы использования классов для текстового ввода-вывода.

Использование *Stream Writer*

Программы генерируют два вида вывода.

- ✓ Некоторые программы пишут блоки данных в виде байтов в чисто бинарном формате. Этот тип вывода полезен для эффективного сохранения объектов (например, файл объектов `Student`, которые сохраняются между запусками программы в файле на диске).
 - ✓ Большинство программ читает и записывает информацию в виде текста, который может читать человек. Классы `StreamWriter` и `StreamReader` являются наиболее гибкими классами для работы с данными в таком виде.
- Данные в удобном для чтения человеком виде ранее назывались ASCII-строками, а сейчас — ANSI-строками. Эти два термина указывают названия организаций по стандартизации, которые определяют соответствующие стандарты. Однако кодировка ANSI работает только с латинским алфавитом

и не имеет кириллических символов, символов иврита, арабского языка или ~~мны~~ не говоря уж о такой экзотике, как корейские, японские или китайские иероглифы. Гораздо более гибким является стандарт Unicode, который включает ANSI-символ как свою начальную часть, а кроме них — массу других алфавитов, включая ~~все~~ перечисленные выше. Unicode имеет несколько форматов, именуемых *кодировками* форматом по умолчанию для C# является UTF8.



Приведенная далее демонстрационная программа FileWrite ~~считывает~~ строки данных с консоли и записывает их в выбранный пользователем файл

```
// FileWrite - запись ввода с консоли в текстовый файл
using System;
using System.IO; // Требуется для работы с файлами
namespace FileWrite
{
    - public class Program
    {
        public static void Main(string[] args)
        {
            // Создание объекта для имени файла — цикл while
            // позволяет пользователю продолжать попытки до тех
            // пор, пока файл не будет успешно открыт
            StreamWriter sw = null;
            string sFileName = "";
            while(true)
            {
                try
                {
                    // Ввод имени файла для вывода (просто Enter для
                    // завершения программы)
                    Console.Write("Введите имя файла "
                        + "(пустое имя для завершения):");
                    sFileName = Console.ReadLine();
                    if (sFileName.Length == 0)
                    {
                        // Имени файла нет — выходим из цикла
                        break;
                    }
                    // Открываем файл для записи; если файл уже
                    // существует, генерируем исключение:
                    // FileMode.CreateNew - для создания файла, если
                    // он еще не существует и генерации исключения при
                    // наличии такого файла; FileMode.Append для
                    // создания нового файла или добавления данных к
                    // существующему файлу; FileMode.Create для
                    // создания нового файла или урезания уже
                    // имеющегося до нулевого размера. Возможные
                    // варианты FileAccess: FileAccess.Read,
                    // FileAccess.Write, FileAccess.ReadWrite
                    FileStream fs = File.Open(sFileName,
                        FileMode.CreateNew,
                        FileAccess.Write);
                    // Генерируем файловый поток с UTF8-символами (по
```

```

// умолчанию второй параметр дает UTF8, так что он
// может быть опущен)
sw = new StreamWriter(fs,
                      System.Text.Encoding.UTF8);
// Считываем по одной строке, выводя каждую из них
// в FileStream для записи
Console.WriteLine("Введите текст " +
                  "(пустую строку для выхода)");

while(true)
{
    // Считываем очередную строку с консоли; если
    // строка пуста, завершаем цикл
    string sInput = Console.ReadLine();
    if (sInput.Length == 0)
    {
        break;
    }
    // Записываем считанную строку в файл вывода
    sw.WriteLine(sInput);
}
// Закрываем созданный файл
sw.Close();
sw = null; // Желательно обнуление ссылочных
           // переменных после использования
}
catch(IOException fe)
{
    // Произошла ошибка при работе с файлом — о ней
    // надо сообщить пользователю вместе с полным
    // именем файла
    string sDir = Directory.GetCurrentDirectory();
    string s = Path.Combine(sDir, sFileName);
    Console.WriteLine("Ошибка с файлом {0}", s);
    // Теперь выводим сообщение об ошибке из
    // исключения
    Console.WriteLine(fe.Message);
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
                      "завершения программы...");
    Console.Read();
}
}

```



Демонстрационная программа FileWrite использует пространства имен System. IO и System. Text. Пространство имен System. IO содержит классы, предназначенные для работы с файлами.



Обратитесь к разделу справочной системы, посвященному пространству имен System. Text. Одним из его более полезных классов является StringBuilder, который предоставляет эффективный подход для работы со сложными строками, состоящими из нескольких частей. Это гораздо более эффективный способ работы, чем использование оператора + для конкатенации большого количества строк.

Программа начинает работу с функции `Main()`, которая включает цикл `while` со держащий `try`-блок. В этом нет ничего необычного для программ, работающих с файлами. (В следующем разделе, где описывается работа с классом `StreamReader`, используется немного другой подход, дающий те же результаты.)



Все функции ввода-вывода вставлены в `try`-блок с перехватом, в котором генерируется соответствующее сообщение об ошибке. В этом надо быть очень аккуратным, так как файловый ввод-вывод является источником множества ошибок, таких как отсутствующие файлы и каталоги, неверные пути и тому подобное.

Цикл `while` служит двум следующим целям.

- ✓ Он позволяет программе вернуться и повторить попытку в случае, если произошла ошибка ввода-вывода. Например, если демонстрационная программа не может найти файл, который планирует читать пользователь, она может запросить у него имя файла еще раз, а не просто оставить его с сообщением об ошибке.
- ✓ Команда `break` в программе переносит вас за `try`-блок, тем самым предоставляет удобный механизм для выхода из функции или программы. Не забывайте о том, что `break` работает только в пределах цикла, в котором вызвана эта команда.

Демонстрационная программа `FileWrite` считывает имя создаваемого файла с клавиатуры. Программа прекращает работу путем выхода из цикла с помощью команды `break`, если пользователь вводит пустое имя файла. Ключ к программе заключается в следующих строках:

```
FileStream fs = File.Open(sFileName, FileMode.CreateNew,
                          FileAccess.Write);

// ...

sw = new StreamWriter(fs, System.Text.Encoding.UTF8);
```

В первой строке программа создает объект `FileStream`, который представляет файл, записываемый на диск. Конструктор `FileStream` использует следующие три аргумента.

- ✓ **Имя файла:** это просто имя файла, который следует открыть. Простое имя файла, например `filename.txt`, предполагает, что файл находится в текущем каталоге (для демонстрационной программы `FileWrite` это подкаталог `bin\Debug` каталога проекта; словом, это каталог, в котором находится сам `.EXE`-файл). Имя файла, начинающееся с обратной косой черты, например `directory\filename.txt`, рассматривается как полный путь на локальной машине. Имя файла, начинающееся с двух обратных косых черт (например, `\\machine\directory\filename.txt`), указывает файл, расположенный на другой машине в вашей сети. Кодировка файла — существенно более сложный вопрос, выходящий за рамки данной книги.
- ✓ **Режим работы с файлом:** этот аргумент определяет, что вы намерены делать с файлом. Основными режимами работы с файлом для записи являются создание (`CreateNew`), добавление к файлу (`Append`) и перезапись (`Create`). `CreateNew` создает новый файл, но генерирует исключение `IOException`, если такой файл уже существует. Простой режим `Create` создает файл, если он отсутствует, но если он есть, то просто перезаписывает его. И наконец, `Append` создает файл,



если он не существует, но если он имеется, открывает его для дописывания информации в конец файла.

Тип доступа: файл может быть открыт для чтения, записи или для обеих операций.



Класс `FileStream` имеет ряд конструкторов, у каждого из которых один или оба аргумента, отвечающие за режим открытия и тип доступа, имеют значения по умолчанию. Однако, по моему скромному мнению, вы должны указывать эти аргументы явно, поскольку это существенно повышает понятность программы. Поверьте, это хороший совет — значения по умолчанию могут быть удобны для программиста, но не для того, кто будет читать его код.

В следующей строке программа "оборачивает" вновь открытый файловый объект `FileStream` в объект `StreamWriter`. Класс `StreamWriter` служит оберткой для объекта `FileStream`, которая предоставляет набор методов для работы с текстом.



Такой вид "оборачивания" одного класса вокруг другого представляет собой полезный программный шаблон проектирования — `StreamWriter` "оборачивается" (содержит ссылку) вокруг другого класса `FileStream` и расширяет интерфейс `FileStream`, добавляя некоторые мелкие удобства. Методы `StreamWriter` вызывают методы внутреннего объекта `FileStream`. Это — рассматривавшееся в главе 12, "Наследование", отношение СОДЕРЖИТ.

Первый аргумент конструктора `StreamWriter` — объект `FileStream`. Второй аргумент указывает используемую кодировку. Кодировка по умолчанию — `UTF8`.

Вы не должны указывать кодировку при чтении файла. Дело в том, что `StreamWriter` записывает тип применяемой кодировки в первых трех байтах файла. `StreamReader` считывает эти три байта при открытии файла и определяет тип используемой кодировки. Скрытие такого рода деталей представляет собой одно из преимуществ хорошей библиотеки.

Затем программа `FileWrite` начинает чтение строк, вводимых с консоли. Программа завершает работу при считывании пустой строки, но до этого она собирает все считанные строки и записывает их, используя метод `WriteLine()` класса `StreamWriter`.



Подобие `StreamWriter.WriteLine()` и `Console.WriteLine()` — больше, чем простое совпадение.

И наконец, файл закрывается с помощью вызова `sw.Close()`.



Обратите внимание, что программа обнуляет ссылку `sw` по закрытии файла. Файловый объект становится бесполезен после того, как файл закрыт. Правила хорошего тона требуют обнулять ссылки после того, как они становятся недействительными, так, чтобы обращений к ним больше не было (если вы попытаетесь это сделать, то будет сгенерировано исключение).

Блок `catch` напоминает футбольного вратаря: он стоит здесь для того, чтобы ловить все исключения, которые могут быть сгенерированы в программе. Он выводит сообщение об ошибке, включая имя вызвавшего ее файла. Однако выводится не просто имя файла, а его полное имя, включая путь к нему. Это делается посредством класса `Directory`, который позволяет получить текущий каталог и добавить его перед введенным именем файла с использованием метода `Path.Combine()` (`Path` — класс, разрабо-

ный для работы с информацией о путях, а `Directory` предоставляет свойства и методы для работы с каталогами).

Путь — это полное имя каталога. Например, если имя файла — `c:\user\directory\!text.txt`, то его путь — `c:\user\directory`.



Метод `Combine()` достаточно интеллектuaлен, чтобы разобраться, что для файла наподобие `c:\test.txt` `Path` не является текущим каталогом. `Path.Combine()` представляет также наиболее безопасный путь, гарантирующий корректное объединение двух частей пути, включая символ-разделитель (`\`) между ними. (В Windows символ-разделитель пути — `\`. Вы можете получить корректный разделитель для операционной системы, под управлением которой запущена программа, с помощью `Path.DirectorySeparatorChar`. Библиотека .NET Framework изобилует такого рода возможностями, существенно облегчая программистам на C# написание программ, которые должны работать под управлением нескольких операционных систем.)

Достигнув конца цикла `while` — либо после выполнения `try`-блока, либо после блока `catch`, — программа возвращается к началу цикла и позволяет пользователю написать другой файл.

Вот как выглядит пример выполнения демонстрационной программы (пользовательский ввод выделен полужирным шрифтом).

Введите имя файла (пустое имя для завершения): **TestFile1.txt**

Введите текст (пустую строку для выхода)

Это какой-то текст

и еще

и еще раз...

Введите имя файла (пустое имя для завершения): **TestFile1.txt**

Ошибка с файлом C:\C#Programs\FileWrite\bin\Debug\TestFile1.txt
The file already exists.

Введите имя файла (пустое имя для завершения): **TestFile2.txt**

Введите текст (пустую строку для выхода)

**Я ошибся - мне надо было ввести
имя файла TestFile2.**

Введите имя файла (пустое имя для завершения):

Нажмите <Enter> для завершения программы...

Все отлично работает, пока некоторый текст вводится в файл `TestFile1.txt`. Но при попытке открыть файл `TestFile1.txt` заново программа выводит сообщение `The file already exists` (файл уже существует). Обратите внимание на полный путь к файлу, выводимый вместе с сообщением об ошибке. Если исправить ошибку и ввести имя `TestFile2.txt`, все продолжает отлично работать.

Повышение скорости чтения с использованием `StreamReader`



Запись файла — дело стоящее, но совершенно бесполезное, если вы не можете позже прочесть записанное. Приведенная далее демонстрационная программа считывает текстовый файл, например, созданный демонстрационной программой `FileWrite` или с помощью Блокнота.

```

// FileRead - читает текстовый файл и выводит на консоль его
// содержимое
using System;
using System.IO;
namespace FileRead
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Нам нужен объект для чтения файла
            StreamReader sr;
            string sFileName = " ";
            // Пытаемся получить корректное имя файла до тех пор,
            // пока наконец его не получим (для выхода из
            // программы надо использовать комбинацию клавиш
            // <Ctrl+C>)
            while(true)
            {
                try
                {
                    // Ввод имени файла
                    Console.WriteLine("Введите имя текстового файла:");
                    sFileName = Console.ReadLine();
                    // Если пользователь ничего не ввел, генерируем
                    // исключение для указания, что такой ввод
                    // неприемлем
                    if (sFileName.Length == 0)
                    {
                        throw new
                            IOException("Введено пустое имя файла");
                    }
                    // Открываем файловый поток для чтения; если файл
                    // не существует - не создаем его
                    FileStream fs = File.Open(sFileName,
                                                FileMode.Open,
                                                FileAccess.Read);
                    // Преобразуем в StreamReader - этот класс
                    // использует первые три байта файла для
                    // определения использованной кодировки (но не для
                    // языка)
                    sr = new StreamReader(fs, true);
                    break;
                }
                // Сообщение об ошибке с указанием имени файла
                catch(IOException fe)
                {
                    Console.WriteLine("{0}\n\n", fe.Message);
                }
            }
            // Чтение содержимого файла
            Console.WriteLine("\nСодержимое файла:");
            try

```

```

    {
        // Чтение по одной строке
        while(true)
        {
            // Считывание строки
            string slnput = sr.ReadLine() ;
            // Выход, когда больше считать строку не удастся
            if (slnput == null)
            {
                break;
            }
            // Вывод считанного на консоль
            Console.WriteLine(slnput);
        }
    }
    catch(IOException fe)

        // перехватывает все ошибки и сообщает о них
        Console.Write(fe.Message);

    // Закрываем файл (игнорируя возможные ошибки)
    try

        sr.Close();

    catch {}
    sr = null;
    // Ожидаем подтверждения пользователя
    Console.WriteLine("Нажмите <Enter> для " +
        "завершения программы... ");
    Console.Read();
}
}

```

Вспомним, что текущим каталогом, использовавшимся демонстрационной программой FileRead, был подкаталог `\bin\Debug` в каталоге проекта FileRead (но не каталог `\bin\Debug` в каталоге проекта FileWrite). Перед тем как вы запустите программу FileRead, поместите текстовый файл в подкаталог `\bin\Debug` каталогом проекта и запомните имя этого файла, чтобы впоследствии вы могли открыть его. Для этого вполне подойдет копия файла `TestFile1.txt`, созданного демонстрационной программой FileWrite.

Демонстрационная программа FileRead применяет другой подход к именам файлов. В ней пользователь считывает один и только один файл. Пользователь должен ввести корректное имя файла, который будет считан программой. После того как программа прочтет файл, она завершает свою работу. Если пользователь хочет прочесть второй файл, он должен заново запустить программу.

Одно из ограничений подхода, использованного в демонстрационной программе FileRead, заключается в том, что попытки получить имя файла от пользователя продолжают до бесконечности. Если пользователь ошибается, он должен продолжать свои попытки. Единственный выход из программы без ввода корректного имени — воспользоваться комбинацией клавиш `<Ctrl+C>` либо щелкнуть на кнопке закрытия консольного окна.

Программа начинается с цикла `while`, как и демонстрационная программа `FileWrite`. В цикле программа получает имя файла для чтения от пользователя. Если имя файла пустое, программа генерирует свое собственное сообщение об ошибке: Введено пустое имя файла. Если имя файла не пустое, оно используется для открытия объекта `FileStream` в режиме для чтения. Вызов `File.Open()` работает так же, как и в демонстрационной программе `FileWrite`.

- ✓ Первый аргумент — это имя файла.
- ✓ Второй аргумент — режим открытия файла. Режим `FileMode.Open` гласит: "Открыть файл, если он существует, и сгенерировать исключение, если его нет". Другой вариант — `OpenNew`, который создает файл нулевой длины в случае отсутствия последнего. Лично я никогда не использовал этот режим (кому надо читать из пустого файла?), но мало ли — может, имеются люди, умеющие напиться из пустого стакана?
- ✓ Последний аргумент указывает на желание читать из объекта `FileStream`. Другие возможные варианты — `Write` и `ReadWrite`. (Кажется странным открывать файл в программе `FileRead` с использованием режима `Write`, не правда ли?)

Полученный в результате объект `fs` класса `FileStream` оборачивается в объект `sr` класса `StreamReader` — для предоставления удобного доступа к текстовому файлу.

Весь раздел открытия файла размещен в `try`-блоке в цикле `while`. Этот `try`-блок предназначен исключительно для открытия файла. Если в процессе открытия файла происходит ошибка, генерируется и перехватывается исключение, выводится сообщение об ошибке, и программа возвращается к запросу имени файла. Однако если процесс завершается благополучно, то команда `break` передает управление за пределы цикла в раздел чтения файла.



Демонстрационные программы `FileRead` и `FileWrite` представляют два разных способа обработки исключений. Вы можете поместить всю программу в один `try`-блок, как в демонстрационной программе `FileWrite`, либо поместить раздел открытия файла в собственный `try`-блок. Обычно использовать отдельные `try`-блоки проще, а кроме того, это позволяет генерировать более точные сообщения об ошибках.

Когда процесс открытия файла завершен, программа `FileRead` считывает строку текста из файла с помощью вызова `ReadLine()`. Программа выводит эту строку на консоль посредством хорошо знакомого вызова `Console.WriteLine()`, после чего возвращается к считыванию очередной строки текста. Когда программа достигает конца файла, вызов `ReadLine()` возвращает значение `null`. Когда это происходит, программа прекращает цикл чтения, закрывает объект и завершает работу.

Обратите внимание, как вызов `Close()` обернут в свой собственный небольшой `try`-блок. Блок `catch` без аргументов перехватывает все классы исключений (что эквивалентно `catch (Exception)`). Любая ошибка, сгенерированная в `Close()`, перехватывается и игнорируется. Этот блок `catch` предназначен для того, чтобы предотвратить распространение исключения и завершение программы. Ошибка игнорируется, поскольку программа ничего не может поделать со сбоем в вызове `Close()`, тем более что через пару строк программа все равно завершается. Вы могли бы поместить вызов `Close()` в блок

finally после блока catch для того, чтобы гарантировать его выполнение в любом случае, но в данной ситуации это излишне.



Пустой catch включен исключительно в демонстрационных целях. Представление вызову собственного try-блока с перехватом всего, что можно, предотвращает завершение программы из-за несущественных ошибок. Оли этот метод можно использовать только если ошибка действительно не критична и не вредит работе программы.

Вот как выглядит пример вывода программы:

```
Введите имя текстового файла:TestFilex.txt
Could not find file "C:\C#Programs\FileRead\TestFilex.txt".
```

```
Введите имя текстового файла:TestFile1.txt
```

```
Содержимое файла:
Это какой-то текст
И еще
И еще раз...
Нажмите <Enter> для завершения программы...
```

Как видите, это тот же текст, который был записан в файл TestFile1.txt, созданный в демонстрационной программе FileWrite (ведь вы не забыли скопировать его! каталог \FileRead\bin\debug?).

Работа с коллекциями

В этой главе...

- У Каталог как коллекция
- > Реализация коллекции `LinkedList`
- > Итеративный обход коллекции `LinkedList`
- > Реализация индексирования для упрощения доступа к объектам коллекций
- > Упрощение циклического обхода коллекции с помощью нового блока итератора `C#`



Файл представляет собой один из типов коллекций данных, но существуют и другие коллекции. Например, каталог можно рассматривать как коллекцию файлов. Кроме того, `C#` предоставляет множество типов контейнеров в оперативной памяти.

Эта глава построена на основе главы 15, "Обобщенное программирование", и материала, посвященного файлам, из главы 19, "Работа с файлами и библиотеками". Основным вопросом, рассматриваемым в данной главе — проход (итерирование) по коллекциям разного вида, от каталогов до массивов и списков всех видов. Вы также узнаете, как написать собственный класс коллекции (более фундаментальный, чем рассматривавшийся в главе 15, "Обобщенное программирование", пример очереди с приоритетами) — связанный список.

Обход каталога файлов

Чтение и запись — вот основное, что необходимо знать для работы с файлами. Именно этим и занимались демонстрационные программы `FileRead` и `FileWrite` из главы 19, "Работа с файлами и библиотеками". Однако в ряде случаев вам просто нужно просканировать каталог файлов в поисках чего-то.



Приведенная далее демонстрационная программа `LoopThroughFiles` просматривает все файлы в данном каталоге, считывая каждый файл и выводя его содержимое на консоль в шестнадцатеричном формате. (Это демонстрирует вам, что файл можно выводить не только в виде строк. Что такое шестнадцатеричный формат — вы узнаете немного позже.)



Если запустить эту программу в каталоге с большим количеством файлов, то вывод шестнадцатеричного дампа может занять длительное время. Много времени требует и вывод дампа большого файла. Либо испытайте программу на каталоге покороче, либо, когда вам надоест, просто нажмите клавиши `<Ctrl+C>`. Эта команда должна прервать выполнение программы в любом консольном окне.

```

// LoopThroughFiles - проход по всем файлам, содержащимся в
// каталоге. Здесь выполняется вывод шестнадцатеричного
// дампа файла на экран, но могут выполняться и любые иные
// действия
using System;
using System.IO;
namespace LoopThroughFiles
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Если каталог не указан...
            string sDirectoryName;
            if (args.Length == 0)
            {
                // ...получаем имя текущего каталога...
                sDirectoryName = Directory.GetCurrentDirectory();
            }
            else
            {
                // ...в противном случае считаем, что первый
                // переданный программе аргумент и есть имя
                // используемого каталога
                sDirectoryName = args[0];
            }
            Console.WriteLine(sDirectoryName);
            // Получение списка всех файлов каталога
            FileInfo[] files = GetFileList(sDirectoryName);
            // Проход по всем файлам списка с выводом
            // шестнадцатеричного дампа каждого файла
            foreach(FileInfo file in files)
            {
                // Вывод имени файла
                Console.WriteLine("\nДамп файла {0}:",
                                file.FullName);
                // Вывод содержимого файла
                DumpHex(file);
                // Ожидание подтверждения пользователя
                Console.WriteLine("\nНажмите <Enter> для вывода " +
                                "следующего файла");
                Console.ReadLine();
            }
            // Файлы закончились!
            Console.WriteLine("\nБольше файлов нет");
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
            Console.ReadLine();
        }
        // GetFileList - получение списка всех файлов в
        // указанном каталоге
        public static FileInfo[]
            GetFileList(string sDirectoryName)

```

```

{
    // Начинаем с пустого списка
    FileInfo[] files = new FileInfo[0];
    try
    {
        // Получаем информацию о каталоге
        DirectoryInfo di =
            new DirectoryInfo(sDirectoryName);
        // В ней имеется список файлов
        files = di.GetFiles();
    }
    catch(Exception e)
    {
        Console.WriteLine("Каталог \"" + sDirectoryName +
            "\" неверен"), -
        Console.WriteLine(e.Message);
    }
    return files;
}

// DumpHex - для заданного файла выводит его содержимое
// на консоль
public static void DumpHex(FileInfo file)
{
    // Открываем файл
    FileStream fs;
    try
    {
        string sFileName = file.FullName;
        fs = new FileStream(sFileName, FileMode.Open,
            FileAccess.Read);

        // В действительности FileInfo предоставляет метод
        // file.OpenRead(), который открывает FileStream за
        // вас, если вы слишком ленивы
    }
    catch(Exception e)
    {
        Console.WriteLine("\nНе могу читать \"" +
            file.FullName + "\"");
        Console.WriteLine(e.Message);
        return;
    }
    // Построчный проход по содержимому файла
    for(int nLine = 1; true; nLine++)
    {
        // Считываем очередные 10 байтов (это все, что можно
        // разместить в одной строке); выходим, когда все
        // байты считаны
        byte[] buffer = new byte[10];
        int numBytes = fs.Read(buffer, 0, buffer.Length);
        if (numBytes == 0)
        {
            return;
        }
    }
}

```



```

        // номером строки
        Console.WriteLine("{0:D3} - ", nLine);
        DumpBuffer(buffer, numBytes), -
        // После каждых 20 строк останавливаемся, так как
        // прокрутка консольного окна отсутствует
        if ((nLine % 20) == 0)
        {
            Console.WriteLine("Нажмите <Enter> для вывода " +
                               "очередных 20 строк");
            Console.ReadLine();
        }
    }
}
// DumpBuffer - вывод буфера символов в виде единой
// строки в шестнадцатеричном формате
public static void DumpBuffer(byte[] buffer,
                               int numBytes)

    for(int index = 0; index < numBytes; index++)

        byte b = buffer[index] ,•
        Console.WriteLine("{0:X2}, ", b) ,-

    Console.WriteLine();
}
}
}
}
}

```

В командной строке пользователь указывает каталог, применяемый в качестве аргумента программы. Приведенная далее команда выведет шестнадцатеричный дамп каждого файла из временного каталога (как текстовых, так и бинарных файлов):

```
loopthroughfiles c:\randy\temp
```

Если не ввести имя файла, программа по умолчанию использует текущий каталог! (Шестнадцатеричный дамп выводит все числа в шестнадцатеричной системе счисления — см. врезку "Шестнадцатеричные числа".)

Шестнадцатеричные числа

Как и бинарные числа (0 и 1), шестнадцатеричные числа также очень важны в компьютерном программировании. В шестнадцатеричной системе счисления цифрами являются обычные десятичные цифры 0-9 и буквы A, B, C, D, E, F — где A = 10, B = 11,

F = 15. Для иллюстрации (префикс 0x указывает на шестнадцатеричность выводимого числа): 0xD = 13. 0x10 = 16: $1 \cdot 16 + 0 \cdot 1$. 0x2A = 42: $2 \cdot 16 + A \cdot 1$ (здесь $A \cdot 1 = 10 \cdot 1$). Буквы могут быть как строчными, так и прописными: F означает то же, что ит. Эти числа выглядят причудливо, но они очень полезны, в особенности при отладке или работе с аппаратной частью или содержимым памяти.



Демонстрационные программы FileRead и FileWrite считывали имена файлов с консоли, в то время как в этой программе имя файла передается в командной строке. Поверьте, вас никто не пытается запутать, а всего лишь предлагаются различные варианты решения одной и той же задачи.

Первая строка демонстрационной программы `LoopThroughFiles` определяет наличие аргумента в командной строке. Если список аргументов пуст (`args.Length` равен 0), программа вызывает `Directory.GetCurrentDirectory()`. Если программа запущена из Visual Studio, а не из командной строки, то по умолчанию будет использоваться подкаталог `bin\Debug` в каталоге проекта `LoopThroughFiles`.



Класс `Directory` предоставляет пользователю набор методов для работы с каталогами, а класс `FileInfo` — методы для перемещения, копирования и удаления файлов.

Затем программа получает список всех файлов в указанном каталоге посредством вызова `GetFileList()`. Эта функция возвращает массив объектов `FileInfo`. Каждый объект `FileInfo` содержит информацию о файле — например, имя файла (как полное имя с путем, `FullName`, так и без пути — `Name`), дату его создания и время последнего изменения. Функция `Main()` проходит по всему списку файлов с помощью цикла `foreach`. Она выводит имя каждого файла и передает его функции `DumpHex()` для вывода содержимого на консоль.

Пауза в конце каждого цикла позволяет программисту просмотреть выведенную `DumpHex()` информацию.

Функция `GetFileList()` начинает работу с создания пустого списка `FileInfo`, который будет возвращен в случае ошибки.

Этот прием стоит запомнить и использовать при работе с функциями `Get...List()`: если происходит ошибка, вывести сообщение о ней и вернуть пустой список.



Будьте внимательны при возврате ссылок. Например, не возвращайте ссылки ни на одну из внутренних очередей в классе `PriorityQueue` в главе 15, "Обобщенное программирование", если не хотите намеренно пригласить пользователей мешать нормальной работе класса (работой не посредством методов класса, а напрямую с очередями). Но `GetFileList()` не дает вам доступа к внутренностям одного из ваших классов, так что в данном случае все в порядке.

Затем функция `GetFileList()` создает объект `DirectoryInfo`. Как и гласит его имя, объект `DirectoryInfo` содержит тот же вид информации о каталоге, что и объект `FileInfo` о файле. Однако у объекта `DirectoryInfo` есть доступ к одной вещи, к которой нет доступа у объекта `FileInfo`, а именно к списку файлов каталога в виде массива `FileInfo`.

Как обычно, функция `GetFileList()` помещает код, работающий с файлами и каталогами, в большой `try`-блок. Конструкция `catch` в конце функции перехватывает все генерируемые ошибки и выводит имя каталога (которое, вероятно, введено неверно, т.е. такого каталога не существует).

Функция `DumpHex()` несколько сложнее из-за трудностей в форматировании вывода.

Функция `DumpHex()` начинает работу с открытия файла. Объект `FileInfo` содержит информацию о файле, но не открывает его. Функция `DumpHex()` получает полное имя файла, включая путь. Затем она открывает `FileStream` в режиме только для чтения с использованием этого имени. Блок `catch` перехватывает исключение, если `FileStream` не в состоянии прочесть файл по той или иной причине.

Затем `DumpHex()` считывает файл по 10 байт за раз и выводит их в одну строку в шестнадцатеричном формате. После вывода каждых 20 строк программа приостанавливает работу в ожидании нажатия пользователем клавиши `<Enter>`.



По вертикали консольное окно по умолчанию имеет 25 строк (правда, пользователь может изменить эту настройку, добавив или убрав строки). Это означает, что вы должны делать паузу после вывода каждых 20 строк или около того. В противном случае данные будут быстро выведены на экран и пользователь не сможет их прочесть.

Операция деления по модулю (%) возвращает остаток после деления. То есть выражение $(nLine \% 20) == 0$ истинно при значениях $nLine$, равных 20, 40, 60, 80.... Словом, идея понятна. Это важный метод, применимый для всех видов циклов, когда нужно выполнять некоторую операцию только с определенной частотой.

Функция `DumpBuffer()` выводит каждый член массива байтов с использованием управляющего элемента форматирования `x2`. `x2` хотя и звучит как название какого-то секретного военного эксперимента, означает всего лишь "вывести число в виде двух шестнадцатеричных цифр" (см. главу 9, "Работа со строками в C#").

Диапазон значений `byte` — от 0 до 255, или `0xFF` — т.е. двух шестнадцатеричных цифр для вывода одного байта достаточно.

Вот как выглядят первые 20 строк при выводе содержимого файла `output.txt`. Даже его собственная мать не узнала бы его в таком виде...

Дамп файла `C:\C#ProgramsVi\holdtank\Test2\bin\output.txt`:

```
001 - 53, 74, 72, 65, 61, 6D, 20, 28, 70, 72,
002 - 6F, 74, 65, 63, 74, 65, 64, 29, 0D, 0A,
003 - 20, 20, 46, 69, 6C, 65, 53, 74, 72, 65,
004 - 61, 6D, 28, 73, 74, 72, 69, 6E, 67, 2C,
005 - 20, 46, 69, 6C, 65, 4D, 6F, 64, 65, 2C,
006 - 20, 46, 69, 6C, 65, 41, 63, 63, 65, 73,
007 - 73, 29, 0D, 0A, 20, 20, 4D, 65, 6D, 6F,
008 - 72, 79, 53, 74, 72, 65, 61, 6D, 28, 29,
009 - 3B, 0D, 0A, 20, 20, 4E, 65, 74, 77, 6F,
010 - 72, 6B, 53, 74, 72, 65, 61, 6D, 0D, 0A,
011 - 20, 20, 42, 75, 66, 66, 65, 72, 53, 74,
012 - 72, 65, 61, 6D, 20, 2D, 20, 62, 75, 66,
013 - 66, 65, 72, 73, 20, 61, 6E, 20, 65, 78,
014 - 69, 73, 74, 69, 6E, 67, 20, 73, 74, 72,
015 - 65, 61, 6D, 20, 6F, 62, 6A, 65, 63, 74,
016 - 0D, 0A, 0D, 0A, 42, 69, 6E, 61, 72, 79,
017 - 52, 65, 61, 64, 65, 72, 20, 2D, 20, 72,
018 - 65, 61, 64, 20, 69, 6E, 20, 76, 61, 72,
019 - 69, 6F, 75, 73, 20, 74, 79, 70, 65, 73,
020 - 20, 28, 43, 68, 61, 72, 2C, 20, 49, 6E,
```

Нажмите <Enter> для вывода очередных 20 строк



Можно восстановить файл в виде строк из вывода в шестнадцатеричном формате. `0xb1` — числовой эквивалент символа `a`. Буквы расположены в алфавитном порядке, так что `0xb5` должно быть символом `e`. `0x20` — пробел. Приведенная здесь первая строка выглядит при обычной записи в виде строк как `"Stream (pr"`. Интригующе, не правда ли? Полностью коды букв вы можете найти в разделе "ASCII, table of codes" справочной системы.

Эти коды корректны и при использовании набора символов Unicode, который применяется C# по умолчанию (побольше о Unicode вы можете узнать, прогулявшись в Интернете в поисках "Unicode characters").

Вот как выглядит вывод программы, если указать неверное имя каталога x:

```
Каталог "x" неверен
Could not find a part of the path
"C:\C#Programs\LoopThroughFiles\bin\Debug\x".

Больше файлов нет
Нажмите <Enter> для завершения программы...
Не впечатляет?...
```

Написание собственного класса коллекции: связанный список

Я из тех учителей, которые по старинке считают, что сначала следует освоить таблицу умножения, а уж потом давать ученику калькулятор. Так что сейчас вы пройдете сквозь дебри создания собственной коллекции, перед тем как познакомиться со встроенными коллекциями, о которых упоминалось в главе 15, "Обобщенное программирование". Здесь будут рассмотрены все "болты и гайки", из которых состоит класс коллекции, и как все они объединяются в одно целое.

Одним из наиболее распространенных видов контейнеров после массива является связанный список, каждый объект которого указывает на предыдущий и последующий элементы списка, т.е. объекты, составляющие список, оказываются соединены в цепочку. Вы используете ссылки на объекты для объединения отдельных узлов в цепь. В каждом таком узле содержатся дополнительные данные, указывающие на следующий узел в цепи. Отдельная переменная, обычно называемая ссылкой на голову списка, указывает на первый объект в списке, в то время как хвост списка указывает на его последний элемент.



Односвязные списки содержат узлы, связанные только с узлами, следующими за ними. По такому списку можно пройти только в одном направлении, следуя связям между узлами. Дважды связанный список содержит узлы, которые указывают как на последующий, так и на предыдущий узлы. По таким спискам можно проходить в обоих направлениях.

Связанный список по сравнению с массивом обладает рядом преимуществ и недостатков.

- ✓ Можно легко вставить элемент в середину списка. Для выполнения вставки программа должна изменить только значения четырех ссылок (в дважды связанном списке), но это простые, быстро вносимые изменения.
- ✓ Точно так же можно легко удалить элемент из связанного списка.
- ✓ Связанный список при необходимости может расти или уменьшаться. Программа начинает работу с пустым связанным списком, а затем по мере необходимости добавляет и удаляет элементы.
- ✓ Доступ к элементу, располагающемуся следующим, быстр и прост, однако элементы связанного списка не индексированы. Таким образом, обращение к определенному элементу списка может потребовать проход по всему списку, что весьма неэффективно.

Связанные списки идеально подходят для хранения последовательностей данных, особенно если программа не знает заранее их точное количество (тем не менее следует серьезно подумать о возможном применении обобщенного класса `List<T>`, который был описан в главе 15, "Обобщенное программирование". Если вам нужен именно связанный список, можно воспользоваться встроенным связанным списком из C# 2.0, а не тем, который разрабатывается в данном разделе. Обратитесь к справочной системе за информацией о пространстве имен `System.Collections.Generic`).

Другие пространства имен коллекций, которыми вы можете захотеть воспользоваться — `System.Collections` и `System.Collections.Specialized`. Поищите информацию о них в справочной системе, но в первую очередь следует искать подходящую коллекцию именно в пространстве имен `System.Collections.Generic`.

Пример связанного списка



Приведенная далее демонстрационная программа иллюстрирует создание и использование связанного списка.

```
// LinkedListContainer - демонстрация "самодельного"
// связанного списка. Этот контейнер реализует интерфейс
// IEnumerable для поддержки таких операторов, как foreach.
// Этот пример включает также итератор, который реализует
// интерфейс IEnumerator
using System;
using System.Collections;

namespace LinkedListContainer
{
    // LLNode - каждый LLNode образует узел списка. Каждый
    // узел LLNode содержит ссылку на целевые данные,
    // встроенные в список
    public class LLNode
    {
        // Это данные, которые хранятся в узле списка
        internal object linkedData = null;
        // Указатели на следующий и предыдущий узлы в списке
        internal LLNode forward = null; // Следующий узел
        internal LLNode backward = null; // Предыдущий узел

        internal LLNode(object linkedData)
        {
            this.linkedData = linkedData;
        }

        // Получение данных, хранящихся в узле
        public object Data
        {
            {
                get
                {
                    return linkedData;
                }
            }
        }
    }
}
```

```

}

// LinkedList - реализация дважды связанного списка
public class LinkedList : IEnumerable
{
    // Концы связанного списка. Спецификатор internal
    // позволяет итераторам обращаться к ним непосредственно
    internal LLNode head = null; // Начало списка
    internal LLNode tail = null; // Конец списка

    public IEnumerator GetEnumerator()
    {
        return new LinkedListIterator(this);
    }

    // AddObject - добавление объекта в конец списка
    public LLNode AddObject(object objectToAdd)
    {
        return AddObject(tail, objectToAdd);
    }

    // AddObject- добавление объекта в список
    public LLNode AddObject(LLNode previousNode,
                            object objectToAdd)
    {
        // Создание нового узла с добавляемым объектом
        LLNode newNode = new LLNode(objectToAdd);

        // Начнем с простейшего случая - пустого списка.
        if (head == null && tail == null)
        {
            // ...теперь в нем один элемент
            head = newNode;
            tail = newNode;
            return newNode;
        }

        // Добавляем ли мы новый узел в середину списка?
        if (previousNode != null &&
            previousNode.forward != null)
        {
            // Просто изменяем указатели
            LLNode nextNode = previousNode.forward;

            // Указатель на следующий узел
            newNode.forward = nextNode;
            previousNode.forward = newNode;

            // Указатель на предыдущий узел
            nextNode.backward = newNode;
            newNode.backward = previousNode;

            return newNode;
        }

        // Добавление в начало списка?
        if (previousNode == null)

```

```

    {
        // Делаем его головой списка
        LLNode nextNode = head;
        newNode.forward = nextNode;
        nextNode.backward = newNode;
        head = newNode;
        return newNode;
    }
    // Добавление в конец списка
    newNode.backward = previousNode;
    previousNode.forward = newNode;
    tail = newNode;
    return newNode;
}

// RemoveObject - удаление объекта из списка
public void RemoveObject(LLNode currentNode)
{
    // Получаем соседей удаляемого узла
    LLNode previousNode = currentNode.backward;
    LLNode nextNode      = currentNode.forward;

    // Обнуляем указатели удаляемого объекта
    currentNode.forward = currentNode.backward = null;

    // Был ли это последний элемент списка?
    if (head == currentNode && tail == currentNode)

        head = tail = null;
        return;

    // Это узел в середине списка?
    if (head != currentNode && tail != currentNode)

        previousNode.forward = nextNode;
        nextNode.backward = previousNode;
        return;

    // Это узел в начале списка?
    if (head == currentNode && tail != currentNode)

        head = nextNode;
        nextNode.backward = null;
        return;

    // Это узел в конце списка...
    tail = previousNode;
    previousNode.forward = null;
}

// LinkedListIterator - дает приложению доступ к спискам

```

```

// LinkedList
public class LinkedListIterator : IEnumerator
{
    // Итерируемый связанный список
    private LinkedList linkedList;

    // "Текущий" и "следующий" элементы связанного списка.
    // Объявлены как private для предотвращения
    // непосредственного обращения извне
    private LLNode currentNode = null;
    private LLNode nextNode = null;

    //LinkedListIterator - конструктор
    public LinkedListIterator(LinkedList linkedList)
    {
        this.linkedList = linkedList;
        Reset ();
    }

    // Current- возвращаем объект данных в текущей позиции
    public object Current
    {
        get
        {
            if (currentNode == null)
            {
                return null;
            }
            return currentNode.linkedData;
        }
    }

    // Reset - перемещение итератора назад, в позицию,
    // непосредственно предшествующую первому узлу списка
    public void Reset()
    {
        currentNode = null;
        nextNode = linkedList.head;
    }

    // MoveNext - переход к следующему элементу списка, пока
    // не будет достигнут его конец
    public bool MoveNext()
    {
        currentNode = nextNode;
        if (currentNode == null)
        {
            return false;
        }
        nextNode = nextNode.forward;
        return true;
    }
}

public class Program

```



```

{
    public static void Main (string [] args)
    {
        // Создаем контейнер и добавляем в него три элемента
        LinkedList l1c = new LinkedList();
        LLNode first = l1c.AddObject("Первая строка");
        LLNode second = l1c.AddObject("Вторая строка");
        LLNode third = l1c.AddObject("Последняя строка");

        // Добавляем элементы в начале и середине списка
        LLNode newfirst
            = l1c.AddObject(null, "Перед первой строкой");
        LLNode newmiddle
            = l1c.AddObject(second, "Между второй и " +
                               "третьей строкой");

        // Итератором можно управлять "вручную"
        Console.WriteLine("Проход по контейнеру вручную");
        LinkedListlterator Hi
            = (LinkedListlterator)l1c.GetEnumerator();
        Hi.Reset();
        while (Hi.MoveNext())
        {
            string s = (string) Hi.Current;
            Console.WriteLine(s);
        }

        // Либо использовать цикл foreach
        Console.WriteLine("\nПоход с использованием foreach");
        foreach (string s in l1c)
        {
            Console.WriteLine(s);
        }

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
                           "завершения программы...");
        Console.Read();
    }
}

```

Классы `LinkedList` и `LLNode` образуют фундамент приведенной демонстрационной программы. На рис. 20.1 показан связанный список с тремя узлами, каждый из которых указывает на (или "содержит") отдельную строку, так что `LinkedList` вполне заслуживает названия "контейнер". Лично я, впрочем, как и большинство в мире .NET, предпочитаю термин коллекция.

Узлы в связанном списке представлены объектами `LLNode`. Для каждого данного узла член `forward` указывает на следующий узел в списке, а член `backward` — на предыдущий. Класс `LinkedList` представляет сам список. Член `head` указывает на первый узел списка, член `tail` — на последний. По сути, это все, что есть в данном классе.

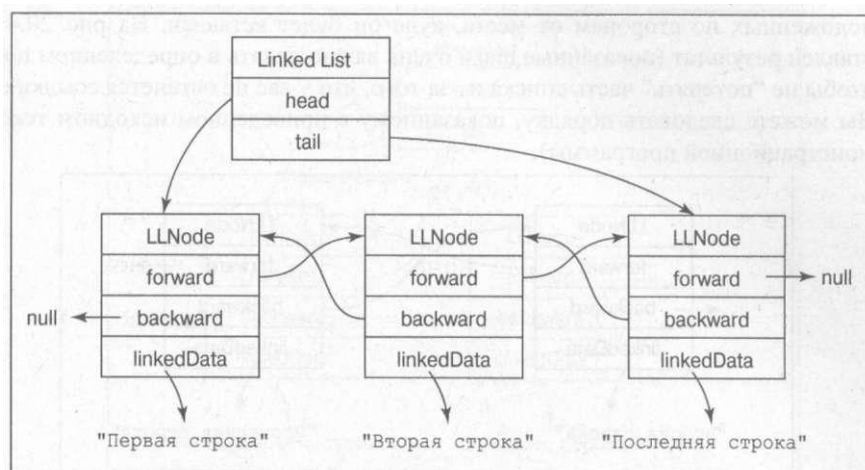


Рис. 20.1. Классы *LinkedList* и *LLNode* совместно создают связанный список

Добавление объекта в связанный список

Основные сложности содержатся в методах `AddObject()` и `RemoveObject()`. Сначала рассмотрим метод `AddObject()`.

Для того чтобы добавить объект в список, вы должны знать, куда именно его нужно поместить. По существу, имеется четыре ситуации.

1. Вы добавляете новый объект в пустой список. Это простейшая из всех ситуаций; она показана на рис. 20.2. Указатель `head` равен `null`, как и указатель `tail`. Следует просто заставить указывать их на добавляемый элемент, и на этом все — вы получите список с одним узлом.

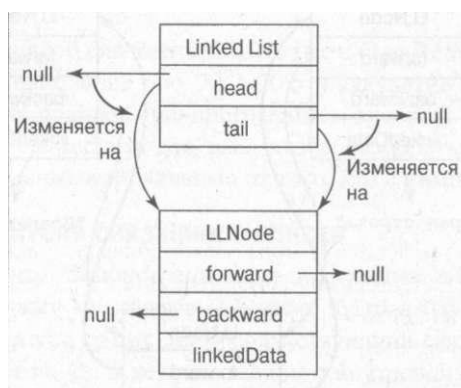


Рис. 20.2. Добавление нового узла в пустой связанный список выполняется в два счета

2. Наиболее сложная ситуация — это добавление элемента в середину списка. В этом случае предыдущий и следующий узлы не равны `null`. На рис. 20.3 показана такая ситуация. Чтобы вставить объект в середину списка, следует настроить указатели `forward` и `backward` как вставляемого объекта, так и объектов, рас-

положенных по сторонам от места, куда он будет вставлен. На рис. 20.4 представлен результат (показанные шаги очень важно делать в определенном порядке, чтобы не "потерять" часть списка из-за того, что у вас не останется ссылки на нее. Вы можете следовать порядку, показанному в приведенном исходном тексте демонстрационной программы).

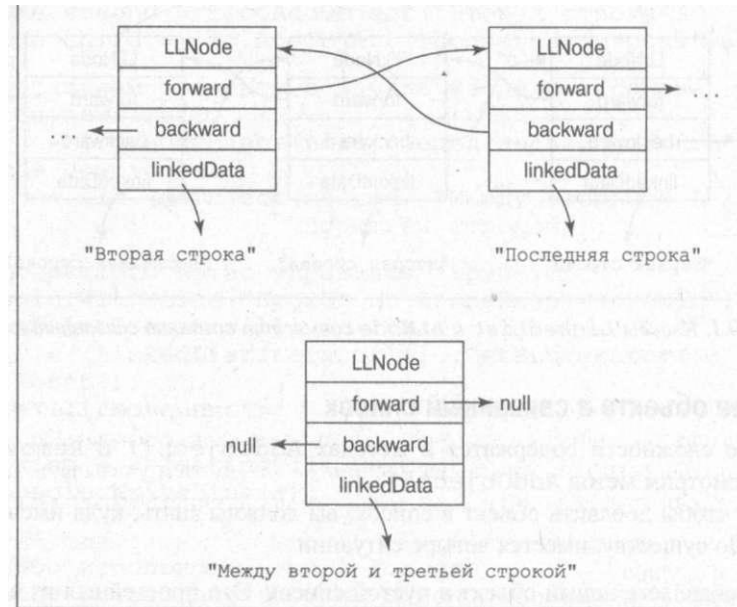


Рис. 20.3. Перед вставкой в список объект не связан ни с чем

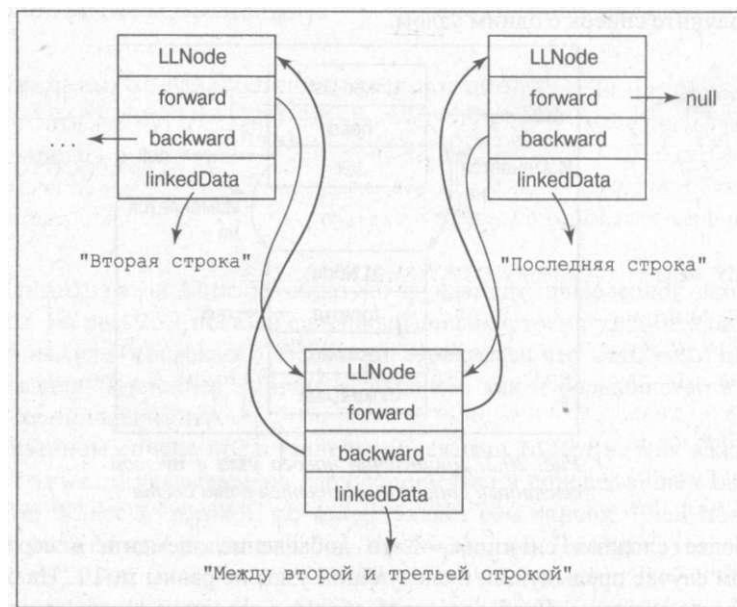


Рис. 20.4. После вставки объекта в список он становится частью команды

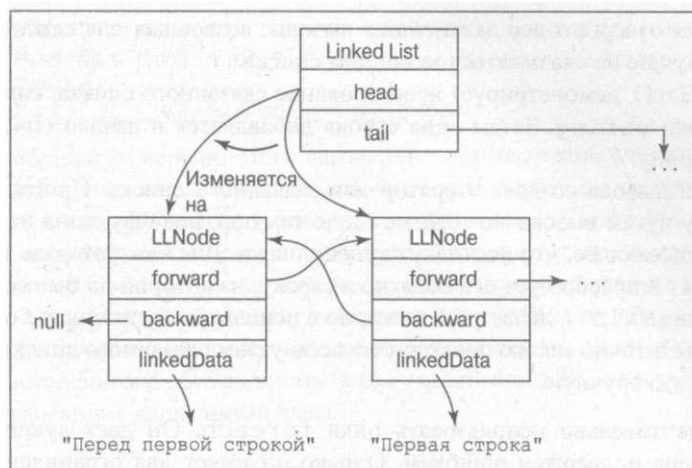


Рис. 20.5. Добавление объекта в голову списка делает его первым в списке

3. Объект может быть добавлен и в голову списка. Это гибрид первых двух случаев. Указатель на голову списка после вставки должен указывать на новый объект, а старый первый объект списка после вставки должен указывать на вставленный узел, как на предыдущий в списке. Все это продемонстрировано на рис. 20.5.
4. И наконец, новый объект может быть вставлен в конец списка. Это ситуация, обратная случаю 3.

Удаление объекта из связанного списка

Метод `RemoveObject()` рассматривает те же четыре ситуации, но в обратном направлении.



Единственный способ следовать `AddObject()` и `RemoveObject()` — нарисовать рисунки наподобие рис. 20.2-20.5 и аккуратно пройти каждый шаг. Не стесняйтесь — не родился еще программист, который в подобной ситуации ни разу не рисовал бы рисунка для того, чтобы разобраться во всем. Сложные вещи остаются сложными независимо от того, кто с ними работает.

Реализация перечислителя связанного списка

Обратите внимание, что демонстрационная программа `LinkedListContainer` в действительности содержит три класса: `LLNode`, `LinkedList` и `LinkedListIterator`. Класс `LinkedListIterator` — сопутствующий связанному списку класс со специальными привилегиями. Он в деталях знаком с внутренним устройством связанного списка, которое недоступно никому во внешнем мире. Внешние клиенты используют его для итерирования связанного списка.

Класс `LinkedListIterator` работает путем отслеживания текущего и следующего за ним узла. Изначально `currentNode` равен `null`, а `nextNode` — первому элементу связанного списка. После каждого вызова `MoveNext()` `currentNode` указывает на то, что перед этим было следующим узлом, а `nextNode` перемещается к следующему за ним узлу. После того как `MoveNext()` достигает конца списка, `currentNode` равно

`null`, и функция отвергает все дальнейшие вызовы, возвращая для каждого из них значение `false`. Лучше не оказываться за концом списка.

Функция `Main()` демонстрирует использование связанного списка, сначала добавляя в него три строки `string`. Затем одна строка добавляется в начало списка, и еще одна — в его середину.

Первый цикл вывода создает итератор для связанного списка. Программа проходит по всему списку путем вызова `MoveNext()` до тех пор, пока функция не вернет значение `false`, указывающее, что достигнут конец списка. Для каждого узла программа получает его объект и преобразует его обратно в строку, из которой он был создан.

Затем функция `Main()` делает все то же, но с использованием цикла `foreach`.

Цикл `foreach` точно так же проходит по всем узлам связанного списка, как функция `Main()` делала это вручную.



Предпочтительно использовать цикл `foreach`. Он дает лучший код, прост и меньше подвержен ошибкам. Однако он имеет два ограничения: в нем нет счетчика `int i`, но можно объявить собственный счетчик перед циклом и самостоятельно увеличивать его в теле цикла. Также нельзя удалять элементы из коллекции внутри цикла `foreach`. В этом случае вам нужна коллекция `removedItems`, в которой вы сохраняете индексы или ссылки на элементы, найденные в цикле `foreach` и которые должны быть удалены из исходной коллекции. Затем используйте цикл `foreach` еще раз для прохода по коллекции `removedItems` и удалите указанные в нем элементы из исходной коллекции. Лучше один раз увидеть, чем сто — услышать, так что вот как это выглядит в исходном тексте:

```
List<string> removedItems = new List<string>();

// Цикл по originalCollection
foreach(string s in originalCollection)
{
    // Если s требуется удалить, сохраняем ссылку или индекс в
    // removedItems
    removedItems.Add(s);
}

foreach(string s in removedItems) // Цикл по removedItems
{
    originalCollection.Remove(s);
}
```

Вывод программы `LinkedListContainer` выглядит следующим образом:

```
Проход по контейнеру вручную
Перед первой строкой
Первая строка
Вторая строка
Между второй и третьей строкой
Последняя строка

Обход с использованием foreach
Перед первой строкой
Первая строка
```

Вторая строка
Между второй и третьей строкой
Последняя строка
Нажмите <Enter> для завершения программы...



Обобщенную версию этого связанного списка можно найти в демонстрационной программе `GenericLinkedListContainer` на прилагаемом компакт-диске. Обратите внимание, что `GenericLinkedListContainer` продолжает использовать интерфейс `IEnumerator`, который будет рассмотрен немного позже, но в некоторых ситуациях следует применять вместо него новую обобщенную версию `IEnumerator<T>`. Однако пока не стоит начинать анализировать обобщенные классы. Кроме того, обратитесь к встроенному обобщенному классу `LinkedList`, который, несомненно, превосходит написанный здесь.

Зачем нужен связанный список

Связанный список может показаться пустыми хлопотами. Его основное преимущество заключается в большой скорости вставки и удаления узлов. "Ну хорошо, — можете сказать вы. — Но ведь добавление `string` в массив из четырех или пяти строк не сложнее перемещения нескольких ссылок для освобождения места." А что вы скажете, если этот массив будет содержать несколько сотен тысяч строк, и вы должны выполнять массу вставок и удалений из него?

Второе преимущество связанного списка в том, что он может расти и уменьшаться. Если вы думаете, что вам будут нужны 1000 объектов, то вы должны создать массив на 1000 элементов, независимо от того, будете вы их использовать или нет. Что еще хуже, если вы в действительности создадите 2000 объектов, то можете считать, что в этот раз вам крупно не повезло. (Да, конечно, можно создать второй массив с большей емкостью и скопировать в него содержимое первого, но что при этом можно сказать об эффективности и затратах памяти?)



На этом принципе основаны многие распространенные вирусы. Например, некоторый исполненный благих намерений программист решает, что 256 символов будет достаточно для любого имени файла, и объявляет массив `char[256]`. Если программист забудет убедиться, что имя на самом деле не длиннее, чем ожидается, то у хакера появляется шанс сломать программу, передав ей неправдоподобно длинное имя, и переписать тем самым часть кода за массивом (это называется *переполнением буфера*). Впрочем, это проблема в первую очередь `C/C++`: `C#` автоматически проверяет выход за границы массива.

Обход коллекций: итераторы

В оставшейся части главы будут проанализированы три разных подхода к общей задаче *итерирования* коллекции. В этом разделе будет продолжено обсуждение наиболее традиционного (как минимум, для программистов на `C#`) подхода с использованием итераторов, которые реализуют интерфейс `IEnumerator`. В качестве примера рассматривается итератор для связанного списка из предыдущего раздела.



Термины *итератор* (iterator) и *перечислитель* (enumerator) являются синонимами. Термин *итератор* более распространен, несмотря на имя реализуемого им интерфейса. От обоих терминов можно произвести глагольную форму — вы можете итерировать контейнер, а можете перечислять. Другими подходами к решению этой же задачи являются индексаторы и новые блоки итераторов.

Доступ к коллекции: общая задача

Различные типы коллекций могут иметь разные схемы доступа. Не все виды коллекции могут быть эффективно доступны с использованием индексов наподобие массивов — таков, например, связанный список. Различия между типами коллекций делают невозможным написание без специальных средств функции наподобие приведенной далее:

```
// передается коллекция любого вида
void myClearFunction(Collection aColl, int index)
{
    aColl[index] = 0; // Индексирование работает не для всех
                     // типов коллекций
    // ... продолжение...
}
```

Коллекции каждого типа могут сами определять свои методы доступа (и делают это). Например, связанный список может предоставить метод `GetNext()` для выборки следующего элемента из цепочки объектов; стек может предложить методы `Push()` и `Pop()` для добавления и удаления объектов и т.д.

Более общий подход состоит в предоставлении для каждого класса коллекции отдельного так называемого *класса итератора*, который знает, как работать с конкретной коллекцией. Каждая коллекция *X* определяет свой собственный класс `IteratorX`. В отличие от *X*, `IteratorX` представляет общий интерфейс `IEnumerator`, золотой стандарт итерирования. Этот метод использует второй объект, именуемый итератором, в качестве указателя внутрь коллекции.

Итератор (перечислитель) обладают следующими преимуществами.

- ✓ Каждый класс коллекции может определить свой собственный класс итератора. Поскольку итератор реализует стандартный интерфейс `IEnumerator`, с ним обычно легко работать.
- ✓ Прикладной код не должен знать о внутреннем устройстве коллекций. Пока программист работает с итератором, тот берет на себя все заботы о деталях. Это — хорошая инкапсуляция.
- ✓ Прикладной код может создать много независимых объектов-итераторов для одной и той же коллекции. Поскольку итератор содержит информацию о своем собственном состоянии (знает, где он находится в процессе итерирования), каждый итератор может независимо проходить по коллекции. Вы можете одновременно выполнять несколько итераций, причем в один и тот же момент все они могут находиться в разных позициях.

Чтобы сделать возможным наличие цикла `foreach`, интерфейс `IEnumerator` должен поддерживать различные типы коллекций — от массивов до связанных списков. Следовательно, его методы должны быть максимально обобщенными, насколько это возможно. Например, нельзя использовать итератор для произвольного доступа к элементам коллекции, поскольку большинство коллекций не обеспечивают подобного доступа.

`IEnumerator` предоставляет три следующих метода.

- ✓ `Reset()` — устанавливает итератор таким образом, чтобы он указывал на начало коллекции. *Примечание:* обобщенная версия `IEnumerator`, `IEnumerator<T>`, не предоставляет метод `Reset()`. В случае обобщенного `LinkedList` просто начинайте работу с вызова `MoveNext()`.
- ✓ `MoveNext()` — перемещает итератор от текущего объекта в контейнере к следующему.
- ✓ `Current` — свойство (не метод), которое дает объект данных, хранящийся в текущей позиции итератора.

Описанный принцип продемонстрирован приведенной далее функцией. Программист класса `MyCollection` (не показанного здесь) создает соответствующий класс итератора — скажем, `IteratorMyCollection` (применяя соглашение об именах `IteratorX`, упоминавшееся ранее). Прикладной программист ранее сохранил ряд объектов `ContainedDataObjects` в коллекции `MyCollection`. Приведенный далее фрагмент исходного текста использует три стандартных метода `IEnumerator` для чтения этих объектов:

```
// Класс MyCollection хранит объекты типа
// ContainedDataObject data
void MyFunction(MyCollection myColl)
{
    // Программист, создавший класс MyCollection, создал также
    // и класс итератора IteratorMyCollection; прикладной
    // программист создает объект итератора для прохода по
    // объекту myColl
    IEnumerator iterator = new IteratorMyCollection(myColl);
    // перемещаем итератор в "следующую позицию" внутри
    // коллекции
    while(iterator.MoveNext ())
    {
        // Получаем ссылку на объект данных в текущей позиции
        // коллекции
        ContainedDataObject containedData; // data
        contained = (ContainedDataObject)iterator.Current;
        // ...используем объект данных contained...
    }
}
```

Функция `MyFunction()` принимает в качестве аргумента коллекцию `ContainedDataObjects`. Она начинается с создания итератора типа `IteratorMyCollection`. Функция начинает цикл с вызова `MoveNext()`. При первом вызове `MoveNext()` перемещает итератор к первому элементу коллекции. При каждом последующем вызове `MoveNext()` перемещает указатель "на одну позицию." Функция `MoveNext()` возвращает `false`, когда коллекция исчерпана и итератор больше нельзя передвинуть.

Свойство `Current` возвращает ссылку на объект данных в текущей позиции итератора. Программа преобразует возвращаемый объект в `ContainedDataObject` перед тем, как присвоить его переменной `contained`. Вызов `Current` некорректен, если предшествующий вызов метода `MoveNext()` не вернул `true`.

Использование foreach

Методы `IEnumerator` достаточно стандартны для того, чтобы C# использовали их автоматически для реализации конструкции `foreach`.

Цикл `foreach` может обращаться к любому классу, реализующему интерфейс `IEnumerable`, как показано в приведенной обобщенной функции, которая может работать с любым классом — от массивов и связанных списков до стеков и очередей:

```
void MyFunction(IEnumerable containerOfStrings)
{
    foreach(string s in containerOfStrings)
    {
        Console.WriteLine("Следующая строка - {0}", s);
    }
}
```

Класс реализует `IEnumerable` путем определения метода `GetEnumerator()`, который возвращает экземпляр `IEnumerator`. Скрыто от посторонних глаз `foreach` вызывает метод `GetEnumerator()` для получения итератора. Цикл использует этот итератор для обхода контейнера. Каждый выбираемый им элемент приводится к соответствующему типу перед тем, как продолжить выполнение тела цикла. Обратите внимание, что `IEnumerable` и `IEnumerator` различные, но связанные интерфейсы. C# 2.0 предоставляет обобщенную версию обоих интерфейсов — см. информацию о пространстве имен `System.Collections.Generic` в справочной системе.

Итак, цикл `foreach` можно записать таким образом:

```
foreach(int nValue in myContainer)
{
    // ...
}

Это эквивалентно следующему циклу for:
for(IEnumerator i =
    myContainer.GetEnumerator(); // Инициализация
    i.MoveNext();                // Условие
    )                             // Пустой инкремент
{
    int nValue = (int)i.Current   // Получение текущего
    //                                     // элемента
}
```

Раздел инициализации цикла `for` получает итератор. Раздел условия использует `MoveNext()` для определения конца контейнера. `MoveNext()` сам увеличивает указатель, т.е. раздел инкремента цикла пуст. (Тем не менее, вам все равно следует указать точку с запятой после раздела условия, после которой не следует никакой код. В приведенном фрагменте исходного текста это точка с запятой после `i.MoveNext()`.) В первой строке цикла выбирается очередной объект и преобразуется к `int` (`Current` всегда возвращает тип `Object`). Если возвращенный объект не является `int`, C# генерирует исключение неверного преобразования типа.

Обращение к коллекциям как к массивам: индексаторы

Обращение к элементам массива очень простое и понятное: команда `container [n]` обеспечивает обращение к *n*-му элементу массива `container`. Было бы хорошо, если бы так же просто можно было обращаться и к другим типам коллекций.

C# позволяет написать вам свою собственную реализацию операции индексирования. Вы можете обеспечить возможность обращения через индекс коллекции, которые таким свойством изначально не обладают. Кроме того, вы можете индексировать с использованием в качестве индексов не только типа `int`, но и других типов, например, `string`.

Формат индексатора

Индексатор выглядит очень похоже на свойство, за тем исключением, что в нем вместо имени свойства появляются ключевое слово `this` и оператор индекса `[]`:

```
class MyArray
{
    public string this[int index] // Обратите внимание на
    {                             // ключевое слово "this"
        get
        {
            return array[index];
        }
        set
        {
            array[index] = value;
        }
    }
}
```

За сценой выражение `s = myArray[i]`; вызывает функцию доступа `get`, передавая ей значение индекса `i`. Выражение `myArray[i] = "строка"`; приводит к вызову функции доступа `set`, которой передаются индекс `i` и строка "строка".

Пример программы с использованием индексатора

Индексы не ограничены типом `int`. Например, вы можете использовать для индексирования коллекции домов имена их владельцев или адреса. Кроме того, свойство индексатора может быть перегружено для различных типов индекса.



Приведенная далее демонстрационная программа `Indexer` генерирует класс виртуального массива `KeyedArray`, который выглядит и функционирует точно так же, как и обычный массив, с тем исключением, что в качестве его индексов применяется значение типа `string`.

```
// Indexer - данная демонстрационная программа иллюстрирует
// использование оператора индекса для обеспечения доступа к
// массиву с использованием строк в качестве индексов
```

```

using System;

namespace Indexer
{
    public class KeyedArray

        // Следующая строка обеспечивает "ключ" к массиву — это
        // строка, которая идентифицирует элемент
        private string[] sKeys;

        // object представляет собой реальные данные, связанные
        // с ключом
        private object [] oArrayElements;

        // KeyedArray - создание KeyedArray фиксированного
        // размера
        public KeyedArray(int nSize)

            sKeys = new string[nSize] ;
            oArrayElements = new object[nSize];

        // Find - поиск индекса записи, соответствующей строке
        // sTargetKey (если запись не найдена, возвращает -1)
        private int Find(string sTargetKey)

            for(int i = 0; i < sKeys.Length; i++)

                if (String.Compare(sKeys[i], sTargetKey) == 0)

                    return i;
            }
        }
        return -1;
    }

    // FindEmpty - поиск свободного места в массиве для
    // новой записи
    private int FindEmpty()

        for (int i = 0; i < sKeys.Length; i++)

            if (sKeys[i] == null)
            {
                return i;
            }

        throw new Exception("Массив заполнен");

    // Ищем содержимое по указанной строке - это и есть
    // индексатор
    public object this[string sKey]
    {
        set
        {

```

```

        // Проверяем, нет ли уже такой строки
        int index = Find(sKey);
        if (index < 0)
        {
            // Если нет - ищем новое место
            index = FindEmpty();
            sKeys[index] = sKey;
        }
        // Сохраняем объект в соответствующей позиции
        oArrayElements[index] = value;
    }

    get
    {
        int index = Find(sKey);
        if (index < 0)
        {
            return null;
        }
        return oArrayElements[index];
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Создаем массив с достаточным количеством элементов
        KeyedArray ma = new KeyedArray(100);
        // Сохраняем возраст членов семьи Симпсонов
        ma["Bart"] = 8;
        ma["Lisa"] = 10;
        ma["Maggie"] = 2;

        // Ищем возраст Lisa
        Console.WriteLine("Ищем возраст Lisa");
        int age = (int)ma["Lisa"];
        Console.WriteLine("Возраст Lisa - {0}", age);

        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы...");

        Console.Read();
    }
}

```



Класс `KeyedArray` включает два обычных массива. Массив `oArrayElements` содержит реальные данные `KeyedArray`. Строки, которые хранятся в массиве `sKeys`, работают в качестве идентификаторов массива объектов, *г*-ый элемент `sKeys` соответствует *г*-ой записи `oArrayElements`. Это позволяет прикладной программе индексировать `KeyedArray` с помощью индексов типа `string`.



Индексы, не являющиеся целыми числами, известны как *ключи* (key). Кстати, можно реализовать `KeyedArray` с использованием `List<T>` (см. главу 15, "Обобщенное программирование") вместо массива фиксированного размера. `List<T>` индексируется, как и массив, так как они оба реализуют интерфейс `IList` (или `IList<T>`). Это позволит сделать `KeyedArray` обобщенным классом и получить большую гибкость, чем при использовании внутреннего массива.

Индексатор `set[string]` начинает с проверки, не имеется ли уже данного индекса в массиве, для чего он применяет функцию `Find()`. Если она возвращает индекс, `set[]` сохраняет новый объект данных в соответствующем элементе `oArrayElements`. Если `Find()` не может найти ключ, `set[]` вызывает `FindEmpty()` для возврата пустого элемента, где и будет сохранен переданный объект.

Функция `get[]` работает с индексом с применением аналогичной логики. Сначала она ищет определенный ключ с использованием метода `FindO`. Если `Find()` возвращает отрицательный индекс, `get[]` возвращает соответствующий член `oArrayElements`, где хранятся запрошенные данные. Если же `FindO` возвращает -1, то метод `get[]` возвращает значение `null`, указывающее, что переданный ключ в списке отсутствует.

Метод `Find()` циклически проходит по всем элементам массива `sKeys` в поисках элемента с тем же значением, что и переданное значение типа `string`. Метод `Find()` возвращает индекс найденного элемента (или значение -1, если элемент не найден). Функция `FindEmpty()` возвращает индекс первого элемента, который не имеет связанного ключевого элемента.

При написании методов `Find()` и `FindEmpty()` не ставилась цель повысить их эффективность, так что имеется множество возможностей сделать их быстрее, но все они не имеют никакого отношения к индексаторам.

Правда, было бы здорово добавить возможность индексирования к классу связанного списка `LinkedList`? Да, это можно сделать. Но вспомните, что даже в классе `KeyedArray` требуется проход по массиву `sKeys` для поиска определенного ключа, а значит, и функций `Find()` и `FindEmpty()`, которые этим занимаются. Точно так же при реализации индексатора для `LinkedList` вам придется осуществлять проход по всему связанному списку, и единственный способ сделать это — пройти по всему списку с использованием итератора `LinkedListIterator`, следуя по ссылкам `forward` от узла к узлу. Индексатор окажется удобным, но очень медлительным.

Заметьте, что вы не можете удалять элементы посредством ключа `null`. Как же реализовать удаление? Как часто говорится в учебниках — "данная задача остается читателю в качестве домашнего упражнения".

Функция `Main()` демонстрирует применение индексатора. Сначала программа создает объект типа `KeyedArray` длины 100 (т.е. со 100 свободными элементами). Далее в этом объекте сохраняется возраст детей семьи Симпсонов с использованием имен в качестве индексов. И наконец, программа получает возраст Лизы с применением выражения `ma["Lisa"]` и выводит его на экран.

Обратите внимание, что программа должна выполнить преобразование типа для значения, возвращенного из `ta[]`, так как `KeyedArray` написан таким образом, что может хранить объекты любого типа. Без такого преобразования типов можно обойтись, если индексатор написан так, что может работать только со значениями типа `int`, или если `KeyedArray` — обобщенный класс (см. главу 15, "Обобщенное программирование").

Вывод программы прост и элегантен:
Ищем возраст Lisa
Возраст Lisa - 10
Нажмите <Enter> для завершения программы...



В качестве отступления— интерфейс `ICollection` описывает класс, предоставляющий целый индекатор в форме `object this[int]`. В C# имеется также интерфейс `IEnumerator`, который позволяет заменить `object` выбранным вами типом `T`. Это устраняет необходимость преобразования типов из предыдущего примера.

Блок итератора

Помните код функции `Main()`, написанный при демонстрации самостоятельно разработанного класса `LinkedList`? (Его можно найти в демонстрационной программе `LinkedListContainer` на прилагаемом компакт-диске.) Вот его фрагмент:

```
public static void Main(string[] args)
{
    // Создаем контейнер и добавляем в него три элемента
    LinkedList llc = new LinkedList();
    // Добавляем объекты...

    Console.WriteLine("Проход по контейнеру вручную");
    LinkedListIterator Hi
        = (LinkedListIterator)llc.GetEnumerator();
    Hi.Reset();
    while (Hi.MoveNext())
    {
        string s = (string)Hi.Current;
        Console.WriteLine(s);
    }
}
```

Данный код получает `LinkedListIterator` и использует его метод `MoveNext()` и свойство `Current` для обхода связанного списка. Однако C# 2.0 может упростить вам этот обход так, что вы получите перечисленные преимущества.

- ✓ Вам не придется вызывать `GetEnumerator()` (и выполнять преобразование типа результатов).
- ✓ Вам не понадобится вызывать `MoveNext()`.
- ✓ Вам не придется вызывать `Current` и выполнять преобразование типа, возвращаемого значения.
- ✓ Вы сможете просто использовать `foreach` для обхода коллекции (C# сделает все остальное за вас).

Если быть честным, то `foreach` работает и для класса `LinkedList` из этой главы. Это связано с наличием метода `GetEnumerator()`. Но я все еще должен самостоятельно писать класс `LinkedListIterator`. Новизна состоит в том, что вы можете пропустить при обходе часть вашего класса.



В создаваемых классах коллекций можно предоставить *блок итератора* (iterate block) вместо написания собственного класса итератора для поддержки коллекции.

Можно использовать блок итератора и для других рутинных работ.

Этот новый подход применяет *блоки итераторов*. Когда вы пишете класс коллекции, такой как `KeyedList` или `PriorityQueue` — вы реализуете блок итератора вместо реализации интерфейса `IEnumerator`. Затем пользователи этого класса могут просить итерировать коллекцию с помощью цикла `foreach`. Приготовьтесь расстаться с частью вашего драгоценного времени, чтобы ознакомиться с несколькими вариантами *блоков итераторов*.



Все примеры в этом разделе являются частью демонстрационной программы `IteratorBlocks` на прилагаемом компакт-диске.

```
// IteratorBlocks - демонстрация применения блоков
// итераторов для написания итераторов коллекций
using System;
namespace IteratorBlocks
{
    class IteratorBlocks
    {
        //Main - демонстрация пяти различных приложений блоков
        // итераторов
        static void Main(string[] args)
        {
            // Итерирование месяцев года, вывод количества дней в
            // каждом из них
            MonthDays md = new MonthDays();
            // Итерируем
            Console.WriteLine("Месяцы:\n");
            foreach (string sMonth in md)
            {
                Console.WriteLine(sMonth);
            }
            // Итерируем коллекцию строк
            StringChunks sc = new StringChunks();
            // Итерируем - выводим текст, помещая каждый фрагмент
            // в своей собственной строке
            Console.WriteLine("\nСтроки:\n");
            foreach (string sChunk in sc)
            {
                Console.WriteLine(sChunk);
            }
            // А теперь выводим их в одну строку
            Console.WriteLine("\nВывод в одну строку:\n");
            foreach (string sChunk in sc)
            {
                Console.Write(sChunk);
            }
            Console.WriteLine();
            // Итерируем простые числа до 13
```

```

YieldBreakEx yb = new YieldBreakEx();
// Итерируем, останавливаясь после 13
Console.WriteLine("\nПростые числа:\n");
foreach (int nPrime in yb)

    Console.WriteLine(nPrime);
}
// Итерируем четные числа в убывающем порядке
EvenNumbers en = new EvenNumbers();
// Вывод четных чисел от 10 до 4
Console.WriteLine("\nЧетные числа:\n");
foreach (int nEven in en.DescendingEvens(11, 3))

    Console.WriteLine(nEven);
}
// Итерируем числа типа double
PropertyIterator prop = new PropertyIterator();
Console.WriteLine("\nЧисла double:\n");
foreach (double db in prop.DoubleProp)

    Console.WriteLine(db);
}
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}
}
// Классы IteratorBlocks
// MonthDays - определяем итератор, который возвращает
// месяцы и количество дней в них
class MonthDays
{
    string[] months =
    { "January 31", "February 28", "March 31",
      "April 30", "May 31", "June 30", "July 31",
      "August 31", "September 30", "October 31",
      "November 30", "December 31" };
    //GetEnumerator - это и есть итератор
    public System.Collections.IEnumerator GetEnumerator()
    {
        foreach (string sMonth in months)
        {
            // Возвращаем по одному месяцу в каждой итерации
            yield return sMonth; // Новый синтаксис
        }
    }
}
//StringChunks - определение итератора, возвращающего
// фрагменты текста
class StringChunks
{
    //GetEnumerator - итератор. Обратите внимание, как он
    // (дважды) вызывается в Main

```



```

public System.Collections.IEnumerator GetEnumerator()
{
    // Возврат разных фрагментов текста на каждой итерации
    yield return "Using iterator ";
    yield return "blocks ";
    yield return "isn't all ";
    yield return "that hard";
    yield return ".";
}
}
//YieldBreakEx - пример использования ключевого слова
// yield break
class YieldBreakEx
{
    int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
    //GetEnumerator - возврат последовательности простых
    // чисел с демонстрацией применения конструкции yield
    // break
    public System.Collections.IEnumerator GetEnumerator()
    {
        foreach (int nPrime in primes)
        {
            if (nPrime > 13) yield break; //Новый синтаксис
            yield return nPrime;
        }
    }
}
//EvenNumbers - числовой итератор, возвращающий четные
// числа между граничными значениями в убывающем порядке
class EvenNumbers
{
    //DescendingEvens - "именованный" итератор, который
    // также демонстрирует использование конструкции yield
    // break
    public System.Collections.IEnumerable
    {
        DescendingEvens(int nTop, int nStop)
        // Начинаем с ближайшего к nTop меньшего четного числа
        if (nTop % 2 != 0)
            nTop -= 1;
        // Итерируем до ближайшего к nStop четного числа,
        // превышающего это значение
        for (int i = nTop; i >= nStop; i -= 2)
        {
            if (i < nStop)
                yield break;
            // Возвращаем на каждой итерации очередное четное
            // число
            yield return i;
        }
    }
}
//PropertyIterator - реализация функции доступа свойства
// класса в качестве блока итератора

```

```

class PropertyIterator
{
    double[] doubles = { 1.0, 2.0, 3.5, 4.67 };
    // DoubleProp - свойство "get" с блоком итератора
    public System.Collections.IEnumerable DoubleProp

        get
        {
            foreach (double db in doubles)
            {
                yield return db;
            }
        }
}

```

Итерация месяцев

Следующий фрагмент из демонстрационной программы `IteratorBlocks` представляет итератор, который проходит по месяцам года:

```

// MonthDays - определяем итератор, который возвращает
// месяцы и количество дней в них
class MonthDays
{
    string[] months =
    { "January 31", "February 28", "March 31",
      "April 30", "May 31", "June 30", "July 31",
      "August 31", "September 30", "October 31",
      "November 30", "December 31" };
    //GetEnumerator - это и есть итератор
    public System.Collections.IEnumerator GetEnumerator()
    {
        foreach (string sMonth in months)
        {
            // Возвращаем по одному месяцу в каждой итерации
            yield return sMonth; // Новый синтаксис
        }
    }
}

```

А вот часть функции `Main()`, где выполняется итерирование этой коллекции с применением цикла `foreach`:

```

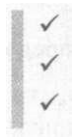
// Итерирование месяцев года, вывод количества дней в
// каждом из них
MonthDays md = new MonthDays();
// Итерируем
Console.WriteLine("Месяцы:\n");
foreach (string sMonth in md)
{
    Console.WriteLine(sMonth);
}

```

Это исключительно простой "класс коллекции", основанный на массиве, как и ~~как~~ `KeyedArray`. Класс содержит массив, элементы которого имеют тип `string`. Клиент итерирует данную коллекцию, ее блок итератора выдает ему эти строки по ~~он~~ *одной*. Каждая строка содержит имя месяца с количеством дней в нем. Тут нет ничего сложного,

Класс определяет собственный блок итератора, в данном случае как метод `GetEnumerator()`. Метод `GetEnumerator()` возвращает объект типа `System.Collections, IEnumerator`. Да, вы должны были писать такой метод и ранее, но вы должны ~~были~~ *были* писать не только его, но и собственный класс-перечислитель для поддержки вашего класса-коллекции. Теперь же вы *пишете только простой метод, возвращающий пере-* *числитель* с использованием новых ключевых слов `yield return`. Все остальное C# делает за вас: создает класс-перечислитель и применяет его метод `MoveNext()` для итерирования. У вас уменьшается количество работы и размер исходного текста.

Ваш класс, содержащий метод `GetEnumerator()`, больше не должен реализовывать интерфейс `IEnumerator`. В следующих разделах вам будет показано несколько вариаций блоков итераторов:



- ✓ обычные итераторы;
- ✓ именованные итераторы;
- ✓ свойства классов, реализованные как итераторы.

Что такое коллекция

Остановимся на минутку и сравним эту небольшую коллекцию с коллекцией `LinkedList`, рассматривавшейся ранее в главе. В то время как `LinkedList` имеет сложную структуру узлов, связанных посредством указателей, приведенная простейшая коллекция месяцев основана на простом массиве с фиксированным содержимым. Но ~~не~~ *все* же следует расширить понятие коллекции.

(Ваш класс коллекции не обязан иметь фиксированное содержимое — большинство коллекций разработаны для хранения объектов путем добавления их в коллекции, ~~на~~ *например*, с помощью метода `Add()` или чего-то в этом роде. Класс `KeyedArray`, к ~~при~~ *примеру*, использует для добавления элементов в коллекцию индексатор. Ваша коллекция также должна обеспечивать метод `Add()`, как и блок итератора, чтобы вы могли ~~рабо~~ *работать* с ней с помощью `foreach`.)

Цель коллекции, в наиболее общем смысле, заключается в хранении множества ~~об-~~ *объектов* и обеспечении возможности их обхода, последовательно выбирая их по одному — хотя иногда может использоваться и произвольная выборка, как в демонстрационной программе `Indexer`. (Конечно, массив и так в состоянии справиться с этим, без ~~допол-~~ *дополнительных* "наворотов" наподобие класса `MonthDays`, но итераторы вполне могут ~~при-~~ *применяться* и за пределами примера `MonthDays`.)

Говоря более обобщенно, независимо от того, что именно происходит за сценой, ~~иер-~~ *иерируемая* коллекция генерирует "поток" значений, который можно получить с помощью `foreach`.

Для лучшего понимания данной концепции ознакомьтесь с еще одним примером ~~про-~~ *простого* класса из демонстрационной программы `IteratorBlocks`, который иллюстрирует чистую идею коллекции:

```
//StringChunks - определение итератора, возвращающего  
// фрагменты текста
```

```

class StringChunks

//GetEnumerator - итератор. Обратите внимание, как он
// (дважды) вызывается в Main
public System.Collections.IEnumerator GetEnumerator()
{
    // Возврат разных фрагментов текста на каждой итерации
    yield return "Using iterator ";
    yield return "blocks ";
    yield return "isn't all ";
    yield return "that hard";
    yield return ".";
}
}

```

Коллекция `StringChunks`, как ни странно, ничего не *хранит* в обычном смысле этого слова. В ней нет даже массива. Так где же тут коллекция? Она — в последовательности вызовов `yield return`, использующих специальный новый синтаксис для возврата элементов один за другим, пока все они не будут возвращены вызывающей функции. Эта коллекция "содержит" пять объектов, каждый из которых представляет собой простую строку, как и в рассмотренном только что примере `MonthDays`. Извне класса, в функции `Main()`, вы можете итерировать эти объекты посредством простого цикла `foreach`, поскольку конструкция `yield return` возвращает *по одной строке за раз*. Вот часть функции `Main()`, в которой выполняется итерирование "коллекции" `StringChunks`:

```

// Итерируем коллекцию строк
StringChunks sc = new StringChunks();
// Итерируем - выводим текст, помещая каждый фрагмент
// в своей собственной строке
Console.WriteLine("\Строки:\n");
foreach (string sChunk in sc)
{
    Console.WriteLine(sChunk);
}

```

Синтаксис итератора

В C# 2.0 вводятся два новых варианта синтаксиса итераторов. Конструкция `yield return` больше всего напоминает старую комбинацию `MoveNext()` и `Current` для получения очередного элемента коллекции. Конструкция `yield break` похожа на оператор `break`, который позволяет прекратить работу цикла или конструкции `switch`.

yield return

Синтаксис `yield return` работает следующим образом.

1. При первом вызове он возвращает первое значение коллекции.
2. При следующем вызове возвращается второе значение. »
3. И так далее...

Это очень похоже на старый метод итератора `MoveNext()`, использовавшийся в `LinkedList`. Каждый вызов `MoveNext()` предоставляет новый элемент коллекции. Однако в данном случае вызов `MoveNext()` не требуется.

Что же подразумевается под очередным вызовом? Давайте еще раз посмотрим на цикл `foreach`, использующийся для итерирования коллекции `StringChunks`:

```
foreach (string sChunk in sc)
{
    Console.WriteLine(sChunk);
}
```

Каждый раз, когда цикл получает новый элемент посредством итератора, последний сохраняет достигнутую им позицию в коллекции. При очередной итерации цикла `foreach` итератор возвращает следующий элемент коллекции.

yield break

Следует упомянуть еще об одном синтаксисе. Можно остановить работу итератора в определенный момент, использовав в нем конструкцию `yield break`. Например достигнут некоторый порог при тестировании определенного условия в блоке итератора класса коллекции, и вы хотите на этом прекратить итерации. Вот краткий пример блока итератора, использующего `yield break` именно таким образом:

```
//YieldBreakEx - пример использования ключевого слова
// yield break
class YieldBreakEx
{
    int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23 },-
    //GetEnumerator - возврат последовательности простых
    // чисел с демонстрацией применения конструкции yield
    // break
    public System.Collections.IEnumerator GetEnumerator()
    {
        foreach (int nPrime in primes)
        {
            if (nPrime > 13) yield break; // Новый синтаксис
            yield return nPrime;
        }
    }
}
```

В рассмотренном случае блок итератора содержит оператор `if`, который проверяет простые числа, возвращаемые итератором (кстати, с применением еще одного цикла `foreach` внутри итератора). Если простое число превышает 13, в блоке выполняется инструкция `yield break`, которая прекращает возврат простых чисел итератором. В противном случае работа итератора продолжалась бы, и каждая инструкция `yield return` давала бы очередное простое число, пока коллекция полностью не исчерпалась бы.

Блоки итераторов произвольного вида и размера

До этого момента блоки итераторов выглядели примерно следующим образом:

```
public System.Collections.IEnumerator GetEnumerator()
{
```

```

    yield return something;
}

```

Однако они могут также принимать и другие формы — *именованных итераторов* и *свойств классов*.

Именованные итераторы

Вместо того чтобы писать блок итератора в виде метода с именем `GetEnumerator()`, можно написать *именованный итератор* — функцию, возвращающую интерфейс `System.Collections.IEnumerable` вместо `IEnumerator`, которая не обязана иметь имя `GetEnumerator()` — можете назвать ее хоть `MyFunction()`.

Вот, например, простая функция, которая может использоваться для итерирования четных чисел от некоторого значения в порядке *убывания* до некоторого конечного значения — да, да, именно в порядке убывания: для итераторов это сущие пустяки!

```

//EvenNumbers - определяет именованный итератор, который
// возвращает четные числа в определенном диапазоне в
// порядке убывания
class EvenNumbers
{
    //DescendingEvens - это "именованный итератор", в котором
    // используется ключевое слово yield break. Обратите
    // внимание на его использование в цикле foreach в функции
    // Main()
    public System.Collections.IEnumerable
        DescendingEvens(int nTop,
                        int nStop)
    {
        // Начинаем с ближайшего к nTop четного числа, не
        // превосходящего его
        if (nTop % 2 != 0) // Если nTop нечетно
            nTop -= 1;
        // Итерации от nTop в порядке уменьшения до ближайшего к
        // nStop четного числа, превосходящего его
        for (int i = nTop; i >= nStop; i -= 2)

            if (i < nStop) yield break;
            // Возвращаем очередное четное число на каждой
            // итерации
            yield return i;
        }
    }
}

```

Метод `DescendingEvens()` получает два аргумента (удобная возможность), определяющих верхнюю и нижнюю границы выводимых четных чисел. Первое четное число равно первому аргументу или, если он нечетен, на 1 меньше него. Последнее генерируемое четное число равно значению второго аргумента `nStop` (или, если `nStop` нечетно, на 1 больше него). Эта функция возвращает не значение типа `int`, а интерфейс `IEnumerable`. Но в ней все равно имеется инструкция `yield return`, которая возвращает четное число и затем ожидает очередного вызова из цикла `foreach`.

Примечание: это еще один пример "коллекции", в основе которой нет никакой "настоящей" коллекции, наподобие уже рассматривавшегося ранее класса `string` -

Chunks. Заметим также, что эта коллекция *вычисляется*— на этот раз возвращаемые значения не жестко закодированы, а вычисляются по мере необходимости. Это еще один способ получить коллекцию без коллекции. (Вы можете получать элементы коллекции откуда угодно — например, из базы данных или от Web-сервиса). И наконец, этот пример демонстрирует, что вы можете итерировать так, как вам заблагорассудится — например с шагом -2, а не стандартным единичным.

Вот как можно вызвать DescendingEvens() из цикла foreach в функции Main() (заодно здесь показано, что произойдет, если передать нечетные граничные значения — еще одно применение оператора %):

```
// Инстанцирование класса "коллекции" EvenNumbers
EvenNumbers en = new EvenNumbers();
// Итерирование: выводим четные числа от 10 до 4
Console.WriteLine("\n Поток убывающих четных чисел:");
foreach(int even in en.DescendingEvens(11, 3))
{
    Console.WriteLine(even);
}
```

Этот вызов дает список четных чисел от 10 до 4. Обратите также внимание, как используется цикл foreach. Вы должны инстанцировать объект EvenNumbers (класс коллекции). Затем в инструкции foreach вызывается метод именованного итератора:

```
EvenNumbers en = new EvenNumbers();
foreach(int even in en.DescendingEvens(nTop, nStop)) ...
```



Если бы DescendingEvens() был статической функцией, можно было бы обойтись без экземпляра класса. В этом случае ее можно было бы вызвать с использованием имени класса, как обычно:

```
foreach(int even in EvenNumbers.DescendingEvens
(nTop,nStop))...
```

Поток идей для потоков объектов

Теперь, когда вы можете сгенерировать "поток" четных чисел таким образом, подумайте о массе других полезных вещей, потоки которых вы можете получить с помощью аналогичных "коллекций" специального назначения: потоки степеней двойки, членов арифметических или геометрических прогрессий, простых чисел или чисел Фибоначчи — да что угодно. Как вам идея потока случайных чисел (чем, собственно, и занимается класс Random) или сгенерированных случайным образом объектов?



Если вы помните демонстрационную программу PriorityQueue из главы 15, "Обобщённое программирование", то можете взглянуть на другую демонстрационную программу — PackageFactoryWithIterator — на прилагаемом компакт-диске. В ней проиллюстрировано использование блока итератора для создания потока сгенерированных случайным образом объектов, представляющих пакеты. Для этого применяется та же функция, что и в классе PackageFactory в демонстрационной программе PriorityQueue, но содержащая блок итератора.

Итерируемые свойства

Можно также реализовать блок итератора в виде *свойства* класса.— конкретнее, в функции доступа `get()` свойства. Вот простой класс со свойством `DoubleProp`. Функция доступа `get()` этого класса работает как блок итератора, возвращающий поток значений типа `double`:

```
// PropertyIterator - демонстрирует реализацию функции
// доступа get свойства класса как блока итератора
class PropertyIterator
{
    [double] doubles = { 1.0, 2.0, 3.5, 4.67 };
    // DoubleProp - свойство "get" с блоком итератора
    public System.Collections.IEnumerable DoubleProp

        get
        {
            foreach(double db in doubles)
            {
                yield return db;
            }
        }
}
```

Заголовок `DoubleProp` пишется так же, как и заголовок метода `DescendingEvens()` в примере именованного итератора. Он возвращает интерфейс `IEnumerable`, но в виде свойства, не использует скобок после имени свойства и имеет только функцию доступа `get()`, но не `set()`. Функция доступа `get()` реализована как цикл `foreach`, который итерирует коллекцию и применяет стандартную инструкцию `yield return` для поочередного возврата элементов из коллекции чисел типа `double`.

Вот как это свойство можно использовать в функции `Main()`:

```
// Инстанцируем класс "коллекции" PropertyIterator
PropertyIterator prop = new PropertyIterator();
// Итерируем ее: генерируем значения типа double по одному
foreach (double db in prop.DoubleProp)
{
    Console.WriteLine(db);
}
```



Вы можете использовать *обобщенные итераторы*. Подробнее о них можно узнать из справочной системы, из раздела, посвященного применению итераторов.

Где надо размещать итераторы

В небольших классах итераторов специального назначения в демонстрационной программе `IteratorBlocks` коллекции размещались *внутри класса итератора*, как, например, в `MonthDays`. В некоторых случаях это вполне корректно, например, когда коллекция похожа на класс `SentenceChunks`, возвращающий части текста, или `DescendingEvens`, который возвращает вычисляемые значения. Но что, если вы хотите

предоставить итератор, основанный на блоке итератора для реального класса коллекции, например такого, как `LinkedList`?



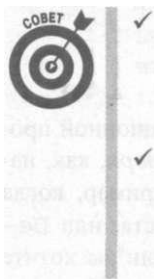
Эта задача решается в демонстрационной программе `LinkedListWithIteratorBlock` на прилагаемом компакт-диске. В ней класс `LinkedList` переписан и использует метод `GetEnumerator()`, реализованный как блок итератора. Он полностью заменяет старый класс `LinkedListIterator`. В приведенном далее листинге представлена только новая версия `GetEnumerator()`. Полностью демонстрационную программу можно найти на прилагаемом компакт-диске.

```
// LinkedListWithIteratorBlock - реализует итератор для
// связанного списка в виде блока итератора
class LinkedList // ": IEnumerator" больше не требуется
{
    ... Остальная часть класса
    // GetEnumerator - реализован как блок итератора
    public IEnumerator GetEnumerator()
    {
        // Проверяем действительность текущего узла. Если он
        // null, он еще не использовался, так что его надо
        // установить так, чтобы он указывал на голову
        // связанного списка
        if(currentNode == null)
        {
            currentNode = head;
        }
        // Здесь выполняются итерации перечислителя,
        // возвращаемого методом GetEnumerator()
        while(currentNode != null)
        {
            yield return currentNode.Data;
            currentNode = currentNode.forward;
        }
    }
}
```

Такой базовый вид блока итератора уже встречался ранее в этой главе:

```
public System.Collections.IEnumerator GetEnumerator() {}
```

Это выглядит точно так же, как и объект `IEnumerator`, который метод `GetEnumerator()` возвращает в исходном классе `LinkedList`. Однако реализация метода `GetEnumerator()` теперь работает совершенно иначе.



✓ Когда вы пишете блок итератора, C# создает для вас скрытый класс `LinkedListIterator`. Вы не пишете этот класс и не видите его код. Он не является частью демонстрационной программы `LinkedListWithIteratorBlock`.

✓ В методе `GetEnumerator()` вы просто используете цикл для обхода всех узлов связанного списка и возврата с помощью `yield return` элементов данных, хранящихся в каждом узле. Этот код приведен в предыдущем листинге.

- ✓ Вам больше не нужно определять ваш класс коллекции, как реализующий `IEnumerator`, что видно из приведенного в листинге заголовка класса.

Все не так просто

При этом нельзя забывать о некоторых неизбежных вещах.

- ✓ Вы должны убедиться, что начинаете обход с начала списка. Для этого в новый класс `LinkedList` добавлен член-данные `currentNode`, посредством которого отслеживается перемещение итератора по списку. Изначально член `currentNode` равен `null`, так что итератор должен проверять это условие. Если это так, он устанавливает `currentNode` таким образом, чтобы тот указывал на голову связанного списка.

Если только `head` не равен `null` (связанный список не пуст), то `currentNode` становится ненулевым до конца итераций. Когда же он достигает конца списка, итератор должен вернуть `null`, что послужит сигналом о прекращении работы для цикла `foreach`.

- ✓ При каждом шаге по списку необходимо осуществлять все действия, которые выполнялись ранее функцией `MoveNext()` по перемещению к следующему узлу:

```
// Действия, выполнявшиеся ранее MoveNextO
while(currentNode != null)
{
    // То, что делало свойство Current
    yield return currentNode...; // Часть кода опущена
    currentNode = currentNode.forward;
}
```

Большинство реализаций блоков итераторов используют цикл для прохода по коллекции— а иногда даже внутренний цикл `foreach` (но пример `String-Chunks` показывает, что это не единственно возможный путь).

- ✓ Когда вы проходите по списку и начинаете возврат данных с помощью `yield return`, вы должны "выковырять" хранящиеся данные из объекта `LLNode`. Узел связанного списка— это всего лишь корзина для хранения `string`, `int`, `Student` и т.п. объектов. Поэтому вы должны вернуть не `currentNode`, а сделать следующее:

```
yield return currentNode.Data; // Вот теперь верно
currentNode = currentNode.forward;
```

То же, но за сценой, делает и исходный перечислитель. Свойство `Data` класса `LLNode` возвращает данные в узле как `Object`. Исходный необобщенный связанный список преднамеренно спроектирован обобщенным настолько, насколько это возможно, поэтому он и хранит объекты класса `Object`.

Теперь цикл `while` с инструкцией `yield break` выполняет то, что ранее вы должны были делать с огромным количеством работы. В результате метод `GetEnumerator()` работает в цикле `foreach` в функции `Main()`, как и ранее.

Если вы немного поразмышляете над этим, то поймете, что такая реализация просто перемещает функциональность старого класса итератора `LinkedListIterator` в класс `LinkedList`.

За сценой цикл `foreach` выполняет за вас необходимые приведения. Так что если вы храните в списке строки `string`, ваш цикл `foreach` ищет именно их:

```
foreach(string s in llc) // foreach выполняет
{                          // приведение типов
    Console.WriteLine(s);
}
```



Обобщенная версия связанного списка из этой главы с блоком итератора использована в демонстрационной программе `GenericLinkedListContainer` на прилагаемом компакт-диске. В этой демонстрационной программе инстанцируется обобщенный класс `LinkedList` для объектов типа `string`, а затем — типа `int`. Чтобы лучше понять, как все это работает, стоит пошагово пройти цикл `foreach` в отладчике. Для сравнения вы можете познакомиться с новым встроенным классом `LinkedList<T>` в пространстве имен `System.Collections.Generic`.



Советую вам при работе полностью забыть о мире необобщенных коллекций — за исключением старого доброго массива, который может оказаться полезен и при этом безопасен с точки зрения типов. Воспользуйтесь лучше обобщенными коллекциями. Правда, лично я собираюсь не следовать эму совету уже в следующем разделе...

Осталось еще немного...

Реализация исходного итератора в `LinkedList` реализует итератор как отдельный класс, спроектированный для работы с классом `LinkedList`. У такого решения есть одна привлекательная возможность, которая отсутствует у решения с использованием блока итератора. Вы можете легко создать несколько экземпляров итераторов и применять каждый из них независимо от других. Так что `iterator1` может пройти потому, когда `iterator2` только начинает обход.



Последняя демонстрационная программа решает этот вопрос (хотя и для более лее простой коллекции, не `LinkedList`). `IteratorBlockIterator` использует объект итератора с доступом к внутреннему устройству коллекции, но сам этот объект реализован с применением блока итератора.

```
// IteratorBlockIterator - реализует отдельный объект
// итератора для работы с классом коллекции, типа
// LinkedList, но сам итератор реализуется с использованием
// блока итератора
using System;
using System.Collections;
namespace IteratorBlockIterator
{
    class Program
    {
        // Создание коллекции и использование двух объектов
        // итератора для независимого итерирования (каждый
        // использует блок итератора)
        static void Main(string[] args)
        {
            string[] strs = new string []
```

```

        { "Joe", "Bob", "Tony", "Fred" };
MyCollection mc = new MyCollection(strs);
// Создание первого итератора и начало итераций
MyCollectionIterator mci1 = mc.GetEnumerator();
foreach (string si in mci1) // Первый итератор

    // Какая-то работа со строками
    Console.WriteLine(si);
    // Ищем босса Тони
    if(si == "Tony")
    {
        // В середине этой итерации начинаем новую с
        // использованием второго итератора
        MyCollectionIterator mci2 = mc.GetEnumerator();
        foreach (string s2 in mci2) // Второй итератор

            // Работа со строками
            if(s2 == "Bob")
            {
                Console.WriteLine("\t{0} - босс {1}", s2, si);
            }
    }
}
// Ожидаем подтверждения пользователя
Console.WriteLine("Нажмите <Enter> для " +
    "завершения программы...");
Console.Read();
}
}
// Простая коллекция строк
public class MyCollection
{
    // Реализация коллекции с использованием ArrayList
    // internal - так что объекты итераторов могут
    // обращаться к строкам
    internal ArrayList list = new ArrayList();
    public MyCollection(string[] strs)

        foreach(string s in strs)
        {
            list.Add(s);
        }
}
// GetEnumerator - как и в LinkedList, возвращает один
// из объектов итераторов
public MyCollectionIterator GetEnumerator()
{
    return new MyCollectionIterator(this);
}
}
// MyCollectionIterator - класс итератора для MyCollection
public class MyCollectionIterator
{

```

```

// Храним ссылку на коллекцию
private MyCollection mc;
public MyCollectionIterator(MyCollection mc)
{
    this.mc = mc;
}
// GetEnumerator - блок итератора, который выполняет
// реальные итерации для объекта итератора
public System.Collections.IEnumerator GetEnumerator()
{
    // Итерируем список связанной коллекции, который
    // доступен, потому что объявлен как internal
    foreach (string s in mc.list)
    {
        yield return s; // Сердце блока итератора
    }
}
}
}

```

Коллекция в `IteratorBlockIterator` представляет собой простой класс-оболочку вокруг класса `ArrayList`. Его метод `GetEnumerator()` просто возвращает новый экземпляр сопутствующего класса итератора, такого же, как для `LinkedList`.

```

// GetEnumerator - как и в LinkedList, возвращает один
// из объектов итераторов
public MyCollectionIterator GetEnumerator()
{
    return new MyCollectionIterator(this);
}

```

Однако внутри самого класса итератора все гораздо интереснее. Он также содержит метод `GetEnumerator()`. Реализованный с применением блока итератора, он выполняет всю работу по итерированию. Вот этот метод:

```

// GetEnumerator - блок итератора, который выполняет
// реальные итерации для объекта итератора
public System.Collections.IEnumerator GetEnumerator()
{
    // Итерируем список связанной коллекции, который
    // доступен, потому что объявлен как internal
    foreach (string s in mc.list)
    {
        yield return s; // Сердце блока итератора
    }
}

```

Данный метод имеет доступ к `ArrayList` из сопутствующей коллекции, так что его инструкция `yield return` может поочередно возвращать хранимые в коллекции строки.

Выигрыш от этих сложностей концентрируется в функции `Main()`, где создаются две копии объекта итератора. Цикл `foreach` для второго итератора вложен в цикл `foreach` для первого, что позволяет получить вывод программы наподобие приведенного:

```

Joe
Bob
Tony

```

```

    Bob - босс Tony-
Fred
// • - •

    Строка с отступом выводится вложенной итерацией.
    Вот как выглядят эти вложенные циклы в функции Main():
MyCollectionIterator mci1 = mc.GetEnumerator();
foreach (string si in mci1)    // Первый итератор

    // Какая-то работа со строками
    Console.WriteLine(si);
    // Ищем босса Тони
    if(si == "Tony")
    {
        // В середине этой итерации начинаем новую с
        // использованием второго итератора
        MyCollectionIterator mci2 = mc.GetEnumerator();
        foreach (string s2 in mci2)    // Второй итератор

            // Работа со строками
            if(s2 == "Bob")
            {
                Console.WriteLine("\t{0} - босс {1}", s2, si);
            }
        }
    }
}

    Впрочем, исходный итератор, с MoveNext() и Current, все равно остается более
гибким и простым...

```

Глава 21

Использование интерфейса Visual Studio

В этой главе...

- > Использование инструментария Visual Studio
- > Настройка рабочего места
- > Отладка программ

Естественно, для работы необходимо знание языка. Программист на С#, не знающий С# — нонсенс. Однако важно также знать и используемый инструментарий — в частности, пользовательский интерфейс пакета Visual Studio, который вы, вероятно, применяете в работе. В этой главе речь пойдет о том, как работать с Visual Studio.

Материал настоящей главы применим ко всем редакциям Visual Studio 2005, включая Visual C# Express. Большая часть времени при работе над консольными приложениями из этой книги была потрачена на работу со следующими четырьмя окнами Visual Studio:

- ✓ Solution Explorer и Class View;
- ✓ Editor;
- ✓ Help;
- ✓ Debugger.

Эти окна перечислены в "хронологическом" порядке, а не в порядке важности. Когда вы пишете большую программу, вы работаете с окном Solution Explorer, а затем вводите исходный текст С# в окне редактора, все время пользуясь окном справки, а также окнами Solution Explorer или Class View. После того как вы ввели программу, вы ищете ошибки в ней в окне отладчика. Пару раз в месяц вы обращаетесь еще к одному окну, за которым сидит кассир, но это окно не относится к Visual Studio.

Перед тем как приступить к работе, обычно настраивается расположение окон так, чтобы это было удобно программисту.



Разработка графических приложений Windows включает ряд дополнительных окон: Form Designer, Toolbox и Properties. Вкратце о них было рассказано в главе 1, "Создание вашей первой Windows-программы на С#".

Настройка расположения окон

Visual Studio организует свои различные инструменты в окна для лучшего использования наиболее ограниченного компьютерного ресурса — экрана монитора.

Состояния окон

Окно может находиться в одном из четырех состояний.

- ✓ Закрытое (Closed)
- ✓ Свободное (Floating)
- ✓ Закрепленное (Docked)
- ✓ Свернутое (Tabbed)

Эти состояния описаны в следующих разделах.

Закрытое окно

Закрытое окно — это окно, убранное с экрана. Единственный способ увидеть его вновь — воспользоваться подменю View, как показано на рис. 21.1. Наиболее часто используемые окна перечислены в середине меню View; менее распространенные — в подменю View^Other Windows. Некоторые отладочные окна доступны только из меню Debug в режиме отладки.

Свободные окна

Свободное окно выглядит парящим над рабочим столом Visual Studio, как показано на рис. 21.2. Свободное окно не является совершенно независимым. Например, его нельзя минимизировать или разместить за главным окном, но зато можно поместить "за пределы" окна Visual Studio, эффективно расширяя рабочий стол последнего.



Каждый режим Visual Studio имеет собственные настройки. У вас может быть одно расположение окон в режиме разработки программы, когда вы редактируете ее исходный текст и компилируете ее, и другое в режиме отладки, как будет описано позже в этой главе.

Закрепленное окно

Вы можете *закрепить* практически любое окно, выбрав его и воспользовавшись командой Window^Dockable. Закрепленное окно "хватается" за другое окно или рамку главного окна Visual Studio. Это состояние сохраняется и при изменении размеров окна Visual Studio — закрепленные окна цепко держатся за границы основного окна и изменяют свои размеры вместе с ним.

На рис. 21.3, например, окно Output закреплено в верхнем правом углу, а окно Error List — в нижней части окна. Перемещая границу между двумя окнами, вы автоматически изменяете размеры каждого из окон.

Свернутое окно

Скрытые закрепленные окна, или, говоря более точно, минимизированные, выглядят как тонкие закладки, закрепленные на внешних границах окна Visual Studio. На рис. 21.3 показаны окна Solution Explorer и Class View, свернутые в закладки у правой границы окна Visual Studio.

Разместите курсор мыши над такой закладкой, чтобы развернуть окно. Чтобы оно осталось в таком состоянии, воспользуйтесь кнопкой в правой верхней части окна с изображением канцелярской кнопки или оставьте его в автоматически скрываемом состоянии, что бывает удобным, но, правда, далеко не всегда.

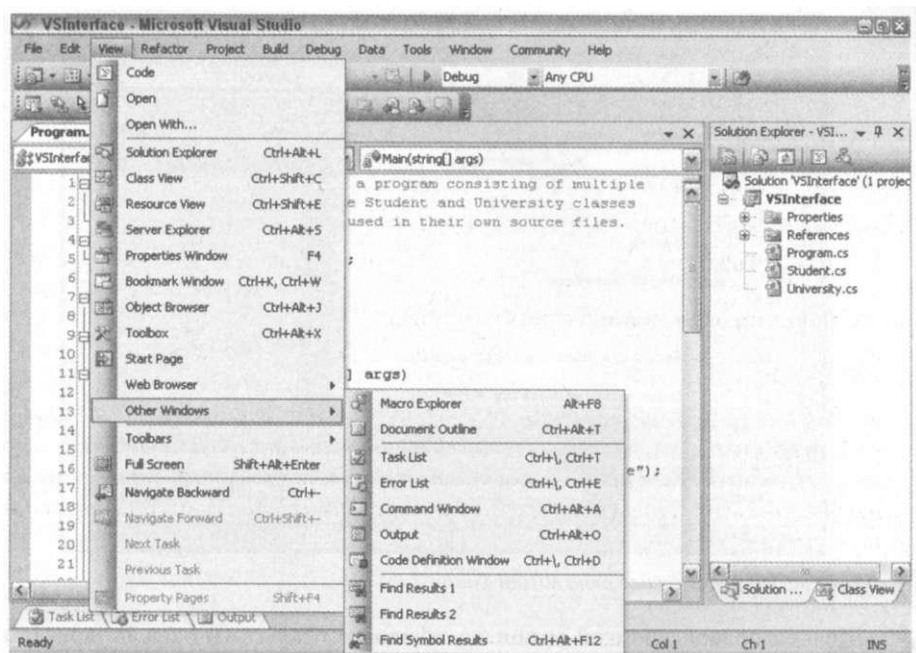


Рис. 21.1. Подменю *View* позволяет открыть все необходимые окна

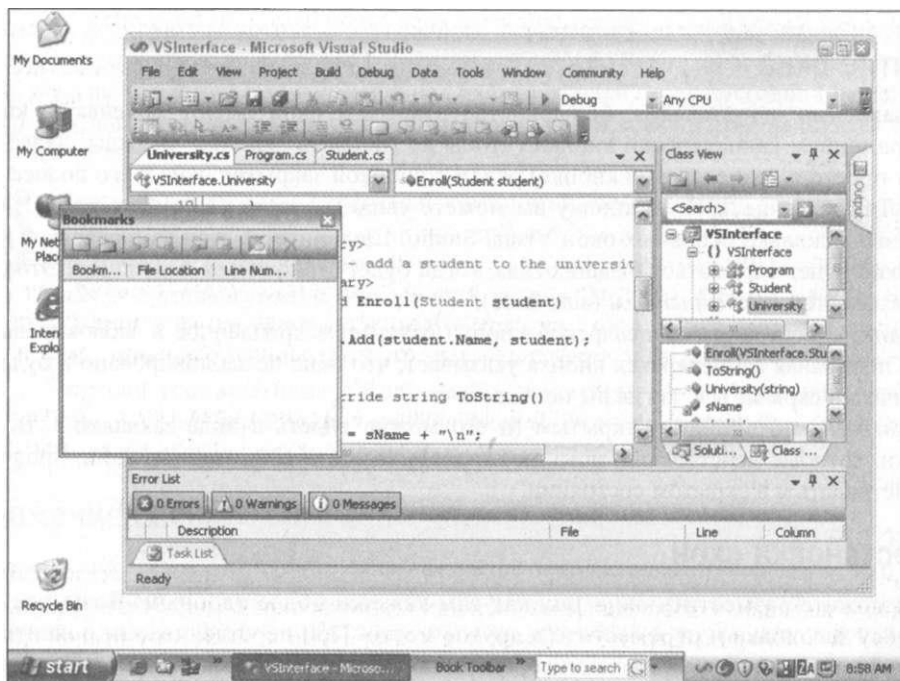


Рис. 21.2. Свободное окно выглядит независимым от Visual Studio

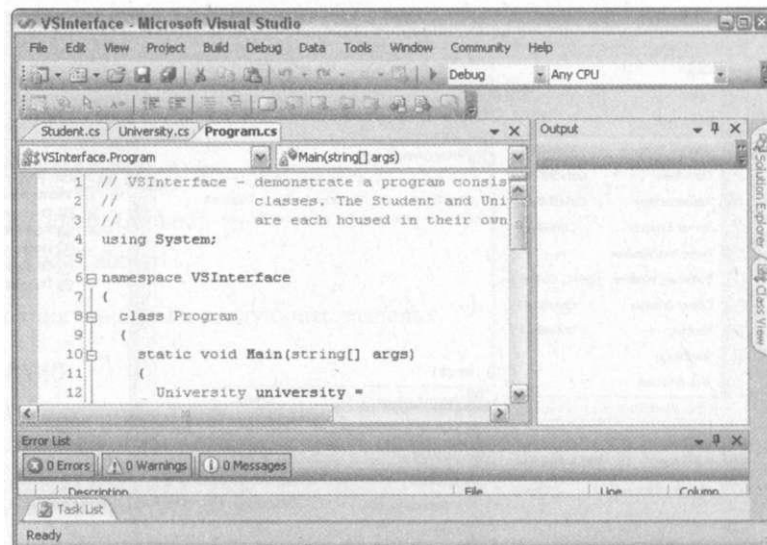


Рис. 27.5. Основные окна Visual Studio



Щелчок правой кнопкой мыши на заголовке открытого окна позволяет изменить его состояние— сделать его свободным, закрепленным, свернутым или скрытым.

Скрытие окна

Независимо от установок, большинство открытых окон имеют маленькую кнопку с изображением канцелярской кнопки (чтобы не говорить "кнопка" дважды, далее речь пойдет просто о канцелярской кнопке) рядом с кнопкой закрытия окна в его полосе заголовка. Такую канцелярскую кнопку вы можете увидеть у окна Output на рис. 21.3, где показаны несколько основных окон Visual Studio. Щелкните на этой канцелярской кнопке, и окно будет скрываться с ваших глаз, когда будет становиться ненужным. Это свойство называется *автоскрытием* (auto-hide).

Поднятое состояние канцелярской кнопки означает закрепленное и заблокированное окно. Опущенная канцелярская кнопка указывает, что окно не заблокировано и будет автоматически скрываться, когда вы покидаете его.

Скрытое окно остается открытым (и его можно видеть в виде закладки). Все настройки, которые действовали, пока оно находилось в открытом состоянии, продолжают действовать и в скрытом состоянии.

Перестановка окон

Вы можете разместить окна так, как вам кажется более удобным. Возьмите окно за полосу заголовка и переместите в другое место. При перетаскивании появится серое изображение окна, указывающее, где окно будет закреплено, если вы перенесете его в это место. На рис. 21.4 показано то же окно Visual Studio, что и на рис. 21.3, после того как окно Output было перемещено для закрепления в верхней части окна Visual Studio.

При перемещении окна можно использовать "направляющий ромб" в центре с четырьмя стрелками, направленными в разные стороны от центра.

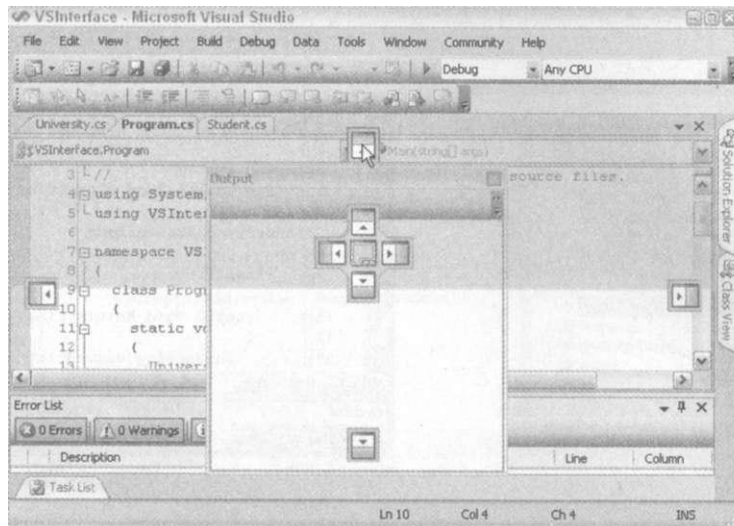


Рис. 21.4. Закрепленное окно можно перезакрепить в новом месте

Для того чтобы перетащить окно, его нужно взять за полосу заголовка, перенести к рамке, за которую вы хотите его закрепить, переместить указатель мыши на направляющую стрелку для этой стороны и отпустить его. Окно будет закреплено в данной позиции, если вы отпустите кнопку мыши над направляющей стрелкой (одной из центрального ромба или ее дубля у края окна — на рис. 21.4 указатель мыши находится как раз над таким дублем).



Расстановка окон — увлекательное занятие, чем-то похожее на игру (можно при этом вспомнить знаменитый кубик Рубика). Вам может потребоваться подправить несколько окон, чтобы достичь желаемого эффекта. Например, начав с конфигурации, показанной на рис. 21.3, вы можете перенести окно Output к левой границе, а окно Error List сместить в нижний правый угол, как показано на рис. 21.5. Чтобы окно Error List было закреплено у всей нижней границы окна Visual Studio, закрепите его за нижнюю рамку (на рис. 21.6 показана данная конфигурация). Экспериментируйте, пока не получите устраивающий вас результат.

Наложение окон

Перетаскивание и отпускание окна на центральном квадрате направляющего ромба позволяет складывать окна в "стопку" (центральная пиктограмма играет роль своеобразного клея). Каждое окно в такой стопке доступно при щелчке на вкладке, которая может быть сверху или внизу окна. На рис. 21.7 показана стопка окон редактирования, состоящая из трех окон — для файлов University.cs, Student.cs и Program.cs. Двойной щелчок на имени файла в Solution Explorer (о нем чуть позже) откроет окно с этим файлом так, что оно окажется верхним в стопке.

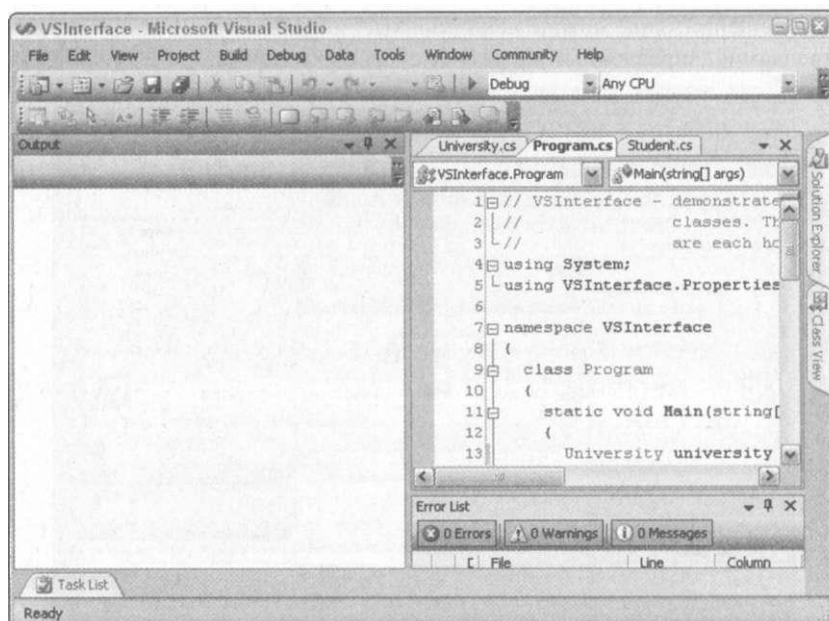


Рис. 21.5. Чтобы получить данную конфигурацию окон из конфигурации на рис. 21.3, требуется два шага. Еще один шаг— и вы получите конфигурацию, показанную на рис. 21.6

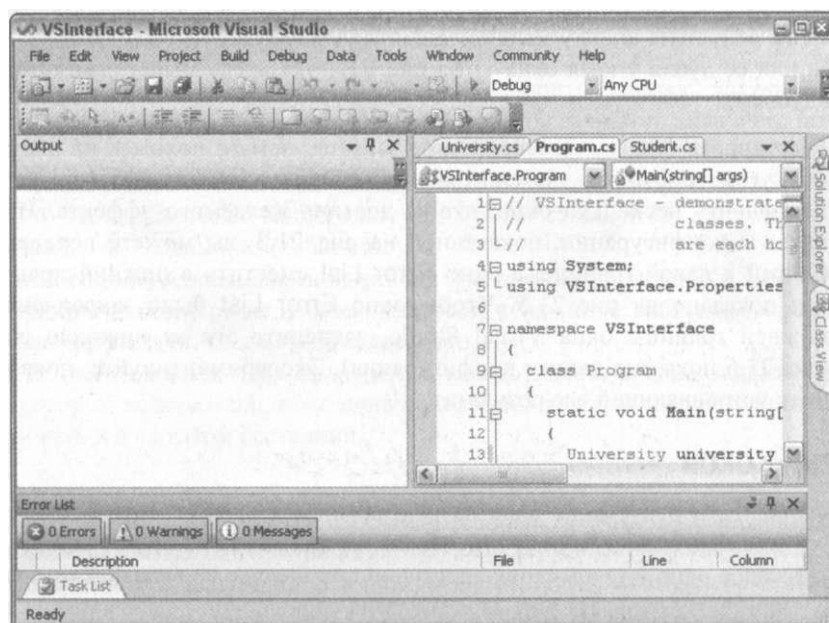


Рис. 21.6. Последовательное закрепление окон позволяет достичь желаемой конфигурации

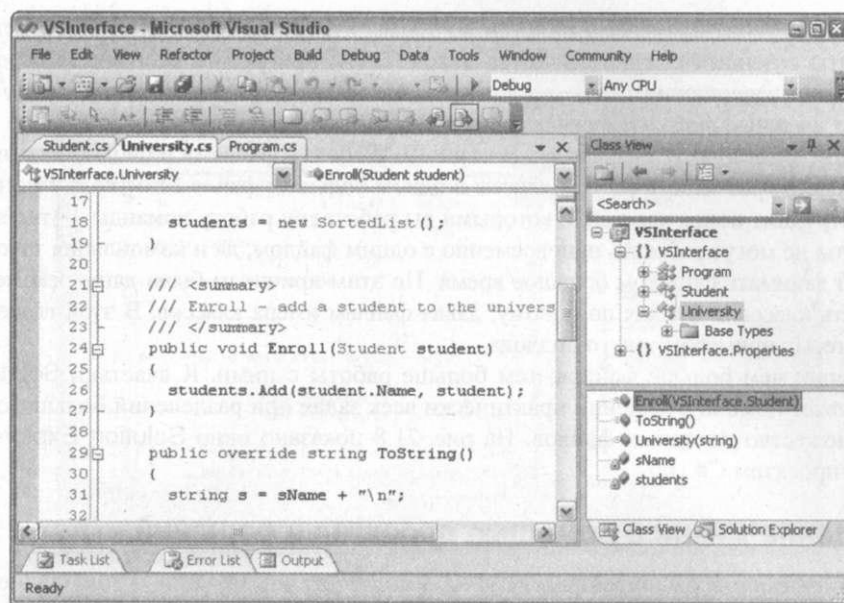


Рис. 21.7. Стопки окон помогают эффективно использовать рабочее пространство окна Visual Studio

Модные штучки



Щелкните правой кнопкой мыши в окне Solution Explorer и выберите пункт View Class Diagram. Visual Studio сгенерирует в проекте новый файл ClassDiagram1.cd. Вы можете открыть его и воспользоваться Class Designer для визуализации и работы со связями в вашей программе с использованием символики в стиле UML (подробнее об этом можно узнать из раздела "class diagram, presentation and documentation" справочной системы). Это не полнофункциональное средство для работы с UML-диаграммами, но оно может помочь визуализировать вашу программу и быть полезным при работе с кодом.



Чтобы познакомиться с другими модными штучками в Visual Studio 2005, обратите внимание на новое меню Refactor и команду Code Snippets Manager в меню Tools, а кроме того, обратитесь к разделу "What's New" справочной системы. Запомните эти комбинации клавиш: <Ctrl+K>, а потом — <Ctrl+X>.

Работа с Solution Explorer

Программа может состоять из любого количества исходных файлов C# — ну, скажем, из любого разумного количества. Несколько тысяч может оказаться слишком большим числом, хотя, вероятно, Visual Studio приходилось сталкиваться с подобным количеством при создании продуктов Microsoft.

"Ну и зачем создавать все эти файлы?" — спросите вы. Реальные программы могут быть очень большими, как уже говорилось в главе 19, "Работа с файлами и библиотеками". В этой главе рассматривалась система продажи авиабилетов, состоящая из многих

частей: интерфейса для заказа билетов по телефону, для работы через Интернет, часть для работы с ценами и налогами и так далее. Такие программы становятся огромным задолго до их завершения.

Такие сложные системы могут состоять из множества отдельных классов, по одному для каждого описанного интерфейса. В главе 19, "Работа с файлами и библиотеками", был предложено не размещать все эти классы в одном большом файле `Program.cs`, поскольку это затруднит поиск классов, с которыми вы работаете, работу команды — так как программисты не могут работать одновременно с одним файлом, да и компиляция такого файла станет занимать слишком большое время. По этим причинам были даны рекомендации размещать классы в файлах по одному, давая файлам имена классов. В этой главе вы познакомитесь с примером такого подхода.

Конечно, чем больше файлов, тем больше работы с ними. К счастью, `Solution Explorer` может помочь в решении практически всех задач при разделении большого проекта на множество исходных файлов. На рис. 21.8 показано окно `Solution Explorer` с открытым проектом `C#`.

Упрощение жизни с помощью проектов и решений

Файл проекта с расширением `.csproj` содержит инструкции о том, какие файлы входят в проект и как именно они должны быть скомбинированы. Именно с этим файлом вы и работаете посредством окна `Solution Explorer`.

Проекты могут объединять программы, которые зависят от одних и тех же пользовательских классов, как правило, сложные программы разделяются на несколько проектов, в совокупности составляющих одно *решение*. Пара стандартных сценариев организации проектов уже была описана в главе 19, "Работа с файлами и библиотеками": объединение программы записи файлов с программой чтения, или программа, которая разделена на код в выполняемом файле, и одна или несколько библиотек классов. В этих сценариях при изменениях в одном проекте остальные перекомпилировались автоматически. Программа записи файла описывалась одним проектом, программа чтения — другим. Аналогично, у вас был один проект для выполняемого файла, и другой — для библиотеки. Набор проектов называется в `Visual Studio` *решением* (файлы решений имеют расширение `.sln`).

Проект описывает не только исходные файлы, которые должны быть собраны вместе в одну программу. Файл проекта включает такие свойства, как, например, имя программы и аргументы, передаваемые ей при запуске из `Visual Studio`.



Каждая программа, независимо от ее размера, описывается решением `Visual Studio`, содержащим как минимум один проект. Чтобы увидеть пример много-проектного решения, обратитесь к решению демонстрационной программы `ClassLibrary` на прилагаемом компакт-диске. Это решение содержит два проекта, один — для небольшой тестовой программы, или "драйвера", и второй — для простой библиотеки классов. Эта программа также рассматривалась в главе 19, "Работа с файлами и библиотеками".

В мире имеются миллионы программ. В следующем разделе будет рассмотрена только одна демонстрационная программа `VSInterface`, определяющая класс `University` и класс `Student`. Каждый класс находится в своем собственном файле. Программа добавляет несколько объектов `Student` в `University`, а затем выводит результат.

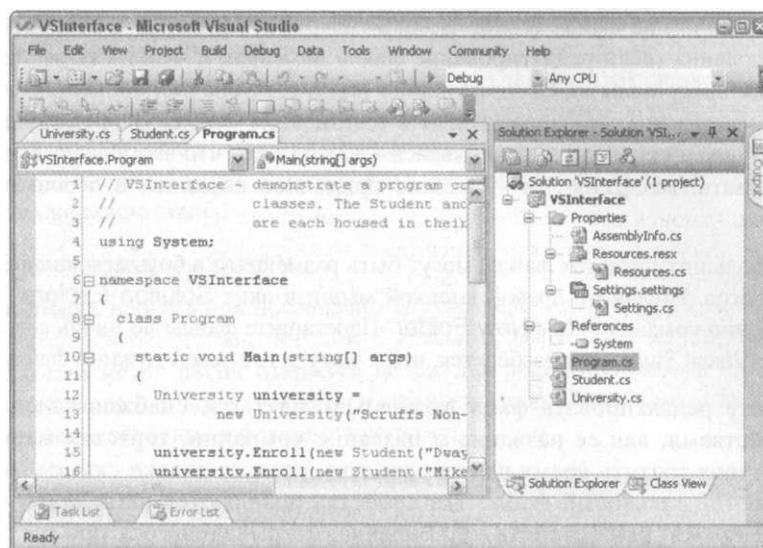


Рис. 21.8. Проект по умолчанию содержит шаблонный класс *Program*, *cs*, выделенный в окне **Solution Explorer**

Отображение проекта

Перечисленные далее шаги создадут схему приложения по умолчанию для программы *VSInterface*.

1. Выберите команду меню **File** → **New** → **Project**.
2. Выберите пиктограмму **Console Application**.
3. Введите имя **VSInterface** и щелкните на кнопке **ОК**.

Выберите команду меню **View** → **Solution Explorer** для того, чтобы увидеть файл проекта *VSInterface*, как показано на рис. 21.8. Таким образом, создано решение *VSInterface*, содержащее один проект с тем же именем *VSInterface*.

Изучение Solution Explorer

Окно **Solution Explorer** показывает две начальные подпапки, **Properties** и **References**. **Properties** содержит файл *AssemblyInfo.cs*, а также некоторые файлы "ресурсов" и "настроек". О настройках речь пойдет чуть позже, а что касается ресурсов, то тут достаточно будет сказать, что они содержат такие вещи, как изображения, пиктограммы, строки, входящие в пользовательский интерфейс (такие как сообщения в диалоговых окнах) и тому подобное. Подробнее о ресурсах можно узнать из справочной системы..

Подпапка **References** содержит все ссылки на внешние пространства имен, указанные с помощью команды меню **Projects** → **Add Reference**. Ссылки, добавленные в ваш проект, соответствуют вашим директивам `using` (включая некоторые "предположения" по умолчанию, которые могут не оправдаться для вашей программы; их можно удалить, если они вам не нужны). В окне также перечислены: исходный файл по умолчанию *Program.cs* и прочие исходные файлы, которые вы добавляете в проект в процессе работы. (Эти файлы немного отличаются для графических программ **Windows**, как вы могли видеть при создании программы, описанной в главе 1, "Создание вашей первой **Windows**-программы на C#".)

Двойной щелчок на файле в окне **Solution Explorer** приводит к открытию окна для его редактирования (если редактирование файла возможно). Файл `Program.cs` содержит трамплин программы на C# — функцию `Main()`. Конечно, большинство программ в этой книге написаны непосредственно в одном этом файле. В предыдущих версиях Visual Studio файл `Program.cs` назывался `Class1.cs`, что заставляло всякий раз его переименовывать. `Program.cs` — вполне приличное название, в переименовании не нуждающееся.



В больших проектах файлы могут быть размещены в большем количестве папках. Щелкните правой кнопкой мыши в окне **Solution Explorer** и выберите в меню команду **Add^New Folder**. Перетащите файлы во вновь созданную папку. Visual Studio сам разберется, что к чему, и где лежат нужные файлы.

Вы можете редактировать файл `AssemblyInfo.cs`, снабжая вашу программу такими свойствами, как ее название и название компании, торговая марка и номер версии. Не стоит тратить время на эти свойства, следует только сказать, что это всего лишь простой текстовый файл. Эти свойства появляются на вкладке **Version information** (Версия) окна свойств .EXE-файла, открывающегося при щелчке правой кнопкой мыши на файле в Проводнике Windows и выборе из раскрывающегося меню команды **Properties**.

Вы можете добавить в проект файлы любого вида. Обычно я храню здесь электронную таблицу или текстовый файл для собственных примечаний. Щелкните правой кнопкой мыши на проекте в окне **Solution Explorer** и выберите команду меню **Add^Existing Item**. Выберите добавляемый файл.

Свойства проекта



Щелкните правой кнопкой мыши на имени проекта в окне **Solution Explorer** и выберите команду **Properties**, чтобы открыть диалоговое окно свойств проекта. Здесь можно обновить ряд настроек проекта. Многие категории в окне **Properties** имеют имена, которые вполне проясняют их предназначение, но вы всегда можете обратиться к справочному материалу: поищите раздел "project properties" в справочной системе. Одна из наиболее полезных вкладок окна **Properties** — вкладка **Settings**. Определите ваши собственные настройки: только для чтения — **Application** и для чтения и записи — **User** — и вы сможете обратиться к ним из своей программы следующим образом:

```
// Эту строку надо добавить в раздел using
using MyAppName.Properties,-
// ... затем где угодно в вашей программе напишите что-то
// вроде:

// для получения значения настройки только для чтения:
string myString = Settings.Default.MyAppSetting;

// для установки значения настройки для чтения и записи:
Settings.Default.MyUserSetting = myUserString;
// Доступ к ресурсам осуществляется практически так же
```

Таким образом, оказывается очень просто сохранить пользовательские настройки без необходимости управления ими самостоятельно с помощью классов `System.IO`, описанных в главе 19, "Работа с файлами и библиотеками". В завершенной программе, ко-

тую вы распространяете, пользовательские настройки хранятся в персональных областях Application Data каждого пользователя. (Эта папка является частью персонального профиля каждого пользователя и обычно хранится в папке C:\Documents and Settings\

Навигация по исходному тексту



В больших программах достаточно трудно переходить от работы над методом А класса В к работе над методом С класса D. Вы можете найти определенный класс или метод путем открытия файла для редактирования и просмотра его содержимого или воспользоваться возможностью Class View, что значительно быстрее — особенно в больших проектах.

В Solution Explorer вы должны дважды щелкнуть на файле для того, чтобы открыть его, а затем прокрутить его или воспользоваться средствами поиска для того, чтобы найти искомый метод. Вы можете также применить два раскрывающихся списка поверх каждого окна редактора для поиска класса и его методов в текущем файле. Левый раскрывающийся список содержит классы текущего файла, а правый — члены выбранного класса. Я часто пользуюсь этими списками, но все же лично мне больше нравится использовать Class View.

Class View рассматривает программу не как множество файлов, а как множество классов и их членов, что позволяет данному средству быть незаменимым помощником при навигации по проекту. Щелкните на классе в верхней панели для того, чтобы увидеть список его членов (с сигнатурами параметров) в нижней панели. Это очень полезно — иметь возможность быстро вспомнить об аргументах и типе возвращаемого значения метода. Вернитесь к рис. 21.7, на котором показан результат двойного щелчка на классе University в окне Class View — открыт соответствующий исходный файл; двойной щелчок на методе Enroll() показывает этот член в открытом исходном файле. (Посредством правого щелчка на классе в окне Class View отображается раскрывающееся меню с некоторыми интересными возможностями.)



Держите окна Solution Explorer и Class View в закрепленном наложенном состоянии, чтобы быстро переключаться между ними с помощью одного щелчка на вкладке.

Добавление класса

Размещать каждый класс в отдельном файле, да еще так, чтобы имя файла совпадало с именем класса — хорошая программистская привычка. Подклассы могут находиться либо в своих собственных файлах, либо в файле базового класса, в зависимости от того, насколько тесно они связаны друг с другом.

Классы School и Student определенно следует разместить в разных файлах. Точно так же нужно разделить классы HighSchool, University и School, поскольку концептуально они достаточно далеки друг от друга. Вместе в одном файле лучше разместить классы наподобие LesserCanadianGoose и GreaterCanadianGoose.



Рис. 21.9. Добавление нового класса в проект с использованием окна *Add New Item*

Для того чтобы добавить класс `University` в программу `VSInterface`, выполните следующие шаги.

1. Щелкните правой кнопкой мыши на имени проекта `VSInterface` в окне `Solution Explorer`, а затем выберите команду меню `Add Add New Item`.
В появившемся окне вам будет предложена масса шаблонов объектов на выбор. Их слишком много, чтобы поместиться на одном рисунке!
2. Выберите `Class`, введите `University.cs` в поле `Name` в нижней части окна и затем щелкните на кнопке `Open`.

На рис. 21.9 показано окно `Add New Item` с выбранным шаблоном `Class`.

Содержимое нового исходного файла `University.cs` выглядит очень похоже на содержимое файла `Program.cs`, который строится по умолчанию при создании новой программы.

3. Повторите процесс для класса `Student`. После этого проект будет содержать файлы `Student.cs` и `University.cs` наряду с `Program.cs`.

Вернитесь к рис. 21.8, на котором показан результат выполнения указанных шагов. Три исходных файла представлены вкладками в окне редактирования.

Завершение демонстрационной программы



Данная версия класса `Student` создает и выводит, информацию о студенте, состоящую из идентификатора и имени.

```
// VSInterface - файл Student.cs
// Student - моделирование студента, который в состоянии сам
// написать свое имя
using System;
namespace VSInterface
```

```

{
    /// <summary>
    /// Student - учащийся школы
    /// </summary>
    public class Student
    {
        private string sStudentName;
        private int nID;
        public Student(string sStudentName, int nID)
        {
            this.sStudentName = sStudentName;
            this.nID = nID;
        }
        /// <summary>
        /// Name - имя учащегося
        /// </summary>
        public string Name { get { return sStudentName; } }
        /// <summary>
        /// ToString - возвращает имя и идентификатор
        /// </summary>
        public override string ToString()
        {
            return String.Format("{0} ({1})", sStudentName, nID);
        }
    }
}

```



Конструктор класса `Student` получает имя и идентификатор студента. Метод `ToString()` перекрывает версию базового класса по умолчанию `Object`. Эта пользовательская версия возвращает имя студента и его идентификатор в скобках. В главе 18, "Эти исключительные исключения", более подробно рассказано о перекрытии `ToString()`.

Класс `Student` включает комментарии документирования, помеченные как `///`. Такое документирование делает код более понятным, в особенности если классы распределены по нескольким файлам, и может оказаться полезным для других программистов (да даже и для самого автора исходного текста через некоторое время). Об использовании таких документирующих комментариев и генерации справочных файлов для ваших программ уже рассказывалось в главе 8, "Методы класса".



Вы можете перекомпилировать вашу программу даже до того, как введете исходный текст файла `Student.cs` — после того, как введете исходный текст файлов `University.cs` и `Program.cs`. Это неплохая идея — инкрементная разработка программ. Перекомпилируйте и исправляйте вашу программу до тех пор, пока компилятор не перестанет выводить сообщения об ошибках или предупреждения. Поступайте так для каждого класса или даже метода, пока не избавитесь от всех ошибок при компиляции.



Исходный текст файла `University.cs` столь же прост, как и его предшественник:

```

// VSInterface - файл University.cs
// University - простейший контейнер для студентов

```

```

using System;
using System.Collections;
namespace VSInterface
{
    /// <summary>
    /// University - учебное заведение
    /// </summary>
    public class University
    {
        private string sName;
        private SortedList students; // Словарь
        public University(string sName)
        {
            this.sName = sName;
            students = new SortedList();
        }
        /// <summary>
        /// Enroll - добавить студента в университет
        /// </summary>
        public void Enroll(Student student)
        {
            students.Add(student.Name, student);
        }
        public override string ToString()
        {
            string s = sName + "\n";
            s += "Список студентов:" + "\n";
            // Итерация по всем студентам университета с
            // использованием обычного перечислителя
            IEnumerator iter = students.GetEnumerator();
            while(iter.MoveNext())
            {
                object o = iter.Current;
                // Следующий подход не работает, потому что итератор
                // для SortedList возвращает записи словаря, которые
                // включают как студента, так и ключ:
                //
                // Student student = (Student)o;
                // // Работоспособен следующий вариант:
                // (обратите внимание на преобразование типов)
                DictionaryEntry de = (DictionaryEntry)o;
                Student student = (Student)de.Value;
                s += student.ToString() + "\n";
            }
            return s;
        }
    }
}

```

Данный файл указывает, что его содержимое является частью пространства имен VSInterface. Класс University состоит не более чем из имени и отсортированной коллекции студентов. Метод Enroll() добавляет объекты типа Student в SortedList с использованием имени студента в качестве ключа сортировки — другими словами, студенты хранятся в списке в отсортированном порядке.

Метод `University.ToString()` выводит название университета, гимн и имена всех студентов. Он делает это путем создания итератора, метод `MoveNext()` которого применяется для перехода от одного элемента списка к другому, получая каждый элемент посредством свойства `Current`. Поскольку класс `SortedList`, в котором хранятся студенты, представляет собой словарь, итератор возвращает не сохраненные объекты, а записи из словаря (объекты класса `DictionaryEntry`), содержащие объект вместе с ключом, применяемым для сортировки. Обратите внимание на использованные преобразования типов.



Коллекции и их итерирование описаны в главах 15, "Обобщенное программирование", и 20, "Работа с коллекциями". Здесь вместо синтаксиса `MoveNext()` можно применить блок итератора, рассмотренный в предыдущей главе, а можно воспользоваться циклом `foreach`:

```
foreach(DictionaryEntry de in students)
{
    Student student = (Student)de.Value;
    s += student.ToString() + "\n";
}
```

Как вы вскоре убедитесь, цикл `foreach` предпочтительнее.

Преобразование классов в программу

Классы `Student` и `University` не образуют программу. Консольное приложение начинается со статического метода `Main()`. Этот метод может быть в любом классе, однако по умолчанию он находится в классе `Program`.



Содержимое исходного файла `Program.cs` изменено следующим образом:

```
// VSInterface - демонстрационная программа, состоящая из
// нескольких классов. Классы Student и University находятся
// в своих собственных исходных файлах.
// Файл Program.cs
using System;
namespace VSInterface
{
    class Program
    {
        static void Main(string[] args)
        {
            University university =
                new University("Институт случайных наук");
            university.Enroll(new Student("Dwayne", 1234));
            university.Enroll(new Student("Mikey", 1235));
            university.Enroll(new Student("Mark", 1236));
            Console.WriteLine(university.ToString());
            Console.WriteLine();
            // Ожидаем подтверждения пользователя
            Console.WriteLine("Нажмите <Enter> для " +
                              "завершения программы...");
        }
    }
}
```

```

        Console.Read();
    }
}

```

При компиляции программы файл проекта говорит Visual Studio о том, что в ~~одну~~ программу следует объединить все три файла— `University.cs`, `Student.cs` и `Program.cs`.

При выполнении программы выводится простой (но отсортированный!) список студентов:

```

Институт случайных наук
Список студентов:
Dwayne  (1234)
Mark    (1236)
Mikey   (1235)

```

Press Enter to terminate...

Как должен выглядеть код

Программы в этой книге написаны так, чтобы максимально сэкономить бумагу. Дополнительные пустые строки опущены, код, который не представляет непосредственный интерес для рассматриваемой темы, зачастую тоже, а оставшийся в основном линейно организован. Должен ли любой ваш код выглядеть таким образом?

Далее приводится несколько рекомендаций по написанию кода, который легко читать (человеку), обновлять, сопровождать и тестировать, и который хорошо организован в концептуальном смысле.

- ✓ **Используйте дополнительные пробелы и пустые строки.** Избегайте излишнего сжатия кода, которое наблюдается в настоящей книге. Вот класс `Student`, переписанный в более свободном формате:

```

// VSInterface - файл Student.cs
// Student - моделирование студента, который в состоянии сам
// написать свое имя

#region Using Directives

using System;

#endregion Using Directives

namespace VSInterface
{
    /// <summary>
    /// Student - учащийся школы
    /// </summary>
    public class Student
    {
        #region Private Data Fields

        // Имена переменных начинаются со строчной буквы, но

```

```

// слова внутри имени переменной начинаются с прописных
// букв
private string sStudentName;
private int nID;

#endregion Private Data Fields

#region Constructors

// Student - конструктор и имя класса начинаются с
// прописной буквы, как и все слова внутри имени
public Student(string sStudentName, int nID)
{
    this.sStudentName = sStudentName;
    this.nID = nID;
}

#endregion Constructors

#region Public Methods and Properties

/// <summary>
/// Name - имя учащегося
/// </summary>
public string Name { get { return sStudentName; } }

/// <summary>
/// ToString - возвращает имя и идентификатор
/// </summary>
public override string ToString()
{
    return String.Format("{0} ({1})", sStudentName, nID);
}

#endregion Public Methods and Properties
}
}

```

✓ **Используйте директивы Visual Studio `#region` и `endregion` для отделения разделов вашего кода.** Это позволит сворачивать и скрывать разделы при работе над другими частями кода. Нажмите `<Ctrl+M>`, а затем `<Ctrl+0>`, чтобы переключиться между свернутым и развернутым состоянием. Давайте вашим разделам описательные имена, такие как показаны в приведенном листинге.

✓ **Используйте XML-комментарии, начинающиеся с `///`.** Данные символы позволяют Visual Studio применять их в механизме автозавершения, выводя комментарии как документацию по данному методу прямо в окне кода при вызове одного из ваших собственных методов. Механизм автозавершения рассматривался в главе 8, "Методы класса", и является одним из простейших способов получить справочную информацию о методе или классе — в том числе и по вашим собственным, если вы используете XML-комментарии.



Вы можете также воспользоваться инструментом NDoc с открытым кодом (<http://ndoc.sourceforge.net>) и автоматически сгенерировать привлекательную документацию в стиле Visual Studio на основании ваших XML-комментариев.

- ✓ **Комментируйте код, но делайте комментарии значимыми.** Хороший комментарий рассказывает о ваших намерениях и назначении кода, а не о механике их реализации. Например, не пишите так:

```
// Цикл по массиву студентов с вызовом метода Display для
// каждого объекта типа Student.
```

Вместо этого достаточно написать:

```
// Вывод информации о студентах.
```

- ✓ **Посмотрите на имена методов или классов, которые собираетесь комментировать, и подумайте, нельзя ли их переименовать так, чтобы комментарии стали излишни.** Метод `DisplayAllStudents()` не требует никаких комментариев.

- ✓ **Используйте хорошие описывающие имена для переменных, методов, классов и прочих объектов.** Начинайте имена методов с глаголов (`DisplayAllStudents()`), логические переменные или методы со слов наподобие *is* или *has* (`IsValid`, `hasItems`, `canPaste`), и делайте все имена понятными и значащими. Не используйте слишком длинных имен. Избегайте применения в именах аббревиатур, в особенности нестандартных.

- ✓ **Хотя в этой книге и используется венгерская нотация (см. главу 3, "Объявление переменных-значений"), существуют и другие соглашения по именованию.** Большинство программистов не используют венгерскую нотацию, в которой в качестве префикса применяется указание типа (наподобие *s* для `string`, *d* для `double` и так далее). В предыдущем примере использован другой стиль именования, который вы встречаете в большей части документации и примеров.

- ✓ **Пишите короткие методы, которые проще для понимания, менее подвержены ошибкам и легче тестируются.** Везде, где это возможно, работа метода должна использовать вызовы других методов. Это называется *разложением* вашего кода. Если категорически не требуется иного, делайте ваши методы закрытыми. Даже однострочный код стоит выделить в отдельный метод, если это делает код исходного метода понятнее. Предположим, например, что у вас есть сложное составное логическое выражение, наподобие

```
if((ypos == -1) & (vowelPos == -1)) ...
```

Его достаточно сложно понять с первого взгляда. Можно использовать комментарии для пояснения сути дела, но маленький метод с хорошим именем ничуть не хуже:

```
public bool HasNoVowels(int indexOfLetterY,
                        int indexOfFirstVowel)
{
    return (indexOfLetterY == -1) &
           (indexOfFirstVowel == -1);
}
```


Этот код (из небольшого переводчика на Pig Latin¹⁰, который я как-то писал) следует за кодом, который пытается найти первую гласную в целевом слове, если таковая существует. Если ее нет, `indexOfFirstVowel` принимает значение -1. Однако буква `y` также может рассматриваться как гласная в некоторых ситуациях, так что этот метод должен принимать во внимание и ее.

В методе, вызывающем `HasNoVowels()`, следующая строка гораздо проще для понимания, чем исходное логическое выражение:

```
if (HasNoVowels(ypos, vowelPos)) { return 'USE_WHOLE_WORD; }
```

Данный пример иллюстрирует *рефакторинг* (реорганизацию кода).

- ✓ **Пишите код, который открывает его предназначение.** Например, следующий метод, реализующий алгоритм преобразования английских слов на "пороссячью латынь" ("убрать буквы перед первой гласной, перенести их в конец слова и добавить 'ay'"), автоматически рассказывает о решаемой задаче даже без комментариев:

```
public string ConvertToPigLatin(string word)
{
    return GetBackPart(word) + GetFrontPart(word) + "ay";
}
```

Код написан на высоком уровне, с использованием имен методов, которые ясно указывают их предназначение, не детализируя, *как* именно они работают — с применением циклов, ветвлений и т.д. Легко увидеть, как минимум в общем, что делает каждый вызов метода. Исходная версия этого метода была полна конструкций `if`, циклов, сложных логических выражений и локальных переменных.

Алгоритм "пороссячьей латыни" прост, но некоторые его составные части несколько запутанны — как, например, поиск первой гласной для разбивки слова. Использование описанного стиля работает сверху вниз (от общего к частному), откладывая детали. Как можно предположить, методы `GetBackPart()` и `GetFrontPart()` написаны одинаково, с явным указанием намерений на каждом шагу и переносом деталей в подчиненные методы. Многие программы в этой книге можно улучшить посредством этого стиля, либо используя его изначально, либо прибегая к рефакторингу.

- ✓ **Можно снизить сложность еще больше, если создать вспомогательные классы, инкапсулирующие часть работы, вместо одного или двух классов, тянущих все на себе.** Всегда старайтесь инкапсулировать мелкие детали в классах или наборах методов. В частности, посмотрите, нет ли кода, который может измениться в будущем, и инкапсулируйте его в собственном классе. Моя любимая книга на эту тему — *Head First Design Patterns* Фриманов (Freeman) (O'Reilly, 2004).

Эти и подобные методы помогут вам справиться с величайшей проблемой программирования: управлением сложностью. Плотный, закрученный код трудно понимаем, а это — прямой путь к ошибкам.

¹⁰ "Пороссячья латынь" — искажение слов английского языка по определенным правилам; в чем-то аналог знакомого с детства "языка" в стиле "э-чи-то-чи дет-чи-ский-чи я-чи-зык-чи". — Примеч. ред.

Помогите мне!

Вряд ли вы обладаете настолько феноменальной памятью, чтобы помнить все классы и методы даже из одного пространства имен, скажем, `System`. Конечно, можно запомнить синтаксис `C#` и несколько других деталей, но все же лучше не забывать о том, как пользоваться справочной системой, поиск нужной информации в которой имеет несколько видов.



Окно справочной системы Visual Studio называется Document Explorer. Знание этого факта может помочь избежать определенной неразберихи.

F1

Помощь по клавише `<F1>` предоставляет быстрый доступ к информации о полях или конструкциях в существующем коде, которые вы плохо помните или не вполне понимаете.

Например предположим, что вам не понятна разница между оператором `new` и одноименным модификатором метода. Вы можете щелкнуть на слове `new` в любом месте в окне редактирования и нажать `<F1>`. Visual Studio откроет окно помощи, как показано на рис. 21.10. (Если вы не понимаете разницу между терминами `new`, см. в главе 6, "Объединение данных — классы и массивы", описание оператора `new`, а в главе 12, "Наследование", — наследования `new`. Или, как видно из приведенной копии экрана, имеется еще ограничение `new()` у обобщенных классов, так что можно заглянуть и в главу 15, "Обобщенное программирование".)

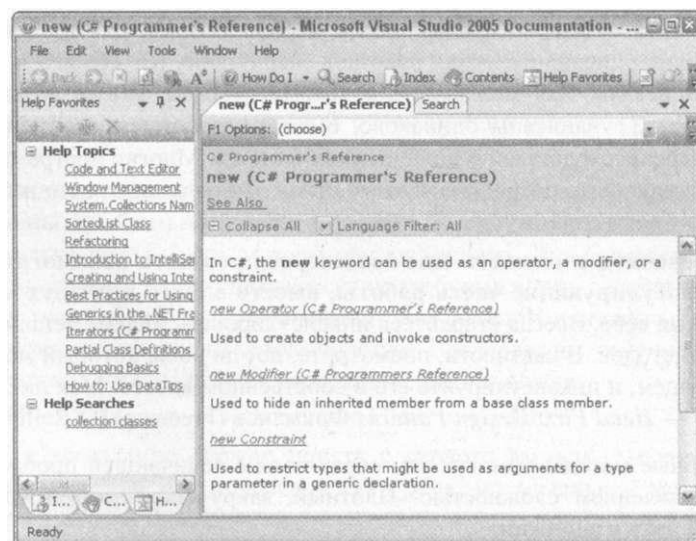


Рис. 21.10. Справочная система Visual Studio поможет разобраться с разными значениями ключевого слова `new`

Если справка содержит несколько статей, соответствующих вашему термину, вы увидите маленькое плавающее окошко с перечислением доступных тем. Дважды щелкните на нужной теме, чтобы увидеть ее.



Visual Studio пытается обеспечить доступ к справочным файлам, установленным на вашем компьютере, и к дополнительным ресурсам Web. Если вы не подключены к Интернету, то можете получить сообщение о том, что справочной системе не предоставляется доступ в Web.

Вы можете выбрать, где будет выводиться окно справочной системы. В Visual Studio выберите команду меню Tools^Options. В разделе Environment слева щелкните на пункте Help, General. Выберите External Help Viewer или Integrated Help Viewer из Show Help Using и щелкните на кнопке OK. Лично я предпочитаю External Help Viewer, когда справочная система запускается в виде отдельной программы и не мешает самому Visual Studio. Integrated Help Viewer помещает справку в свернутое окно вместе с вашими исходными файлами. Но попробуйте оба варианта и решите сами, что вам больше нравится.

Предметный указатель

Если справка <F1> — не то, что вам нужно, поскольку у вас нет соответствующего ключевого слова или идентификатора, вы можете продолжить поиск в предметном указателе (Index Help). Предметный указатель наиболее полезен, когда вы знаете тему, которая может вам помочь, но не уверены в деталях.

Например, вам может потребоваться коллекция некоторого вида, и при этом известно, что большинство классов коллекций находятся в пространстве имен System.Collections. Для поиска следует выбрать команду меню Help^Index, а затем в окне Index ввести **collections** в поле ввода Look For, что предоставит список тем, связанных со словом *collections*. (Этот список находится в левой части окна Help. В правой части выводится текст найденной вами темы.) Двойной щелчок на элементе .NET Framework в списке тем дает вам окно, показанное на рис. 21.11.

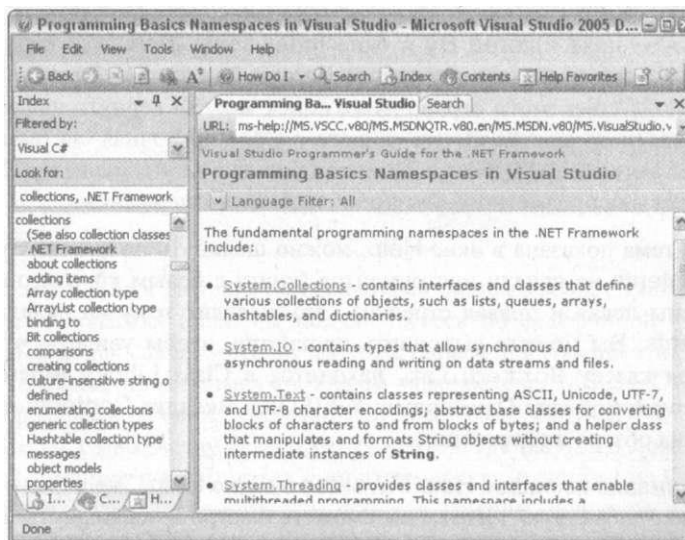


Рис. 21.11. Окно предметного указателя особенно полезно, если известна часть ответа на задаваемый вопрос

Затем следует щелкнуть на `System.Collections`, и эта тема открывает список членов пространства имен `Collections`. При прокрутке списка в нем можно найти класс `SortedList`. В соответствии с кратким описанием справа это именно то, что нужно. Итак, результаты поиска выглядят так, как показано на рис. 21.12.

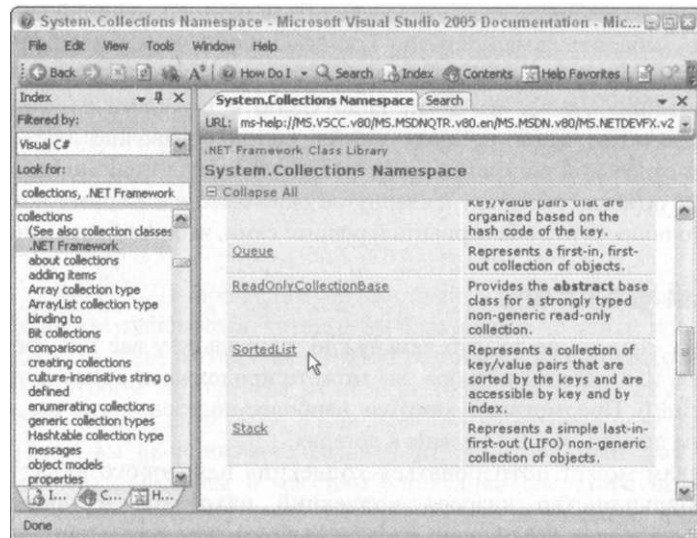


Рис. 21.12. Найдена информация по определенному классу

Каждый член слева в окне тем представляет собой гиперссылку. Щелчок на `SortedList` открывает информацию об этом классе, включая гиперссылки на члены класса, так что вы можете легко получить более детальную информацию.



Текстовое поле **Filtered By** в окне **Index Help** позволяет ограничить список тем, в которых выполняется поиск. На рис. 21.11 и 21.12 поиск велся в рамках "Visual C#". Без этого ограничения поиск мог бы вернуть информацию о коллекциях, не имеющих ничего общего с C#. Справочная система Microsoft Developer Network (MSDN) существенно больше, чем справка по C#. Фильтрация работает для предметного указателя, поиска и содержания.



Когда тема показана в окне **Help**, можно щелкнуть на кнопке **Sync with Table of Contents** на панели инструментов (кнопка с белым кругом, на котором изображены левая и правая стрелки). Это выделит тему на вкладке содержания **Contents**. Вы можете выполнить прокрутку, чтобы увидеть, что тема, посвященная классу `SortedList`, находится в **Class Library Reference** для .NET Framework Software Development Kit (SDK). Вкладка **Contents** полезна для получения обзора информации.

Обратите внимание на опцию **Help Favorites** в меню **Help**. Эта вкладка позволяет сохранить тему как "избранную". Позже вы сможете быстро к ней вернуться. На рис. 21.10 показано окно **Help Favorites** с некоторыми из избранных тем. Чтобы добавить текущую тему из предметного указателя в список избранного, щелкните на окне темы правой кнопкой мыши и выберите команду **Add to Help Favorites**.

Поиск

Опция **Search** в меню **Help** наиболее полезна, когда вы в точности не знаете, что именно вам нужно. Это полнотекстовый поиск по всем темам справочной системы.

Например, требуется коллекция, отсортированная в алфавитном порядке. Для поиска следует выбрать **HelpOSearch** для того, чтобы открыть вкладку **Search**. Но при вводе **sorted** в поле **Search For** полученные результаты оказываются не слишком полезными, так что лучше ввести **collection classes**.

На рис. 21.13 показаны результаты поиска для "collection classes". Если вы максимизируете окно **Help**, то увидите несколько закладок справа от окна: **Local Help**, **MSDN Online**, **Codezone Community** и **Questions**. По умолчанию вы получаете результаты поиска в локальных файлах.

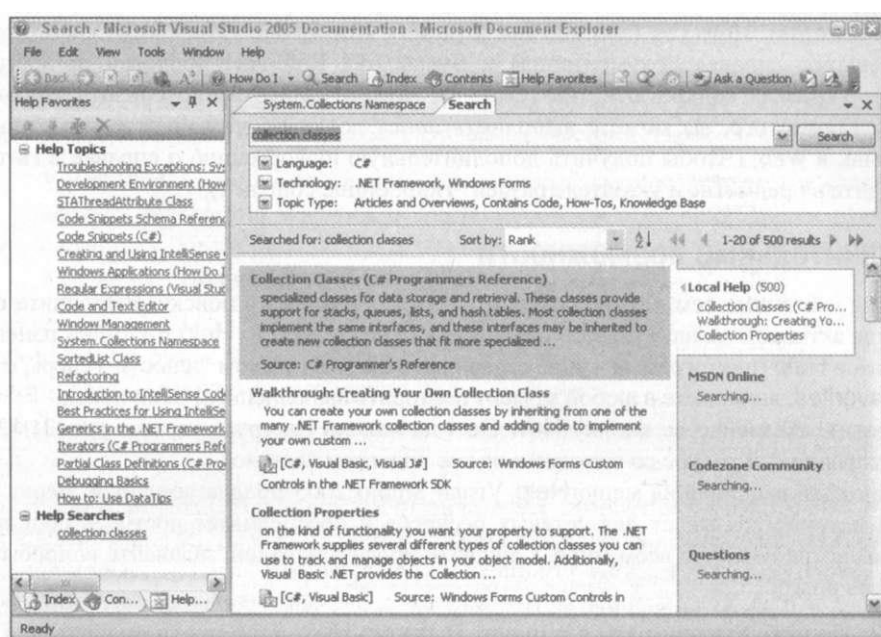


Рис. 21.13. Используйте полнотекстовый поиск, если вам не помогли ни контекстный поиск по <F1>, ни предметный указатель

Если вы подключены к Интернету, справочная система вернет также темы, расположенные в других областях (если вы подключаетесь по телефонной линии, это будет очень медленный поиск). Щелкните на закладке справа для вывода этих тем. **Local Help** обращается к файлам, хранящимся на вашем компьютере. **MSDN Online** обращается к справочным ресурсам на сайте Microsoft Developer Network (MSDN). **Codezone Community** обращается к множеству независимых сайтов, где вы часто можете найти дополнительную информацию и пообщаться с другими программистами на C# в форумах. Закладка **Questions** позволяет выполнить поиск в группах новостей, посвященных X# и вопросам, связанным с .NET.

(Чтобы получить советы о том, как составлять хорошие запросы, откройте справку, выберите **Help^Help on Help**, щелкните на **Techniques for Locating Help**, а затем на **Full-Text Searches**.)



Поиск может помочь найти класс или метод, который нужен вам для ваших целей, но при этом можно потратить много времени, продираясь через дебри ненужных тем.

Такой широкий поиск, как "collection class", возвращает сотни возможных тем (максимальное количество выводимых в окне — 500), так что вы получаете их так же, как страницы с результатами поиска в Web. Для перехода к следующей или предыдущей странице результатов поиска щелкните на стрелке в правом верхнем углу Local Help на вкладке Search. Большинство этих тем будут для вас бесполезны.

Как и в случае предметного указателя, можно улучшить полнотекстовый поиск с помощью фильтра. Можно фильтровать поиск по языку, технологии (такой как .NET Windows Forms или Office Applications) и типу темы. На рис. 21.13 установлен весьма широкий тип тем: Articles and Overviews (статьи и обзоры), Contains Code (с содержанием исходных текстов), How-Tos (краткие инструкции), Knowledge Base (базы знаний), Other Documentation (прочая документация) и Syntax/API Reference (справка по синтаксису/API). Указывая конкретный тип темы, вы можете существенно снизить количество мусора. Кроме того, вы можете выполнить поиск локально на вашем компьютере или глобально, в Web. (Чтобы получить дополнительную информацию о справке в Интернете, найдите в предметном указателе раздел "Help, online content".)

Дополнительные возможности

Кроме избранных тем, можно сохранить в Help Favorites и поиски. Выполните поиск, затем при активной вкладке Search щелкните на кнопке Add to Help Favorites панели инструментов Help (пиктограмма в виде странички с желтым знаком "плюс"). Теперь, открыв Help Favorites, вы можете в любой момент повторить выполненный вами поиск.

Обратите внимание на кнопку How Do I на панели инструментов на рис. 21.13. Это новый справочный ресурс со ссылками на все виды тем "how-to".

В качестве расширения меню Help Visual Studio 2005 предлагает новое меню Community, которое связывает ряд сетевых ресурсов и обеспечивает доступ к сообществу программистов на C# во всем мире. Слушайте профессионалов, задавайте вопросы и набирайтесь опыта...



Попробуйте поиграться с окном Dynamic Help. Оно предназначено для отражения контекста того, с чем вы работаете в данный момент — класс библиотеки .NET Framework, ключевое слово C# и так далее. Честно говоря, данное усовершенствование не такое уж и важное, хотя идея, конечно, привлекательная.



Лично я считаю наиболее важными и полезными возможностями справочной системы контекстную справку <F1> и предметный указатель. Старайтесь начинать с контекстной справки <F1>. Затем переходите к предметному указателю. Он напоминает предметный указатель книги. Если же и здесь вы не получили помощь, переходите к полнотекстовому поиску. Поиск похож... ну, на прогулку в Web, но не такую эффективную. И наконец, обратитесь к карте: вкладке содержания. Содержание похоже на оглавление книги. Это неплохое место, если вы хотите получить не напоминание, а обзор на какую-то тему.

Автоперечисление членов

"Автоперечисление членов" в Visual Studio часто делает излишним обращение к меню Help. При вводе имени класса или метода Visual Studio пытается предоставить вам справку на основании введенного во всплывающем окне.



Автоперечисление можно отключить. Выберите команду меню Tools^Options. В окне Options щелкните на пункте Text Editor в левой панели и выберите команду All Languages^General. И наконец, проверьте установку флага Auto List Members.

Чтобы увидеть, чем может помочь указанная возможность, рассмотрим знакомую ситуацию: я знаю, что класс коллекции некоторого типа хранит элементы в отсортированном порядке. Поскольку я знаю, что этот класс находится где-то в пространстве имен System.Collections, следует поместить курсор на начало пустой строки в редакторе исходного текста и ввести new System.Collections. Как только будет введена точка в конце "Collections", Visual Studio откроет меню, в котором перечислены все классы, составляющие пространство имен Collections. Это самый быстрый и простой вид помощи.



Visual Studio перечисляет в данной ситуации только неабстрактные классы, поскольку только они могут быть инстанцированы с использованием ключевого слова new. Подробнее об абстрактных и конкретных классах можно прочесть в главе 13, "Полиморфизм".

В прокручиваемом списке возможных классов находится и класс SortedList. После выбора класса Visual Studio открывает его описание, как показано на рис. 21.14. Похоже, этот класс — именно то, что нужно.

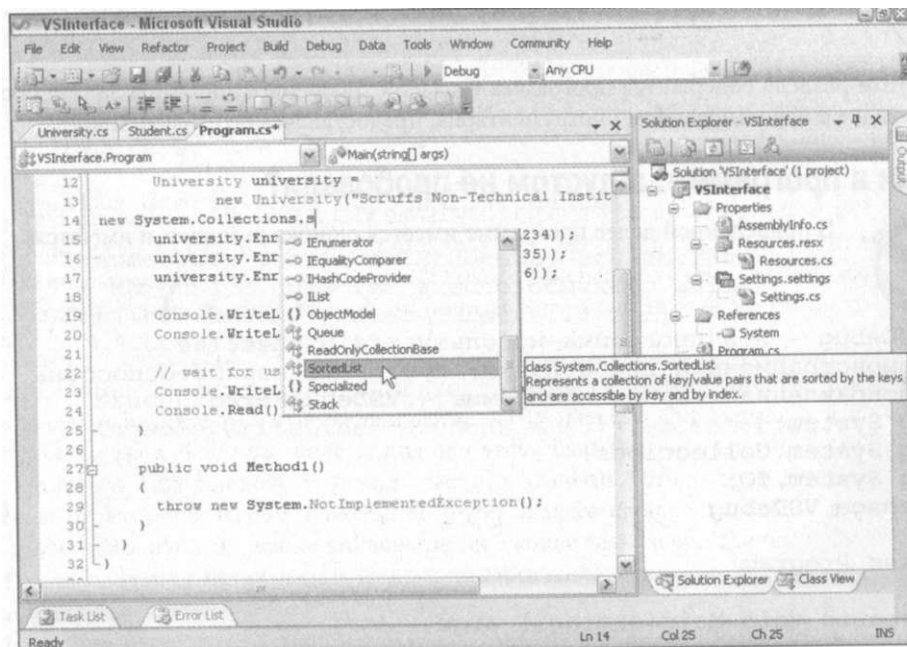


Рис. 21.14. Автоперечисление — мощное подспорье в работе программиста

После того как вы нашли то, что искали, можно удалить временный текстовый файл `new System.Collections.SortedList`.

При нормальном течении событий при вводе реального кода автоперечисление является частью автозавершения, о котором подробно рассказывалось в главе 8, "Методы класса".

Отладка

Программы в этой книге, не боясь этого слова, можно смело назвать безупречными. Но это результат определенного труда — нетривиальные программы никогда не работают с первого раза (наверняка это следствие определения тривиальной программы как таковой, которая корректно работает сразу же после создания).

Строгий синтаксис C# позволяет отловить массу ошибок. Например, пропущенная инициализация переменной перед ее использованием всегда была бичом для более ранних языков программирования. Теперь в C# невозможно допустить такую ошибку, так как он отслеживает, когда и где переменной впервые присваивается значение, и где эта переменная применяется. Если ее использование предшествует инициализации, C# бьет колокола. (Говорю честно — я пытался, но никак не могу придумать трюк, как создать программу, использующую неинициализированную переменную.)

Однако компилятор не в состоянии обнаружить все ошибки программиста (если бы это было так, программисты бы быстро удалили его со своих жестких дисков, чтобы не оставаться безработными). Всегда существует необходимость поиска и исправления ошибок времени выполнения.



В коммерческом программном обеспечении ошибки времени выполнения часто именуют *особенностями* программы.

В этом разделе содержится программа с массой "особенностей". Моя задача состоит в ее отладке с использованием инструментария, предоставляемого Visual Studio.

Жучки в программе: а дуством не пробовали?



В приведенной далее программе имеется ошибка (а может, и имеются).

```
// VSDebug - эта программа используется в качестве
// демонстрационной для отладки; программа неработоспособна
// (исправленная версия программы — VSDebugFixed)
using System;
using System.Collections;
using System.IO;
namespace VSDebug
{
    class Program
    {
        static void Main(string[] args)
        {
            // Я должен вставить это предупреждение, чтобы
```



```

        // избежать тысяч писем с указанием, что моя программа
        // нее работает
        Console.WriteLine("Эта программа не работает!");
        Student si = new Student("Student 1", 1);
        Student s2 = new Student("Student 2", 2);
        // display the two students
        Console.WriteLine("Student 1 = ", si.ToString());
        Console.WriteLine("Student 2 = ", s2.ToString());
        // Теперь требуем от класса Student вывести всех
        // студентов
        Student.OutputAllStudents();
        // Ожидаем подтверждения пользователя
        Console.WriteLine("Нажмите <Enter> для " +
            "завершения программы., .");
        Console.Read();
    }
}

public class Student
{
    static ArrayList allStudents = new ArrayList();
    private string sStudentName;
    private int nID;
    public Student(string sName, int nID)
    {
        sStudentName = sName;
        nID = nID;
        allStudents.Add(this);
    }
    // ToString - возвращает имя и идентификатор студента
    public override string ToString()
    {
        string s = String.Format("{0} ({1})",
            sStudentName, nID);
        return s;
    }
    public static void OutputAllStudents()
    {
        IEnumerator iter = allStudents.GetEnumerator();
        // Используя цикл for вместо обычного while
        for(iter.Reset(); iter.Current != null;
            iter.MoveNext())
        {
            Student s = (Student)iter.Current;
            Console.WriteLine("Student = {0}", s.ToString());
        }
    }
}
}

```

После удаления из программы всех ошибок времени компиляции можно переходить к делу.

Несмотря на то что все ошибки изгнаны, остались еще два предупреждения (они появляются в окне Error List с пиктограммой в виде желтого треугольника). Предупрежде-

ния означают потенциальные проблемы, которые недостаточно серьезны, чтобы считать их ошибками. Однако это не значит, что их можно игнорировать. Хорошенько изучите их, поскольку они могут указать на ошибки в вашем коде. Хотя, конечно, расшифровка сообщений компилятора порой труднее, чем расшифровка египетских иероглифов.



Просто для интересующихся — на прилагаемом компакт-диске имеется демонстрационная программа `VSDDebugGeneri.es`, в которой вместо хранения `allStudents` в `ArrayList` используется `List<T>`, метод `OutputAllStudents()` оказывается гораздо проще — и все это работает!

Пошаговая отладка

При обнаружении наличия ошибки в программе одним из наилучших первых шагов в ее локализации и устранении является возможность отладчика, известная под названием *пошагового выполнения* (single stepping). Для этого воспользуйтесь командой меню `Debug^Step Over` или клавишей `<F10>`.



Даже если вы не очень хорошо запоминаете функциональные клавиши, в этом случае следует сделать исключение. Использовать меню `Debug` для доступа к пошаговому выполнению `Step Over` и `Step Into` — непозволительно медленно (панель инструментов лучше меню, но и ей далеко до клавиатуры).

Нажатие `<F10>` приведет к выполнению демонстрационной программы `VSDDebug` до открывающей фигурной скобки функции `Main()`. Повторное нажатие `<F10>` выполнит функцию `Main()` до первой *выполнимой инструкции* (executable statement) — инструкции, которая что-то делает на самом деле. Комментарии и объявления такими инструкциями не являются, а вот `Console.WriteLine()` определенно осуществляет некоторые действия. На рис. 21.15 изображено окно Visual Studio после двукратного пошагового выполнения.

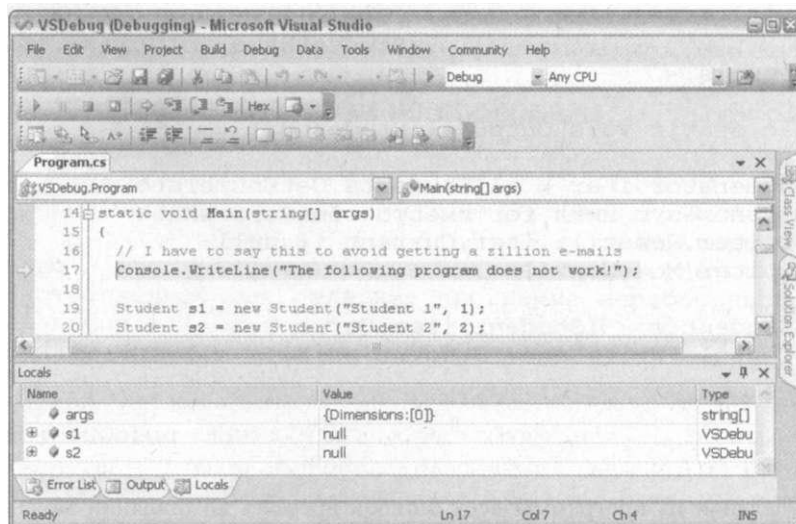


Рис. 21.15. Пошаговый режим приводит к выполнению программы по одной инструкции



Перед началом процесса отладки Visual Studio перекомпилирует программу, так что не нервничайте преждевременно.

Обратите внимание, что первая строка программы подцвечена. Это — очередная инструкция, которая будет выполнена в пошаговом режиме. Запомните — желтая строка еще не выполнена.

Обратите также внимание на окно, открытое внизу, с именем Locals (оно может находиться в минимизированном состоянии; на рис. 21.15 оно показано открытым).

В этом окне выводится список трех *локальных* переменных, т.е. переменных, объявленных в текущей функции. Переменные `si` и `s2` типа `VSDebug.Student` имеют значения `null`, поскольку им еще не были присвоены значения. Столбец `Type` предоставляет информацию о полном имени класса, включая пространство имен. Переменная `args` типа `string[]` (массив строк) с длиной 0 означает, что программе не были переданы аргументы. Это тоже очень важная возможность отладчика.

Еще одно нажатие <F10> приводит к выполнению вывода предупреждения функцией `WriteLine()`. Чтобы убедиться в этом, нажмите комбинацию клавиш <Alt+Tab> для того, чтобы переключиться на программу VSDebug. В консольном окне вы увидите одну строку — Эта программа не работает!. Это именно то, что и ожидалось. Еще раз нажмите <Alt+Tab> для возврата в Visual Studio.



<Alt+Tab> — команда переключения между программами Windows, используемая для передачи управления от одной программы к другой. Она "активизирует" главное окно программы, в которую вы переключаетесь. Когда активен отладчик, программа VSDebug выполняется, но находится в приостановленном состоянии. Клавиши <Alt+Tab> можно использовать где угодно, а не только в Visual Studio.

Естественно, очередное нажатие <F10> приводит к выполнению строки `Student si = ...`. Поток управления останавливается на следующей строке функции `Main()`, выполняя конструктор первого объекта `Student` за один шаг. Конструктор всегда выполняется, даже если вы этого и не видите.

В окне Locals переменная `s2` остается равной `null`, но переменная `si` теперь содержит объект класса `Student`. Небольшой знак "плюс" слева от `si` означает, что объект можно открыть и посмотреть на его "внутренности". После щелчка на этом значке окно Locals приобрело вид, показанный на рис. 21.16, раскрывая содержимое объекта `si`.

Первая и вторая записи в экземпляре объекта — члены `nID` типа `int` и `sStudentName` типа `string`. Третья запись — заголовок списка статических членов класса — в данном случае это единственный член данных `allStudents` типа `ArrayList`. Поскольку данный объект также имеет свое содержимое, слева от него находится маленький значок "плюс", позволяющий ознакомиться с этим содержимым.

Взгляните внимательнее на значения двух членов экземпляра: `sStudentName` имеет значение "`Student 1`", которое выглядит вполне корректно, а `nID` имеет значение 0, что корректным не назовешь. Это значение должно быть равно 1 — значению, переданному конструктору. Что-то в конструкторе пошло не так...

Беспокоясь о программировании на C# вообще и о своей карьере в частности, я выбираю команду меню `Debug<=>Stop Debugging`. Затем я нажимаю <F10> три раза: один раз для перезапуска программы, и два раза для того, чтобы пройти `WriteLine()` —

тут, похоже, все нормально работает. Но вместо того чтобы нажать <F10> еще раз и **вы**полнить конструктор, не заходя в него, я нажимаю <F11> и захожу в конструктор.

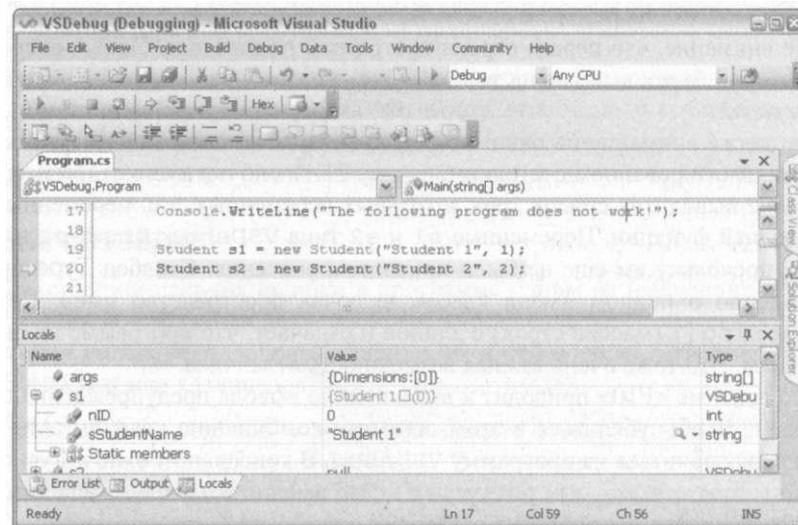


Рис. 21.16. Окно *Locals* позволяет получить детальную информацию о состоянии объекта



Действие клавиш <F10> и <F11> идентично, когда вы выполняете инструкцию, не являющуюся вызовом функции некоторого вида. Пошаговое выполнение посредством <F11> (step into) приводит к пошаговому выполнению вызываемой функции. Однако оба пошаговых режима не позволяют заходить в библиотечные функции .NET. Исходный текст реализации библиотеки закрыт для входа.

В этот раз в окне появляется конструктор с выделенной первой строкой. Далее следует открыть интересующий объект `this` в окне *Locals*. Затем несколько раз нажать <F10>, чтобы перейти к точке инициализации `nID`.



Каждое изменяемое значение в окне *Locals* выделяется красным цветом.

Весь в ожидании, я нажимаю <F10> еще раз. Интересно— значение `this.nID` не изменяется, несмотря на то что значение `nID` в окне *Locals* стало равным 1.



Если вы скомпилируете демонстрационную программу *VSDebug* с использованием команды `Debug^Build VSDebug` вместо применения клавиши <F10> для пошагового прохода в отладчике, в окне *Error List* вы увидите два упомянутых предупреждения. Если вы посмотрите на конструктор класса *Student*, то увидите волнистую пурпурную линию под присваиванием `nID = nID`, указывающую на проблему. Если бы вы сделали это перед тем, как переходить к отладке, возможно, вы бы смогли исправить ошибку, не прибегая к отладчику. К сожалению, эта линия не видна на рис. 21.16, так как на нем показан код в отладчике.

Вернемся к анализируемой строке. Следующее выражение просто присваивает значение `nID` самому себе:

```
nID = nID;
```

Это законно, но бесполезно и совсем не то, что требовалось. Вот что было нужно на самом деле:

```
this.nID = nID; // Присваивание аргумента переменной-члену
```



Можно прекратить отладку и перекомпилировать программу, после чего, поместив курсор над этим присваиванием, посмотреть еще раз на предупреждение во всплывающем окне. Оно гласит *"Assignment made to same variable; did you mean to assign something else?"* ("Выполняется присваивание той же переменной. Не намеревались ли вы выполнить иное присваивание?"). Можно ли выразиться понятнее?

Можно попробовать изменить строку на `this.nID = nID` и снова пошагово выполнить программу. (С некоторыми ограничениями вы можете также просто изменить исходный текст и продолжить отладку — эта возможность Visual Studio называется *Edit and Continue* — поищите информацию о ней в справочной системе.)

В этот раз следует аккуратно проверить объекты `s1` и `s2` в окне *Locals* после их конструирования, но, кажется, все выглядит хорошо. Однако пошаговое выполнение очередного вызова `WriteLine()` дает странный вывод на экран:

```
Эта программа не работает!  
Student 1 =
```

Что могло случиться на этот раз? Вероятно, `ToString()` ничего не возвращает. Необходимо начать сначала, так что пока что я прекращаю отладку.

Главное - вовремя остановиться

Вероятно, вы уже замаялись в очередной раз пошагово проходить программу. Пошаговый проход большой программы представляется вообще сплошным кошмаром.

Отладчик Visual Studio позволяет указать, что вы хотите остановить программу в ее конкретной точке. Это достигается путем создания так называемой *точки останова* (breakpoint).

Для этого нужно щелкнуть мышью на области слева от интересующей команды `WriteLine()`, где предполагается приостановить выполнение программы. Возле строки появляется маленький красный кружок, а сама строка подцвечивается красным цветом, что свидетельствует о наличии точки останова. Теперь можно указать программе, что она может начинать выполнение, посредством команды меню *Debug^Start* или клавиши `<F5>`. В результате ни о чем не нужно беспокоиться, зная, что программа остановится, дойдя до указанной строки.

Как и должно быть, программа начинает работу и останавливается в заданной точке, как показано на рис. 21.17. Обратите внимание на желтую стрелку, появляющуюся в красном кружке, и на подцвечивание инструкции `WriteLine()` желтым цветом.

Далее следует вновь три раза нажать `<F10>`, чтобы добраться до инструкции `WriteLine()`, которая выводит информацию о студенте 1, и затем — `<F11>`, чтобы попасть в метод `ToString()`.

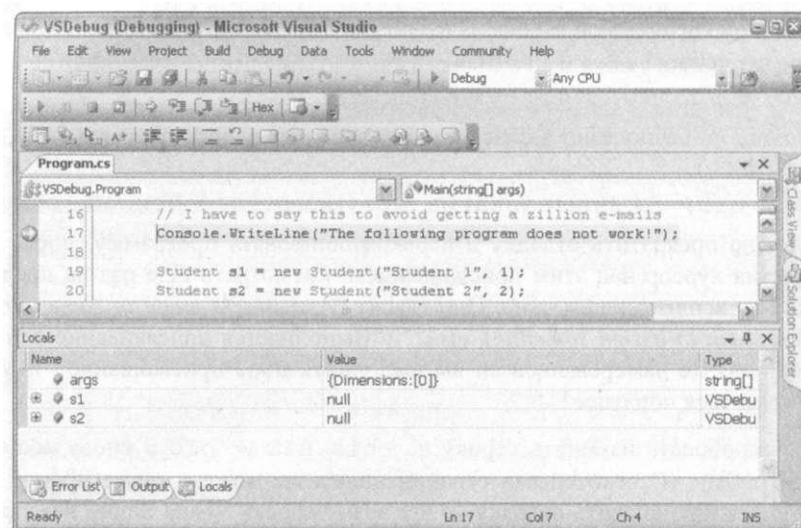


Рис. 21.17. Желтая стрелка указывает, где остановилась программа из-за наличия точки останова

Я не должен был передавать результат вызова `String.Format()` оператору `return`, как показано в следующей строке:

```
return String.Format("{0} ({1})", sStudentName, nID);
```

Вместо этого следовало бы переписать `ToString()` с использованием временной переменной `s`:

```
public override string ToString()
{
    string s = String.Format("{0} ({1})", sStudentName, nID);
    return s;
}
```



Присваивание возвращаемого значения промежуточной переменной дает возможность просмотреть его в отладчике. (Помимо этого, нет никаких иных причин поступать таким образом, так что после отладки можно удалить эту промежуточную переменную.)

Нажмите `<FU>` для пошагового выполнения строки, вычисляющей значение `s` — строки, возвращаемой функцией `ToString()`. В окне `Locals` все выглядит вполне корректно, как видно из рис. 21.18. (Примечание: `\t`, которое вы видите в строке 3, представляет собой символ табуляции. Я нажал `<Tab>` вместо пробела, когда вводил строку `String.Format`. На самом деле в этом нет ничего страшного. Вне отладчика выберите команду меню `Edit^Advanced^Show White Space` для того, чтобы вместо пробелов выводилась точка, а вместо символов табуляции — стрелочка. Отключить этот режим можно аналогичным способом.)

Неприятность найдена

Проблема должна заключаться в самом вызове `WriteLine()`. Следует дважды нажать `<F10>`, чтобы вернуться к этой строке. Ага! Управляющий элемент `{0}`, который

должен выводить строку, возвращаемую `ToString()`, отсутствует. То есть в функцию передано значение для него, но сам элемент забыт. Чтобы исправить ошибку, две команды `WriteLine()` надо переписать следующим образом:

```
// display the two students
Console.WriteLine("Student 1 = {0}", si.ToString());
Console.WriteLine("Student 2 = {0}", s2.ToString());
```

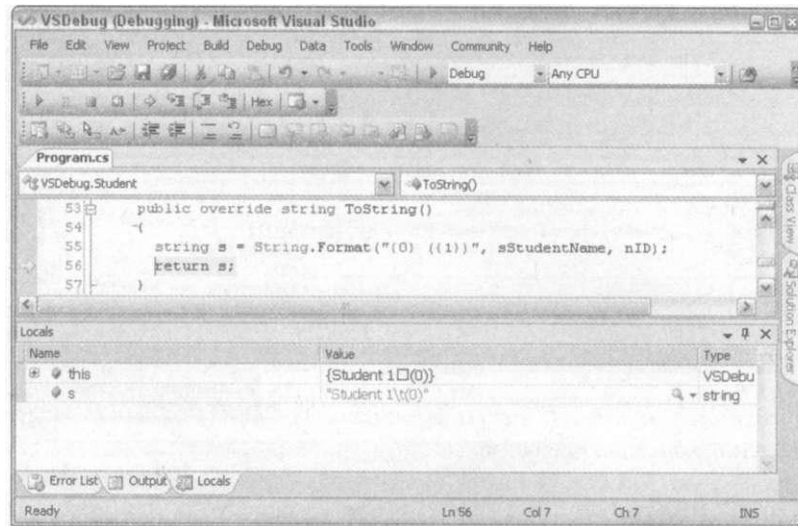


Рис. 21.18. Возвращаемое значение корректно. Так что же происходит?

Вероятно, ошибка произошла в результате перепутывания двух видов функций `WriteLine()`:

```
// С использованием управляющего элемента
Console.WriteLine("Student 1 = " + si.ToString());
// С использованием управляющего элемента
Console.WriteLine("Student 1 = {0}", si.ToString());
```

Подсказка о данных

Сейчас самое время рассказать об одном очень ценном нововведении в отладчике Visual Studio 2005: подсказке о данных (DataTip). Такая подсказка представляет собой небольшой прямоугольник, который появляется, когда вы останавливаете курсор над переменной во время останова в отладчике. После того как я дважды нажал <F10> для возврата в функцию `Main()`, я помещаю курсор над переменной `si` внутри вызова `WriteLine()` (без щелчка) и вижу появившееся окошко с `si` и его значением `ToString(): {Student ID (1)}`. Я игнорирую маленький квадратик, происхождение которого из-за введенного символа табуляции я пояснял ранее. Информацию о подсказке о данных можно получить из раздела "DataTip" справочной системы.



Подсказки работают только когда переменная находится "в контексте", т.е. либо в изучаемой в настоящий момент функции, либо является членом-данными текущего класса, и вы уже выполнили строку, в которой инициализируется эта переменная.

Теперь о существенном усовершенствовании подсказок, видимом из рис. 21.19. Поместите курсор мыши над знаком + в окошке s1. При этом откроется детальная информация об объекте s1. Если вы переместите курсор на значок + перед Static members, а потом — перед allStudents, а затем — перед [0] — нулевым членом ArrayList объекта allStudents — то вы опять увидите s1 — в этот раз уже внутри ArrayList объекта allStudents.

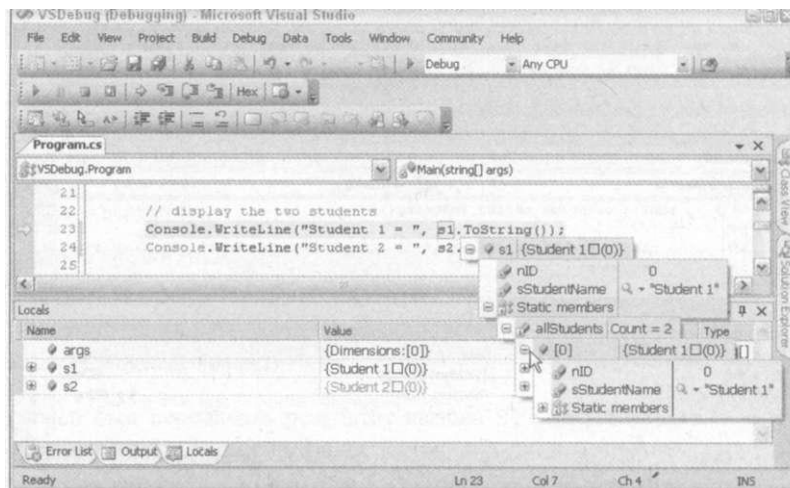


Рис. 21.19. Подсказка о данных — отличное средство, чтобы быстро разобратся с содержимым объекта в отладчике

Подсказки позволяют погружаться все глубже и глубже в сложные объекты. (Ранее для получения этой информации необходимо было открывать окно Watch или Quick-Watch для данной переменной, либо использовать окно Locals.)



Снова щелкните на красном кружке (см. рис. 21.17) для того, чтобы удалить точку останова. (Вы можете также воспользоваться командой меню Debugs Delete All Breakpoints.) В меню Debug имеется еще одна команда Debugs Windows¹Breakpoints, которая предоставляет доступ ко всем возможностям точек останова.

Стек вызовов

Далее я ставлю точку останова на вызове OutputAllStudents(), следующем непосредственно за двумя только что исправленными вызовами WriteLine(). Я нажимаю <F5> для выполнения программы до этой точки и смотрю, что выведено в окне Console. Все выглядит как надо.

Затем я еще раз нажимаю <F10>, чтобы пропустить вызов OutputAllStudents(). И вот тут-то это и происходит.

Появляется сообщение об ошибке наподобие показанного на рис. 21.20. Оно привязано к строке с циклом for, а именно к условию цикла, в котором вызывается свойство Current итератора (об итераторах см. главу 20, "Работа с коллекциями"). Сообщение об ошибке гласит: Enumeration has not started. Call MoveNext (Перечисление не начато. Вызовите MoveNext) — все, что следует знать о происшедшем.

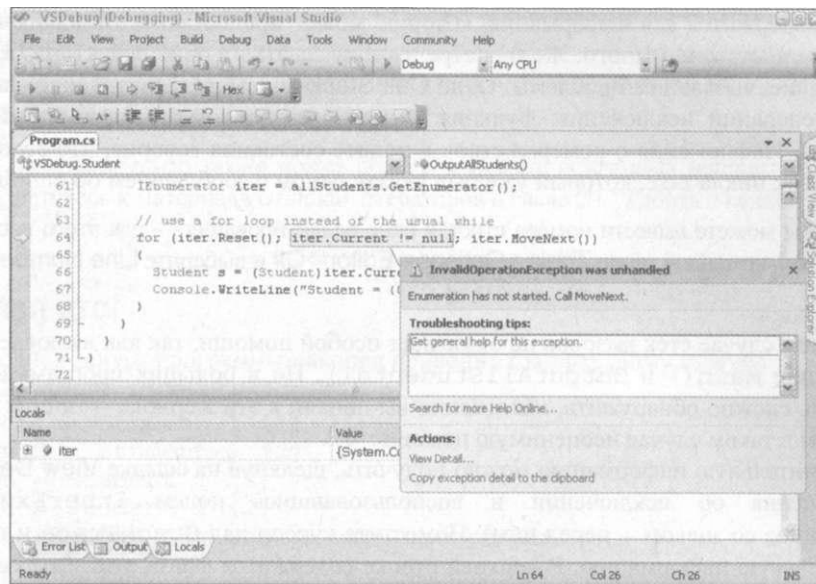


Рис. 21.20. Visual Studio говорит о том, что забыт начальный вызов MoveNext ()

Я закрываю окно сообщения об ошибке и, чтобы получить немного дополнительной информации, командой меню `Debug > Windows > Call Stack` открываю окно стека вызовов Call Stack, показанное на рис. 21.21 (здесь оно раскрыто для того, чтобы было лучше видно представленную им информацию).

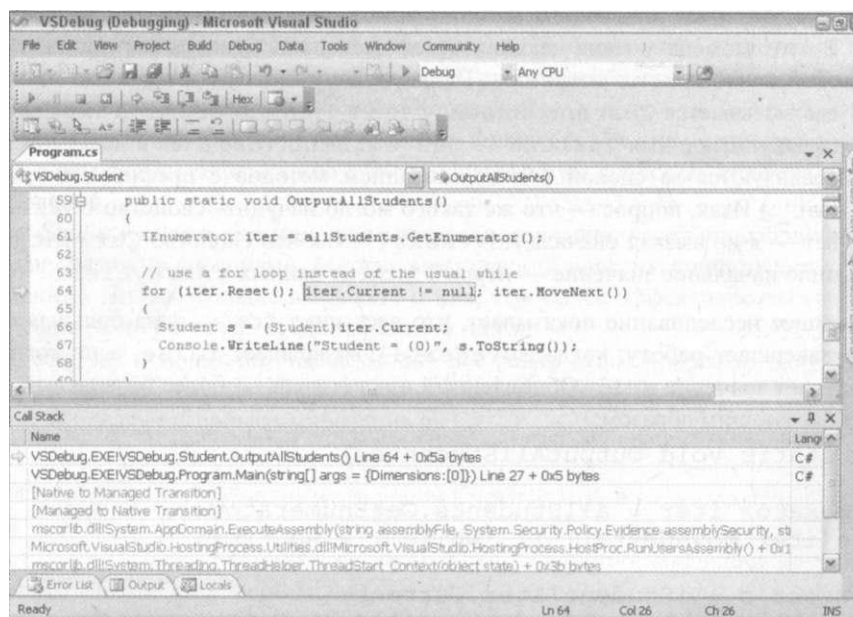


Рис. 21.21. Окно **Call Stack** полезно при поиске источника фатальной ошибки и для определения вашего местоположения

Здесь содержится вся информация, которую может предоставить отладчик — и, как правило, ее достаточно много. Желтая стрелка и подцветка в окне редактора указывают на выражение, вызвавшее проблемы. Окно Call Stack описывает, как именно мы попали в точку генерации исключения: функция `Main()` вызвала `OutputAllStudents()` в строке 64. Информация о номерах строк в полосе состояния говорит, что строка 64 — это заголовок цикла `for`, который уже был указан окном с сообщением об ошибке.



Вы можете вывести номера строк в окне редактирования — для этого воспользуйтесь командой меню `Tools^Options^Editor^C*` и выберите `Line Numbers`.

В данном случае стек вызовов не оказывает особой помощи, так как включает только две функции: `Main()` и `OutputAllStudents()`. Но в больших программах может быть очень сложно обнаружить, как именно вы попали в эти жернова — и стек вызовов окажет вам в таком случае неоценимую помощь.

Дополнительную информацию можно получить, щелкнув на ссылке `View Detail` в окне сообщения об исключении и воспользовавшись полем `InnerException` (единственное со знаком `+` перед ним). Поместите курсор над `StackTrace` и получите дополнительную информацию. В верхней строке содержится фраза `get_Current`. Вот где настоящая неприятность — в вызове свойства `Current` итератора.

Беглый взгляд на документацию по свойству `IEnumerator.Current` проясняет, что не вызван метод `MoveNext()` перед попыткой получения первого элемента. Теперь, когда стало понятно, в чем дело, следует остановить отладчик щелчком на кнопке `Stop Debugging` в полосе инструментов `Debug` и вернуться в режим редактирования. (Кнопка панели инструментов — это ярлык команды меню `Debug^Stop Debugging`; аналогичного эффекта можно добиться и с помощью клавиш `<Shift+F5>`.)



В этот момент у меня накапливается несколько симптомов, указывающих на свойство `Current` итератора. Исключение подцветивает условие цикла `for`, где вызывается `Current`, `StackTrace` в `InnerException` также упоминает закулисное имя `Current` — `get_Current`. (Свойства в действительности реализуются за сценой с использованием методов с префиксами `get_` или `set_`.) Итак, вопрос — что же такого могло начудить свойство `Current`? Ответ — я не вызвал сначала `MoveNext()`, так что свойство `Current` не получило начальное значение — первый элемент данных в `ArrayList`.

Дальнейшее исследование показывает, что весь цикл `for` — одна большая ошибка. Итератор завершает работу, когда `MoveNext()` возвращает `false`, а не когда `Current` получает значение `null`. Обновленный цикл (теперь — более безопасный `while`) выглядит следующим образом:

```
public static void OutputAllStudents()
{
    IEnumerator iter = allStudents.GetEnumerator();
    while(iter.MoveNext()) // 'while', а не 'for'
    {
        Student s = (Student)iter.Current;
        Console.WriteLine("Student = {0}", s.ToString());
    }
}
```

Теперь программа использует `MoveNext ()` для итераций по контейнеру объектов `Student`. Каждый `Student` возвращается свойством `Current`. Цикл завершает работу, когда вызов `MoveNext ()` возвращает `false`, что указывает на то, что в коллекции больше нет не просмотренных элементов.



Цикл `foreach` также может помочь избежать описанных неприятностей—обратитесь к материалу о блоках итераторов в главе 20, "Работа с коллекциями".

Я сделал это!

Очередной запуск программы наконец приводит к корректному выводу:

Эта программа не работает!

```
Student 1 = Student 1 (1)
```

```
Student 2 = Student 2 (2)
```

```
Student = Student 1 (1)
```

```
Student = Student 2 (2)
```

Нажмите <Enter> для завершения программы...

Хорошо, что главной в данной демонстрационной программе была ее отладка, а не создание красивого вывода... Кстати, первая строка вывода более не актуальна.



Исправленная версия демонстрационной программы хранится на прилагаемом компакт-диске под именем `VSDebugFixed`.

Не важно, насколько мощный инструмент отладчик — чем меньше вы будете к нему обращаться, тем лучше. Последние веяния в программировании, такие как первоначальная разработка тестов, непрерывный рефакторинг, шаблоны проектирования и другие аспекты того, что именуется "экстремальным" программированием, могут существенно снизить количество времени, проводимое в отладчике, и повысить производительность программирования (не говоря об уменьшении количества седины у программиста). Поищите литературу или информацию в Web, посвященную вопросам экстремального программирования.

Visual Studio — очень богатая среда программирования, часто способная выполнить одну задачу разными способами. Многие возможности наверняка не дождутся, когда вы их примените. Но чем больше вы знаете о них, тем более эффективно вы сможете выполнять свою работу в качестве программиста на C#.

Даже если вы используете Visual Studio, все равно стоит прочесть главу 22, "C# по дешевке", в которой рассматриваются альтернативы Visual Studio. Это только углубит ваше понимание среды программирования на C#.

Глава 22

С# по дешевке

В этой главе...

- > Поиск альтернатив Microsoft Visual Studio 2005
- > Работа без сети — но не без платформы .NET
- > Программирование на С# в SharpDevelop
- > Программирование на С# в TextPad
- > Использование отладчиков .NET вне Visual Studio 2005
- > Тестирование кода С# посредством инструментария NUnit
- > Проверка возможности запуска ваших программ пользователями

Самым мощным средством для программирования на С# является, вне всяких сомнений, пакет Visual Studio 2005 компании Microsoft. Он объединяет весь процесс разработки в одну интегрированную среду разработки (integrated development environment— IDE), описанную в главе 21, "Использование интерфейса Visual Studio". Вы можете создавать, отлаживать и выполнять свои программы С# в одной среде.

Пакет Visual Studio особенно полезен для разработки программ Windows с графическим интерфейсом пользователя (GUI) и приложений, основанных на Web-страницах с применением технологии ASP.NET, потому что этот пакет предоставляет визуальные методы расположения окон и диалогов. Помимо этого, пакет обладает богатым набором дополнений, без которых, как вы сами можете убедиться, работать достаточно трудно.

Однако пакет Visual Studio стоит недешево. Если у вас его еще нет, вы можете думать: "Я хотел бы попробовать программировать на С#, но как я могу это себе позволить?"

К счастью, в наши дни у вас имеется выбор. Одним вариантом может быть несколько урезанная версия Express языка Visual С# (см. последний раздел данной главы), другим является среда SharpDevelop IDE, которая бесплатно имитирует базовые функциональные возможности Visual Studio. Вы также можете программировать на С# в недорогом редакторе TextPad, как это делают многие программисты на языках Java и Perl. (Прочие варианты можно найти, набрав в строке поиска Google "С# development environment").

В этой главе рассматриваются инструментальные средства, которые позволят вам работать без Visual Studio. Здесь вы познакомитесь с SharpDevelop, TextPad и NUnit, узнаете, как устанавливать и использовать несколько очень дешевых рабочих сред С#. Попутно вам даже будет показано, как написать простое приложение Windows Forms с окном и элементами управления при полном отсутствии проектировщика форм Visual Studio.

Работа без сети — но не без платформы .NET

Первое, что вам понадобится, — это набор бесплатных элементов .NET. Независимо от того, какие инструменты вы выберете, базовые составляющие для программирования на C# включают в себя следующее:

- ✓ текстовый редактор для написания кода, например Блокнот, TextPad или редактор кода SharpDevelop;
- ✓ компилятор C#, `Csc.exe`;
- ✓ один из отладчиков, который поставляется вместе с языком C#: `CorDbg.exe` или `DbgCLR.exe`, предпочтительнее последний;
- ✓ окно командной строки `Command.com` или `Cmd.com` (в зависимости от вашей версии Windows), которое входит в состав Windows.

Ряд других отличных бесплатных дополнений поставляется вместе с языком C#. О нескольких из них чуть больше будет рассказано ближе к концу главы.

Для многих из этих составляющих в дополнение к документации по языку C#, которую можно загрузить со страницы компании Microsoft (о чем речь пойдет в следующем разделе), вам понадобится дополнительная информация. Практически невозможно программировать на C# без справочной информации под рукой, так как у вас будет появляться все большее и большее количество вопросов.

Большинство из того, что вам необходимо, доступно из таких ресурсов, как база знаний (Knowledge Base) Microsoft на Web-сайте сети разработчиков (Microsoft's Developer Network— MSDN) по адресу <http://msdn.microsoft.com>. Там можно в избытке получить информацию о языке C#, платформе .NET, Windows и многом другом. Чтобы найти доступный для разработчиков инструментарий, поищите на сайте MSDN "инструменты платформы .NET". Страница, посвященная Visual C#, расположена по адресу <http://msdn.microsoft.com/vcsharp/2005/>. Центр разработчиков платформы .NET находится по адресу <http://msdn.microsoft.com/netframework/>. Эти страницы содержат ссылки на дополнительные ресурсы, включая группы новостей и форумы, на которых вы можете задавать вопросы. Ссылки Communities и Newsgroups на сайте Visual C# помогут вам найти информацию и помощь по языку C#. Кроме того, вы можете обратиться к Web-сайтам, список которых приведен в конце введения.

Получение бесплатных компонентов

Вы можете получить инструменты, описанные в предыдущем разделе, следующими способами.

- ✓ Путем покупки пакета Visual Studio или Visual C# Express (конечно, это означает, что вы не нуждаетесь в дешевом решении, и тем не менее эта глава может оказаться полезной для вас, поскольку в ней содержится уйма информации о том, что происходит за прекрасным обликом Visual Studio).
- ✓ Путем загрузки бесплатного набора инструментов для разработки программного обеспечения (SDK) платформы .NET, который включает все необходимые инст-

рументы. На сайте MSDN щелкните на вкладке Download, чтобы перейти в раздел Download & Code Center. Там вы можете получить самую последнюю версию набора .NET SDK, который содержит все, что вам необходимо. Выбирайте версию в зависимости от вашего компьютера — вероятнее всего, вам необходима версия x86. Доступны также 64-битовые версии, но для них вам нужен компьютер с 64-битовым процессором.



Набор SDK велик по объему; вероятно, вам понадобится высокоскоростное соединение с Интернетом, но можно заказать этот же набор на компакт-диске на сайте MSDN.

При любом из этих подходов устанавливается платформа .NET, программное обеспечение, содержащее все типы данных и классов, на которых основано программирование на C# — в частности, входящие в пространство имен `System` и другие.

У вас уже могут быть многие из необходимых инструментов, поскольку ряд из них поставляется с последними версиями операционной системы Windows XP. Поищите на своем жестком диске компилятор C#, `Csc.exe`. Если вы его найдете, вероятно, остальные инструменты у вас тоже есть.

Если вы уже установили платформу .NET, большинство инструментов C# обычно расположено в папке `C:\Windows\Microsoft .NET\Framework\v2.0.n`, где *n* означает номер версии. Во время написания этих строк я запускал вторую бета-версию тестового выпуска платформы .NET версии 2.0, поэтому *n* у меня равен 50215, но этот номер, конечно же, изменился, когда был выпущен пакет Visual Studio 2005. Вероятно, на вашей машине эти инструменты расположены в такой же папке.

Наиболее вероятное альтернативное расположение некоторых инструментов — в папке `C:\Program Files`. Туда обычно устанавливаются пакеты Microsoft .NET SDK и Visual Studio. Поищите папку `\GuiDebug` в иерархии папок SDK. Отладчик, который вам нужен (`DbgCLR.exe`), находится там. (Для поиска всегда можно использовать средства Windows.)

Обзор цикла разработки

Основной шаблон разработки программы C# в любой среде программирования довольно прост. Выполните следующие действия.

1. Напишите программу в текстовом редакторе (которым может быть Visual Studio, SharpDevelop, TextPad или даже простой Блокнот). Избегайте текстовых процессоров, подобных Microsoft Word или WordPad. Они делают работу с простыми текстовыми файлами слишком громоздкой.
2. Скомпилируйте программу с помощью компилятора C#, используя Visual Studio, SharpDevelop, TextPad или командную строку. Блокнот для этой цели не подходит.
3. Вернитесь в редактор и при помощи справочной системы и чашки кофе устрани-те ошибки, которые обнаружил компилятор, после чего снова скомпилируйте программу.
4. Запустите программу для ее проверки с помощью Visual Studio, SharpDevelop, TextPad, Windows Explorer, командной строки или инструмента NUnit, который рассматривается далее в этой главе.

5. Посредством отладчика исправьте логические ошибки и другие дефекты, используя Visual Studio, TextPad, SharpDevelop или командную строку.

Намылить, сполоснуть, повторить...

Программирование на C# в программе SharpDevelop

Хорошей, но неполной заменой пакету Visual Studio является программа SharpDevelop, известная также как #develop. Программа SharpDevelop (www.icsharpcode.net) является бесплатной, как и большая часть программного обеспечения с открытым исходным кодом, до тех пор, пока вы придерживаетесь довольно нетребовательного лицензионного соглашения. И если вы не в состоянии позволить себе приобрести Visual Studio, то стоит попробовать поработать с SharpDevelop.



Программа SharpDevelop содержится на прилагаемом к книге компакт-диске, так что испытать ее — дело не сложное.

Изучение SharpDevelop

SharpDevelop прекрасно подходит для написания, компилирования и выполнения программы на C#. Эта программа совсем немного похожа на интегрированную среду разработки Visual Studio (точнее, на более старую, чем Visual Studio 2005, версию, но SharpDevelop работает с C# 2.0), как показано на рис. 22.1. На этом рисунке изображены многочисленные окна документов и инструментов, в достаточной степени соответствующие окнам в Visual Studio.

Возможно, вы заметили, что элементы имеют несколько отличающиеся имена в SharpDevelop и в Visual Studio. В табл. 22.1 сравниваются термины SharpDevelop с аналогичными терминами Visual Studio.

Таблица 22.1. Сравнение терминов SharpDevelop и Visual Studio

SharpDevelop	Visual Studio
Combine	Solution (Project остается Project)
Project Scout	Solution Explorer
Classes Scout	Class View
Properties Scout	Properties Window
Code Window	Code Editor
Task List Pad	Task List Window
Output Pad	Output Window
File Scout	(Нет эквивалента)
Tools Scout	Toolbox Window

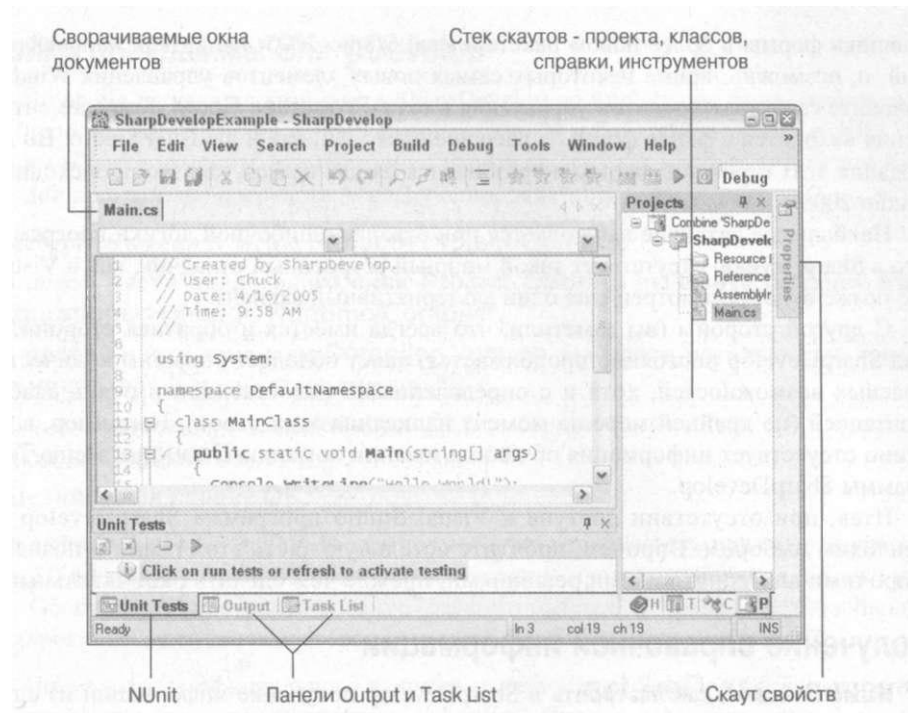


Рис. 22.1. Среда разработки SharpDevelop во многом выглядит (и работает) наподобие Visual Studio

Окна инструментов пакета Visual Studio (например, Output, Toolbox, Properties) в SharpDevelop называются "панелями" или "скаутами".

Если вы будете помнить об этих различиях в названиях, а также о некоторых других вещах, о которых речь пойдет в следующем разделе, то сможете использовать многое из главы 21, "Использование интерфейса Visual Studio", в SharpDevelop — но не материал разделов, посвященных справочной системе и размещению окон.

Сравнение возможностей SharpDevelop и Visual Studio

Для использования SharpDevelop необходимо создать новое объединение (Combain), в которое будет добавлен формируемый проект. Вы можете просматривать файлы и ссылки в этом объединении с помощью окна Project Scout. Из этого окна или из окна Classes Scout можно открывать файлы в окне кода, в котором по умолчанию они появляются на вкладках. Редактирование кода практически идентично редактированию в Visual Studio, включая аналог автозавершения кода в SharpDevelop.

После окончания написания кода его можно скомпилировать посредством меню Build, как и в Visual Studio. Ошибки появляются на панели Error List. Вы можете изменить заданную по умолчанию конфигурацию Debug на конфигурацию Release, а также определить свои собственные настройки.

Если объединение, которое вы создадите, предназначено для построения графического приложения Windows, вы увидите форму, на которой можно разместить элементы управления таким же образом, как и в Visual Studio (за исключением прелестей проекти-

ровщика формы в более новом пакете Visual Studio 2005, например, направляющих линий, и, возможно, кроме некоторых самых новых элементов управления Windows). Установите свойства элементов управления в окне **Properties Scout**. Код элементов управления находится в файле формы с расширением `.CS`, как и в Visual Studio. Во время написания этих строк код формы не разбивался на два класса, как это происходит в Visual Studio 2005.

Наибольшее различие наблюдается при отладке ошибочной логики программы. Пока что в SharpDevelop отсутствует такой мощный встроенный отладчик, как в Visual Studio. Но позже будет рассмотрен еще один альтернативный вариант.

С другой стороны (вы заметили, что всегда имеется и обратная сторона?), работа над SharpDevelop постоянно продолжается, пакет обладает большим количеством прекрасных возможностей, хотя и с определенными недостатками и очень слабой документацией (по крайней мере на момент написания этих строк). Например, в документации отсутствует информация об использовании команды **Debugger** меню **Tools** программы SharpDevelop.

Итак, при отсутствии доступа к Visual Studio программа SharpDevelop является неплохим выбором. Впрочем, прочтите остальную часть этой главы и познакомьтесь с прочими альтернативными решениями, прежде чем сделать окончательный выбор.

Получение справочной информации

Ниже описано, как настроить в SharpDevelop получение информации из справочной системы .NET SDK.

- ✓ Настройте команду в меню **Tools** для открытия справки SDK в вашем Web-браузере. Выберите команду меню **Tools^Options^Tools**. Щелкните на кнопке **Add**. Вызовите инструмент **"Browse .NET Docs"**. В поле **Command** перейдите к вашему Web-браузеру. Для Internet Explorer путь окажется, вероятно, следующим: `C:\Program Files\Internet Explorer\IExplore.exe`. В поле **Arguments** введите путь к документу `StartHere.htm` из папки пакета .NET SDK. Этот пакет находится, вероятно, где-то в папке `C:\Program Files`. На моей машине документ `StartHere.htm` расположен в папке `C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\StartHere.htm`, которая является частью инсталляции пакета Visual Studio. Если этот пакет не установлен, то необходимый документ нужно искать в папке установки пакета Microsoft .NET SDK. Щелкните на кнопке **OK**. Для просмотра документации выберите инструмент в меню **Tools** программы SharpDevelop.
- ✓ Можно просмотреть детальную справку по инструментам .NET SDK, таким как отладчик, путем двойного щелчка на файле `Cptools.chm` в подкаталоге `\Docs` в папке вашего пакета .NET SDK.
- ✓ Также можно обратиться к разделу "Настройка остальных инструментов" далее в этой главе. В нем рассматривается несколько дополнительных инструментов, которые могут помочь вам получить больше информации. Эти инструменты описаны во взаимосвязи с программой TextPad, но их можно использовать и с SharpDevelop.

3. В правой части щелкните на кнопке Add.

Ниже станут доступными поля текстового ввода. Поле Title содержит текст "New Tool".

4. Замените текст "New Tool" в поле Title на что-нибудь наподобие Debugger.

5. Щелкните на кнопке Browse рядом с полем ввода Command и перейдите к каталогу с установленным пакетом .NET Framework SDK. Откройте папку пакета SDK и затем подкаталог GuiDebug. Выберите файл DbgCLR.exe и щелкните на кнопке Open.

Ранее уже рассматривался вопрос о том, где должен находиться ваш пакет SDK.

6. Вернитесь в окно Options, щелкнув на кнопке OK.

Только что созданный инструмент Debugger открывает отладчик CLR и ничего более. В следующих нескольких разделах объясняется, как запускать инструмент, загружать в него файлы и использовать отладчик.

Запуск отладчика из SharpDevelop

После того как вы скомпилировали отладочную версию своей программы, ее можно построчно проверить в отладчике CLR. В этом разделе объясняется, как начать использование отладчика CLR.

CLR является визуальным отладчиком, который выглядит и в основном работает точно так же, как и его коллега в Visual Studio.

Вы работаете в комфортном окне с кодом, который открыт перед вами, и получаете удобные отметки наподобие желтой подсветки текущей линии и красной подсветки строк с контрольными точками, а также можете вызывать знакомые команды из меню Debug с помощью панели инструментов или комбинаций клавиш. Вы можете легко проверять содержимое переменных и отслеживать значения нескольких переменных одновременно. Вы можете исследовать стек вызовов, показывающий последовательности методов, вызывавшихся перед тем методом, через который вы сейчас проходите.

Однако многие из возможностей отладчика Visual Studio 2005 здесь отсутствуют, включая замечательную подсказку о данных, рассматривавшуюся в предыдущей главе. В этом отладчике имеются только старые возможности, но они вполне пригодны для работы.

Загрузка отладчика

Исправив все ошибки компиляции, вы можете обнаружить, что программа после запуска ведет себя странно. Это именно тот случай, когда нужно воспользоваться отладчиком CLR.

Для запуска отладчика не нужно выбирать меню Debug — его просто нет! Вместо этого выберите в SharpDevelop команду меню Tools^Debugger (или аналогичную команду в TextPad).

Когда вы в первый раз отлаживаете какую-то программу, выполните следующие действия.

1. В окне отладчика выберите команду меню DebugsProgram to Debug. Выберите в диалоговом окне исполняемый файл вашего проекта, например, MyCode.exe.

Настройка программы SharpDevelop



Во время написания этой книги SharpDevelop по умолчанию была настроена на использование ранних версий компилятора C# и библиотеки классов .NET. Но вы можете изменить эти настройки (заметьте, что это можно сделать для каждой программы) посредством следующих действий.

1. Выберите команду меню **Project ^Project Options**.
2. В левой части диалогового окна **Project Options** выберите команду **Configurations ^ Debug ^ Runtime/Compiler**.
3. В правой части в панели **Compiler Version** выберите необходимые версии компилятора и среды выполнения .NET.
Для использования последней версии 2.0 выберите `v2 . 0 . n` (где *n* — номер выпуска для версии 2.0).
4. Щелкните на кнопке **ОК**.

После выполнения этих действий можно свободно использовать новые возможности языка C# 2.0 в своих программах, включая обобщенное программирование и блоки итераторов. Обобщенное программирование рассматривается в главе 15, "Обобщенное программирование", а блоки итераторов — в главе 20, "Работа с коллекциями".



После выхода официального выпуска пакета Visual Studio 2005 вам не нужно выполнять вышеперечисленные действия. Просто загрузите самую свежую доступную версию программы SharpDevelop (она бесплатная) и используйте принятую по умолчанию (Standard) версию компилятора.



Если у вас есть пакет Visual Studio или Visual C# Express, вы, вероятно, предпочтете один из них программе SharpDevelop.

Добавление инструмента для запуска отладчика

Самым главным недостатком программы SharpDevelop является отсутствие отладчика с возможностями встроенного отладчика Visual Studio (который описан в главе 21, "Использование интерфейса Visual Studio"). Но здесь будет показано, как добавить в меню Tools программы SharpDevelop инструмент, который запускает еще один визуальный отладчик компании Microsoft— CLR, поставляемый с пакетом .NET Framework SDK. Затем вы узнаете, как запускать и использовать этот отладчик.



Этот отладчик можно использовать как в SharpDevelop, так и в TextPad.

Для добавления инструмента Debugger в программу SharpDevelop выполните следующее.

1. Выберите команду меню **Tools ^Options**.
2. В левой части окна **Options** выберите каталог **Tools** и затем **External Tools**.

Файл с расширением .EXE должен находиться в подкаталоге bin\Debug папки вашего проекта.

2. Выберите команду меню **File^Open^File** и главный файл вашего проекта с расширением .CS (это файл с функцией Main()), к примеру, Mycode.cs.
3. Если программа состоит из нескольких файлов .CS, повторите процедуру открытия файла, описанную во втором шаге, для каждого дополнительного файла.



Выбор нескольких файлов в диалоговом окне File Open позволяет открыть их все одновременно.

На рис. 22.2 показано окно отладчика с прерванным в точке останова проектом C#.

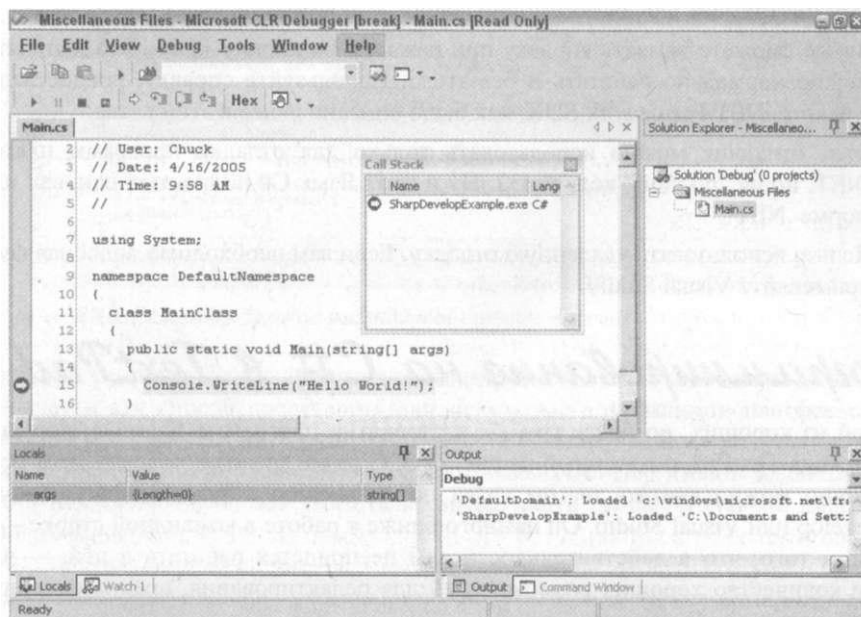


Рис. 22.2. Проект C#, открытый в отладчике CLR



При выборе команды меню **File^Close Solution** можно сохранить конфигурацию отладчика, называемую решением, наподобие решения C# в Visual Studio. Файл решения отладчика имеет расширение .DLN. В следующий раз, когда потребуется загрузить информацию об отладке для этого проекта, вы можете загрузить только файл решения, выбрав в отладчике команду меню **File^Open Solution**. Это избавит от необходимости повторного открытия файла .EXE и всех файлов .CS.

Приготовился, настроил, отладил!

После настройки отладчика с использованием решения для вашей программы вы можете решать обычные отладочные задачи. Из-за большой схожести отладчика со встроенным в Visual Studio не стоит тратить время на посвящение вас во все подробности, так как за ними можно обратиться к предыдущей главе.



Отладчик CLR очень похож на отладчик Visual Studio, поскольку по сути он основан на последнем. Этому отладчику не хватает всего лишь нескольких из наиболее продвинутых возможностей отладчика Visual Studio.

Отсутствующие возможности отладчика

Чего же нет в отладчике CLR, что может обеспечить Visual Studio? В нем отсутствует несколько вещей, но это не вызовет у вас большого огорчения. Ниже перечислены возможности, которых не хватает отладчику CLR по сравнению с отладчиком Visual Studio.

- ✓ Окна Registers, Disassembly и Auto в отладчике CLR работают не так, как в Visual Studio, так что они бесполезны для языка C#. Вероятно, вам не потребуются окна Auto или Registers; а при необходимости вы можете использовать отдельный инструмент для дизассемблирования — Ildasm.
- ✓ Вы не сможете вызвать справку при нажатии на кнопку <F1>. Это неприятно, но можно нормально работать и без этого. Используйте справку, предоставляемую в пакете .NET Framework SDK, как было описано ранее в этой главе.
- ✓ Этот отладчик можно использовать только для отладки программ платформы .NET, но не "родной" код Win32. Ну и что? Язык C# полностью основан на платформе .NET.
- ✓ Нельзя использовать удаленную отладку. Если вам необходима подобная функция, применяйте Visual Studio.

Программирование на C# в TextPad

Одной из хороших, но более грубых альтернатив программе SharpDevelop является редактор кода TextPad, показанный на рис. 22.3. С точки зрения комфорта, этот редактор определенно можно считать шагом назад по сравнению с такими программами, как SharpDevelop или Visual Studio. Он намного ближе к работе в командной строке — за исключением того, что в действительности вам не придется работать в ней, — и имеет большое количество хороших возможностей для редактирования, предназначенных для программистов.



Пробная версия редактора TextPad находится на прилагаемом компакт-диске. Если вы захотите использовать этот редактор, вы должны его приобрести. Посетите Web-сайт по адресу www.textpad.com, заплатите (около 30 дол.) и зарегистрируйте свой редактор.

На Web-сайте программы TextPad также находится форум, на котором можно получить помощь от опытных пользователей TextPad — в основном приверженцев языков Java и Perl, но все же их помощь может быть полезной и для программиста на C#. Некоторые программисты C# предпочитают редактор TextPad программе SharpDevelop, за исключением программирования Windows Forms и Web-программирования. Но все же вы должны попробовать обе программы, чтобы определить свои собственные предпочтения. Эта глава может стать решающей причиной для программирования на C# с помощью TextPad.



На прилагаемом компакт-диске находится демонстрационная программа `PriorityQueueTextPad`, показывающая, как настроить файлы в "проекте" для `TextPad`. Этот пример основан на программе `PriorityQueue`, которая рассматривается в главе 15, "Обобщенное программирование".

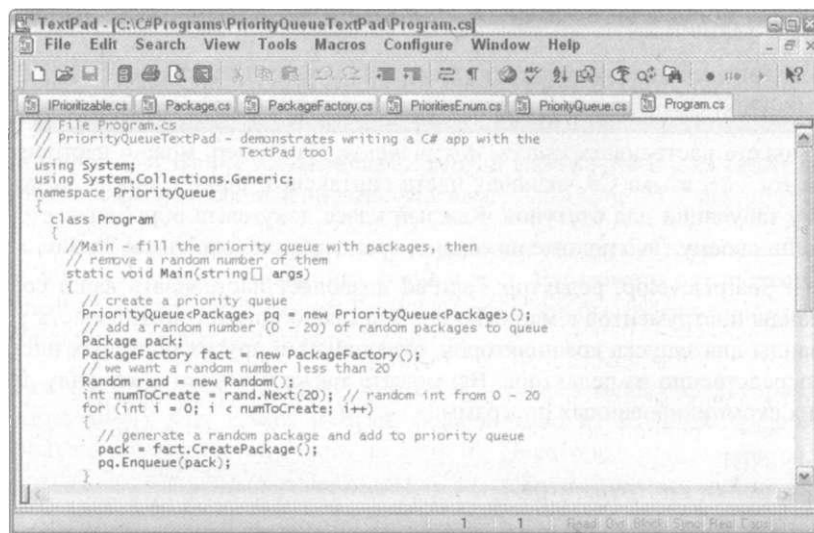


Рис. 22.3. Редактор `TextPad` выглядит обманчиво просто

Поскольку редактор `TextPad` настолько универсален, его следует описать более подробно, отчасти как способ представить вам некоторые возможности платформы .NET, которые я не мог бы объяснить в другом месте. Это такие возможности, которые сложные среды разработки наподобие `Visual Studio` и `SharpDevelop` имеют тенденцию скрывать, но о них стоит знать. Без этого невозможно стать профессионально подготовленным программистом.

Хотя редактор `TextPad` на рис. 22.3 выглядит довольно просто, его можно настроить для выполнения удивительного разнообразия полезных задач программирования. Далее приведен список только некоторых возможностей, которые вы обнаружите, познакомившись с программой поближе.

- ✓ **Работает со многими языками программирования:** `TextPad` специально разработан как редактор исходных текстов программ и отлично работает почти с любым языком программирования; по крайней мере, он имеет встроенную поддержку для очень многих из них, включая `C#` и `Visual Basic`.
- ✓ **Имеет возможность сворачивания документов:** в редакторе `TextPad` вы можете легко работать с множеством исходных файлов программ, как и в его более мощных собратьях.
- ✓ **Предоставляет отличные возможности редактирования:** эти возможности включают поиск пар скобок, закладки для часто посещаемых мест в файлах, отображение номера строки, проверку правописания, сортировку строк, способность записывать и воспроизводить клавиатурные макрокоманды, ряд встроенных мак-

рокоманд и многое другое. Блокнот на фоне TextPad — черепаха по сравнению с гепардом.

✓ **Обеспечивает хорошие встроенные средства управления файлами:** они включают в себя инструмент для сравнения двух файлов, полезный в ряде ситуаций отладки, и команду, которая открывает Windows Explorer в каталоге, содержащем текущий файл.

Но самые главные возможности программы включают следующие действительно полезные функции.

✓ Вы можете настраивать *классы документов*. Например, можно настроить тип документа .CS языка C#, включая цвета синтаксиса, шрифт, опции печати и обработку табуляции для отступов. Каждый класс документа обрабатывается в редакторе по-своему. Это похоже на сервис, предоставляемый Visual Studio.

✓ Как и SharpDevelop, редактор TextPad позволяет настраивать ваши собственные команды инструментов в меню Tools. Это предоставляет возможность установить команды для запуска компиляторов, отладчиков и других полезных инструментов непосредственно из редактора. Вы можете также установить команду для запуска своих скомпилированных программ.

Опции настройки инструментов

✓ **Prompt for Parameters (запрос параметров).** Вашей программе необходимо получать параметры из командной строки? Это так, если вы планируете, что ваше консольное приложение будет запускаться из командной строки, и вы действительно используете параметр args функции Main(). Обычно лучше пропустить настройку этой опции.

✓ **Run Minimized (выполнять минимизированным).** Выбирайте эту опцию, если ваша программа абсолютно никак не взаимодействует (в смысле ввода-вывода) с пользователем. У меня еще не было необходимости использовать эту функцию.

✓ **Save All Documents First (сначала сохранять все документы).** При выборе данной опции происходит сохранение любых документов, открытых вами в редакторе TextPad, перед запуском команды. Я всегда устанавливаю эту опцию.

✓ **Capture Output (перехват вывода).** Выбор данной опции зависит от того, может ли выполнение команды приводить к сообщениям об ошибках и знаете ли вы правильный синтаксис "регулярных выражений" для перехвата ошибок. Далее будет описан этот правильный синтаксис.

Одна возможная причина для выбора этой опции возникает, когда, а) вы настраиваете инструмент для фактического запуска своей успешно скомпилированной программы, и б) вы не предполагаете, что пользователь введет что-либо в командной строке во время выполнения программы. Если программа не является диалоговой, вы можете при желании использовать эту опцию, чтобы направить поток вывода программы в окно Command Results редактора TextPad вместо окна командной строки. Для команды Run следует предпочесть окно командной строки, так что можно не использовать опцию Capture Output.

Но совсем другое дело команда **Build**, которая может (извините, *будет!*) возвращать сообщения об ошибках. При знании синтаксиса волшебных регулярных выражений для языка **C#** вы можете выбрать опцию **Capture Output** для этих команд, но не для других.

- ✓ **Suppress Output Until Completed (запрет вывода до завершения).** Данная опция запрещает вывод, пока программа не закончится. Я обычно оставляю эту опцию неотмеченной.
- ✓ **Sound Alert When Completed (подача звукового предупреждения при завершении).** Если ваша программа долго выполняется (например, долго компилируется), и вы хотели бы в это время выполнять другую работу, эта опция предупредит вас звуковым сигналом о завершении программы.

Чего недостает редактору **TextPad**, так это удобных инструментов наподобие **Solution Explorer**, **Class View**, проектировщика форм и т. д. Вы можете предпочесть **SharpDeveloper**, но даже в этом случае редактор **TextPad** удобен для быстрого редактирования программы или для универсальной замены Блокнота.



Как известно, существует множество других прекрасных редакторов программ, включая **Vi**, **Fte**, **Emacs** и **Brief**, большинство из которых предлагают даже больше возможностей (часто за деньги). Некоторые программисты даже клянутся своим любимым редактором.

Из следующих разделов вы узнаете, как настроить редактор **TextPad** для компилирования, выполнения и отладки своих программ **C#**. Действия довольно сложные, но вы попутно получите большое количество интересной дополнительной информации.

Необходимо настроить две вещи в редакторе **TextPad**, прежде чем он сможет работать с **C#**: класс документов **.CS** для языка **C#** и несколько связанных с **C#** команд в меню **Tools**.

Создание класса документов **.CS** для языка **C#**

Для создания нового класса документов языка **C#** в редакторе **TextPad** выполните следующие действия (которые не проиллюстрированы рисунком).

1. Выберите команду меню **Configure^New Document Class**.
2. В мастере **Document Class Wizard** введите **C#**, чтобы дать название классу, и затем щелкните на кнопке **Next**.
3. Введите ***.cs** в качестве "члена класса". Щелкните на кнопке **Next**.
4. Установите флажок **Enable Syntax Highlighting** и затем из списка файлов подцветки синтаксиса выберите **csharp.syn**. Щелкните на кнопке **Next**, а затем — на кнопке **Finish**.
5. Выберите команду меню **Configure^Preferences**. В иерархическом списке слева выберите пункт **Document Classes^Ctt**. Справа отметьте желаемые опции для обработки файлов **C#**, как показано на рис. 22.4.

Я выбрал опции **Maintain Indentation**, **Automatically Indent Blocks**, **Strip Trailing Spaces from Lines When Saving** и **Check Spelling of Comments**. Все остальные опции я оставил такими, какими они были по умолчанию.

6. В иерархическом списке слева выберите подпункт **Tabulation** в пункте **C#**. Введите количество пробелов табуляции и число пробелов для отступа. Затем щелкните на кнопке **Apply**.

Я выбрал размер табуляции и отступа, равный двум для сохранения пространства при печати примеров программ в книге, но вы можете выбрать более удобные размеры. Я также выбрал опции **Convert New Tabs to Spaces** и **Convert Existing Tabs to Spaces When Saving Files**. Некоторые программисты предпочитают табуляцию.

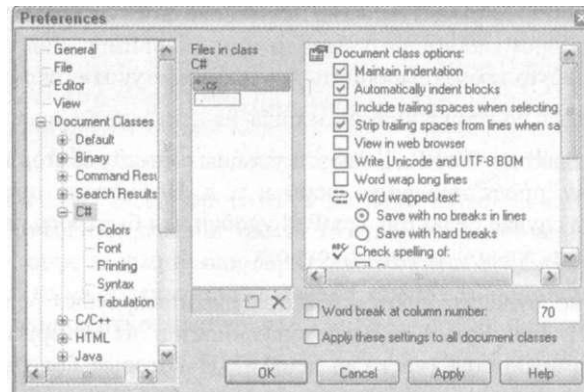


Рис. 22.4. Здесь в редакторе TextPad настраивается класс документов C#

7. Щелкните также слева на пункте **Font** и установите предпочитаемый шрифт для программы. Затем щелкните на кнопке **Apply**.

Я выбрал шрифт Lucida Console, обычный, размер 9 пунктов для экрана и принтера.

8. Находясь в диалоговом окне **Preferences**, просмотрите прочие настройки предпочтений и при желании измените что-либо.

Имейте в виду, что установки для индивидуальных классов документов отменяют общие настройки.

9. При выполнении восьмого шага установите еще одну опцию, которая пригодится позже. В левой панели окна **Preferences** щелкните на пункте **Editor** (прямо под пунктами **General** и **File**). Справа отметьте последнее поле опции **Use POSIX Regular Expression Syntax**.

10. Наконец, когда вы закончите, щелкните на кнопке **Apply** и затем — на кнопке **OK**.

Добавление собственных инструментов: Build C# Debug

Настройка команд меню **Tools** для компиляции, отладки и запуска программ C# включает в себя добавление в меню инструмента для каждого действия, изменение текста, который появляется в меню **Tools**, и дальнейшую настройку инструмента.

В качестве примера выполните следующие действия, чтобы добавить инструмент **Build C# Debug** для компиляции версии вашей программы, которая может выполняться в отладчике.

1. Выберите команду меню **Configured Preferences**. В левой части диалогового окна **Preferences** щелкните на пункте **Tools**.
2. В правой части выберите команду **AdddProgram**. В диалоговом окне **Select a File** перейдите к папке, в которой находится исполняемый файл (.EXE) инструмента.

Чтобы найти инструменты, рассматриваемые в этой главе, обратитесь к разделу "Получение бесплатных компонентов". Обратите внимание, что отладчик CLR находится в ином месте, чем большинство из них. Он описан в разделе "Запуск отладчика из SharpDevelop" этой главы.

3. Выберите требуемый инструмент — в данном случае **C# . exe**.

В списке справа появится новая команда. В настоящий момент это название самого исполняемого файла, но вы можете затем изменить это название на что-нибудь более значащее.



Между группами инструментов в меню можно разместить тонкую сплошную разделительную линию. Начните с нее при создании нового инструмента. Выберите команду **AddMenu Separator** (вместо **Program**). Для перемещения разделителя (или любого инструмента) используйте волнистые стрелки вверх списка инструментов.

4. Для изменения названия инструмента щелкните на нем *дважды* в списке справа — *но немного медленнее, чем при двойном щелчке*. Затем введите текст, который появится в меню **Tools** для этой команды. По завершении ввода нажмите **<Enter>** и затем щелкните на кнопке **Apply**.

Для своей первой команды компилирования я ввел текст **Build C# Debug**.

5. Щелкните на новом инструменте, который вы добавили, в пункте **Tools** диалогового окна **Preferences**, чтобы появились дополнительные опции, как показано на рис. 22.5.

Справа появятся дополнительные опции, как показано на рис. 22.5, с настройками по умолчанию. Их можно изменить, выполнив следующие действия. В правой части поле **Command** уже должно содержать путь к исполняемому файлу инструмента, в поле **Parameters**, возможно, содержится текст **\$File**, а в поле **Initial Folder** — **\$FileDir**. Для большинства команд это совершенно верно, но вы измените эти значения для нескольких первых инструментов, которые настраиваете.

6. В поле ввода **Parameters** введите следующее:

```
/debug /out:$FileDir\bin\debug\Program.exe ©refs.rsp *.cs
```

7. Установите флажки **Save All Documents First** и **Capture Output**. Затем щелкните на кнопке **Apply**.

Оставьте другие флажки неотмеченными.

8. Выберите поле **Regular Expression to Match Output** и аккуратно введите следующую строку, затем щелкните на кнопке **Apply**:

```
"( [*.' ]+.св)\( ( [0-9]+) , ( [0-9]+) \)
```

Ниже будет кое-что пояснено из этого марсианского стихотворения.

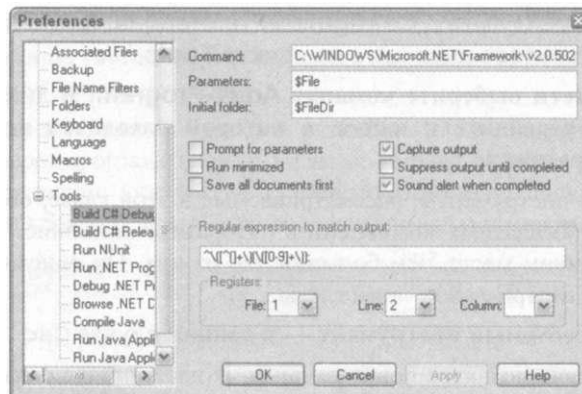


Рис. 22.5. Опции настройки вашего первого инструмента

9. Установите поля **Registers** так, чтобы в них были значения **1, 2 и 3**.
10. Щелкните на кнопке **Apply** и затем на кнопке **OK**, чтобы закрыть диалоговое окно **Preferences**.

Далее вы настроите дополнительные инструменты таким же двухэтапным способом, описанным ниже.

1. Создайте инструмент, как описано выше.

В следующих разделах будут рекомендоваться названия для каждого инструмента.

2. Настройте свой новый инструмент путем его выбора в левой части диалогового окна **Preferences** и заполнения детальной информации в правой части.

В следующих разделах такая детальная информация будет представлена для каждого инструмента.

Настройка инструмента для компиляции финальной версии

После полной отладки и проверки программы вы упаковываете финальную версию и выпускаете ее в свет. Для настройки инструмента компиляции окончательной версии следуйте приведенной инструкции.

- ✓ Program to Run (программа для запуска). При создании инструмента перейдите к Csc.exe (ищите папку, описанную в разделе "Получение бесплатных компонентов" этой главы).
- ✓ Menu Name (название меню). Введите Build C# Release.
- ✓ Command (команда). Настраивается, когда вы создаете пункт меню инструмента и указываете программу, которая запускается для этого инструмента: Csc.exe.
- ✓ Parameters (параметры). При настройке инструмента введите в это поле текст /out:\$FileDir\bin\release\Program.exe @refs.rsp *.cs (теперь без ключа /debug). Затем заполните пункты в оставшейся части этого списка.
- ✓ Initial Folder (начальный каталог). Оставьте это поле без изменений.

- ✓ **Options (опции).** Установите флажки **Save All Documents First** и **Capture Output** (остальные не установлены).
- ✓ **Regular Expression (регулярное выражение).** Щелкните на кнопке **Apply**, снова откройте инструмент **Build C# Debug**, выделите регулярное выражение, нажмите комбинацию клавиш **<Ctrl+C>** для его копирования, выберите инструмент **Build C# Release** и поле **Regular Expression** и нажмите комбинацию клавиш **<Ctrl+V>** для вставки регулярного выражения. Щелкните на кнопке **Apply**. Избегайте повторного ввода этого монстра!
- ✓ **Registers (регистры).** Введите в ячейки 1,2 и 3.

Теперь в редакторе TextPad имеются два инструмента. Позже вы добавите еще несколько, но те инструменты, которые были только что настроены, требуют небольшого пояснения и некоторых настроек.

Объяснение опций настройки инструментов Debug и Release

Для некоторых компиляций командная строка может нуждаться в изменениях. В этих случаях вы должны временно перенастраивать инструменты компилирования в редакторе TextPad или создавать специальный файл для сохранения дополнительных опций. Я предпочитаю последний подход.

В следующих разделах рассматриваются все опции в полях **Parameters** инструментов **Build C# Debug** и **Build C# Release**, а также объясняются опции **Regular Expression** и **Registers**.

Ключ /debug

Первым параметром в поле **Parameters** инструмента **Build C# Debug** (но не инструмента **Build C# Release**) является ключ **/debug**.

В некоторых программах используются *ключи командной строки*, которые предоставляют пользователю возможность настраивать поведение программы. Для инструментов редактора TextPad, которым необходимы такие ключи, они вводятся в поле **Parameters**. Например, ввод строки **/debug * .cs** дает команду компилятору C# запустить отладочную версию процесса компиляции.



Ключи командной строки отличаются от ее параметров. Символ **\$File** — это параметр, указывающий редактору TextPad имя файла, который является текущим в данный момент в редакторе. Редактор TextPad передает эту информацию инструменту при его запуске.

Но если вам, к примеру, нужно указать отладочную версию при запуске компилятора, то необходимо использовать *ключ*, который понимает компилятор: для языка C# таким ключом является **/debug**. Другие команды определяют другие ключи. Выяснить, какие ключи доступны, обычно можно путем ввода в окне команды имени команды, за которым следует пробел, символ наклонной черты и вопросительный знак. Например, можно ввести **esc.exe /?**, чтобы узнать, какие ключи компилятора C# имеются (как и уточнить остальную часть синтаксиса команды). Ключи имеют названия и начинаются с символа **/** или **-**.



В своем консольном приложении вы также можете определить ключи командной строки. Не забудьте включить метод для предоставления справки по вашим ключам, когда пользователь введет **YourProgram.exe /?** Другими словами, функция `Main()` должна получать аргументы командной строки и искать среди них ключ `/?`, а затем выводить справочную информацию.

Ключ /out

Ключ `/out` в поле **Parameters** позволяет указать, где компилятор должен помещать компилируемую программу для работы. Значением по умолчанию является `$FileDir`. В следующем списке объясняется информация, использованная в ключе `/out` для команды **Build C# Debug**.



✓ `$FileDir`: этот "макропараметр" сообщает редактору TextPad о месте расположения ваших исходных файлов C# (* .cs). При компиляции программы из редактора TextPad компилятор помещает результирующие файлы .EXE, .DLL и другие файлы компиляции, такие как информация отладки (в файле .PDB), в тот же каталог по умолчанию.

Жизнь будет проще, если скопировать структуру каталогов, которая используется в программах Visual Studio и SharpDevelop. Это позволяет компилировать одинаковую программу в любой из этих сред с аналогичной структурой. В Visual Studio файлы компиляции помещаются в подкаталог подкаталога `$FileDir`: для отладочной версии это `$FileDir\bin\Debug`, для окончательной — `$FileDir\bin\Release`.

Для подражания программе Visual Studio создайте вручную подкаталоги в папке вашего проекта и включите ключ `/out` в поле **Parameters** по образцу: `/out: $FileDir\bin\release\Program.exe`. Выполняйте эти действия как часть настройки каждого проекта.

Для поиска других макросов наподобие `$FileDir` введите "tools" в справочной системе TextPad и выберите тему "Tool Parameter Macros".

✓ `Program.exe`: ключ `/out` можно использовать для определения имени программы. В Visual Studio класс, включающий функцию `Main()`, по умолчанию называется `Program`, и файл, содержащий этот класс, называется `Program.cs`. (при желании можно изменить одно или оба эти названия.) Вы также можете вручную изменить имя файла с расширением .EXE в ключе `/out` на что-нибудь другое. Возможно, вам не нравится называть каждый файл с расширением .EXE "Program". Для ввода другого названия просто откройте диалоговое окно **Configuration** для инструмента компиляции и замените строку `Program` в ключе `/out` другим названием. Я использую строку "Program.exe" в качестве заполнителя, так как в поле **Parameters** нужно что-нибудь ввести.

Параметр @refs.rsp

Параметр `@refs.rsp` позволяет упростить уже пугающую командную строку для инструментов компиляции редактора TextPad. Хитрость заключается в том, чтобы сохранить набор дополнительных параметров в текстовом файле в каталоге вашего проекта и обращаться к нему из командной строки, указывая этот файл. Компилятор C# читает

файл и добавляет его содержимое к командной строке. Этот файл используется как с отладочной, так и с финальной версиями.

Для использования данной возможности командной строки вашего инструмента выполните следующие действия.

1. В редакторе TextPad выберите команду меню **File^New** для создания нового файла.
2. В начале файла введите: **# Файл настройки для компиляций C#.**
3. Сохраните файл с именем **refs. rsp** в каталоге вашего проекта.

Убедитесь в том, что поле Save as Type в диалоговом окне Save as установлено в **All Files (*.*)**.

Что содержится в файле настройки

Эти файлы в основном применяются для ссылок на пространства имен, которые используются вами. Каждый раз, когда директива `using`, такая как `using System;`, помещается в начало исходного файла C#, компилятору необходима информация о том, где искать классы из этого пространства имен, т.е. имя и расположение файла .DLL, который связан с пространством имен, упомянутым в директиве `using`.

В Visual Studio вы отправляете компилятор к соответствующему файлу .DLL (называемому *сборкой* (assembly)), выбирая команду меню **Project^Add Reference**. (Этот шаг не нужен для подмножества библиотеки .NET, для которой в компиляторе уже имеются собственные предопределенные ссылки: обычно используемые элементы находятся в файле `Mscorlib.dll`. Ниже будет дано объяснение, как определить, что это за элементы.)

В инструментах компиляции редактора TextPad (Debug и Release) вы выполняете ту же операцию посредством ключа `/reference` (/r для краткости). Вы *могли бы* поместить свои ключи /r (по одному для каждой директивы `using`) прямо в командной строке (другими словами, в поле **Parameters** для ваших инструментов компиляции, вместо файла `@refs.rsp`).

Да, вы могли бы это сделать. Но только представьте себе, каким длинным стало бы содержимое командной строки в таком небольшом поле **Parameters**!

Пример файла настроек

Размещение элементов наподобие ключей /r — это именно то, для чего предназначен файл настройки. В приведенном далее файле содержится пара ключей /r:

```
# файл настройки для приложений C#
/r:System.Windows.Forms.dll
/r:"C:\Program Files\NUnit 2.2\bin\nunit.framework.dll"
```

Этот простой файл настройки включает комментарий (первая строка, начинающаяся с #) и два ключа `/reference`, каждый на отдельной строке. Первый ключ ссылается на файл .DLL, содержащий классы в пространстве имен `System.Windows.Forms`, а второй — на файл .DLL для инструмента тестирования NUnit. Данный инструмент является библиотекой классов .DLL сторонних производителей. Эти строки не являются кодом языка C#, поэтому они не заканчиваются точкой с запятой. Вторая строка заключена в кавычки, потому что путь к файлу содержит пробелы.



Большинство библиотечных классов, которые вы указываете в директивах `using` (или в полностью квалифицированных именах в своем исходном тексте наподобие `System.Windows.Forms.Form`), являются частью *библиотеки базовых классов* (Base Class Library — BCL) платформы .NET Framework. Данные классы — это файлы `.DLL`, которые уже сохранены в *глобальном кэше сборок* (Global Assembly Cache — GAC), центральном хранилище в каталоге Windows. Компилятор может найти эти классы на основе ссылки, просто по имени файла `.DLL` наподобие `System.Windows.Forms.dll`.

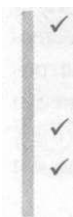
Но иногда необходимо использовать классы, определенные в библиотеках классов сторонних производителей или в библиотеках классов, созданных вами самостоятельно. Например, чтобы применить инструмент тестирования NUnit с вашим проверяемым кодом, понадобится директива `using NUnit.Framework`; во всех файлах, содержащих тестируемые классы; кроме того, нужна ссылка (/г) на каталог, в котором находится файл `nunit.framework.dll`.

Файлы библиотеки инструмента NUnit не хранятся в GAC. Чтобы помочь компилятору их найти, ссылка должна определить полный путь к файлу `.DLL`. Именно поэтому в предшествующем примере во втором ключе /г содержится полный путь.

Следует упомянуть еще вот о чем: в файле настройки, как и в командной строке (в поле Parameters для инструмента), необходимо определить ключи, параметры и имена файлов для компиляции в такой последовательности:

ключи	// например: <code>/debug</code> и <code>/out</code>
параметры	// например: <code>Orfs.rsp</code>
файлы	// например: <code>*.cs</code>

Ознакомьтесь с кратким описанием примера.



✓ `/debug` — это ключ. (Его лучше разместить здесь, чем в поле Parameters, если только вы не используете этот файл настройки для обеих компиляций — Debug и Release).

✓ `@debug.rsp` — это параметр (но этот параметр должен быть в командной строке).

✓ `*.cs` — список файлов для компиляции. (Вы можете иногда встретить дополнительные ключи после имен файлов.)

Большинство из этих элементов могут встречаться как в командной строке, так и в файле параметров.



Размещайте файл параметров в каталоге, на который указывает параметр `$FileDir` в редакторе TextPad. Для команд компиляции это каталог, содержащий исходные файлы `.CS`.

Во время компиляции компилятор раскрывает командную строку с использованием информации из файла настроек `@refs.rsp` (в командной строке они разделялись бы пробелами).



Компилятор C# всегда использует дополнительный заданный по умолчанию файл настройки, называемый `csc.rsp`, для ссылок на все общие сборки, такие как `System.dll`. Таким образом компилятор знает о них без вашего вмешательства. Найдите этот файл в том же месте, где находится компилятор C#, `csc.exe`, и просмотрите его.

Работа над ошибками компиляции

Оба инструмента — и Build C# Debug, и Build C# Release — запускают компилятор. В любом случае, даже при компиляции окончательной версии, вы можете получить от компилятора сообщения об ошибках. При правильной настройке инструментов любые сообщения компилятора об ошибках появляются в окне Command Results редактора TextPad. Типичное сообщение об ошибке выглядит так:

```
mycode.cs(11,17): error CS0246: The type or namespace name
'joeyTypes' could not be found (are you
missing a using directive or an assembly
reference?)
```

Первая часть — это имя файла, в котором произошла ошибка. Числа в круглых скобках — номера строки и столбца в этой строке. Остальная часть просто подробнее описывает ошибку.



Волшебное регулярное выражение, которое вы добавили при настройке инструментов, предназначено для выделения трех частей информации из этого сообщения: имени файла, номера строки и номера столбца. Редактор TextPad перехватывает сообщение об ошибке от компилятора, применяет регулярное выражение для извлечения этих элементов и помещает их в те "регистры", которые вы настроили как 1, 2 и 3. Вот в чем был весь фокус-покус!

Синтаксис регулярного выражения примерно так же прост, как и общая теория относительности. (В действительности он не так уж и плох, если только вы поймете его суть.)



В окне Command Results можно дважды щелкнуть на сообщении об ошибке и перейти к указанной строке и столбцу. Очень полезная возможность!

Конечно же, вы должны разобраться в том, что же именно произошло неправильно, но, естественно, это останется упражнением для самостоятельного изучения.

Настройка остальных инструментов

Теперь добавьте, переименуйте и настройте инструменты для выполнения компилируемой программы, запуска инструмента тестирования NUnit, отладки программы и просмотра документации пакета .NET SDK. В следующих разделах перечислены настройки конфигурации каждого из упомянутых инструментов. Щелкайте на кнопке Apply после внесения изменений в каждую установку.

Два инструмента для запуска вашей программы

После успешной компиляции своей программы вы, как правило, захотите ее запустить. Поскольку отладочная и финальная компиляции помещают результат в различные каталоги, вам потребуются два инструмента. Чтобы настроить инструменты для запуска вашей программы, используйте настройки из следующего списка.



Program to Run (программа для запуска). При создании инструмента укажите следующую программу для его запуска: `Csc.exe`. Чуть позже вы измените название программы на другое, не доступное во время создания инструмента. Эта установка одинакова для обоих инструментов запуска, описанных здесь.

- ✓ **Menu Name (название меню).** Введите `Run Debug` для одного инструмента и `Run Release` для другого.
- ✓ **Command (команда).** При настройке инструмента замените все, что находится в этом поле, строкой `$FileDir\bin\debug\Program.exe` для первого инструмента и `$FileDir\bin\release\Program.exe` для второго. (Если вы изменили свои командные строки компиляции в поле `Parameters`, чтобы использовать название, отличное от `"Program"`, воспользуйтесь этим названием и здесь). Затем завершите настройку остальных элементов в этом списке.
- ✓ **Parameters (параметры).** Оставьте это поле без изменений для обоих инструментов.
- ✓ **Options (опции).** Установите флажок `Save All Documents First` (остальные опции не установлены).
- ✓ **Remaining options (остальные опции).** Пропустите.



При использовании своего инструмента `Run Debug` или `Run Release` убедитесь в том, что файл, содержащий функцию `Main()`, находится в редакторе `TextPad` на переднем плане.

Инструмент для модульного тестирования вашей программы

В процессе программирования можно существенно увеличить свою уверенность в программе путем написания и частого выполнения небольших тестов для классов и методов. Инструмент `NUnit`, упрощающий такое тестирование, описан далее в этой главе. Для настройки инструмента, запускающего тестирование `NUnit`, используйте следующие установки.

- ✓ **Program to Run (программа для запуска).** При создании инструмента введите название или перейдите к программе `C:\Program Files\NUnit 2.2\bin\nunit-gui.exe` (хотя, возможно, у вас имеется более свежая версия программы `NUnit`).
- ✓ **Menu Name (название меню).** Введите `Run NUnit`.
- ✓ **Command (команда).** Это поле устанавливается после создания инструмента.
- ✓ **Other options (другие опции).** При настройке инструмента оставьте их все без изменений.

Инструмент для отладки вашей программы

После того как вы скомпилировали свою программу с помощью команды `Build C# Debug`, которую настроили ранее, вам зачастую необходимо запустить программу в отладчике для поиска ошибок. Чтобы настроить инструмент для запуска отладчика, добавьте инструмент `Debug .NET Program` в редактор `TextPad`, используя следующие установки.

- ✓ **Program to Run (программа для запуска).** При создании инструмента перейдите к папке, в которой установлен пакет `.NET Framework SDK`, обычно она находится в каталоге `C:\Program Files`. Затем найдите подкаталог `\GuiDebug` и откройте файл `DbgCLR.exe`. Обратитесь к разделу "Получение бесплатных компонентов" выше в этой главе.
- ✓ **Menu Name (название меню).** Введите `Debug .NET Program`.

- ✓ **Command (команда).** Это поле устанавливается после создания инструмента.
- ✓ **Parameters and Initial folder (параметры и начальный каталог).** При настройке инструмента очистите эти поля. Затем завершите настройку остальных элементов этого списка.
- ✓ **Options (опции).** Установите флажок **Save All Documents First** (никаких регулярных выражений или регистров).

Эта команда работает для любого языка платформы .NET, а не только для C#. Подробнее применение данного отладчика было описано ранее в этой главе. Один и тот же отладчик используется как в TextPad, так и в SharpDevelop.

Инструмент для просмотра документации пакета .NET SDK

Поскольку редактор TextPad ничего не знает о языке C# или библиотеке .NET Framework (в отличие от SharpDevelop), вы будете вынуждены воспользоваться инструментом для просмотра документации из .NET Framework SDK. Для настройки инструмента, который открывает документацию в вашем Web-браузере, добавьте инструмент **Browse .NET Docs**, используя следующие настройки.

- ✓ **Program to Run (программа для запуска).** При создании инструмента перейдите к папке, в которой установлен ваш Web-браузер. Например, для использования программы Internet Explorer перейдите к программе C:\Program Files\Internet Explorer\IExplore.exe.
- ✓ **Menu Name (название меню).** Введите **Browse .NET Docs**.
- ✓ **Command (команда).** Это поле устанавливается после создания инструмента.
- ✓ **Parameters (параметры).** В Windows Explorer найдите папку с пакетом .NET SDK; вероятно, она находится в каталоге C:\Program Files. При настройке инструмента введите путь к этой папке, за которым следует \StartHere .htm.
- ✓ **Other options (другие опции).** Пропустите.



Существует еще один отличный инструмент. Программа WinCV позволяет просматривать пространства имен и классы в библиотеках платформы .NET. Исполняемый файл WinCV.exe расположен в папке \bin в каталоге пакета .NET SDK. Все, что вам нужно — это указать путь в поле **Command** в окне **Preferences**.

Дизассемблер промежуточного языка (Intermediate Language Disassembler) позволит вам получить удобочитаемую версию того, во что скомпилирована ваша программа C#. Постепенно из полной абракадабры она будет превращаться во все более и более понятный для вас код. Программа Ildasm.exe расположена в подкаталоге \bin каталога .NET SDK. Все, что вам нужно — это указать путь в поле **Command**. При создании инструмента можно выбрать опцию **Close DOS Window on Exit**.



Для получения специальной справки по инструментам SDK дважды щелкните на файле \Docs\CpTools.chm в окне Windows Explorer в каталоге .NET SDK.

Итак, вы завершили настройку редактора TextPad для работы с языком C#. На рис. 22.6 показан окончательный вид меню **Tools**. Теперь редактор TextPad стал очень полезным инструментом для программирования на C#, а вы узнали, что делают про-

граммы Visual Studio и SharpDevelop под своим блестящим покрывалом. Далее вы узнаете, как тестировать свои программы.

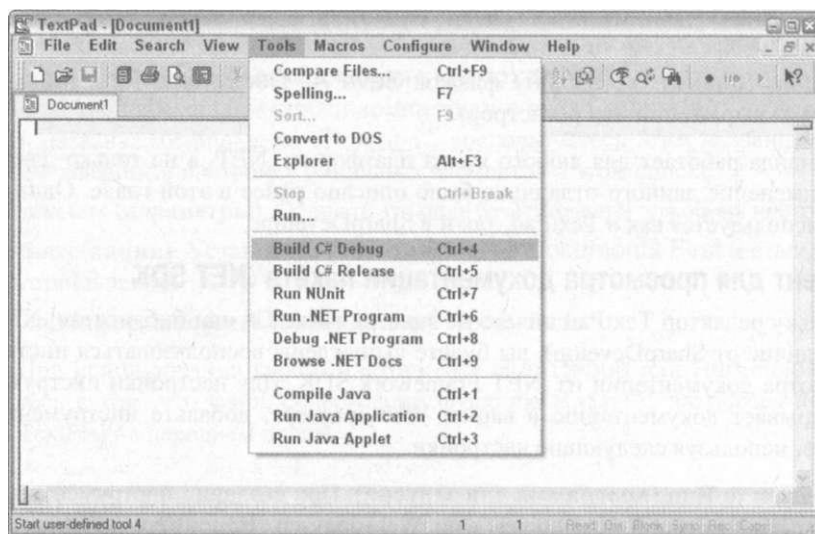


Рис. 22.6. Простое меню Tools в редакторе TextPad— вашей новой среде программирования на C#

Тестирование с помощью программы NUnit

Выше объяснялось, как настроить инструмент для запуска NUnit. Эта программа доступна на Web-сайте по адресу www.nunit.org и на прилагаемом компакт-диске. Это — программное обеспечение с открытым исходным кодом, поэтому вы можете свободно использовать его, придерживаясь лицензии.

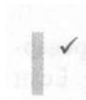


Данный раздел применим как к работе в Visual Studio, так и к работе в TextPad или SharpDevelop.

Запуск программы NUnit

Чтобы можно было запускать программу NUnit из какой-нибудь рассматриваемой в этой главе среды разработки, вы должны сначала настроить ее в меню Tools. Ознакомьтесь с командами для запуска программы NUnit.

- ✓ Для запуска программы NUnit из пакета Visual Studio выберите команду меню Tools'^*Ваше название*.
- ✓ Для запуска программы NUnit из редактора TextPad выберите команду меню Tools^Run NUnit.
- ✓ Для запуска программы NUnit из SharpDevelop выберите команду меню Tools^NUnit.



Программу NUnit также можно запустить из меню Start в Windows, из Windows Explorer или из командной строки.

Конечно, сначала необходимо кое-что подготовить. Программа NUnit проста в использовании, но вы должны ознакомиться с ее соглашениями и методами.

На рис. 22.7 показана программа NUnit с частично успешным тестовым запуском, который только что завершился.

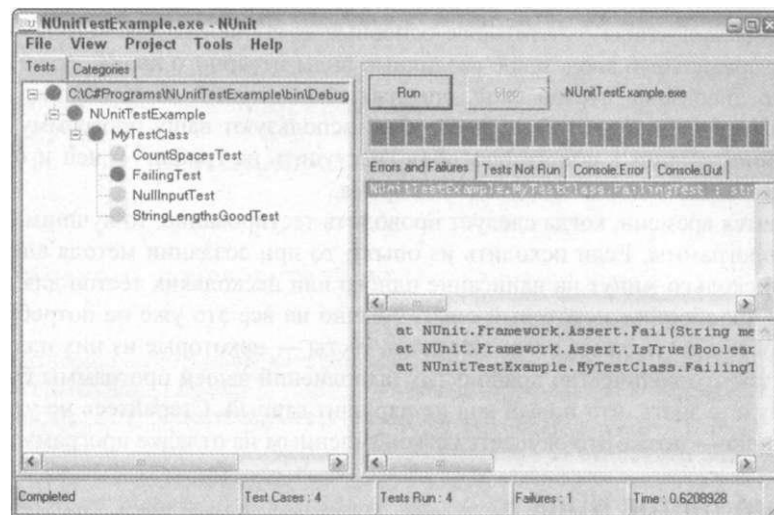


Рис. 22.7. Тестирование в NUnit и простое, и мощное

Тестирование

Эта книга о программировании на языке C#, поэтому вы можете удивиться, почему в ней говорится о тестировании, и к тому же — зачем вы должны этим заниматься.

Если вы ожидаете пользы от программы, то требуется некоторая гарантия, что она действительно делает то, для чего предназначена. Без соответствующей проверки у вас не будет уверенности даже для самостоятельного использования программы, не говоря уж о ее распространении среди других пользователей. Попрограммируйте некоторое время, и вы поймете, что имеется в виду.

Но, может, кто-то другой несет ответственность за тестирование? Да, в больших программных проектах обычно имеется отдельная группа тестировщиков, которая проверяет программное обеспечение. Но если вы работаете в одиночку, то вы и есть эта группа в полном составе. Но даже если вы работаете в команде, вы все равно несете ответственность за тестирование программы, которую пишете.



Тестирование, осуществляемое в NUnit, относится к типу *модульного тестирования*. "Модули", которые вы проверяете, обычно являются отдельными классами и их методами. Чтобы убедиться в том, что класс `BankAccount` не будет обманывать вашего работодателя (банк) или его клиентов, вы должны выполнить множество вводов информации для его методов и удостовериться в корректности получаемых результатов.



Модульное тестирование выполняет две задачи. Первая заключается в проверке правильности поведения программы. Вторая состоит в *поиске ошибок*. Если они в программе есть (а это неизбежно), вы должны обнаружить их и искоренить. Пишите свои тесты так, чтобы вы могли найти ошибки.

Типичный *класс тестирования* в NUnit содержит значительное количество — обычно небольших — *методов тестирования*. Каждый **метод тестирования** проверяет один из методов вашего класса. Может потребоваться несколько или даже множество *методов тестирования*, чтобы охватить все основные компоненты для метода, который вы проверяете: правильный ввод, плюс различные виды неверного ввода, включая ввод вне допустимого диапазона, пустой ввод, опасный ввод, глупый ввод, никогда не случающийся ввод и т.д. Люди, которые фактически используют вашу программу, будут очередными Эйнштейнами в поиске способов "наступить на грабли" в ней и будут давать знать о себе обычно в самое неподходящее время.

Что касается времени, когда следует проводить тестирование, то лучшим будет время написания программы. Если исходить из опыта, то при создании метода следует всегда потратить несколько минут на написание одного или нескольких тестов для него. После того как вы приобретете некоторый опыт, обычно на все это уже не потребуется много времени, и затем вы сможете легко запускать тесты — некоторые из них или все — снова и снова, так что количество правильных выполнений вашей программы будет возрастать, а вы будете знать, что новый код не нарушит старый. Старайтесь не уклоняться от этого принципа — позже это окупается сэкономленным на отладке программ временем.

Написание тестов NUnit

Для работы в программе NUnit добавьте один или несколько *классов тестирования* — они могут быть в любом проекте решения программы, хотя зачастую в целях доступности их легче поместить в тот же проект, в котором находится тестируемая программа. (Если поместить эти классы в отдельный проект, то методы, которые тестируются, должны быть объявлены как `public`, чего обычно делать не следует. Общим правилом должно являться сохранение методов скрытыми, когда это только возможно).

Класс тестирования NUnit может иметь следующий вид:

```
using System;
using NUnit.Framework;      // Эта директива необходима
namespace NUnitTestExample
{
    // Элементы в скобках [] называются "атрибутами"

    [TestFixture]           // Атрибут: это термин NUnit для класса
                             // тестирования
    public class MyTestClass
    {
        // Здесь размещаются все данные, необходимые для
        // большинства или всех тестов, в виде переменных-
        // членов, а также конструктор(ы) (если они
        // необходимы)
        public MyTestClass() { }

        // Метод настройки — вызывается перед каждым методом
        // тестирования; используется для установки одинаковых
```

```

// начальных условий для тестов, так что ни один тест не
// влияет на результаты последующих тестов
[SetUp]           // Атрибут SetUp (обратите внимание на
                  // правильное написание)

public void MySetup()
{
    // Здесь выполняется настройка, необходимая для всех
    // методов тестирования
}
// Здесь вы пишете методы тестирования
// Методы тестирования имеют "атрибут" [Test] и всегда
// объявлены как public void, без аргументов
[Test]
public void StringLengthsGoodTest()
{
}
[Test]
public void CountSpacesTest()
{
}
// Вспомогательный метод для тестов, но не сам тест (нет
// атрибута [Test])
private int CountSpaces(string s)
{
}
} // другие тесты...

```



Вы можете найти законченный пример реального класса тестирования NUnit `NUnitTestFixture` на прилагаемом компакт-диске. На рис. 22.8 показан пример класса тестирования NUnit в редакторе TextPad. Этот же класс тестирования прекрасно работает в Visual Studio и SharpDevelop (в последнем вы можете запускать тесты прямо в среде SharpDevelop с поддержкой программы NUnit).

Изучение класса тестирования программы NUnit

Вот что необходимо знать о классах тестирования.

✓ **Директива using и ссылка.** Вам необходима директива `using` для `NUnit.Framework`, а также ссылка (ключ `/r`) на файл `nunit.framework.dll` в подкаталоге `\bin` в папке программы NUnit на вашей машине; вероятно, это `C:\Program Files\NUnit`.

✓ **Приспособление для теста и тесты.** Программа содержит класс тестирования, или "приспособление для теста" (test fixture), который состоит из методов тестирования и, возможно, дополнительных методов поддержки. Скоро вы познакомитесь с методами тестирования, а на компакт-диске в демонстрационной программе `NUnitTestFixture` представлено несколько их разновидностей.

Демонстрационная программа представляет собой простую программу, основанную на методе `TrimAndPad()` в демонстрационной программе `AlignOutput`

из главы 9, "Работа со строками в C#". Полный текст программы NUnitTestExample здесь приводиться не будет.

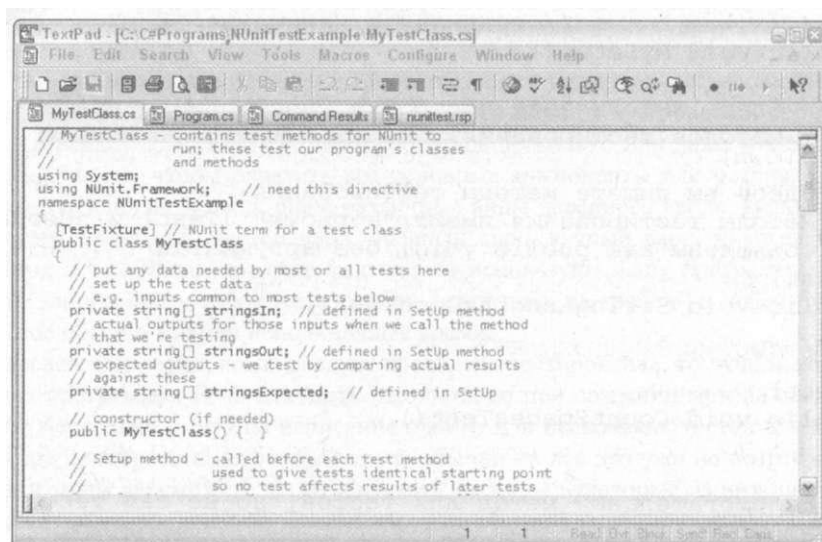


Рис. 22.8. Класс тестирования, загруженный в редактор TextPad, выглядит так же, как и в Visual Studio

✓ **Атрибуты.** Атрибуты — это возможность языка C#, которая больше нигде не рассматривается в данной книге. Они своего рода "украшение", которое можно одеть на классы, методы и другие объекты языка C# для различных целей. Класс тестирования украшен атрибутом [TestFixture], а методы тестирования — атрибутом [Test].

NUnit определяет разные атрибуты, используемые этой программой, чтобы пробраться в ваш скомпилированный файл .EXE или .DLL посредством методики, называемой *отражением* (эта тема также выходит за рамки данной книги). С помощью "включения отражения" сборки .NET программа NUnit может определить все классы (по атрибуту [TestFixture] !) и все методы (по атрибуту [Test]) тестирования. Затем она может просто вызывать только те методы, которые выполняют ваши тесты (без реального запуска всей программы).

Вы увидите несколько других атрибутов в этой главе и в примере NUnitTestExample на компакт-диске.

✓ **Установка и очистка.** Кроме методов тестирования, можно при желании определить по одному методу с атрибутами [Setup], [TearDown], [FixtureSetUp] и [FixtureTearDown].

Методы тестирования должны быть разработаны так, чтобы ни один тест не влиял на любой другой тест. Каждый метод тестирования получает новую песочницу для игр.

Программа NUnit вызывает метод, который вы украсили атрибутом [Setup], всякий раз перед запуском каждого метода тестирования. Это дает возможность обеспечить любые необходимые действия настройки. Например, вы можете использовать этот метод для открытия соединения с базой данных, создания необходимого

объекта, инициализации массива с некоторыми тестовыми вводами и так далее. Аналогичным методом [Setup] является метод [TearDown]. Ваш метод [TearDown], который вы должны выбрать, вызывается прямо после запуска каждого метода тестирования. Используйте его для очистки после тестирования, установки ссылок объектов в null, отключения от базы данных или сети и так далее.

Программа NUnit вызывает аналогичные методы [FixtureSetUp] и [FixtureTearDown], если вы их предоставляете, один раз для каждого полного выполнения испытаний. Метод [FixtureSetUp] запускается перед запуском любого теста, а метод [FixtureTearDown] после завершения работы всех тестов. Иногда можно локализовать действия установки и очистки так, чтобы они не повторялись для каждого метода тестирования, устраняя дублирование.

Написание тестовой программы NUnit

Методы тестирования программы NUnit имеют достаточно стандартную структуру, хотя можно, конечно, проявить творческий подход и использовать их для проверки чего-нибудь значительно большего. (Например, существует программа, которая проверяет, был ли установлен определенный пиксель на экране!) Вот метод тестирования из программы NUnitTestExample, находящейся на компакт-диске.

```
// StringLengthsGoodTest — проверяет корректный ввод (в
// противоположность ошибочному вводу)
// Методы тестирования начинаются с "атрибута" [Test]
[Test]
public void StringLengthsGoodTest() // всегда public void,
// без аргументов
{
    Console.WriteLine("StringLengthsGoodTest:");
    // Здесь выполняется настройка ввода и тому подобного
    // (если вы еще не сделали этого в методе Setup)
    stringsIn = new string[] { "Joe ", "Rumpelstiltskin",
                             " Vanderbilt" };

    // Вызов тестируемого метода
    // Генерируются измененные строки
    stringsOut = Program.TrimAndPad(stringsIn);

    // Сравнение фактических результатов вызова с ожидаемыми
    // В этом тесте мы ожидаем, что все строки будут длиной 16
    // символов. NUnit.Framework предоставляет класс Assert с
    // несколькими методами, включая IsTrue, IsFalse и
    // AreEqual — все они проверяют выполнение логических
    // условий. Первый параметр метода IsTrue является
    // проверяемым логическим условием, вторым параметром
    // является сообщение, которое выводится NUnit, если
    // логическое условие ложно
    Assert.IsTrue(stringsOut[0].Length == 16,
                  "строка0 имеет неверную длину");
    Assert.IsTrue(stringsOut[1].Length == 16,
                  "строка1 имеет неверную длину");
    Assert.IsTrue(stringsOut[2].Length == 16,
                  "строка2 имеет неверную длину");
}
```


Далее описана обычная последовательность метода тестирования в NUnit.

1. **Выполняются все необходимые настройки теста (если вы еще не сделали этого в методе с атрибутами [Setup] или [FixtureSetUp]).**
2. **Вызывается тестируемый метод, получаются результаты его вызова.**

Если метод ничего не возвращает, вам, вероятно, придется проявить творческий подход. Например, если метод копирует файлы из одного места в другое, вам придется использовать язык C#, чтобы подсчитать количество файлов в источнике и получателе, или сравнить все имена файлов, например, применяя методы наподобие описанных в главах 19, "Работа с файлами и библиотеками", и 20, "Работа с коллекциями". Вы можете передать часть работы вспомогательным функциям, а платформа .NET Framework предоставляет большое число классов, которые можно использовать для помощи.



Это служит лишним подтверждением тому, что лучше использовать короткие, простые методы — отчасти потому, что их проще проверять. Вы можете "разложить" большой метод на несколько меньших и протестировать их. (Посетите Web-сайт www.refactoring.com).

3. **Выполняется фактический тест: сравнение полученных результатов с результатами, которых вы ожидаете. Если они совпадают, тест пройден. Если нет, то тест завершился неудачно.**



Вы должны *знать*, чего ожидаете от функции, вводя в нее данные. Используйте контролируемые условия для тестирования и выполняйте тесты с "тестирующими", а не с реальными данными.

Если тест пройден, в окне программы NUnit он будет отмечен зеленой точкой. Если не пройден, он будет отмечен красным, и отобразится информация о том, где произошла ошибка, и что при этом происходило. Используйте эту информацию для поиска неприятностей в коде своей программы.

4. **Выполните все необходимые действия по "уборке за собой".**

В частности, не должны сложиться условия, которые могут влиять на последующие тесты. Это можно сделать в методах [TearDown] или [FixtureTearDown] или же в конце самого метода тестирования.

Иногда может потребоваться добавить один или несколько вспомогательных методов или свойств к *проверяемому классу*, чтобы облегчить тестирование. Я присваиваю таким методам доступ `internal` (не `private` и не `public`) и помечаю их как поддержку тестирования в блоке `#region/#endregion`. Вы можете также создать в своей программе специальный класс `TestSupport`. Делайте методы и свойства этого класса статическими, чтобы их можно было вызывать так, как показано в следующей строке:
`TestSupport.StoreSomethingForTestToCheck(something);`

Дополнительные преимущества этого вызова состоят в том, что он сохраняет большую часть связанного с тестом кода вне программы и помечает элемент поддержки, не внося беспорядка. В своем классе `TestSupport` вы можете предоставить член для хранения данных, а также метод или свойство, которое может использоваться в тестах для получения этих сохраненных данных.

Использование Assert

Механизмом для проведения фактических испытаний и передачи их результатов в программу NUnit для отображения является класс `Assert`, который предоставляет NUnit. Класс `Assert` имеет многочисленные методы для сравнения объектов и выполнения логических проверок.



Методы класса `Assert` включают `IsTrue()`, `IsFalse()`, `IsNull()`, `IsNotNull()`, `AreEqual()`, `AreSame()`, `Fail()` и `Ignore()`. Для получения информации о классе `Assert` и его методах перейдите в подкаталог `\doc` в каталоге NUnit и дважды щелкните на файле `Assertions.html`. (Другие расположенные там файлы содержат полезную информацию об NUnit, доступную также на Web-сайте по адресу www.nunit.org.)



В языке C# используется похожая технология `Assert`, в частности в классе `System.Debug`.

Идея заключается в том, чтобы "сделать заявление" (или "утверждать"), что имеет место некоторое логическое условие. Утверждение терпит неудачу, если это условие ложно, или оно успешно, если условие истинно. В случае неудачного исхода в NUnit отображается сообщение, которое передается в качестве последнего параметра методам `Assert`.

Вы можете проявить творческий подход в тестировании утверждений. Например, если вашим тестирующим вводом является массив данных, который вы передаете тестируемому методу, можно поместить код утверждения внутри цикла так, чтобы утверждение выполнялось для каждого элемента массива, как в приведенном ниже фрагменте из еще одного метода тестирования в программе `NUnitTestExample`.

```
// Подсчет количества пробелов в строке и проверка
// ожидаемого результата
for(int i = 0; i < stringsOut.Length; i++)
{
    // Вспомогательный метод
    int nSpaces = CountSpaces(stringsOut[i]);
    // Проверка утверждения для каждого элемента массива
    // stringsOut
    Assert.AreEqual(nSpaces, spacesExpected[i],
        "Строка " + i + " содержит неверное " +
        "количество пробелов");
}
```

Это хороший способ — проверять сразу целый диапазон вводов в одном тестирующем методе.

Запуск набора тестов NUnit

После того как вы написали свою программу и тесты для нее, необходимо запускать тесты следующим образом.

1. Скомпилируйте свою программу.

Компиляция должна пройти без ошибок, прежде чем вы сможете запускать тесты NUnit.

2. Запустите отдельную программу NUnit.

Она существует в двух вариантах — в форме консольного и графического приложений. Вы, вероятно, предпочтете графическую версию.

В программе SharpDevelop можно запустить NUnit из меню Tools, и результат будет отображаться прямо в среде SharpDevelop в окне Unit Tests.



Вы также можете загрузить инструмент [TestDriven.NET](http://www.testdriven.net) (с Web-сайта www.testdriven.net), который предоставляет похожее отображение результатов тестирования NUnit в среде Visual Studio.

3. В NUnit выберите команду меню **File^Open** и перейдите к папке **\bin\Debug** или **\bin\Release**, где находится программа, только что скомпилированная вами, в виде файла **.EXE** или **.DLL**. В окне открытия файла NUnit дважды щелкните на файле **.EXE** или **.DLL**, который содержит вашу тестируемую программу.

В NUnit будет отображен иерархический список приспособлений для тестирования (классов), а также методов тестирования в окне Test.

4. Щелкните на кнопке **Run**.

В NUnit запустятся тесты (в алфавитном порядке!), и серые точки перед названием каждого метода тестирования станут зелеными (успешно заверченный тест) или красными (произошла ошибка).

5. Взгляните на вкладку **Errors and Failures** справа для просмотра информации о неудачных тестах.

Щелкните на записи об ошибке, чтобы просмотреть стек вызовов в панели под методом, потерпевшим неудачу: вы увидите последовательность вызовов метода, которая привела к неудаче; для каждого вызова приведены имя файла и номер строки (стек вызовов описан в главе 21, "Использование интерфейса Visual Studio"). (В NUnit отображается и другая полезная информация, такая как обычный вывод программы в консоль на вкладке **Console.Output** и ошибочный вывод программы на вкладке **Console.Error**.)

В следующий раз, когда вы захотите выполнить тесты, перекомпилируйте свою программу и переключитесь в окно NUnit. Программа автоматически перезагрузит ваш измененный код (вы также можете сделать это вручную посредством команды **Reload** в меню **File**). Щелкните на кнопке **Run**.



Вы также можете щелкнуть на одном тесте в программе NUnit для запуска только его.

На рис. 22.7, приведенном выше, показан ряд тестов программы NUnit, которые только что были выполнены. Один из них прошел неудачно (это было сделано преднамеренно). **FailingTest** отмечен более темной (красной) точкой (и, как результат, **MyTestClass** и другие элементы иерархии, расположенные над ним). Индикатор выполнения под кнопкой **Run** также красный. Успешно заверченные тесты отмечены светлыми (зелеными) точками. В программе отображается также стек вызовов для неудачного теста.

Исправление ошибок в проверяемой программе

Не исключено, что вы могли совершить ошибки и в самих тестирующих методах, так что любые результаты, полученные в программе NUnit, являются ошибочными независимо от проверяемого метода или от метода тестирования. Бррр... Кто будет следить за наблюдателями? Когда такое происходит, было бы хорошо иметь возможность пройти методы тестирования в отладчике, чтобы можно было проверить переменные и отследить проблему.

Имеется только одна загвоздка: вашу проверяемую программу запускает NUnit, а не Visual Studio или TextPad. И что знает программа NUnit об отладчиках для языка C#?

Однако все же имеется способ отладить тестирующую программу. (В действительности их несколько. Вам будет показан один из них.)



Зачем присоединяться к процессу NUnit?

Каждая запущенная программа запускает в Windows *процесс*, отдельную область памяти, выделенную только для данной программы. Внутри этой области программа сохраняет свои переменные и работает практически так же, как если бы она имела в своем распоряжении весь компьютер. Когда программы Visual Studio и NUnit выполняются одновременно, каждая из них находится в своем собственном процессе, и вы должны предоставить программе Visual Studio (или отладчику CLR) доступ к процессу NUnit для отладки.



В следующих разделах описано, как отлаживать тесты NUnit из таких отладчиков:

- ✓ отладчик Visual Studio (когда вы работаете в Visual Studio);
- ✓ отладчик CLR (когда вы работаете в SharpDevelop, TextPad или из командной строки).

Запуск тестов NUnit в отладчике Visual Studio

Выполните следующие действия для отладки своей программы тестирования, если вы работаете в Visual Studio (в следующем разделе рассматривается работа в TextPad или SharpDevelop, или из командной строки).

1. **Скомпилируйте программу без ошибок времени компиляции.**
2. **Запустите NUnit и убедитесь, что загружен проект, который вы хотите отладить.**
Не забывайте об этом шаге и обязательно проверьте, файл какой именно программы вы загрузили в NUnit.
3. **В Visual Studio откройте файл, содержащий класс тестирования. Установите точку останова в строке, в которой хотите остановить отладчик.**
Эта строка будет в одном из ваших методов тестирования.
4. **В Visual Studio выберите команду меню **Debugs Attach to Process**.**
О том, зачем это делать, рассказывается во врезке "Зачем присоединяться к процессу NUnit?"
5. **В диалоговом окне **Attach to Process** найдите в списке **Available Processes** процесс **nun.it-gui.exe** (в левой колонке).**

В поле Title этого "процесса" будет написано что-то вроде NUnitTestExample.exe — NUnit (в зависимости от названия тестируемой программы). На рис. 22.9 показано, как выглядит диалоговое окно Attach to Process с выбранным процессом NUnit для присоединения.

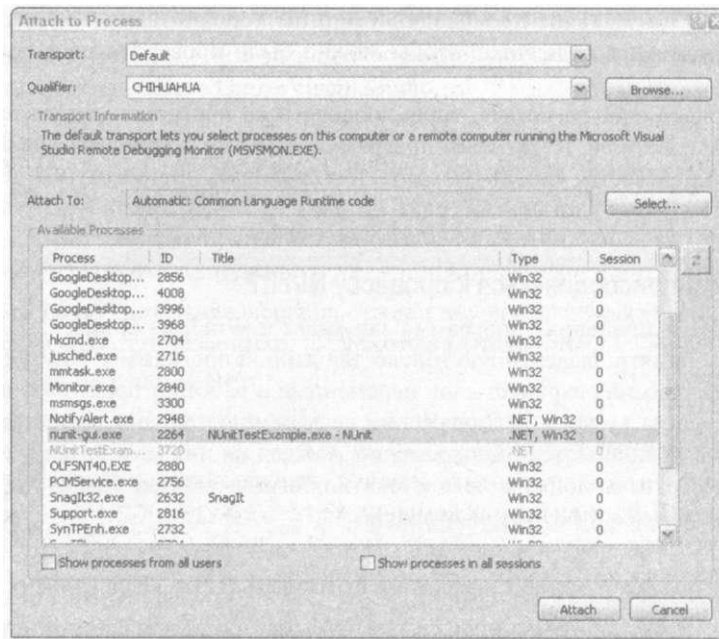


Рис. 22.9. Присоединитесь к NUnit как ко внешнему процессу для отладки программы тестирования

6. Щелкните на кнопке **Attach**.

7. Нажмите комбинацию клавиш <Alt+Tab> для переключения в программу NUnit. Щелкните на кнопке **Run**.

Отладчик остановится на строке с точкой останова и подцветит ее. Вы можете начать пошаговое выполнение программы с этой точки.



Можно установить точки останова как в программе тестирования, так и в тестируемой программе, но в тесте такая точка должна быть только одна. С нее и начинается отладка.

Запуск тестов NUnit в отладчике CLR

Выполнение тестов NUnit в отладчике CLR очень похоже на их выполнение в отладчике Visual Studio. Вам необходимо делать это, когда вы работаете с NUnit из программ TextPad или SharpDevelop.

Для запуска программы тестирования в отладчике CLR выполните следующие действия.

1. Запустите отладчик CLR как обычно, из меню **Tools** в программе TextPad или SharpDevelop, или при необходимости запустите его как самостоятельное приложение.

2. Запустите NUnit (графическую или консольную версию).
3. В отладчике, как обычно, загрузите программу для отладки и соответствующие ей файлы .CS.
4. Установите в отладчике точку останова в строке исполняемой программы *в одном из ваших методов тестирования*.
5. Выберите команду меню **Debug^Processes**. В диалоговом окне **Processes** выберите процесс `nunit-gui.exe` (или `nunit-console.exe`) и щелкните на кнопке **Attach**. Затем — на кнопке **Close**.
6. Нажмите клавишу <F10> для выполнения программы с пропуском вызовов функций или <F11> для пошагового выполнения с заходом в функции.
7. Отладчик остановится в установленной вами точке останова.
8. Продолжайте пошаговое выполнение, проверку переменных и т.д.

Этот процесс может перевести вас из испытательного кода в код вашей программы по вызову испытательным методом. Ошибка может быть в любой из данных областей.

Прекращение отладки



Для того чтобы прекратить отладку, переключитесь в Visual Studio (или отладчик CLR) и выберите команду меню **Debug<=>Stop Debugging**. Это приведет к отсоединению от процесса NUnit, и вы сможете продолжать программирование.

В любой момент, когда вы захотите отладить тесты вашего проекта, повторите предыдущие шаги.

Написание исходного текста Windows Forms без Form Designer

Одной из наиболее недостающих возможностей в TextPad является отсутствие визуальных средств для размещения управляющих элементов в ваших формах Windows. Большинство читателей этой книги хотят создавать программы для Windows, и все, что можно здесь сделать — это порекомендовать SharpDevelop. Но можно показать и еще одну альтернативу — написание такого кода вручную.

Это всего лишь код

Не забывайте, что код, генерируемый визуальными средствами для разработки форм — всего лишь код. Он следует определенным соглашениям, к которым вы должны присоединиться *при его написании вручную*, но это не более чем код на C#. Вы можете написать его самостоятельно следующим образом.



Просто собезьянничав и скопировав код, который был разработан визуальными средствами.



✓ Написав важные части кода самостоятельно и отбросив ненужное, поскольку вы не играете в эти игры.

Далее будут вкратце рассмотрены оба подхода.

Работа в стиле визуального инструмента

Написание такого же кода, как и создаваемый визуальными инструментами, упирается в знание формата.

В главе 1, "Создание вашей первой Windows-программы на C#", вы вкратце знакомились с программированием Windows Forms и создавали маленькое приложение, которое выводило форму (окно) с двумя текстовыми полями и одной кнопкой. При щелчке на кнопке программа копировала текст из верхнего поля в нижнее.



Если у вас нет Visual Studio, вы не сможете следовать инструкции из главы 1, "Создание вашей первой Windows-программы на C#", так что идентичный код предоставлен в демонстрационной программе *Imitating-FormDesigner* на прилагаемом компакт-диске. Вы можете увидеть тот же код, что и в главе 1, "Создание вашей первой Windows-программы на C#", но он написан мною самостоятельно, без визуального инструментария.

За красивыми формами, создаваемыми при помощи мыши, Visual Studio скрывает код, который создает все эти управляющие элементы и заставляет их работать.

Visual Studio создает весь код за исключением того, что происходит при взаимодействии с пользователем. Например, когда пользователь щелкает на кнопке, выполняются какие-то действия, и написать код для этих действий — это уже ваша забота. В описанном примере действие представляет собой одну строку на C#, которая копирует текст из `textBox1` в `textBox2`:

```
textBox2.Text = textBox1.Text;
```

Когда вы создаете новое приложение Windows (в отличие от консольных приложений, которые использовались во всей этой книге), Visual Studio 2005 генерирует три исходных файла. Два из них содержат часть класса `Form1`, который лежит в основе формы, представляющей пользовательский интерфейс вашей программы. В третьем файле находится класс `Program`, содержащий функцию `Main()`.

Вот более детальное описание полученных вами файлов.



✓ `Form1.cs`: этот файл содержит объявление *частичного класса* (partial class) для класса `Form1`. (Что такое частичный класс, будет объяснено позже.) Это только половина класса `Form1` (вторая его половина находится в другом файле), но это та половина, в которой вы пишете весь код, который должен быть создан вручную.



✓ `Form1.Designer.cs`: данный файл содержит вторую половину кода частичного класса `Form1`. Это часть, которую Visual Studio 2005 создает для вас.

Не трогайте этот файл. Добавляйте код только в первую половину класса `Form1` в файле `Form1.cs`. Когда вы добавляете или модифицируете управляющий элемент в проектировщике форм, он сам изменит код — вы не должны в это вмешиваться.



✓ `Program.cs`: этот файл содержит функцию `Main()`.

В демонстрационной программе `ImitatingFormDesigner` я написал определенный код в файле `Form1.cs` для того, чтобы программа выполняла некоторые действия: обработчик щелчка `button1` копирует текст из `textBox1` в `textBox2`, как и в главе 1, "Создание вашей первой Windows-программы на C#", в которой на рис. 1.10 показано, как это выглядит на экране компьютера.

Взгляните на код второй демонстрационной программы на прилагаемом компакт-диске — `Forms YourWay`. Этот код похож на неприкасаемый код из файла `Form1.Designer.cs` демонстрационной программы `ImitatingFormDesigner`, но в него включены комментарии, поясняющие, что и как работает.



Главное заключается в том, что при возможности использовать проектировщик форм Visual Studio (или его коллегу из SharpDevelop) дизайнер выполняет массу работы за вас, но все, что он делает — это всего лишь код, который — сам собой, с большими затратами труда — вы можете написать и самостоятельно.

Частичные классы

Частичные классы — новая возможность C# 2.0. Идея заключается в разбиении класса на два или большее количество файлов. Каждый файл содержит *частичный класс* (partial class), помеченный новым ключевым словом `partial`:

```
// Файл 1
public partial class MyClass
{
    // Часть класса MyClass, но не весь — остальная его часть
    // находится в другом файле
}

// Файл 2
public partial class MyClass // Тот же заголовок
{
    // Еще одна часть класса MyClass, но не весь он — словом,
    // идея вам понятна?
}
```

Главное преимущество частичных классов в том, что теперь вы можете сделать следующее:



- ✓ разделить работу над одним классом одновременно между несколькими программистами;
- ✓ отделить часть класса, которую вы создали с помощью некоторого инструмента, генерирующего код, от части(ей), написанной человеком.

Вряд ли вы будете часто использовать эту возможность, но там, где она полезна — она очень и очень нужна.

Главная причина для отделения сгенерированного машиной кода от кода, написанного человеком, в том, что при каждом запуске генератора для добавления или внесения изменений (например, для добавления новой кнопки) он генерирует код заново. Если в этот код были добавлены ваши изменения, при повторной генерации они пропадут. Но если вы отделите свой код от машинного, то тем самым спасете его от механического вмешательства компьютера.



Позже вы сможете перенести код из сгенерированной части в написанную вами, но это приведет к тому, что дизайнер не сможет с ним работать. Например, если вы перенесли весь код, касающийся некоторой кнопки, включая ее создание, в вашу половину, то кнопка больше не будет появляться в дизайнера, хотя будет выводиться при работе программы. Именно так вы можете добавлять визуальные элементы в форму или модифицировать ее динамически, во время выполнения программы. Если вы переместите только часть кода, при повторной генерации дизайнер может восстановить ее, что приведет к дублированию и ошибкам компиляции. Такое перемещение следует выполнять очень осторожно.

Как уже упоминалось, частичные классы связаны в первую очередь с кодом дизайнера форм Visual Studio, что наглядно показывает демонстрационная программа `ImitatingFormDesigner`.

Самостоятельное написание



Чтобы посмотреть, какой код вы *обязаны* написать при отсутствии визуального инструментария, взгляните на предельно упрощенную схему демонстрационной программы `FormsYourWay` на прилагаемом компакт-диске. Комментарии в этом коде указывают, куда следует добавить код управляющих элементов.

Код демонстрационной программы `FormsYourWay` состоит в основном из фигурных скобок, комментариев и кода, связанного с управляющими элементами формы.

Одна из интересных особенностей этой программы состоит в том, что в ней нет файла `Program.cs` с функцией `Main()`. Вместо этого функция `Main()` сделана методом самого класса `Form1`. Данная функция может находиться в любом классе. Просто убедитесь, что в программе имеется ровно одна функция `Main()`. Кстати, взгляните на эту функцию `Main()`, чтобы иметь представление, как выглядит типичная функция `Main()` для графического приложения Windows.

Ознакомьтесь со сводкой элементов кода, необходимых для размещения управляющих элементов в форме с помощью одного лишь кода, без применения дизайнера.

- ✓ Объявить управляющие элементы (как члены-данные класса формы). Они должны иметь классы наподобие `Button`, `TextBox` или `Label` из пространства имен `System.Windows.Forms`.

```
System.Windows.Forms.Button button1;
```
- ✓ При желании можно использовать директиву `using` для пространства имен `System.Windows.Forms`.
- ✓ Инстанцировать каждый управляющий элемент (в конструкторе формы или методе, который он вызывает).

```
button1 = new System.Windows.Forms.Button()
```
- ✓ Установить свойства каждого управляющего элемента (в конструкторе формы или методе, который он вызывает). Вы должны поэкспериментировать, чтобы определить корректные координаты `Location` и значения `Size` (обратите внимание, что в примере параметр `y` у `Size` не используется, так что указано значение 0).

```
button1.Location = new System.Drawing.Point(40, 80);  
button1.Name = "button1";
```

```
button1.TabIndex = 1;
button1.Text = "Copy";
```

- ✓ Добавить обработчик для каждого события, которое вы хотите обработать, для каждого из управляющих элементов (в конструкторе). Добавление обработчика для кнопки выглядит примерно так:

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

В представленном примере единственное событие, требующее обработки, — это событие Click кнопки. Однако вам могут потребоваться и другие события. Аргумент в скобках именуется методом Form1, который отвечает на щелчок кнопки. В данном случае он называется button1_Click().

- ✓ Добавить каждый угадывающий элемент в коллекцию Controls (в конструкторе), `this.Controls.Add(button1);`

- ✓ Написать обработчики для всех событий, связанных с каждым управляющим элементом (в качестве методов формы; после конструктора). Метод обработчика кнопки имеет примерно следующий вид:

```
private void button1_Click(object sender, System.EventArgs e)
{
    // Некоторые действия в ответ на щелчок на кнопке
    MessageBox.Show("Hello");
    // Или, как в демонстрационной программе FormsYourWay:
    textBox2.Text = textBox1.Text;
}
```

Чрезвычайно простой код демонстрационной программы FormsYourWay использует единственное небольшое интерактивное окно, показанное на рис. 1.10. Другими словами, результат идентичен полученному в главе 1, "Создание вашей первой Windows-программы на C#", и ранее в этой главе. Однако сами коды несколько отличаются друг от друга.



Вам потребуется масса справочной информации о свойствах классов управляющих элементов, таких как Button, TextBox, ListBox и т.д., а также о классах Form и его базовых классах. Все это можно найти в справочной системе .NET SDK.

Убедитесь, что пользователи смогут запустить вашу программу

Представьте, что вы написали программу, выпустили ее окончательную версию и отправили в бурное море людей и событий, где кто-то установит ее на свой компьютер и попытается запустить? Да, эта программа работала (вполне корректно) на вашем компьютере, но будет ли она функционировать на чужой машине? Вы должны перед тем как пускать программу по водам, проверить ее по возможности на максимально большем количестве компьютеров с разными конфигурациями.

Что, если на целевой машине не установлен .NET? Да, политика Microsoft такова, чтобы предельно забить винчестеры пользователей всем, что только можно продать, ну а вдруг?...

Вы не Microsoft и не должны требовать от ваших пользователей установить .NET SDK. Вместо этого вы можете распространять с вашей программой пакет *.NET redistributable package* Dotnetfx.exe. Его можно бесплатно загрузить с MSDN и распространять с вашими программами. Сделайте установку этого пакета частью установки вашей программы, и пользовательская машина будет корректно настроена для выполнения программ .NET — включая все необходимые библиотеки времени выполнения, библиотеки базовых классов и компилятор ЛТ. Если на целевом компьютере уже имеется старая версия .NET, но не та, в которой нуждается ваша программа, данная инсталляция не затронет ее и установит новую версию "рядом", так что они обе будут к услугам нуждающихся в них программ (само собой, Dotnetfx.exe изменяется с каждой новой версией .NET).



Один из типов проектов, которые можно создавать в Visual Studio (или SharpDevelop) — проект установки. Он создает .MSI-файл (Microsoft Installer), на котором пользователь может дважды щелкнуть мышью для установки вашей программы на своем компьютере. Это возможность выходит за рамки настоящей книги, но документирована в .NET SDK. Однако для небольших программ можно не создавать .MSI-файл, а просто скопировать ее на пользовательский жесткий диск. Microsoft также предоставляет способ инсталляции Интернет-приложений, написанных с использованием ASP.NET, через Интернет.

Visual Studio для бедных

Если вы попытаетесь программировать в Visual Studio, вы обязательно захотите приобрести его.



Учтите, что наряду с предельно дорогими профессиональными версиями имеются недорогие (около 50 долл.) версии Visual Studio. В главе 19, "Работа с файлами и библиотеками", упоминалась версия Visual C# Express — подумайте о возможности ее приобретения.

Но тем не менее все описанные в данной главе инструменты — весьма достойная замена Visual Studio.

Предметный указатель

.NET, 31

A

as, 274

B

base, 278; 290

bool, 65

C

catch, 400

char, 66

const, 398

D

decimal, 64

default, 360

DLL, 421

double, 62

E

else, 89

enum, 398

Exception, 401

F

finally, 400

float, 62

foreach, 133; 460; 464

G

goto, 111

I

if, 86

int, 58

interface, 313

internal, 235

is, 272

J

Java, 31

N

namespace, 422

O

out, 157

override, 294

P

partial, 561

private, 234; 235

protected, 235

public, 231; 235

R

readonly, 398

Real numbers, 61

ref, 157

return, 162

Rounding, 61

S

sealed, 308

signed, 60

string, 67; 199–221

struct, 327
switch, 109

T

this, 185
throw, 400
Truncation, 61
try, 400

U

UML Lite, 299
Unicode, 436
unsigned, 60
using, 425; 426

V

value, 242
Value types, 67
virtual, 294
Visual Studio, 32
 Solution Explorer, 493
 Автозавершение, 190; 503
 Генерация XML-документации, 197
 Дополнение справочной системы, 193
 Настройка расположения окон, 487
 Справка по встроенным функциям, 191
 Справочная система, 506
void, 166

W

where, 359
Windows Forms, 36
WriteLine(), 173

Y

yield, 474
yield break, 476
yield return, 475

A

Абстрактный класс, 302

Абстракция, 225
Автозавершение, 190; 503
Аргументы по умолчанию, 156
Асинхронный ввод-вывод, 434

Б

Базовый класс, 263
Беззнаковые целые числа, 60
Библиотеки классов, 430
Блок итератора, 469

В

Венгерская запись, 69
Вложенный цикл, 106
Возврат значения из функции, 162
Вызов функции, 88

Д

Действительные числа, 61
Деструктор, 281
Директива using, 426

З

Зацикливание, 97
Знаковые целые числа, 60

И

Индекс массива, 125
Индексатор, 465
Инициализация, 59
Инстанцирование, 117; 339
Интерфейс, 228; 312
 Абстрактный, 323
 Обобщенный, 364
 Предопределенный, 316
Исключение, 400
 Пользовательское, 405
 Регенерация, 411
Итератор, 461
 Блок, 469
 Именованный, 477
 Синтаксис, 475

К

Класс, 775
 object, 273
 Абстрактный, 302
 Базовый, 263
 Библиотека, 430
 Конструктор, 244
 Конструктор по умолчанию, 246
 Метод, 177; 182
 Обобщенный, 339
 Оболочка, 348
 Объект, 117
 Ограничение доступа, 231
 Опечатывание, 308
 Определение, 116
 Определение метода, 181
 Определение функции, 179
 Перегрузка конструкторов, 253
 Свойство, 242
 Статические члены, 123
 Статическое свойство, 243
 Функции доступа, 242
 Функция-член, 141
 Частичный, 561
 Члены, 116
 Экземпляр, 117; 181
Классификация, 227
Код ошибки, 395
Коллекция, 474
Комментарий, 57
 Документирующий, 794; 499
Конструктор по умолчанию, 246

Л

Логическое сравнение, 77

М

Массив, 724
 Индекс, 725
 Свойство Length, 729
 Фиксированного размера, 725
Метод, 182
Модуль, 235; 421

Н

Наследование, 230; 261
 Конструктор базового класса, 276
Недетерминистическая деструкция, 287
Нулевой объект, 720

О

Область видимости, 704
Обобщенные классы, 339
 Создание, 347
Объект, 777
 Нулевой, 720
Объявление, 59
Окно DOS, 50
Окно документа, 38
Окно управления, 38
Округление, 67
Оператор, 73
 as, 274
 break, 98
 continue, 98
 if, 86
 is, 272
 Безусловного перехода, 777
 Бинарный, 74
 Декремента, 77
 Деления по модулю, 74
 Инкремента, 76
 Префиксный и постфиксный, 76
 Приоритеты, 74
 Присваивания, 58; 75
 Присваивания составной, 76
 Составной логический, 79
 Тернарный, 83
 Умножения, 73
Отладка, 572
 Пошаговое выполнение, 574
 Точка останова, 577
Отношение
 МОЖЕТ_ИСПОЛЬЗОВАТЬСЯ_КАК,
 311
 СОДЕРЖИТ, 269
 ЯВЛЯЕТСЯ, 263

п

Перегрузка функции, 753; 253; 284
Передача аргументов в программу, 767
Переменная, 57
 Инициализация, 59
 Объявление, 57
Переполнение буфера, 467
Перечислитель, 462; 474
Повышение типа, 81
Позднее связывание, 292
Полиморфизм, 230; 291
Полностью квалифицированное имя, 425
Понижение типа, 82
Преобразование типов, 70
Присваивание, 59
Пробельный символ, 207
Проект, 174; 420; 494
Пространство имен, 421; 422
 Объявление, 422
Пузырьковая сортировка, 755
Пустая строка, 67

р

Работа с файлами, 434
Разделение программы, 419
Разложение классов, 300
Разложение кода, 504
Регистр, 67
Рекурсия, 290
Рефакторинг, 148; 505
Решение, 174; 420; 494

с

Сборка, 427
Сборка мусора, 727
Связанный список, 457
Символ
 Пробельный, 207
Соглашения по именованию, 69
Сокращенное вычисление, 80
Скрытие метода базового класса, 285
Специальные символы, 66
Сравнение
 Чисел с плавающей точкой, 78
Ссылка, 720; 131

Строка, 67; 799-227
 Использование switch, 205
 Конкатенация, 200
 Неизменность, 207
 Сравнение, 207
 Сравнение без учета регистра, 205
 Форматирование, 272; 218
 Модификаторы, 218
 Форматная, 275
Структура, 327
 Конструктор, 329
 Предопределенные типы, 333

т

Тип
 Повышение, 81
 Понижение, 82
Типы с плавающей точкой, 62
Типы-значения, 67
Точка останова, 57 7

у

Уровень абстракции, 226
Усечение, 67

ф

Файл
 FileStream, 438
 StreamWriter и StreamReader, 435
 Проекта, 420; 494
Форма, 36
Функция, 747
 Аргументы, 749
 Передача по значению, 756
 Передача по ссылке, 757
 По умолчанию, 756
 Возврат значения, 762
 Вызов, 88
 Перегрузка, 755; 253; 284
 Функция-член, 747

ц

Целочисленные типы, 60
Цикл, 93

do...while, 98
for, 104
foreach, 134
while, 93
Бесконечный, 203
Вложенный, 106
Счетчик, 97

щ

Шаблон консольного приложения, 47
Шаблоны программы, 33

Э

Экземпляр, 117; 227
Элемент управления, 38

Числа с плавающей точкой, 61

Научно-популярное издание

Стефан Рэнди Дэвис, Чак Сфер

С# 2005 для "чайников"

В издании использованы карикатуры американского художника Рича Теннанта

Литературный редактор	<i>Т.Г. Сковородникова</i>
Верстка	<i>О.В. Романенко</i>
Художественные редакторы	<i>В.Г. Павлютин, Т.А. Тараброва</i>
Корректор	<i>Л.А. Гордиенко</i>

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 14.11.2007. Формат 70х100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 27,6. Уч.-изд. л. 46,44.

Тираж 1000 экз. Заказ № 5558

Отпечатано по технологии СtР
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.



C# 2005 "чайников"™



Шпаргалка

Операторы

Приоритет	Операторы	Унарность	Ассоциативность
Высокий	() [] . new typeof	Унарный	Слева направо
	! ~ + - ++ -- (приведение типа)	Унарный	Слева направо
	* / %	Бинарный	Слева направо
	+ -	Бинарный	Слева направо
	< <= > >= is as	Бинарный	Слева направо
	== !=	Бинарный	Слева направо
	&	Бинарный	Слева направо
	^	Бинарный	Слева направо
		Бинарный	Слева направо
	&&	Бинарный	Слева направо
Низкий		Бинарный	Слева направо
	?:	Тернарный	Справа налево
	= *= /= %= += -= &= ^= = <<= >>=	Бинарный	Справа налево

Типы целочисленных переменных

Тип	Размер (байты)	Диапазон значений	Использование
sbyte	1	от -128 до 127	sbyte sb = -12;
byte	1	от 0 до 255	byte b = 12;
short	2	от -32 768 до 32 767	short sn = -123;
ushort	2	от 0 до 65 535	ushort usn = 123;
int	4	от -2 147 483 648 до 2 147 483 647	int n = 123;
uint	4	от 0 до 4 294 967 295	uint un = 123U;
long	8	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	long l = 123L;
ulong	8	от 0 до 18 446 744 073 709 551 615	long ul = 123UL;

Другие типы переменных

Тип	Диапазон	Использование
decimal	до 28 цифр	decimal d = 123M;
char	от 0 до 65,535 (коды в наборе символов Unicode)	char x = 'c'; char newline = '\n';
string	от пустой строки ("") до очень большого количества символов из набора Unicode	string s = "my name"; string empty = "";
bool	true и false	bool b = true;



C# 2005 "чайников"™



Типы переменных с плавающей точкой

Шпаргалка

Тип	Размер (байты)	Диапазон	Точность	Использование
float	8	от $1.5 \cdot 10^{-45}$ до $3.4 \cdot 10^{38}$	6-7 цифр	float f = 1.2F;
double	16	от $5.0 \cdot 10^{-324}$ до $1.7 \cdot 10^{308}$	15-16 цифр	double d = 1.2;

Управление потоком выполнения

```
if (i < 10)
{
    // Выполняется, если i меньше 10
}
else
{
    // Выполняется в противном случае
}

while(i < 10)
{
    // Цикл выполняется, пока i меньше 10
}

for(int i = 0; i < 10; i++)
{
    // Выполнение 10 итераций
}

foreach(MyClass mc in myCollection)
{
    // Выполняется по одному разу для каждого объекта mc из
    // коллекции myCollection
}
```

Определение класса

```
[access][<abstract | sealed>]class MyClassName
[ : [BaseClass] [, Interface, ...] ]
{
    [static][access]type dataMember;
    [<static|virtual|abstract|new|override>]
    [access]type method(... args ...)
```

Для классов `access` может быть

`public|protected|internal|private`

Для членов класса `access` может также принимать значения

`protected internal`

Примечания:

[feature]

<feature1 | feature2>

...

Необязательный параметр

Либо feature1, либо feature2

Неопределенное количество инструкций или выражений

Файл был скачен с сайта

<http://Knigaluby.Ru>

Данный файл представлен только для ознакомления. После ознакомления данный файл необходимо удалить. Сохраняя данный файл, Вы несете ответственность в соответствии с законодательством.