THE EXPERT'S VOICE ® IN NET

C# 2008

УСКОРЕННЫЙ КУРС

ДЛЯ ПРОФЕССИОНАЛОВ

www.williamspublishing.com

Apress®

www.apress.com

Трей Нэш

Предисловие Веса Дайера, участника команды разработки языка C# в Microsoft

ACCELERATED C# 2008

Trey Nash

Apress®

C# 2008
УСКОРЕННЫЙ КУРС

ДЛЯ ПРОФЕССИОНАЛОВ

Трей Нэш

Москва • Санкт-Петербург • Киев
2008

ББК 32.973.26-018.2.75
Н95
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *Н.А. Мухина*

Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Нэш, Трей.

Н95 C# 2008: ускоренный курс для профессионалов. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 576 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1377-7 (рус.)

Книга ведущего специалиста в области технологий .NET представляет собой интенсивный курс по новейшей версии языка C#, воплотившей в себе важные дополнения и предлагающей среду, в которой функциональное программирование может органично переплетаться с обычным стилем императивного программирования на C#. Подробно рассматриваются такие темы, как фундаментальные принципы объектно-ориентированного проектирования, основные структуры данных; обработка исключений, делегаты, анонимные функции, контракты и интерфейсы, события, обобщения и многопоточность, а также нововведения наподобие лямбда-выражений, расширяющих методов и языка LINQ. Книга изобилует множеством примеров, которые не только иллюстрируют концепции, но также демонстрируют способы правильной разработки и умеренного их применения в реальных условиях.

Книга рассчитана на программистов разной квалификации, а также будет полезна студентам и преподавателям дисциплин, связанных с программированием и разработкой для.NET.

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 APress, Berkeley, CA.

Authorized translation from the English language edition published by APress, Copyright © 2008 by Weldon W. Nash, III

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2008.

ISBN 978-5-8459-1377-7 (рус.)
ISBN 978-1-59059-873-3 (англ.)

© Издательский дом "Вильямс", 2008
© by Weldon W. Nash, III, 2008

Оглавление

Предисловие 14

Об авторе 15

О техническом редакторе 15

Благодарности 16

Введение 17

Глава 1. Обзор C# 22

Глава 2. C# и CLR 31

Глава 3. Обзор синтаксиса C# 39

Глава 4. Классы, структуры и объекты 63

Глава 5. Интерфейсы и контракты 159

Глава 6. Перегрузка операций 186

Глава 7. Исключения: безопасность и обработка 199

Глава 8. Работа со строками 235

Глава 9. Массивы, типы коллекций и итераторы 264

Глава 10. Делегаты, анонимные функции и события 300

Глава 11. Обобщения 328

Глава 12. Многопоточность в C# 369

Глава 13. В поисках канонических форм C# 423

Глава 14. Расширяющие методы 482

Глава 15. Лямбда-выражения 509

Глава 16. LINQ: язык интегрированных запросов 533

Приложение. Справочная информация 567

Предметный указатель 570

Содержание

Предисловие 14

Об авторе 15

О техническом редакторе 15

Благодарности 16

Введение 17

- Как организована эта книга 18
- От издательства 21

Глава 1. Обзор C# 22

- Отличия между C# и C++ 22
  - Язык C# 22
  - Язык C++ 23
  - Сборка мусора CLR 24
- Пример программы на C# 24
- Обзор средств, добавленных в C# 2.0 26
- Обзор новшеств C# 3.0 28
- Резюме 29

Глава 2. C# и CLR 31

- JIT-компилятор и CLR 32
- Сборки и загрузчик сборок 33
  - Минимизация рабочего набора приложения 34
  - Присвоение имен сборкам 35
  - Загрузка сборок 35
- Метаданные 36
- Межязыковые возможности 38
- Резюме 38

Глава 3. Обзор синтаксиса C# 39

- C# — строго типизированный язык 39
- Выражения 40
- Операторы и выражения 41
- Типы и переменные 42
  - Типы значений 43
  - Ссылочные типы 46
  - Инициализация переменных по умолчанию 47
  - Неявно типизированные локальные переменные 48
  - Преобразования типов 50
  - Операции `as` и `is` 52
  - Обобщения 54

Пространства имен 56
Определение пространств имен 57
Использование пространств имен 58
Поток управления 59
if-else, while, do-while и for 60
switch 60
foreach 61
break, continue, goto, return и throw 61
Резюме 62

Глава 4. Классы, структуры и объекты 63

Определения классов 65
Поля 66
Конструкторы 70
Методы 70
Свойства 72
Инкапсуляция 77
Доступность 81
Интерфейсы 83
Наследование 85
Герметизированные классы 92
Абстрактные классы 93
Вложенные классы 94
Индексаторы 97
Частичные классы 100
Частичные методы 101
Статические классы 102
Зарезервированные имена членов 104
Определения типов значений 105
Смысл this 108
Финализаторы 111
Интерфейсы 111
Анонимные типы 112
Инициализаторы объектов 115
Упаковка и распаковка 118
Когда происходит упаковка 122
Эффективность и путаница 124
System.Object 125
Эквивалентность и ее смысл 126
Интерфейс IComparable 127
Создание объектов 127
Ключевое слово new 127
Инициализация полей 128
Статические конструкторы (класса) 130
Конструктор экземпляра и порядок создания 133
Уничтожение объектов 137

Финализаторы 137
Детерминированное уничтожение 139
Обработка исключений 140
Одноразовые (disposable) объекты 140
Интерфейс IDisposable 140
Ключевое слово using 143
Типы параметров методов 144
Аргументы-значения 145
Аргументы ref 145
Параметры out 147
Массивы params 148
Перегрузка методов 148
Наследование и виртуальные методы 149
Виртуальные и абстрактные методы 149
Методы new и override 150
Методы sealed 152
Завершающие несколько слов о виртуальных методах C# 153
Наследование, включение и делегирование 153
Выбор между интерфейсом и наследованием класса 154
Сравнение делегирования и композиции и наследования 155
Резюме 158

Глава 5. Интерфейсы и контракты 159

Интерфейсы определяют типы 160
Определение интерфейсов 161
Что может быть интерфейсом? 162
Наследование интерфейсов и сокрытие членов 163
Реализация интерфейсов 165
Неявная реализация интерфейса 165
Явная реализация интерфейса 166
Переопределение реализаций интерфейсов в производных классах 168
Берегитесь побочных эффектов от реализации интерфейсов типами значений 171
Правила сопоставления членов интерфейсов 172
Явная реализация интерфейса с типами значений 176
Соображения, касающиеся версий 178
Контракты 179
Контракты, реализованные классами 179
Интерфейсные контракты 181
Выбор между интерфейсами и классами 182
Резюме 185

Глава 6. Перегрузка операций 186

Можете — не значит должны 186
Типы и форматы перегруженных операций 186
Операции не должны изменять свои операнды 188

Имеет ли значение порядок параметров? 188
Перегрузка операции сложения 189
Операции, допускающие перегрузку 190
Операции сравнения 191
Операции преобразования 193
Булевские операции 195
Резюме 198

Глава 7. Исключения: безопасность и обработка 199

Как CLR трактует исключения 199
Механика обработки исключений в C# 200
Генерация исключений 200
Изменения, касающиеся необработанных исключений, которые появились в .NET 2.0 201
Обзор синтаксиса оператора `try` 202
Повторная генерация и трансляция исключений 204
Исключения, сгенерированные в блоке `finally` 207
Исключения, сгенерированные в финализаторах 207
Исключения, сгенерированные в статических конструкторах 209
Кто должен обрабатывать исключения? 210
Избегайте применения исключений для управления потоком выполнения 210
Обеспечение нейтральности к исключениям 211
Базовая структура нейтрального к исключениям кода 212
Ограниченные области выполнения 217
Критичные финализаторы и `SafeHandle` 220
Создание пользовательских классов исключений 224
Работа с выделенными ресурсами и исключениями 226
Обеспечение поведения отката 230
Резюме 233

Глава 8. Работа со строками 235

Обзор `String` 235
Строковые литералы 236
Спецификаторы формата и глобализация 237
`Object.ToString`, `IFormattable` и `CultureInfo` 238
Создание и регистрация пользовательских типов `CultureInfo` 239
Форматные строки 241
`Console.WriteLine` и `String.Format` 242
Примеры строкового форматирования в пользовательских типах 244
`ICustomFormatter` 245
Сравнение строк 247
Работа со строками из внешних источников 249
`StringBuilder` 251
Поиск строк с помощью регулярных выражений 253
Поиск с помощью регулярных выражений 254
Поиск и группирование 255

Замена текста с помощью Regex 259
Варианты создания Regex 261
Резюме 263

Глава 9. Массивы, типы коллекций и итераторы 264

Представление массивов 264
Неявно типизированные массивы 265
Конвертируемость и ковариантность 268
Возможности сортировки и поиска 268
Синхронизация 269
Векторы против массивов 270
Многомерные прямоугольные массивы 271
Многомерные зубчатые массивы 273
Типы коллекций 275
Сравнение ICollection<T> с ICollection 275
Синхронизация коллекций 277
Списки 278
Словари 279
Наборы 279
System.Collections.ObjectModel 280
Эффективность 283
IEnumerable<T>, IEnumerator<T>, IEnumerable и IEnumerator 284
Типы, производящие коллекции 288
Итераторы 289
Прямые, обратные и двунаправленные итераторы 294
Инициализаторы коллекций 298
Резюме 299

Глава 10. Делегаты, анонимные функции и события 300

Обзор делегатов 300
Создание и использование делегатов 301
Одиночный делегат 302
Цепочки делегатов 303
Итерация по цепочкам делегатов 305
Несвязанные делегаты (открытые экземпляры) 306
События 309
Анонимные методы 314
Остерегайтесь сюрпризов захваченных переменных 319
Анонимные методы как привязки параметров делегатов 321
Шаблон Strategy 325
Резюме 327

Глава 11. Обобщения 328

Разница между обобщениями и шаблонами C++ 329
Эффективность и безопасность типов обобщений 330
Определения обобщенных конструируемых типов 332

Обобщенные классы и структуры 333
Обобщенные интерфейсы 336
Обобщенные методы 336
Обобщенные делегаты 338
Преобразование обобщенного типа 342
Выражение значения по умолчанию 343
Типы, допускающие значения null 344
Контроль доступа к конструируемым типам 346
Обобщения и наследование 347
Ограничения 348
Ограничения на неклассовых типах 353
Обобщенные системные коллекции 354
Обобщенные системные интерфейсы 356
Проблемы выбора и их решение 357
Преобразования и операции внутри обобщенных типов 357
Динамическое создание конструируемых типов 366
Резюме 368

Глава 12. Многопоточность в C# 369

Многопоточность в C# и .NET 369
Запуск потоков 370
Шаблон IOU и асинхронные вызовы методов 373
Состояния потока 373
Завершение потоков 376
Останавливающиеся и пробуждающиеся потоки 378
Ожидание завершения потока 379
Потоки переднего плана и фоновые потоки 380
Локальное хранилище потока 381
Как неуправляемые потоки и апартаменты COM приспособлены друг к другу 384
Синхронизация между потоками 386
Легковесная синхронизация с помощью класса Interlocked 387
Класс Monitor 393
Блокирующие объекты 401
Семафоры 407
События 408
Объекты синхронизации Win32 и WaitHandle 409
Использование ThreadPool 412
Асинхронные вызовы методов 413
Таймеры 421
Резюме 422

Глава 13. В поисках канонических форм C# 423

Канонические формы ссылочных типов 424
Классы должны помечаться как sealed по умолчанию 424
Использование шаблона NVI 425

Является ли Object клонируемым? 428
Является ли Object одноразовым? 434
Нужен ли финализатор Object? 437
Если переопределили Equals, переопределите и GetHashCode 451
Поддерживает ли объект упорядочивание? 454
Является ли Object форматируемым? 457
Является ли Object преобразуемым? 460
Всегда отдавайте предпочтение безопасности типов 463
Использование неизменных ссылочных типов 467
Канонические формы типов значений 470
Переопределение Equals для повышения производительности 471
Поддерживают ли значения этого типа какие-либо интерфейсы? 475
Реализация безопасных к типам форм членов интерфейса и унаследованных методов 476
Резюме 479
Список вопросов для ссылочных типов 479
Список вопросов для типов значений 481

Глава 14. Расширяющие методы 482

Введение в расширяющие методы 482
Как компилятор находит расширяющие методы? 483
За кулисами 486
Читабельность или понятность 487
Рекомендации по использованию 488
Использование расширяющих методов вместо наследования 488
Изоляция расширяющих методов в отдельном пространстве имен 490
Изменение контракта типа может разрушить расширяющие методы 491
Трансформации 491
Цепочки операций 496
Пользовательские итераторы 497
Заимствование из функционального программирования 499
Шаблон Visitor 504
Резюме 508

Глава 15. Лямбда-выражения 509

Введение в лямбда-выражения 509
Лямбда-выражения 510
Лямбда-операторы 515
Деревья выражений 515
Операции над выражениями 518
Функции как данные 519
Полезные применения лямбда-выражений 520
Вернемся к итераторам и генераторам 520
Замыкания (захват переменной) и мемоизация 523
Приправа 528
Анонимная рекурсия 530
Резюме 531

Глава 16. LINQ: язык интегрированных запросов 533
Мост к данным 534
Выражения запросов 534
Вернемся к расширяющим методам и лямбда-выражениям 536
Стандартные операции запросов 537
Ключевые слова запросов C# 539
Конструкция `from` и переменные диапазона 539
Конструкция `join` 540
Конструкция `where` и фильтры 543
Конструкция `orderby` 543
Конструкция `select` и проекция 544
Конструкция `let` 546
Конструкция `group` 547
Конструкция `into` и продолжение 550
Добродетель лени 551
Поощрение лени итераторами C# 552
Ниспровержение лени 553
Немедленное выполнение запросов 555
Еще раз о деревьях выражений 555
Приемы функционального программирования 556
Пользовательские стандартные операции запросов и “ленивое вычисление” 556
Замена операторов `foreach` 564
Резюме 565

Приложение. Справочная информация 567

Предметный указатель 570

Предисловие

Программирование доставляет удовольствие. Это — захватывающее путешествие сквозь проблемы сознания, украшенное чудесными пейзажами открытий. На протяжении всего процесса программистское видение проблем и их решений формируется сквозь призму используемого языка и продиктованного им образа мышления. Поэтому для программиста чрезвычайно важно хорошее знание этого языка.

Проблемы, над которыми трудятся разработчики, настолько разнообразны, насколько разнообразны и сами разработчики. Они простираются от медицинских приложений до программного обеспечения страхового дела, от мультимедийных приложений до инструментов добычи данных. Эволюция C# практически повторяет эволюцию проблем, с которыми сталкиваются разработчики. По мере усложнения проблем происходило упрощение языка и повышение его мощности, позволяющее справиться с возрастающей сложностью.

Язык C# начинался как способ описания многократно используемых компонентов, которые работают в широком разнообразии исполняющих сред. На этой фазе он зарекомендовал себя в роли великолепного языка для описания архитектуры компонентов и систем, при этом сохраняя в себе и расширяя свои корни C-подобного языка. Одним из главных приобретений C# стала полностью объектно-ориентированная система типов, унифицирующая концепции примитивных и сложных типов, а также сборщик мусора.

В то время как первая версия C# была главным достижением, вторая версия стала определяющим моментом в формировании языка. Система типов стала намного богаче с появлением обобщений (generics). Язык также начал поддерживать такие средства, как итераторы и анонимные методы, открывшие возможность более простого и элегантного дизайна. Эти средства позволили разрабатывать более гибкие и мощные каркасы.

Третья версия C# действительно открыла новую страницу в развитии языка. Она размыла линию между кодом и данными. Она представила декларативный синтаксис запросов. Она снабдила программистов новыми функциональными средствами. Все эти новшества позволили программистам легче справляться с данными, находящимися в памяти, в базе данных или поступающими от Web-сервера. Программисты убедятся, что все эти средства вернули им радость от программирования.

В настоящей книге Нэш представил свежее, ясное толкование языка C#. Он не только глубоко понимает C#, но также умеет провести читателя сквозь весь процесс изучения для совершенствования знаний языка. Он умеет сделать путь познания увлекательным, приводя стимулирующие мышление примеры, находя время убедительно мотивировать полезность каждого средства, демонстрирует распространенные приемы и передовой опыт. Я уверен, что, изучив эту книгу, читатель станет писать лучший код.

Вес Дайер (Wes Dyer)
Команда проектировщиков языка и компилятора C#
Microsoft Corporation

Об авторе

Трей Нэш (Trey Nash) в настоящее время разрабатывает программное обеспечение в одной из лидирующих на рынке программных компаний, занимающихся вопросами безопасности. До этого он в течение пяти лет работал на Macromedia Inc. В Macromedia он несколько лет трудился в команде межпродуктной разработки, создавая решения для широкого диапазона продуктов компании, включая Flash и Fireworks. Он специализировался на технологии COM/DCOM с использованием C/C++/ATL, до тех пор, пока не грянула революция .NET. Он увлекся компьютерами с того момента, как получил в свое распоряжение свой первый TI-99/4A, когда ему было всего 13 лет. Он поразил своих родителей, превратив детское увлечение в пристойную высокооплачиваемую карьеру, несмотря на все их опасения. Трей получил степень бакалавра в области электроники в Техасском Университете A&M. Когда он не сидит за компьютером, вы обнаружите его работающим в гараже, упражняющимся в карточных фокусах (странно, но это правда), играющим на пианино, изучающим иностранные языки (в настоящее время — русский и исландский) либо играющим в хоккей.

О техническом редакторе

Шон Вилдермат (Shawn Wildermuth) — сертифицированный специалист Microsoft MVP (C#), MCSD.NET, MCT, являющийся основателем Wildermuth Consulting Services, LLC — компании, занимающейся разработкой архитектур программных систем, обучением и поставкой программных решений в Атланте, шт. Джорджия. Также он является лектором INETA Speaker's Bureau и выступал на нескольких национальных конференциях, посвященных разным темам. В настоящее время занимается преподаванием Silverlight в рамках программы Silverlight Tour (`http://www.silverlight-tour.com`).

Шон также является автором нескольких книг, включая *Pragmatic ADO.NET* издательства Addison-Wesley, соавтором четырех программ обучения и сертификации Microsoft для Microsoft Press, а также соавтором готовящейся к выходу в свет книги *Prescriptive Data Architectures*.

За несколько лет Шон написал ряд статей для различных журналов и Web-сайтов, включая *MSDN*, *MSDN Online*, *DevSource*, *InformIT*, *Windows IT Pro*, *The ServerSide*, *ONDotNet* и для серии *Rich Client* от Intel. Уже более 20 лет Шон получает удовольствие от разработки управляемого данными программного обеспечения. Вы можете обратиться к нему через его Web-сайт по адресу:

```
http://www.wildermuthconsulting.com
```

Благодарности

Написание этой книги было длительным и трудным процессом, на протяжении которого я постоянно получал поддержку от моей семьи и друзей, за которую искренне благодарен. Не будь этой поддержки близких мне людей, процесс был бы намного более трудным и существенно менее плодотворным.

Я хотел бы специально отметить следующих людей за их вклад в работу над первым изданием. Хочу поблагодарить (не в каком-то определенном порядке) Дэвида Веллера (David Weller), Стивена Тоуба (Stephen Toub), Рекса Джаеске (Rex Jaeschke), Владимира Левина (Vladimir Levin), Джерри Мареска (Jerry Maresca), Криса Пелса (Chris Pels), Кристофера Т. Мак-Набба (Christopher T. McNabb), Брэда Вилсона (Brad Wilson), Питера Парча (Peter Partch), Поля Стаббса (Paul Stubbs), Руфуса Литтлфилда (Rufus Littlefield), Томаса Рестрепо (Tomas Restrepo), Джона Ламберта (John Lambert), Джоан Мюррей (Joan Murray), Шерри Кейн (Sheri Cain), Джессику Д'Амико (Jessica D'Amico), Карен Геттман (Karen Gettman), Джима Хаддлстона (Jim Huddleston), Ричарда Дэла Порто (Richard Dal Porto), Гарри Корнелла (Gary Cornell), Брэда Абрамса (Brad Abrams), Элли Фонтейн (Ellie Fountain), Николь Абрамович (Nicole Abramowitz), всю команду издательства Apress, и, наконец, Шелли Нэш.

Во время работы над вторым изданием я хотел бы отметить следующих людей за их помощь и поддержку (опять же, без определенного порядка): Шона Вилдермата (Shawn Wildermuth), Софию Марчант (Sofia Marchant), Джима Комптона (Jim Compton), Доминик Шейкшафт (Dominic Shakeshaft), Веса Дайера (Wes Dyer), Келли Винквист (Kelly Winquist), и Лору Чеу (Laura Cheu).

Если я кого-то забыл — то только по ошибке, а не умышленно. Без вашей поддержки эта работа была бы невозможной. Спасибо вам всем!

Введение

Visual C# .NET (C#) изучить относительно легко любому, кто знаком с другим объектно-ориентированным языком. Даже человек, знакомый с Visual Basic 6.0, которому нужен объектно-ориентированный язык, обнаружит, что понять C# нетрудно. Однако, несмотря на то, что C# вместе с .NET Framework предоставляют легкий путь создания простых приложений, все же вам понадобится немало знаний и понимания того, как их правильно применять для создания сложных, надежных и устойчивых к сбоям приложений C#. В этой книге я научу вас всему, что необходимо знать, и расскажу, как наилучшим образом применять знания для того, чтобы быстро достичь уровня настоящего специалиста в C#.

Идиомы и шаблоны проектирования незаменимы для повышения уровня мастерства разработчика и применения его на практике, так что я покажу вам, как использовать многие из них для создания приложений — эффективных, надежных, устойчивых к сбоям и безопасных в отношении исключений. Хотя многие из этих шаблонов знакомы программистам на C++ и Java, все же некоторые уникальны для .NET и его общеязыковой исполняющей среды (Common Language Runtime — CLR). В последующих главах будет показано, как применять эти обязательные идиомы и приемы проектирования для гладкой интеграции ваших приложений C# с исполняющей системой .NET, причем основное внимание сосредоточивается на новых средствах C# 3.0.

Шаблоны проектирования документируют передовой опыт в проектировании приложений, который накоплен множеством программистов в течение длительного времени. Фактически .NET Framework сам по себе реализует многие хорошо известные шаблоны проектирования. К тому же последние три версии .NET Framework и две версии C# высветили многие новые идиомы и передовые приемы. Вы будете знакомиться с ними на протяжении всей книги. К тому же важно отметить, что этот бесценный набор технологий постоянно эволюционирует.

С появлением C# 3.0 вы получили возможность легко включать приемы функционального программирования, используя лямбда-выражения, расширяющие методы и язык интегрированных запросов (Language Integrated Query — LINQ). Лямбда-выражения облегчают объявление и создание экземпляров делегатов функций в одном месте. Вдобавок к лямбда-выражениям появилась возможность создавать функционалы — функции, принимающие в качестве аргументов функции и обычно возвращающие другие функции. Несмотря на то что вы могли реализовать технику программирования функционалов на C# (хотя и с некоторыми трудностями), новые средства языка C# 3.0 обеспечили среду, в которой программирование функционалов может органично переплетаться с обычным стилем императивного программирования C#. LINQ позволяет выразить операции запроса данных (которые обычно по природе своей являются функционалами), используя естественный для языка синтаксис. Однажды увидев, как работает LINQ, вы поймете, что можете

сделать много больше, чем простой запрос данных; вы можете использовать его для реализации сложных функциональным программ.

.NET и CLR предоставляют уникальную и стабильную межплатформенную исполняющую среду. C# — только один из языков, ориентированных на эту мощную исполняющую систему. Вы обнаружите, что многие из приемов, описанных в этой книге, также применимы к любому языку, ориентированному на исполняющую систему .NET. Тем из вас, у кого есть серьезный опыт работы на C++, и кто знаком с такими концепциями, как канонические формы C++, безопасность исключений, RAII (Resource Acquisition Is Initialization — захват ресурсов является инициализацией), а также корректность констант, книга объяснит, как применить все эти концепции в C#. Если вы — программист на Java или Visual Basic, который потратил годы на разработку собственного набора приемов, и хотите знать, как эффективно применить их в C#, вы найдете здесь ответ и на этот вопрос.

Как вы вскоре убедитесь, для того, чтобы стать экспертом в C#, не нужно тратить годы на приобретение опыта методом проб и ошибок. Вам просто нужно изучить правильные вещи и правильные способы того, как их следует делать. Именно потому я написал эту книгу для вас.

Как организована эта книга

Я предполагаю, что вы уже имеете практический опыт работы с некоторым объектно-ориентированным языком, таким как C++, Java или Visual Basic .NET. Поскольку C# унаследовал свой синтаксис от C++ и Java, я не стану тратить много времени на описание синтаксиса C#, за исключением тех моментов, в которых он отличается от C++ или Java. Если вы уже немного знаете C#, то можете лишь пролистать или вообще пропустить главы с 1 по 3.

Глава 1, "Обзор C#", даст вам первоначальное представление о том, как выглядит простое приложение C#, и предложит описание некоторых базовых отличий среды программирования C# от среды "родного" C++.

Глава 2, "C# и CLR", разовьет тему главы 1 и кратко ознакомит с управляемой средой, внутри которой выполняется приложение C#. Я расскажу о сборках — базовых строительных блоках приложений, в которые компилируются файлы кода C#. Вдобавок вы увидите, как метаданные делают сборки самодостаточными.

Глава 3, "Обзор синтаксиса C#", предоставит описание синтаксиса языка C#. Я продемонстрирую две фундаментальных группы типов CLR: типы значений и ссылочные типы. Также я опишу пространства имен и то, как вы можете использовать их для логического разбиения типов и функциональности внутри ваших приложений.

Главы с 4 по 13 содержат углубленные описания применения полезных идиом, шаблонов проектирования и передовых приемов в ваших программах и проектах C#. Я попытался выстроить эти главы в логическом порядке, но неизбежно одни главы могут ссылаться на приемы или темы, описанные в других (последующих) главах.

Глава 4, "Классы, структуры и объекты", содержит подробности определения типов в C#. Вы узнаете больше о типах значений и ссылочных типах в CLR. Я также коснусь "родной" поддержки интерфейсов внутри CLR и C#. Вы увидите, как работает наследование в C#, а также каким образом каждый объект наследуется

от типа System.Object. Эта глава также содержит богатую информацию об управляемой среде и о том, что вам нужно знать, чтобы определять типы, удобные для нее. Я представлю многие из таких тем в этой главе и продолжу дискуссию более подробно в последующих главах.

Глава 5, "Интерфейсы и контракты", посвящена интерфейсам и роли, которую они играют в языке C#. Интерфейсы представляют собой контракт функциональности, которую могут реализовывать типы. Вы узнаете о многих способах реализации интерфейсов типами, а также о том, как исполняющая система выбирает, какие методы нужно вызывать при вызове методов интерфейса.

Глава 6, "Перегрузка операций", детализирует способы создания специальной функциональности встроенных операций языка C#, когда они применяются к вашим собственным типам. Вы увидите, как перегрузить действие операций, хотя не все управляемые языки, которые компилируются в код для CLR, способны использовать перегруженные операции.

Глава 7, "Исключения: безопасность и обработка", посвящена средствам обработки исключений языка C# и CLR. Хотя синтаксис подобен C++, создание безопасного и нейтрального в отношении исключений кода не так просто — даже сложнее создания безопасного к исключениям кода на "родном" C++. Вы увидите, что написание устойчивого к сбоям, безопасного к исключениям кода вообще не требует применения конструкций try, catch или finally. Я также опишу некоторые новые возможности, добавленные в исполняющую систему .NET 2.0, которые позволят вам создавать более устойчивый к сбоям код, чем это было возможно в .NET 1.1.

Глава 8, "Работа со строками", описывает строки — первоклассный тип CLR — и методы их эффективного применения в C#. Значительная часть этой главы посвящена средствам строкового форматирования различных типов в .NET Framework и тому, как заставить определенные вами типы вести себя подобным образом — посредством реализации Iformattable. Дополнительно я представлю средства глобализации каркаса и расскажу, как создавать собственные CultureInfo для культур и регионов, о которых .NET Framework не имеет понятия.

Глава 9, "Массивы, типы коллекций и итераторы", расскажет о разнообразных массивах и типах коллекций, доступных в C#. Вы можете создавать два типа многомерных массивов наряду с собственными типами коллекций, используя служебные классы коллекций. Вы увидите, как определять прямые, обратные и двунаправленные итераторы, применяя новый синтаксис итераторов, представленный в C# 2.0, так что ваши типы коллекций смогут хорошо работать с операторами foreach.

Глава 10, "Делегаты, анонимные функции и события", продемонстрирует механизмы, используемые внутри C# для обеспечения обратных вызовов. C# пошел на один шаг дальше, и инкапсулирует обратные вызовы в вызываемые объекты, называемые *делегатами*. Вдобавок C# 2.0 позволяет создавать делегаты с сокращенным синтаксисом, называемые *анонимными функциями*. Анонимные функции подобны лямбда-функциям в функциональном программировании. К тому же вы увидите, как на основе делегатов каркас реализует механизм уведомления публикации/подписки на события, позволяя отделять источник события от его потребителя.

Глава 11, "Обобщения", представит, пожалуй, наиболее впечатляющее средство, появившееся в C# 2.0 и CLR. Тем, кто знаком с шаблонами C++, обобщения

покажутся знакомыми, хотя между ними есть многие фундаментальные отличия. Используя обобщения, вы можете создавать оболочку функциональности, внутри которой во время выполнения определяются более специфичные типы. Обобщения наиболее полезны для типов коллекций и обеспечивают значительную эффективность по сравнению с коллекциями из предыдущих версий .NET.

Глава 12, "Многопоточность в C#", описывает то, что необходимо сделать при создании многопоточных приложений в управляемой виртуальной исполняющей системе C#. Если вы знакомы с потоками в родной среде Win32, вы отметите существенные отличия. Более того, управляемая среда предоставляет более развитую инфраструктуру для облегчения работы. Вы увидите, как делегаты с использованием шаблона IOU ("I Owe You") предоставляют отличные ворота в пул потоков обработки. Вероятно, синхронизация — наиболее важная концепция, когда нужно заставить несколько потоков работать параллельно. В этой главе описаны различные средства синхронизации, доступные вашим приложениям.

Глава 13, "В поисках канонических форм C#", — это своего рода диссертация о передовых приемах проектирования при определении новых типов, и о том, как сделать их такими, чтобы их применение было естественным, и чтобы их потребители не могли использовать их неправильно. Я буду касаться этой темы и в других главах, но здесь она обсуждается в деталях. Эта глава снабжена подробным перечнем того, что следует принимать во внимание при определении новых типов на C#.

Глава 14, "Расширяющие методы", раскрывает средство, новое для C# 3.0. Поскольку вы можете вызывать расширяющие методы как обычные методы экземпляра с типом, который они расширяют, их можно воспринимать как развитие контракта типов. Но на самом деле они представляют собой нечто большее. В этой главе я покажу, как расширяющие методы могут открыть мир функционального программирования на C#.

Глава 15, "Лямбда-выражения", посвящена еще одному новому средству C# 3.0. Вы можете объявлять и создавать экземпляры делегатов, используя лямбда-выражения с применением краткого и визуально выразительного синтаксиса. Хотя той же цели могут служить и анонимные функции, они намного более многословны и менее синтаксически элегантны. Однако в C# 3.0 вы можете конвертировать лямбда-выражения в деревья выражений. То есть язык имеет встроенную способность преобразовывать код в структуры данных. Само по себе это средство полезно, но не настолько, как в сочетании в языком интегрированных запросов (Language Integrated Query — LINQ). В сочетании с расширяющими методами лямбда-выражения действительно завершают полный цикл функционального программирования на C#.

Глава 16, "LINQ: язык интегрированных запросов", — это кульминация всех новых средств C# 3.0. Используя выражения LINQ через новые LINQ-ориентированные ключевые слова C# 3.0, вы можете плавно интегрировать запросы к данным в код. LINQ формирует мост между обычным миром императивного программирования C# и миром функционального программирования запросов к данным. Выражения LINQ могут быть использованы для манипуляций нормальными объектами, равно как и данными, полученными из баз данных SQL, наборов данных и XML — и это далеко не полный перечень.

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

ГЛАВА 1

Обзор C#

Поскольку настоящая книга предназначена для опытных разработчиков, использующих объектно-ориентированный подход, я исхожу из того, что вы уже имеете некоторое представление об исполняющей системе .NET. Есть замечательная книга, специально посвященная исполняющей системе .NET — *Essential .NET Volume 1: The Common Language Runtime by Don Box* (Boston, MA: Addison-Wesley, 2002 г.). Кроме того, очень важно обратить внимание на некоторые сходства и различия между C# и C++, прежде чем писать элементарный пример вида "Hello World!". Если у вас уже есть опыт построения приложений .NET, вы можете спокойно пропустить настоящую главу. Однако прочесть раздел "Обзор новшеств C# 3.0" все же стоит.

Отличия между C# и C++

C# — строго типизированный объектно-ориентированный язык, чей код внешне похож на C++ (и Java). Это решение проектировщиков языка C# позволяет разработчикам на C++ легко воспользоваться своими знаниями, чтобы быстро освоить C#. Синтаксис C# в некоторых отношениях отличается от C++, но большинство отличий между этими языками носят семантический и поведенческий характер, что обусловлено отличиями исполняющих сред, в которых работают программы, написанные на этих двух языках.

Язык C#

Исходный код C# компилируется в так называемый *управляемый код* (managed code). Как вам, возможно, уже известно, управляемый код представляет собой код на промежуточном языке IL (Intermediate Language), который находится посредине между высокоуровневым языком (C#) и языком более низкого уровня (ассемблером/машинным кодом). Во время выполнения среда CLR (Common Language Runtime — общеязыковая исполняющая среда) на лету компилирует код IL в машинный код, применяя для этого оперативную компиляцию (Just In Time — JIT). Как и любой аспект проектирования, эта техника имеет свои достоинства и недостатки. Очевидным недостатком может показаться неэффективность компиляции кода во время выполнения. Этот процесс отличается от интерпретации, которая обычно применяется в языках сценариев вроде Perl и JScript. Компилятор JIT не компилирует функцию или метод при каждом его вызове; он делает это только первый раз, при этом продуцируя машинный код, родной по отношению к платформе, на которой он выполняется. Очевидное достоинство компиляции JIT — сокраще-

ние рабочего множества приложения, поскольку "отпечаток памяти" промежуточного кода существенно меньше. Во время выполнения приложения компилируется лишь тот код, который необходим. Например, если ваше приложение содержит код вывода на печать, который никогда не требуется, если пользователь не печатает документ, то компилятор JIT никогда его и не компилирует. Более того, CLR может оптимизировать выполнение программы на лету. Например, CLR может определить способ сокращения "непопаданий" на страницу памяти в диспетчере памяти, реорганизуя скомпилированный код в памяти, и все это он делает во время выполнения. Если учесть вместе все недостатки, то вы обнаружите, что для большинства приложений достоинства такого подхода перевешивают.

На заметку! В действительности вы можете выбирать, кодировать ли программы в "сырой" IL, когда строите их с помощью IL Assembler (ILASM). Однако, скорее всего, это будет неэффективное использование вашего времени. Высокоуровневые языки почти всегда могут предоставить любые возможности, которые доступны вам с "сырым" IL-кодом.

Язык C++

В отличие от C#, код C++ традиционно компилируется в *родной (native) код*. Родной код — это машинный код, используемый процессором, для которого скомпилирована программа. Для простоты предположим, что мы говорим о скомпилированном в машинный код C++, а не об управляемом C++, который обеспечивает C++/CLI. Если вы хотите, чтобы ваше родное приложение C++ работало на разных платформах — 32- и 64-разрядных — вы должны компилировать их по отдельности для каждой из них. Родной двоичный код, полученный на выходе компилятора, обычно не совместим между платформами.

С другой стороны, IL совместим между платформами, поскольку он, наряду с общей инфраструктурой языка (Common Language Infrastructure — CLI), на которой построена CLR, определен международным стандартом[1]. Этот стандарт быстро завоевывает признание и уже реализован вне платформы Microsoft Windows.

На заметку! Рекомендуется посмотреть, как дела у команды Mono[2], занимающейся реализацией альтернативной исполняющей системы для других платформ с открытым кодом — Virtual Execution Systems (VES).

В стандарт CLI входит также файловый формат PE (Portable Executable — переносимый исполняемый модуль) для управляемых модулей. Таким образом, вы действительно можете компилировать программу C# на платформе Windows, а выполнять и на Windows, и на Linux, причем без перекомпиляции, поскольку даже формат файлов стандартизован[3].

[1] Документ по стандарту CLI — Ecma-335 — доступен по адресу `www.ecma-international.org`. Там же можно найти и Ecma-334 — документ по стандарту языка C#.

[2] Информацию о проекте Mono вы найдете на сайте `www.mono-project.com`.

[3] Конечно, на целевой платформе также должны быть установлены все зависимые библиотеки. Это быстро становится реальностью за счет распространения библиотеки .NET Standard Library. Например, загляните на `www.go-mono.com/docs/`, чтобы увидеть, насколько велико покрытие библиотек в проекте Mono.

Степень переносимости чрезвычайно удобна и всегда занимала умы и сердца проектировщиков COM/DCOM в прежние дни, но по разным причинам не была достигнута в нужной мере между разными платформами[4]. Одна из главных причин неудачи состоит в том, что COM не достает выразительного и расширяемого механизма описания типов и их зависимостей. Спецификация CLI изящно решает эту проблему, ввода метаданные, которые рассматриваются в главе 2.

Сборка мусора CLR

Одним из ключевых средств CLR является *сборщик мусора* (Garbage Collector — GC). GC избавляет вас от забот об управлении выделением и освобождением памяти, что является причиной многих ошибок в программном обеспечении. Однако GC не избавляет вас от управления ресурсами, в чем вы убедитесь в главе 4. Например, дескриптор файла — это ресурс, который должен быть каким-то образом освобожден. GC имеет дело напрямую лишь с ресурсами памяти. Чтобы обработать ресурсы, не связанные с памятью, такие как подключения к базе данных и дескрипторы файлов, вы можете использовать финализатор (он рассматривается в главе 13), чтобы освобождать ресурсы тогда, когда GC известит об уничтожении объекта. Однако еще лучше для этой задачи использовать шаблон Disposable, который будет продемонстрирован в главах 4 и 13.

На заметку! CLR ссылается на все объекты *ссылочных типов* опосредовано, подобно тому, как вы работаете с указателями и ссылками в C++, но без применения синтаксиса указателей. Когда вы объявляете переменную ссылочного типа в C#, то тем самым в действительности резервируете место под хранение для ассоциированного с ней типа, либо в куче, либо в стеке, и эта переменная хранит ссылку на объект. Поэтому когда вы копируете ссылку на объект из одной переменной в другую, то в результате получаете две переменных, ссылающихся на один и тот же объект. Все экземпляры ссылочных типов располагаются в управляемой куче. CLR управляет местоположением этих объектов, и когда возникает необходимость переместить их в памяти, она обновляет все ссылки на перемещаемые объекты, чтобы они указывали на новое местоположение. Также в CLR существуют *типы значений*, и их экземпляры находятся в стеке либо существуют в виде полей объектов, находящихся в куче. Их применение связано со многими ограничениями и нюансами. Обычно вы используете их, когда необходима облегченная структура для управления некоторыми связанными данными. Типы значений также удобны для моделирования неизменяемых фрагментов данных. Эта тема более подробно раскрывается в главе 4.

C# позволяет быстро разрабатывать приложения, имея дело с меньшим количеством рутинных деталей, чем это возможно в среде C++. В то же время C# представляет собой язык, который выглядит знакомым для разработчиков на C++ и Java.

Пример программы на C#

Давайте внимательно рассмотрим программу на C#. Обратимся к традиционному примеру "Hello World!", который все знают и любят. Ее консольная версия на C# выглядит так, как показано ниже.

[4] Если вас интересуют подробности, рекомендую прочесть книгу Дона Бокса (Don Box) и Криса Селлса (Chris Sells) *Essential .NET, Volume I: The Common Language Runtime* (Boston, MA: Addison-Wesley Professional, 2002 г.). (Заголовок дает основания ожидать, что рано или поздно появится и том 2.)

```
class EntryPoint {
    static void Main() {
        System.Console.WriteLine( "Hello World!" );
    }
}
```

Обратите внимание на структуру этой программы на C#. В ней объявляется тип (класс по имени EntryPoint) и член этого типа (метод по имени Main). Это отличает ее от C++, где тип декларируется в заголовке, а определяется в отдельной единице компиляции — обычно в файле .cpp. К тому же метаданные (описывающие все типы в модуле и прозрачно генерируемые компилятором C#) избавляют от необходимости опережающего объявления и включения заголовков, как того требует C++. Фактически опережающее объявление в C# вообще отсутствует.

Программисты на C++ найдут статический метод Main знакомым, но за исключением того факта, что его имя начинается с заглавной буквы. Каждая программа нуждается в точке входа, и в случае C# такой точкой служит статический метод Main. Есть и другие отличия. Например, метод Main объявлен внутри класса (в данном случае — EntryPoint). В C# вы обязаны объявлять все методы внутри определения типа. Здесь не существует статических свободных функций, как в C++. Типом возврата метода Main может быть либо int, либо void, в зависимости от ваших потребностей. В моем примере метод Main не имеет параметров, но если вам нужен доступ к параметрам командной строки, то ваш метод Main может объявлять параметр (массив строк) для обращения к ним.

На заметку! Если ваше приложение содержит несколько типов со статическим методом Main, вы можете выбирать, какой нужно запустить, с помощью ключа компилятора /main.

Может показаться, что вызов WriteLine выглядит многословным. Я должен был квалифицировать имя метода именем класса Console, а также указать имя пространства имен, в котором находится класс Console (в данном случае — System). .NET (и, следовательно, C#) поддерживает пространства имен во избежание конфликтов имен в огромном глобальном пространстве. Однако вместо того, чтобы всегда требовать указания полностью квалифицированного имени, включая пространство имен, C# предлагает директиву using, которая является аналогом import из языка Java и using namespace — из C++. Поэтому вы можете слегка изменить предыдущий пример программы, переписав его так, как показано в листинге 1.1.

Листинг 1.1. hello_world.cs

```
using System;
class EntryPoint {
    static void Main() {
        Console.WriteLine( "Hello World!" );
    }
}
```

При наличии директивы using System; вы можете пропустить имя пространства System при вызове Console.WriteLine.

Чтобы скомпилировать этот пример, выполните следующую команду в командной строке Windows:

```
csc.exe /r:mscorlib.dll /target:exe hello_world.cs
```

Давайте разберемся, что именно делает эта команда.

- `csc.exe` — это компилятор Microsoft C#.
- Опция `/r` указывает зависимости сборок данной программы. Сборки подобный концепции DLL-библиотек в мире "родного" кода. `mscorlib.dll` — это место, где определен объект `System.Console`. В действительности ссылаться на сборку `mscorlib` не нужно, поскольку компилятор сошлется на нее автоматически, если только не указать опцию `/nostdlib`.
- Опция `/target:exe` сообщает компилятору, что вы собираете консольное приложение, и если эта опция не указана, то по умолчанию так оно и есть. Другие варианты, которые можно здесь задать: `/target:winexe` — для построения Windows-приложения с графическим интерфейсом пользователя, `/target:library` — для генерации сборки DLL с расширением `.dll` и `/target:module` — для получения DLL с расширением `.netmodule`. Последняя опция генерирует модули, не содержащие манифеста сборки, так что вы должны включить его в сборку позднее, применив компоновщик сборок `al.exe`. Это позволяет создавать многофайловые сборки.
- `hello_world.cs` — компилируемая программа C#. Если в проекте присутствует множество файлов C#, вы можете просто перечислить их в конце командной строки.

После выполнения приведенной выше команды вы получите файл `hello_world.exe`, который можно запустить из командной строки и увидеть ожидаемый результат. Если хотите, можете пересобрать код с опцией `/debug`. Тогда можно будет осуществить пошаговое выполнение внутри отладчика. Чтобы продемонстрировать независимость от платформы C#, в случае наличия у вас работающей ОС Linux с инсталлированным на ней пакетом Mono VES, вы можете скопировать модуль `hello_world.exe` непосредственно в его двоичном виде и запустить, получив тот же результат, если в Linux все настроено правильно.

Обзор средств, добавленных в C# 2.0

С появления начального выпуска в конце 2000 г. язык C# заметно эволюционировал. Эта эволюция стала возможной благодаря его широкому распространению. С появлением версии Visual Studio 2005 и .NET Framework 2.0, компилятор C# стал поддерживать расширения языка C# 2.0. Это стало отличной новостью, поскольку в C# 2.0 были включены некоторые удобные средства, обеспечивающие более естественный стиль программирования с большей эффективностью. В настоящем разделе приводится обзор этих новых средств, а также даются ссылки на последующие главы, содержащие более детальную информацию.

Возможно, наиболее важным дополнением C# 2.0 стала поддержка обобщений. Их синтаксис подобен шаблонам C++, но основное отличие состоит в том, что конструируемые типы, созданные на основе обобщений .NET, являются динамичными по своей природе, т.е. они привязываются и конструируются во время выполнения.

Это отличает их от конкретных типов C++, создаваемых из шаблонов, которые являются статичными — в том смысле, что связываются и создаются во время компиляции[5]. Обобщения наиболее удобны, когда они применяются с контейнерными типами вроде векторов, списков и хеш-таблиц, где они обеспечивают великолепную эффективность. Обобщения могут трактовать типы, которые они содержат, как специфические типы, а не как базовый тип всех объектов — `System.Object`. Тема обобщений будет раскрыта в главе 11, а о коллекциях мы поговорим в главе 9.

В C# 2.0 была добавлена поддержка анонимных методов. Анонимный метод — это то, что иногда называют *лямбда-функция*, что пришло из дисциплин функционального программирования. Анонимные методы C# исключительно полезны для работы с делегатами и событиями. Делегаты и события — это конструкции, используемые для регистрации методов обратного вызова, которые вызываются при их инициации. Обычно вы привязываете их к некоторому где-то определенному методу. Но с анонимными методами вы определяете код делегатов или событий методом встраивания — прямо в той точке, где делегат или событие устанавливается. Это удобно, если ваш делегат просто должен выполнить некоторый небольшой кусочек работы, для которого определение отдельного метода было бы избыточно. Что еще лучше, так это то, что тело анонимного метода имеет доступ ко всем переменным, находящимся в контексте той точки, где этот метод определен[6]. Я расскажу об анонимных методах в главе 10. Лямбда-выражения — новшество в C# 3.0 — призваны прийти на замену анонимным методам, и позволяют писать более читабельный код.

В C# 2.0 добавилась поддержка итераторов. Любой, кто знаком со стандартной библиотекой шаблонов C++ (Standard Template Library — STL), знаком с итераторами и их полезностью. В C# вы обычно используете оператор `foreach` для выполнения прохода (итерации) по объекту, ведущему себя как коллекция. Этот объект коллекции должен реализовать интерфейс `IEnumerable`, включающий метод `GetEnumerator`. Реализация метода `GetEnumerator` для контейнерных типов обычно очень утомительна. Однако при использовании итераторов C# реализация этого метода становится элементарной. Более подробную информацию об итераторах вы узнаете из главы 9.

И, наконец, в C# 2.0 была добавлена поддержка *частичных типов* (partial types). До появления C# 2.0 вы должны были определять каждый класс C# целиком в одном файле (также называемом *единицей компиляции*). Это требование было снято для того, чтобы можно было создавать "скелетный" код. Например, вы можете использовать мастера (wizards) Visual Studio для генерации таких полезных вещей, как типы-наследники `System.Data.DataSet`, для обращения к информации в базе данных. До появления C# 2.0 вносить изменения в сгенерированный код было проблематично. Вы должны были либо наследоваться от него, либо редактировать автоматически сгенерированный код. Редактирование сгенерированного кода — рискованное предприятие, поскольку обычно вы теряете такие изменения, когда по той или иной причине мастер выполняет перегенерацию этого кода. Частичные типы решили эту проблему, поскольку теперь вы можете поместить сгенерированный код в отдельный файл, так что ваши изменения не будут потеряны при его

[5] Работая с C++/CLI, стандартизованным в Ecma-372 и впервые доступным в Visual Studio 2005, вы можете совместно использовать обобщения и шаблоны.

[6] Это называется *герметизацией* (closure) или *захватом переменных*.

перегенерации. Чтобы увидеть отличный пример применения частичных типов, загляните в код, сгенерированный, когда вы создаете приложение Windows Forms в Visual Studio. Подробнее о частичных типах рассказывается в главе 4.

Обзор новшеств C# 3.0

Язык C# 3.0 включает несколько великолепных новых средств. Большинство из них предназначено для поддержки языка интегрированных запросов (Language Integrated Query — LINQ). Тем не менее, все они исключительно полезны и при использовании отдельно, вне контекста LINQ. Многие из них позволяют программистам легче применять приемы функционального программирования.

Теперь C# поддерживает неявно типизированные локальные переменные за счет использования ключевого слова `var`. Важно отметить, что эти переменные не являются бестиповыми; скорее, их тип выводится во время компиляции. Подробнее о них читайте в главе 3.

Хотело ли вам когда-нибудь создать простой тип для хранения взаимосвязанных данных, но без утомительной необходимости создания целого нового класса? Во многих случаях здесь придет на помощь новая поддержка анонимных типов, которая избавит вас от этой обязанности. Применяя анонимные типы, вы можете определять и создавать экземпляры типа в одном составном операторе. Об анонимных типах речь пойдет в главе 4.

Автоматически реализуемые свойства — еще одно полезное новое средство для сокращения объема клавиатурного ввода и вероятности внесения ошибок. Сколько раз вам приходилось просто объявлять класс для хранения нескольких фрагментов данных и испытывать досаду от необходимости набирать большой объем кода средств доступа (`set` и `get`) к этим данным? В конце концов, это полезный прием инкапсуляции. К счастью, теперь автоматически реализуемые свойства позволили значительно сократить объем рутинной работы, необходимой для определения свойств в типах. Подробнее об этом читайте в главе 4.

Если говорить об удобстве, то стоит вспомнить еще и о том, что в C# 3.0 появились два новых средства, помогающие создавать и инициализировать экземпляры объектов. Применяя инициализаторы объектов и коллекций, вы можете создавать экземпляры и инициализировать как объект, так и коллекцию в одном составном операторе. Инициализаторы объектов рассматриваются в главе 4, а инициализаторы коллекций — в главе 9.

В C# 2.0 были представлены определения частичных классов для облегчения применения генераторов кода. В C# 3.0 к этому добавились частичные методы. Применяя частичные методы, генератор кода может объявлять сигнатуру метода, а потребитель этого сгенерированного кода — тот, кто создает остальную часть определения частичного класса — может решать, реализовывать его или нет. Подробнее о частичных методах рассказывается в главе 4.

Расширяющие методы — одно из самых впечатляющих новых средств. На первый взгляд, это просто статические методы, которые могут быть вызваны так, будто они являются методами экземпляра. Они не имеют никакого специального доступа к экземпляру, с которым работают, так что в этом отношении они — простые статические методы. Однако их синтаксис позволяет программировать в более функциональной манере, в результате создавая более ясный и читабельный код.

Теме расширяющих методов и тому, что с ними можно делать, посвящена целая глава 14.

Возможно, еще более впечатляющим новшеством, чем расширяющие методы, является поддержка лямбда-выражений. Лямбда-выражения вытесняют поддержку анонимных методов. То есть, если бы лямбда-выражения появились уже в C# 2.0, то вообще не было бы необходимости в анонимных методах. Однако лямбда-выражения дают много больше, чем анонимные методы, поскольку они могут быть преобразованы как в делегаты, так и в деревья выражений. Лямбда-выражения рассматриваются в главе 15.

"Патриархом" всех новых средств C# 3.0 должен быть LINQ, который вобрал в себя все новые средства — особенно расширяющие методы, лямбда-выражения и анонимные типы. Также он добавил некоторые ключевые слова в язык, позволяющие кодировать интуитивно понятные операторы запросов, гладко заполняя брешь между объектно-ориентированным миром и миром данных. Вы можете применять LINQ для доступа к данным из множества источников. Visual Studio предоставляет возможность использования LINQ на родных объектах коллекций, хранилищах данных SQL и XML. Вскоре ожидается поддержка многих других источников данных, как от Microsoft, так и от независимых поставщиков. Например, вы сможете применять LINQ для подключения к WMI (Windows Management Instrumentation — инструментальные средства управления средой Windows), объектной модели документов (Document Object Model — DOM) и Web. Вдобавок в разработке находятся реализации, которые позволят использовать LINQ для осуществления поиска с помощью популярных Web-сайтов, таких как Google и Flickr. LINQ посвящена глава 16.

Резюме

В этой главе я коснулся высокоуровневых характеристик программ, написанных на C#: весь код компилируется в IL вместо "родных" машинных инструкций определенный платформы. Вдобавок CLR реализует сборщик мусора (GC) для управления распределением и освобождением "сырой" памяти, избавляя вас от необходимости беспокоиться о наиболее распространенных ошибках в разработке программного обеспечения — неправильном управлении памятью. Однако, как и в большинстве инженерных компромиссов, здесь есть и другие аспекты (читай: сложности) управления памятью и ресурсами, которые может привнести GC в определенных ситуациях.

На традиционном примере программы "Hello World!" я кратко продемонстрировал удобство пространств имен, наряду с тем фактом, что C# лишен любого рода синтаксиса включения, который есть в C++. Вместо этого все внешние типы включаются в компиляцию через метаданные, представляющие в богатом описательном формате типы, содержащиеся внутри сборки. Таким образом, метаданные и скомпилированные типы всегда находятся в одном пакете.

Обобщения открыли огромную новую область разработки, с которой вам придется иметь дело, изучая и разрабатывая искусные трюки ее применения, в течение нескольких следующих лет. Некоторые из этих трюков могут быть позаимствованы из мира шаблонов C++, но не все, поскольку эти две концепции фундаментально отличаются. Итераторы и анонимные методы предоставляют лаконичный способ выражения распространенных идиом, таких как перечисления и методы обратного

вызова, а поддержка объявления частичных типов внутри C# облегчает работу с кодом, автоматически генерируемым инструментальными средствами.

C# 3.0 предлагает много новых впечатляющих средств, позволяющих очень легко использовать приемы функционального программирования с минимальными накладными расходами. Некоторые из этих новых средств добавляют удобства программированию на C#. LINQ предоставляет гладкий механизм — мост между миром хранилищ данных и объектно-ориентированным миром.

В следующей главе я кратко раскрою некоторые детали, касающиеся процесса компиляции JIT. Дополнительно я опишу внутреннюю организацию сборок и содержащиеся в них метаданные. Сборки — это базовые строительные блоки приложений C#, аналог DLL-библиотек из мира "родных" приложений Windows.

ГЛАВА 2

C# и CLR

Как упоминалось в предыдущей главе, управляемые приложения и "родные" приложения имеют много различий — главным образом потому, что управляемые приложения выполняются внутри Microsoft CLR. CLR (Common Language Runtime — общеязыковая исполняющая среда) — это виртуальная исполняющая система (Virtual Execution System — VES), реализующая CLI. CLR предоставляет много удобных средств для управляемых приложений, включая, помимо прочего, четко налаженный сборщик мусора для управления памятью, слой безопасности доступа к коду, богатую самоописательную систему типов. В этой главе я покажу вам, как CLR компилирует, упаковывает и выполняет программы C#.

На заметку! Углубленное изучение CLR выходит за рамки тематики настоящей книги, поскольку здесь большее внимание сосредоточено на концепциях и использовании C#. Однако я рекомендую вам поближе познакомиться с CLR. Всегда лучше знать и понимать целевую платформу, и в случае управляемых приложений на C# такой платформой выступает .NET CLR. Для углубленного изучения CLR и всего, о чем будет говориться в настоящей главе, обратитесь к книге Дона Бокса и Криса Селлса *Essential .NET, Volume I: The Common Language Runtime* (Boston, MA: Addison-Wesley Professional, 2002 г.), а также к книге Эндрю Троелсена (Andrew Troelsen) *Pro C# 2005 and the .NET 2.0 Platform, Third Edition* (*Язык программирования C# 2005 и платформа .NET 2.0*, ИД "Вильямс", 2007 г.). Впоследствии вы сможете найти и много других, более специфичных и информативных книг по CLR. За полным освещением слоя CLR, обеспечивающего полную функциональную совместимость с "родными" средами, такими как объекты COM и лежащая в их основе платформа, я рекомендую обратиться к книге Адама Натана (Adam Nathan) *.NET and COM: The Complete Interoperability Guide* (Indianapolis, IN: Sams, 2002 г.). Тематике безопасности доступа кода .NET посвящена книга *.NET Framework Security*, написанная Брайаном А. Ла-Маччиа (Brian A. LaMacchia) и др. (Upper Saddle River, NJ: Pearson Education, 2002 г.). Поскольку ни один разработчик не должен игнорировать безопасность платформы при разработке новых систем, я рекомендую почитать книгу Кейта Брауна (Keith Brown) *The .NET Developer's Guide to Windows Security* (Boston, MA: Addison-Wesley Professional, 2004 г.).

В настоящей главе предлагается высокоуровневое и поверхностное описание механизмов, участвующих в процессе компиляции C# и загрузке кода на выполнение. Как только код загружен, он должен быть скомпилирован в "родной" машинный код платформы, на которой он выполняется. Поэтому вам нужно четко понимать концепцию JIT-компиляции.

JIT-компилятор и CLR

C# компилируется в IL, а IL — это то, что обрабатывает CLR. Спецификация IL включена в стандарт CLI. Вы можете увидеть, как выглядит IL, загрузив приложение "Hello World!" (из предыдущей главы) в программу Intermediate Language Disassembler (ILDASM), входящую в .NET SDK[1]. ILDASM покажет древовидное представление типов данных из сборки, и вы сможете открывать индивидуальные методы и просматривать код IL, который для них сгенерировал компилятор C#. Как видно в листинге 2.1, IL выглядит подобно языку ассемблера; по сути, это и есть язык ассемблера CLR. Он называется IL, потому что является промежуточным шагом между определенным языком и определенной платформой.

Листинг 2.1. Метод `Main` в `HelloWorld.exe` на языке IL

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 13 (0xd)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Hello World!"
    IL_0006: call void [mscorlib]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: ret
} // end of method EntryPoint::Main
```

CLR — это не интерпретатор. Он не транслирует повторно код IL при каждом его выполнении. Хотя на основе интерпретаторов существует много гибких решений (таких, например, как интерпретатор JScript для сервера сценариев Windows (Windows Scripting Host)), обычно они не являются эффективными исполняющими платформами. CLR в действительности компилирует код IL в машинный код, прежде чем выполнять его, т.е. осуществляет JIT-компиляцию. Этот процесс требует некоторого времени, но для каждой части программы это обычно означает лишь одноразовое понижение производительности на весь процесс. Как только код скомпилирован, CLR сохраняет его и просто выполняет скомпилированную версию всякий раз, когда этот код понадобится вновь — почти так же быстро, как код, скомпилированный традиционным образом (а иногда и быстрее).

Хотя фаза JIT-компиляции добавляет сложности и изначально отражается на производительности, все же преимущества компилятора JIT в паре с CLR могут перевесить затраты времени на JIT-компиляцию по следующим причинам.

- *Управляемые приложения могут потреблять меньше памяти.* В общем случае IL-код занимает меньше места в памяти, чем "родной" машинный код. Другими словами, *рабочий набор* управляемых приложений, т.е. количество страниц памяти, потребляемых приложением, обычно меньше, чем у "род

[1] Вы найдете `ILDASM.exe` в каталоге `bin` комплекта .NET SDK, или если у вас инсталлирована среда Visual Studio 2008, то в каталоге `C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin\ildasm.exe`. Вы можете легко запустить ее после открытия экземпляра сеанса командной строки Visual Studio 2008.

ных" приложений. Приложив определенные усилия, вы можете сократить рабочий набор родного приложения до размеров, сравнимых с управляемыми приложениями, но благодаря CLR вы получаете это бесплатно.

- *JIT-компиляции подвергается только тот код, который выполняется.* IL-код обычно более компактен, чем машинный код, поэтому сокращение компилируемого кода до минимума уменьшает образ памяти приложения.
- *CLR может отслеживать частоту вызовов.* Если CLR видит, что раздел JIT-компилированного кода не вызывался в течение длительного времени, то может освободить место, занятое им. При следующем вызове код будет перекомпилирован снова.

CLR также может осуществлять оптимизацию во время выполнения. В родных приложениях вы определяете оптимизацию во время компиляции. Но поскольку в CLR оптимизация происходит во время выполнения, она может применить ее в любой момент. Может случиться, что реализация CLR сможет компилировать код быстрее, но с меньшей степенью оптимизации, и она может так поступать по умолчанию. Однако для кода, который вызывается часто, может быть выполнена перекомпиляция с включенной оптимизацией, чтобы он выполнялся быстрее. Например, модель эффективности CLR может значительно меняться в зависимости от того, сколько центральных процессоров установлено на целевой платформе, или даже в зависимости от того, к какому семейству принадлежит центральный процессор. Для родных приложений вам приходится выполнять много ручной работы — будь то во время выполнения или во время компиляции, — чтобы приспособиться к такой ситуации. Но CLR предоставляет свои средства, так что вы можете намного легче увеличить производительность приложения на многопроцессорных системах. Вдобавок, если CLR определит, что разные части кода, разбросанные по всему приложению, вызываются довольно часто, она может переместить их в памяти так, чтобы они располагались внутри одной группы страниц памяти, тем самым сводя к минимуму количество случаев непопадания на страницу и повышая эффективность кэша кода при выполнении приложения.

Это лишь некоторые аргументы, доказывающие, что CLR — гибкая целевая платформа, и почему ее преимущества перевешивают ущерб для производительности, вызванный начальной JIT-компиляцией.

Сборки и загрузчик сборок

Сборка (assembly) — это дискретная единица многократно используемого кода внутри CLR, по природе своей подобная DLL-библиотеке из мира неуправляемого кода, но на этом все сходство кончается. Сборка может состоять из множества модулей, которые объединяет вместе *манифест*, описывающий содержимое сборки. С точки зрения операционной системы, модуль идентичен DLL. Сборки могут иметь версии, присоединенные к ним, так что несколько одноименных сборок, но с разными версиями, идентифицируются отдельно. Сборки также содержат метаданные, описывающие содержащиеся в них типы. Когда вы поставляете "родную" DLL, обычно вы включаете заголовочный файл и/или документацию, описывающую экспортированные функции. Метаданные удовлетворяют этим потребностям, полностью описывая типы, содержащиеся внутри сборки. Короче говоря, сборки —

это снабженные версиями, самоописательные единицы повторно используемого кода в среде CLR.

Как было сказано в предыдущей главе, когда вы компилируете программу "Hello World!", то в результате получается файл .exe, который фактически является сборкой. Вы можете создавать управляемые сборки, используя любой управляемый язык. Более того, в большинстве случаев любой другой управляемый язык может применять управляемые сборки, написанные на любом управляемом языке. Таким образом, вы легко можете создавать сложные системы, разрабатываемые на нескольких управляемых языках. Например, для создания низкоуровневых типов C++/CLI может быть наиболее естественным языком, а пользовательский интерфейс верхнего уровня лучше создавать на C# или Visual Basic. Чтобы обеспечить функциональную совместимость между разными языками, CLI определяет подмножество системы типов, известную как спецификация общего языка (Common Language Specification — CLS). Если вы применяете только CLS-совместимые типы в своих сборках, то можете быть уверены, что любой управляемый язык сможет их использовать.

Минимизация рабочего набора приложения

В примере "Hello World!" результирующая сборка состоит всего из одного файла. Однако сборки могут состоять и из многих файлов. Эти файлы могут включать скомпилированные модули, ресурсы и любые другие компоненты, перечисленные в *манифесте сборки*. Манифест сборки обычно включается в главный модуль сборки и содержит важную идентифицирующую информацию, включая то, какие части относятся к сборке. Используя эту информацию, загрузчик сборок может определить, помимо прочего, комплектность сборки или факт ее подделки. Сборки могут быть строго или не строго именованными. Строго именованная сборка имеет хеш-код, встроенный в ее манифест, который может быть использован загрузчиком для проверки целостности сборки. Сборки могут также снабжаться электронной подписью, чтобы идентифицировать их разработчика.

Когда запускается исполняемая программа C#, CLR загружает сборку и запускает метод точки входа. Конечно, прежде чем сделать это, он должен быть JIT-компилирован. На этой стадии CLR может потребоваться разрешить некоторые внешние ссылки, чтобы иметь возможность выполнить JIT-компиляцию кода. Например, если ваш метод `Main` создает экземпляр класса по имени `Employee`, то CLR должна найти и загрузить сборку, содержащую тип `Employee`, прежде чем продолжать JIT-компиляцию. Однако замечательное свойство CLR состоит в том, что она загружает сборки по мере необходимости. Поэтому, если у вас есть тип, представляющий метод для печати документа, и он расположен в сборке, отличной от той, что содержит главное приложение, и при этом приложение никогда не реализует эту зависимость, то эта отдельная сборка и не будет никогда загружена. Это предохраняет рабочий набор приложения от чрезмерного роста. Таким образом, при проектировании приложений имеет смысл выделять менее часто используемые средства в отдельные сборки, чтобы CLR загружал их только по мере надобности. В любой момент вы можете сократить рабочий набор приложения, сократить время запуска, а также уменьшить размер "отпечатка памяти" работающего приложения. Ключ к этому — разбиение кода на отдельные единицы, или сборки. Нет смысла создавать приложение из многих сборок, если выполняемый совместно код разбро-

сан по разным сборкам, поскольку в этом случае вы утрачиваете преимущества, связанные с наличием множества сборок.

Присвоение имен сборкам

Вы можете именовать сборки двумя основными способами.

- *Строгое (полное) именование.* Такая сборка имеет имя, состоящее из четырех частей: краткое имя сборки, номер версии, идентификатор культуры в формате ISO, и хеш-маркер. Если имя сборки состоит из всех четырех частей, она считается строго именованной.
- *Частичное именование.* Такая сборка имеет имя, в котором пропущены некоторые детали строго именованной сборки.

Чтобы получить представление о том, как выглядит имя сборки, откройте Windows Explorer и перейдите к вашему глобальному кэшу сборок (Global Assembly Cache — GAC), находящемуся в каталоге `%systemroot%\assembly`. В действительности структура каталогов очень сложна, но подключаемый модуль GAC Explorer представляет то, что вы видите, в браузере. Если вы перейдете в тот же каталог с помощью командной строки, то увидите имена каталогов, используемые GAC для хранения сборок. Не пытайтесь вмешиваться в эту структуру каталогов, иначе вы можете серьезно повредить GAC. Сосредоточившись на представлении в Explorer, вы можете увидеть имена сборок, состоящие из четырех частей. Если часть Culture имени сборки пуста, это значит, что данная сборка нейтральна в отношении культуры, что присуще сборкам, содержащим только код. Я рекомендую изолировать все ваши ресурсы в отдельной сборке, чтобы можно было сделать их специфичными для культуры и легко заменять, не затрагивая код приложения. Подобные рекомендации существуют многие годы и в "родном" программировании с помощью Win32, что позволяет легко локализовать ваше приложение для других языков.

Преимущество строго именованных сборок состоит в том, что они могут быть зарегистрированы в GAC и стать доступными для использования всем приложениям системы. Регистрация сборки в GAC аналогична регистрации в системном реестре COM-сервера. Если сборка не является строго именованной, то приложение может использовать ее лишь локально. Другими словами, сборка должна находиться где-то в каталоге приложения, использующего ее, либо в его подкаталогах. Такие сборки часто называют *приватными*.

Загрузка сборок

Загрузчик сборок проходит сквозь сложную процедуру, когда загружает сборку. Часть этого процесса определяет, какую версию сборки нужно загрузить. Используя конфигурационные файлы приложения, вы можете предоставить загрузчику некоторые подсказки во время определения версии. CLR может загружать сборки по мере надобности, или же вы можете загрузить их явно, вызовом `AppDomain.Load()`. Загрузчик просматривает частично именованные сборки в том же каталоге, где находится работающее приложение, или его подкаталогах. (Такие сборки называются *локальными*.) Загрузчик может также обратиться к GAC при поиске сборки; например, при загрузке сборки с полностью квалифицированным именем загрузчик ищет в GAC перед тем, как обратиться к локальным каталогам.

Версии играют ключевую роль во время загрузки сборок, и все сборки снабжены информацией о версии. Контроль версий встроен в загрузчик CLR с самого начала, и это исключает такую неприятность, которая называется *"адом DLL"* (DLL Hell), когда замена совместно используемой DLL-библиотеки новой версией разрушает приложения, использующие старую версию. Если вы — ветеран, которому пришлось разрабатывать программное обеспечение для Windows в течение последних 15 лет или около того, то вы определенно испытали это на себе. В CLR множество версий одной и той же сборки могут мирно сосуществовать на одной и той же машине без каких-либо конфликтов между собой. Более того, приложения могут выбирать по умолчанию использование наиболее свежей версии, имеющейся на машине, или же специфицировать точную версию, применяя политику версий в своих конфигурационных файлах.

На заметку! Загрузка сборок и работа с версиями — исключительно сложная тема, и ее освещение выходит за рамки этой книги. Пред тем, как загрузить сборку, загрузчик использует различные эвристические приемы для определения того, какую версию следует загрузить. Определив версию, он передает эту информацию вниз — низкоуровневому методу загрузки сборок. Подробнее о загрузке сборок читайте в книге Дона Бокса и Криса Селлса *Essential .NET, Volume I: The Common Language Runtime* (Boston, MA: Addison-Wesley Professional, 2002 г.).

Метаданные

Давайте внимательнее присмотримся к примеру "Hello World!" из листинга 1.1 и сравним его с тем, что вы могли бы делать, если пришли из мира C++. Для начала обратите внимание, что здесь нет никаких включений заголовков. Это потому, что C# не нуждается во включении заголовков. Вместо этого он использует нечто более надежное и информативное, а именно — метаданные. За счет применения метаданных управляемые модули являются самоописательными. В мире C++ для подключения библиотеки к вашему приложению вам нужны две вещи: статическая библиотека или DLL и, обычно, заголовочный файл. Поскольку они существуют как две отдельные сущности, которые вы должны трактовать как одно целое, вполне возможно, что заголовочный файл окажется не соответствующим библиотеке, если вы не проявите должную осторожность. Это может принести неприятности. Управляемые модули, с другой стороны, содержат всю необходимую информацию внутри метаданных, находящихся в самом модуле. Единицей повторного использования в управляемом мире является сборка. То есть сборка, фактически, самодостаточна.

Метаданные также являются расширяемыми, что позволяет вам определять новые типы и атрибуты, которые могут содержаться в метаданных. И в завершение картины — вы можете иметь доступ к метаданным во время выполнения. Например, во время выполнения вы можете пройти по всем полям произвольного типа класса, не зная его объявления заранее, во время компиляции. Сообразительный читатель поймет, что это открывает возможность генерации целых программ и типов во время выполнения, что также совершенно невозможно в родном C++, если только вы не встроите в приложение целый компилятор C++.

Метаданные — это расширяемый формат описания содержимого сборок. К тому же если это не слишком дорого для вас, вы можете определять собственные специ-

альные "атрибуты", которые легко включаются в метаданные типа. В мире управляемого кода почти любая сущность программы с типом может иметь присоединенные к ней метаданные, включая классы, методы, параметры, возвращаемые значения, сборки и т.д. Вы можете определять специальные атрибуты, наследуя их от класса `System.Attribute`, а затем легко ассоциировать экземпляр специального атрибута почти с любой сущностью в вашей сборке.

С метаданными вы можете обращаться и исследовать определения типа и атрибуты, прикрепленные к ним. Метаданные могут рассказать вам о том, поддерживает ли определенный класс объектов данный метод, прежде чем пытаться вызвать его, либо наследует ли данный класс определенный другой. Процесс просмотра метаданных называется *рефлексией*. Обычно при рефлексии типов в сборке вы начинаете с объекта `System.Type`. Вы можете хранить один из этих экземпляров типов, используя ключевое слово C# `typeof`, вызвав `System.Assembly.GetType()`, или несколькими другими способами. В общем случае ключевое слово `typeof` более эффективно, потому что вычисляется во время компиляции, в то время как метод `GetType()`, хотя и более гибкий, поскольку вы можете передать ему произвольную строку, запускается во время выполнения. Получив объект типа, вы можете определить, является ли он классом, интерфейсом, структурой и т.п., какие методы у него есть, количество и типы содержащихся в нем полей.

На заметку! Если вы не понимаете, зачем нужны метаданные, отмечу, что COM/DCOM использует некоторые другие приемы. Если вы когда-либо создавали компоненты COM, то должны быть знакомы с языком описания интерфейсов (Interface Description Language — IDL), который представляет собой независимый от платформы язык описания интерфейсов и компонентов. Обычно вы предоставляете своему заказчику COM-компонент, упакованный в DLL-библиотеку или исполняемую программу, вместе с IDL. Это служит той же цели, что и заголовочный файл для библиотек C++ или документация, сопровождающая DLL. Обычно вы берете IDL и прогоняете его через компилятор IDL, чтобы сгенерировать родной код, с которым вы затем можете взаимодействовать. Библиотека типов (Type Library — TLB) служит почти той же цели, что IDL, но имеет двоичный формат, который потребляют высокоуровневые языки вроде Visual Basic. К сожалению, IDL и TBL не полностью перекрывают друг друга. Некоторые вещи могут быть описаны на IDL, но не на TBL, и наоборот.

Поскольку сборки являются самоописательными, единственное, что нужно компилятору C# для того, чтобы разрешить использование типов — это список сборок, на которые ссылается данная, чтобы скомпилировать и построить программу. Имея такой список, он может обратиться к метаданным, содержащимся внутри них, чтобы разрешить зависимости. Это прекрасная система, и она исключает некоторые чреватые ошибками детали из процесса кодирования C#.

В управляемом мире вам больше не нужно заботиться о дополнительном багаже в виде заголовочных файлов или файлов IDL. Я не стану, правда, утверждать, что вы не должны предоставлять и никакой документации, потому что документация полезна всегда. Но, имея сборку, вы получаете полноценно упакованную сущность, содержащую и код, и описания, необходимые для его использования. Если ваша сборка состоит из единственного файла, как чаще всего и бывает, то этот файл содержит все необходимое для того, чтобы использовать код, находящийся в этой сборке.

Межязыковые возможности

Поскольку сборки самоописательны и содержат в себе переносимый код IL, их легко разделять между многими языками. Наконец-то вы получили жизнеспособное решение для построения сложных систем, в которых некоторые компоненты кодируются на одном языке, а другие — на другом. Например, в сложной системе, используемой для инженерного анализа, вы можете иметь группу разработчиков на C#, кодирующих инфраструктуру системы, и группу инженеров, разрабатывающих математические компоненты. Многие инженеры все еще программируют на таком языке, как Fortran. Это нормально, потому что доступны компиляторы Fortran, генерирующие IL и создающие управляемые сборки. То есть каждая группа разработчиков может работать на языке, более естественном для этой группы и проблемной области, которой она занимается.

Метаданные важны для такого разделения. Джим Миллер (Jim Miller) и Сьюзанн Рэгсдейл (Susann Ragsdale) полностью описывают формат метаданных в книге *The Common Language Infrastructure Annotated Standard* (Boston, MA:Addison-Wesley Professional, 2003 г.). Я рекомендую прочесть ее или же ознакомиться с документацией по стандарту CLI Ecma[2], чтобы лучше понять CLR, а также как генерируются и потребляются метаданные.

Резюме

В настоящей главе было кратко описано, как код на C# компилируется, упаковывается и выполняется.

Я рассказал, как компиляция JIT может конкурировать с традиционно скомпилированными приложениями в плане производительности. Одним из требований для оптимизации JIT-компиляции является выразительный и расширяемый механизм типов, который может быть понятен компилятору. Упаковывая IL в самодокументированные сборки, и компилятор CLR, и компилятор JIT имеют полную информацию, которая нужна им для управления выполнением кода. Вдобавок вы можете явно загружать сборку по требованию, указывая либо строгое, либо частичное ее имя. Сборки позволяют запускать различные версии кода, избегая ситуации "ада DLL", и также предоставляя базу для разработки и разделения компонентов между языками.

В следующей главе вы ознакомитесь с синтаксисом языка C#. Поскольку для раскрытия всех его деталей места не хватит, я рекомендую вам также обратиться к спецификации языка C#.

[2] Документ Ecma-334 описывает стандарт Ecma CLI, а Ecma-334, доступный по адресу www.ecma-international.org — язык C#. ISO/IEC23271 также раскрывает CLI, а ISO/IEC23270 на www.iso.org также посвящен C#. Однако стандарты Ecma обычно более свежие, и вы можете свободно загрузить их на свой компьютер.

ГЛАВА 3

Обзор синтаксиса C#

Эта глава ознакомит вас с синтаксисом языка C#. Предполагается, что вы уже имеете некоторый опыт работы в C++ или Java, потому что C# разделяет сходный с ними синтаксис. Это не случайно. Проектировщики C# сознательно решили использовать знания тех разработчиков, которые уже имели дело с C++ и Java — доминирующими языками в мире объектно-ориентированной разработки программного обеспечения.

Я буду специально отмечать нюансы и отличия, присущие языку C#. Если вы знакомы с C++ или Java, с синтаксисом C# вы почувствуете себя как дома.

C# — строго типизированный язык

Подобно C++ и Java, C# является строго типизированным языком, а это означает, что каждая переменная и экземпляр объекта в системе принадлежат к четко определенному типу. Это позволяет компилятору проверять операции, которые вы пытаетесь выполнить с переменными или экземплярами объектов. Например, предположим, что у вас есть метод, вычисляющий среднее значение из двух целых чисел и возвращающий результат. Вы можете объявить метод C# следующим образом:

```
double ComputeAvg( int param1, int param2 )
{
    return (param1 + param2) / 2.0;
}
```

Приведенный код говорит компилятору, что этот метод принимает два целых числа и возвращает число с плавающей точкой двойной точности (`double`). Таким образом, если компилятор попытается скомпилировать код, где вы вдруг передадите этому методу экземпляр типа `Apple`, он выдаст сообщение об ошибке и прекратит работу. Предположим, вы напишете метод несколько иначе:

```
object ComputeAvg( object param1, object param2 )
{
    return ((int) param1 + (int) param2) / 2.0;
}
```

Вторая версия `ComputeAvg` также верна, но в этом случае вы исключили информацию о типе. Каждый объект и значение в C# неявно унаследованы от `System.Object`. Ключевое слово `object` в C# является псевдонимом класса `System.Object`. Поэтому вполне законно объявлять эти параметры с типом `object`. Хотя вторая версия мето-

да может показаться более гибкой, она допускает разные случайности. Что если некоторый код в приложении попытается передать в ComputeAvg объект типа Apple? Компьютер это пропустит, поскольку Apple наследуется от System.Object, как и любой другой класс. Однако вы получите неприятный сюрприз во время выполнения, когда ваше приложение сгенерирует исключение, сообщающее, что оно не может преобразовать экземпляр Apple в целое число. Метод потерпит неудачу, и если только вы не предусмотрите обработку исключений, он сможет прервать выполнение вашего приложения. Вряд ли вас устроит, чтобы такое случилось в приложении, когда оно будет запущено где-то на рабочем сервере.

Всегда лучше находить ошибки во время компиляции, чем во время выполнения. В этом мораль данной истории. Если вы используете первую версию ComputeAvg, то компилятор сообщит вам о том, что нелепо передавать методу экземпляр Apple. И это намного лучше, чем узнать об этом от рассерженного заказчика, чей коммерческий сервер только что выдал сбой. Компилятор — ваш друг, так что будьте вежливы с ним и предоставляйте ему как можно больше информации о своих намерениях.

Выражения

Выражения в C# практически идентичны выражениям в C++ и Java. При построении выражений важно помнить о приоритетах операций. Выражения C# состоят из операндов — обычно переменных или типов вашего приложения — и операций. Многие из операций могут быть также перегружены. Перегрузка операций рассматривается в главе 6. В табл. 3.1 перечислены приоритеты групп операций. Те, что находятся в верхней части таблицы, имеют более высокий приоритет, а операции внутри одной и той же категории — равный приоритет.

Таблица 3.1. Приоритеты операций C#

Группа операций	Входящие операции	Описание
Первичные	x.m	Доступ к члену.
	x(...)	Вызов метода.
	x[...]	Доступ к массиву.
	x++, x--	Постинкремент и постдекремент.
	new T(...), new T[...]	Создание объекта и массива.
	typeof(T)	Получение объекта System.Type для T.
	checked(x), unchecked(x)	Вычисление выражения в управляемой и неуправляемой среде.
Унарные	+x, -x	Идентичность и отрицание.
	!x	Логическое отрицание.
	~x	Двоичное отрицание.
	++x, --x	Преинкремент и предекремент.
	(T) x	Операция приведения.
Мультипликативные	x*y, x/y, x%y	Умножение, деление и получение остатка.
Аддитивные	x+y, x-y	Сложение и вычитание.
Сдвига	x<<y, x>>y	Сдвиг влево и вправо.

Окончание табл. 3.1

<table>
<tr><th>Группа операций</th><th>Входящие операции</th><th>Описание</th></tr>
<tr><td>Отношений и проверки типа</td><td>x<y, x>y, x<=y, x>=y
x is T
x as T</td><td>Меньше чем, больше чем, меньше или равно, больше или равно.
true, если x может быть преобразовано в T, иначе — false.
Возвращает x, преобразованное в T, или null, если преобразование невозможно.</td></tr>
<tr><td>Эквивалентности</td><td>x == y, x != y</td><td>Эквивалентно и неэквивалентно.</td></tr>
<tr><td>Логического И</td><td>x & y</td><td>Целочисленное побитовое "И", булевское логическое "И".</td></tr>
<tr><td>Логического исключающее ИЛИ</td><td>x ^ y</td><td>Целочисленное побитовое исключающее "ИЛИ", булевское логическое исключающее "ИЛИ".</td></tr>
<tr><td>Логического ИЛИ</td><td>x | y</td><td>Целочисленное побитовое "ИЛИ", булевское логическое "ИЛИ"</td></tr>
<tr><td>Условное И</td><td>x && y</td><td>Возвращается y, только если x истинно.</td></tr>
<tr><td>Условное ИЛИ</td><td>x || y</td><td>Возвращается y, только если x ложно.</td></tr>
<tr><td>Сравнения с null</td><td>x ?? y</td><td>Если x не равно null, возвращается x, иначе — y.</td></tr>
<tr><td>Условные</td><td>x ? y : z</td><td>Возвращается y, если x истинно, иначе — z.</td></tr>
<tr><td>Присваивания</td><td>x = y
x op= y</td><td>Простое присваивание.
Составное присваивание: может быть *=, /=, %=, +=, -=, <<=, >>=, &=, ^= или |=.</td></tr>
</table>

На заметку! Эти операции могут иметь разное значение в разных контекстах. Независимо от этого, их приоритеты никогда не меняются. Например, операция + может означать конкатенацию строк, если вы применяете ее к строковым операндам. Используя перегрузку операций при определении ваших собственных типов, вы можете заставить эти операции выполнять нечто семантически осмысленное для данного типа. Тем не менее, приоритеты операций вы не можете изменить, за исключением тех случаев, когда применяются скобки.

Операторы и выражения

Операторы в C# идентичны по форме операторам C++ и Java. Точка с запятой завершает однострочный оператор. Однако блоки кода, заключенные в фигурные скобки (вроде тех, что следуют за операторами if и while), не должны завершаться точкой с запятой. Добавление точки с запятой в конец блока не обязательно.

Большинство операторов, доступных в C++ и Java, также доступны в C#, включая операторы объявления переменных, условные операторы, операторы управления потоком исполнения, операторы try/catch и т.д. Тме не менее, в C# (как и в Java) доступны некоторые операторы, которых нет в C++. Например, в C# предусмотрен оператор try/finally, который подробно обсуждается в главе 7. В главе 12 будет описан оператор lock, синхронизирующий доступ к блокам кода, за счет использования блока синхронизации объекта. В C# также перегружается ключевое слово using, так что вы можете использовать его и как директиву, и как опера-

тор. Вы можете применять оператор `using` в сочетании с шаблоном Disposable, о котором говорится в главах 4 и 13. Оператор `foreach`, облегчающий итерацию по элементам коллекций, также заслуживает упоминания. Подробнее обо всех этих операторах рассказывается в главе 9, когда пойдет речь о массивах.

Типы и переменные

Каждая сущность в программе C# является объектом, находящимся либо в стеке, либо в управляемой куче. Каждый метод определен в объявлении класса или структуры. Здесь нет таких вещей, как свободные функции, определенные вне контекста объявления `class` или `struct`, как в C++. Даже встроенные типы значений, такие как `int`, `long`, `long double` и т.д., имеют методы, ассоциированные с ними неявно. Поэтому в C# вполне допускается написать оператор вроде следующего:

```
System.Console.WriteLine( 42.ToString() );
```

Оператор, подобный этому, где метод вызывается на значении `42`, покажется незнакомым тем из вас, кто работал на C++ или Java. Однако он подчеркивает, что в C# все является объектами, даже самые базовые типы. Фактически ключевые слова встроенных типов в C# на самом деле отображаются на типы из пространства имен `System`, представляющие их. Вы даже можете не использовать встроенные типы C#, а вместо этого явно применять типы из пространства имен `System`, на которые они отображаются (хотя такая практика нежелательна из соображений стиля). В табл. 3.2 описаны встроенные типы с указанием их размера и типов, на которые они отображаются в пространстве имен `System`.

Таблица 3.2. Встроенные типы C#

Тип C#	Размер в битах	Тип System	Совместимость с CLS
sbyte	8	System.SByte	Нет
short	16	System.Int16	Да
int	32	System.Int32	Да
long	64	System.Int64	Да
byte	8	System.Byte	Да
ushort	16	System.Uint16	Нет
uint	32	System.Uint32	Нет
ulong	64	System.Uint64	Нет
char	16	System.Char	Да
bool	8	System.Boolean	Да
float	32	System.Single	Да
double	64	System.Double	Да
decimal	128	System.Decimal	Да
string	–	System.String	Да
object	–	System.Object	Да

Для каждого элемента этой таблицы в последней колонке указана совместимость данного типа со спецификацией общего языка (Common Language Specification — CLS). CLS определена как часть стандарта CLI для облегчения функциональной совместимости между языками. CLS — подмножество общей системы типов (Common Type System — CTS). Даже несмотря на то, что CLR поддерживает богатый набор встроенных типов, не все языки, компилируемые в управляемый код, поддерживают весь перечень. Однако все управляемые языки должны поддерживать типы из CLS. Например, Visual Basic традиционно никогда не поддерживал беззнаковых типов. Поэтому проектировщики CLI определили CLS для стандартизации типов с целью облегчения функциональной совместимости между языками. Если ваше приложение будет целиком написано на C#, и не будет создавать никаких компонентов, позаимствованных из других языков, то вам не стоит беспокоиться о соответствии строгим требованиями CLS. Но если вы работаете над проектом, в котором компоненты строятся с использованием различных языков, то соответствие CLS становится намного более важным.

В управляемом мире CLR существует два вида типов.

- *Типы значений* (value types). Определяются в C# с применением ключевого слова `struct`. Экземпляры типов значений — единственный вид экземпляров, которые могут располагаться в стеке. В куче они находятся, если являются членами ссылочных типов или же упакованы в объекты ссылочных типов, о чем будет рассказываться позднее. Эти типы подобны структурам C++ в том смысле, что по умолчанию они копируются по значению при передаче в виде параметров методам или присвоении другим переменным. Хотя встроенные типы C# представляют тот же вид значений, что и примитивные типы Java, полных аналогов в Java для них нет.
- *Ссылочные типы* (reference types). Определяются в C# с применением ключевого слова `class`. Называются ссылочными типами потому, что переменные, используемые для манипуляций ими, в действительности являются ссылками на объекты из управляемой кучи. Фактически переменные ссылочных типов CLR подобны типам значений, которые ссылаются на объекты в куче. В этом смысле языки C# и Java идентичны. Программисты C++ могут воспринимать их как указатели, которые не нужно разыменовывать для обращения к объекту. Некоторые программисты на C++ предпочитают воспринимать их как "интеллектуальные указатели" (smart pointers).

Типы значений

Типы значений могут находиться как в стеке, так и в управляемой куче. Вы используете их обычно тогда, когда вам нужно представить некоторые неизменяемые данные, обычно не занимающие много места в памяти. Вы можете определять типы значений в C#, используя ключевое слово `struct`.

На заметку! Несмотря на то что в C++ есть ключевое слово `struct`, его значение в C# отличается тем, что это единственный способ создания типов значений в C#.

Определенные пользователем типы значений ведут себя так же, как встроенные типы значений.

Когда вы создаете значение в потоке исполнения, это значение создается в стеке, как показано в следующем фрагменте кода:

```
int theAnswer = 42;
System.Console.WriteLine( theAnswer.ToString() );
```

Экземпляр theAnswer не только создается в стеке, но при передаче его методу этот метод получает его копию. Типы значений обычно используются в управляемых приложениях для представления облегченных порций или наборов данных, подобно использованию встроенных типов и структур в C++, а также примитивных типов Java.

Значения также могут находиться и в управляемой куче, но не сами по себе. Единственная возможность — это когда ссылочный тип имеет поле типа значения. Несмотря на то что тип значения внутри объекта расположен в управляемой куче, он все равно ведет себя как тип значения в стеке, когда речь идет о передаче его методу; т.е. метод по умолчанию получит его копию. Любые изменения, внесенные в экземпляр значения, являются лишь локальными, если только значение не было передано по ссылке. В следующем коде иллюстрируются эти концепции.

```
public struct Coordinate //это тип значения
{
    public int x;
    public int y;
}
public class EntryPoint  //это ссылочный тип
{
    public static void AttemptToModifyCoord( Coordinate coord ) {
        coord.x = 1;
        coord.y = 3;
    }
    public static void ModifyCoord( ref Coordinate coord ) {
        coord.x = 10;
        coord.y = 10;
    }
    static void Main() {
        Coordinate location;
        location.x = 50;
        location.y = 50;
        AttemptToModifyCoord( location );
        System.Console.WriteLine( "( {0}, {1} )", location.x,
                                  location.y );
        ModifyCoord( ref location );
        System.Console.WriteLine( "( {0}, {1} )", location.x,
                                  location.y );
    }
}
```

В методе Main вызов AttemptToModifyCoord на самом деле ничего не делает со значением location. Это объясняется тем, что AttemptToModifyCoord модифицирует локальную копию значения, которая была создана при вызове метода. Значение location передается по ссылке методу ModifyCoord. Поэтому все изменения в методе ModifyCoord в действительности проводятся над значением

location из вызывающего контекста. Это подобно передаче значения через указатель в C++. Вывод этого примера выглядит так:

```
( 50, 50 )
( 10, 10 )
```

Перечисления

Перечисления (enum) в C# подобны перечислениям в C++, и синтаксис их определения практически идентичен. Но в точке использования, однако, вы должны полностью квалифицировать значения из перечисления, используя имя типа этого перечисления. Все перечисления основаны на лежащем в основе целочисленном типе int, если не указано иначе.

На заметку! Лежащий в основе enum тип должен быть целочисленным, т.е. одним из следующих: byte, sbyte, short, ushort, int, uint, long или ulong.

Каждая константа, определенная в перечислении, должна быть определена со значением, находящимся в пределах диапазона лежащего в основе типа. Если вы не указываете явно значения константы перечисления, то значение получается по умолчанию 0 (если это первая константа в перечислении) или 1 + значение предыдущей константы. Вот пример перечисления на базе long:

```
public enum Color : long
{
    Red,
    Green = 50,
    Blue
}
```

В данном примере, если пропустить двоеточие и ключевое слово long после идентификатора типа Color, то перечисление было бы типа int. Обратите внимание, что значение Red равно 0, значение Green — 50, а значение Blue — 51.

Чтобы использовать это перечисление, напишите код вроде следующего:

```
static void Main() {
    Color color = Color.Red;
    System.Console.WriteLine( "Color is {0}", color.ToString() );
}
```

Если вы откомпилируете и запустите этот код, то увидите, что вывод на самом деле использует имя перечисления вместо обычного значения 0. Реализация метода ToString() типа System.Enum отвечает за всю "магию".

Часто константы перечисления используются для представления битовых флагов. Вы можете присоединить к перечислению атрибут из пространства имен System по имени System.FlagsAttribute, чтобы сделать это явно. Атрибут сохраняется в метаданных, и вы можете сослаться на него во время проектирования для определения того, предназначены ли члены перечисления для использования в качестве битовых флагов. К тому же вы можете ссылаться на эти атрибуты в некоторых местах во время выполнения — например, когда значение перечисления преобразуется в строку. Обратите внимание, что System.FlagsAttribute не модифицирует поведение значений, определенных перечислением. Во время выпол-

нения, однако, некоторые компоненты могут использовать метаданные, сгенерированные атрибутом, для обработки значения иным образом. Это отличный пример того, как можно эффективно использовать метаданные в стиле аспектно-ориентированного программирования (aspect-oriented programming — AOP).

На заметку! Понятие AOP, также известное, как аспектно-ориентированная разработка программного обеспечения (aspect-oriented software development — AOSD) — это концепция, изначально разработанная Грегором Кицзалесом (Gregory Kiczales) и его командой из Xerox PARC. Объектно-ориентированные методологии обычно замечательно справляются с разделением функциональности, или отношений, при проектировании взаимодействующих единиц. Однако некоторые отношения, называемые *перекрестными отношениями* (crosscutting concerns), невозможно легко смоделировать в рамках стандартного объектно-ориентированного проектирования. Например, представим, что вам нужно протоколировать вход и выход в разные методы. Было бы сплошным кошмаром модифицировать код каждого метода, который нужно протоколировать. Намного проще было бы присоединить свойство — или в данном случае, атрибут — к самому методу, чтобы исполняющая система протоколировала его вызовы. Это избавит вас от необходимости модифицировать метод, и такое требование применяется вне зависимости от его реализации. Сервер транзакций Microsoft (Microsoft Transaction Server — MTS) стал одним из первых широко известных примеров внедрения AOP.

Используя метаданные и тот факт, что вы можете присоединять произвольные специальные атрибуты к типам, методам, свойствам и т.д., вы с успехом можете применить AOP в своем собственном проекте.

Ниже приведен пример перечисления битовых флагов:

```
[Flags]
public enum AccessFlags
{
    NoAccess = 0x0,
    ReadAccess = 0x1,
    WriteAccess = 0x2,
    ExecuteAccess = 0x4
}
```

А вот пример использования перечисления `AccessFlags`:

```
static void Main() {
    AccessFlags access = AccessFlags.ReadAccess |
          AccessFlags.WriteAccess;
    System.Console.WriteLine( "Access is {0}", access );
}
```

Если скомпилировать и запустить этот пример, вы увидите, что метод `Enum.ToString`, неявно вызываемый из `WriteLine`, фактически выводит разделенный запятыми список всех бит, установленных в этом значении.

Ссылочные типы

Сборщик мусора (GC) внутри CLR управляет всем, что касается размещения объектов. Он может перемещать объекты в любое время. При этом CLR обновляет переменные, ссылающиеся на эти объекты. Обычно вас не заботит точное местоположение объекта в куче, и вам не нужно беспокоиться о том, перемещен он или

нет. Но есть редкие случаи, например, при взаимодействии с "родными" DLL-библиотеками, когда вам может понадобиться получить прямой указатель памяти на объект в куче. Это возможно посредством техники *небезопасного* (unsafe) (или *неуправляемого*) кода, однако эта тема выходит за рамки настоящей книги.

На заметку! Условно термин *объект* ссылается на экземпляр ссылочного типа, в то время как термин *значение* — на экземпляр типа значений, но все экземпляры любого типа также унаследованы от типа `object`.

Переменные ссылочного типа инициализируются либо посредством операции `new` для создания объекта в управляемой куче, либо инициализируются присвоением другой переменной совместимого типа. Следующий фрагмент кода устанавливает две переменных указывать на один и тот же объект:

```
string idTag = "423-XYZ";
string theTag = idTag;
```

Подобно исполняющей системе Java, CLR управляет всеми ссылками на объекты в куче. В C++ вы должны явно удалять объекты, расположенные в куче — в некоторый тщательно выбранный момент. Но в управляемой среде CLR за вас это делает GC. Это избавляет вас от необходимости заботиться об удалении объектов из памяти и минимизирует утечки памяти. GC может определить в любой момент времени, сколько ссылок существует на определенный объект в куче. Если он определяет, что их нет, значит, можно начать процесс уничтожения объекта в куче. (В главе 13 обсуждается сложность этого процесса и факторы, которые на нее влияют.) Предыдущий фрагмент кода включает две ссылки на один и тот же объект. Вы инициализируете один из них — `idTag`, создавая строковый объект. Второй — `theTag`, вы инициализируете из `idTag`. GC не уберет этот строковый объект из кучи до тех пор, пока обе ссылки не выйдут из их области видимости. Если бы метод возвращал копию ссылки тому, кто вызывает его, то GC продолжал бы отслеживать ссылку на данный объект, даже несмотря на то, что метод, создавший его, уже завершился.

На заметку! Для тех из вас, кто пришел из мира C++: фундаментальный способ трактовки объектов в C++ "вывернут наизнанку" в мире C#. В C++ объекты размещаются в стеке, если только вы не создаете их явно операцией `new`, которая возвращает указатель на объект в "родной" куче. В C# вы не можете создавать объекты ссылочных типов в стеке. Они могут существовать только в куче. Так что это похоже на то, как если бы вы писали код C++, в котором создавали каждый объект в куче, не заботясь потом о его явном удалении для очистки памяти.

Инициализация переменных по умолчанию

По умолчанию компилятор C# производит то, что называется *безопасным кодом* (safe code). Одним из аспектов безопасности является то, что программа не использует неинициализированную память. Компилятор требует, чтобы каждой переменной было присвоено значение прежде, чем можно будет работать с ней, так что полезно знать, как инициализируются переменные разных типов.

Значением по умолчанию для ссылок на объекты является `null`. В точке объявления вы можете необязательно присваивать ссылки, полученные в результате вызова операции `new`; в противном случае они остаются установленными в `null`. Когда вы создаете объект, исполняющая система инициализирует его внутренние поля. Поля, являющиеся ссылками на объекты, разумеется, инициализируются значением `null`. Поля, относящиеся к типам значений, инициализируются установкой всех битов значения в ноль. По сути, вы можете представить, что все, что делает исполняющая система — это установка лежащего в основе хранилища в 0. Для ссылок на объекты это соответствует `null`-ссылке, а для типов значений — нулевому значению.

Для типов значений, которые вы объявляете в стеке, компилятор не выполняет автоматически инициализации нулями. Однако он требует, чтобы вы инициализировали память прежде, чем использовать значение.

На заметку! Поскольку перечисления на самом деле являются типами значений, вы всегда должны объявлять член перечисления, равный нулю, даже если именем члена является `InvalidValue`, `None` или что-то другое, лишенное смысла. Если перечисление объявлено как поле класса, экземпляры этого класса будут иметь это поле установленным в ноль при инициализации по умолчанию. Объявление члена, равного нулю, позволит пользователям вашего перечисления легко справляться с такой ситуацией.

Неявно типизированные локальные переменные

Поскольку C# — строго типизированный язык, каждая объявленная в коде переменная должна иметь определенный тип, ассоциированный с ней. Когда CLR сохраняет содержимое переменной в определенном месте памяти, она также ассоциирует тип с этим местоположением. Но иногда при написании кода для строго типизированных языков объем ввода, необходимый для объявления таких переменных, может быть довольно большим, особенно если речь идет об обобщенном типе. Для сложных типов, таких как переменные запросов, созданные с применением языка интегрированных запросов (LINQ), о которых пойдет речь в главе 16, имена типов могут оказаться очень громоздкими. В этом случае обратитесь к неявно типизированным переменным.

Объявляя локальную переменную с использованием нового ключевого слова `var`, вы в действительности просите компьютер зарезервировать локальный участок памяти и затем присоединить к нему каким-то образом выведенный тип. Во время компиляции у компилятора достаточно доступной информации в точке инициализации переменной, чтобы вывести действительный тип этой переменной, без необходимости с вашей стороны указывать тип явно. Посмотрим, как это выглядит.

Ниже приведен пример объявления неявно типизированной переменной:

```
using System;
using System.Collections.Generic;
public class EntryPoint
{
    static void Main() {
        var myList = new List<int>();
        myList.Add( 1 );
```

```
            myList.Add( 2 );
            myList.Add( 3 );
            foreach( var i in myList ) {
                Console.WriteLine( i );
            }
        }
    }
```

Первое, что вы должны заметить — выделенные полужирным ключевые слова, демонстрирующие использование нового ключевого слова `var`. В первом случае я объявил переменную по имени `myList` и попросил компилятор установить ее тип на основе типа выражения, которое ей присваивается. Важно отметить, что объявление неявно типизированной переменной должно включать инициализатор. Если вы пытаетесь в своем коде написать нечто, подобное тому, что вы видите ниже, то получите от компилятора предупреждение CS0810, сообщающее: "Implicitly typed locals must be initialized" (Неявно типизированные локальные переменные должны быть инициализированы):

```
var newValue; // выдаст ошибку CS0818
```

Аналогично, если вы попытаетесь объявить поле-член класса как неявно типизированную переменную, даже если она будет иметь инициализатор, вы получите ошибку компиляции CS0825, сообщающую: "The contextual keyword 'var' may only appears within a local variable declaration" (Контекстуальное ключевое слово 'var' может применяться только в объявлении локальной переменной).

И, наконец, вы можете использовать объявление нескольких переменных в одной строке, отделяя каждую из них запятой, как в следующих двух примерах:

```
int a = 2, b = 1;
int x, y = 4;
```

Однако вы не можете делать это с неявно типизированными переменными. Компилятор выдаст ошибку CS0819, сообщающую: "Implicitly typed locals cannot have multiple declarations" (Неявно типизированные локальные переменные не допускают множественного объявления).

На заметку! Ранние черновые редакции стандарта C# 3.0, гуляющие по Internet, указывают, что неявно типизированные переменные могут быть инициализированы с использованием множественных объявлений, пока каждый инициализатор дает в результате один и тот же тип. Однако предварительная версия компилятора, которую я использую при написании настоящей книги, не поддерживает этого.

Сложность добавления новых ключевых слов к языку

Добавление к компилятору новых средств, подобных неявно типизированным переменным — не такая тривиальная задача, как может показаться на первый взгляд. Это связано с тем, что всякий раз, когда вы вводите новое слово в язык, вам следует учесть возможность нарушения работы существующего кода и позаботиться об обратной совместимости. Например, представьте, что случится, если у вас есть громадная база кода, написанного на C# 1.0 или C# 2.0, в которой присутствует некий тип, скажем, класс, по имени `var`. Теперь вы собираетесь перенести это все на C# 3.0, и компилируете приложение с применением

нового компилятора. Ясно, что у вас, скорее всего, есть некоторые объявления переменных, создающих экземпляры вашего класса var. Что должен делать компилятор? Ответить на этот вопрос очень непросто.

В тестах с предварительной версией компилятора я попробовал сделать то, что описал — мой компилятор не делал ничего. Но должно ли так быть? Он мог бы выдать предупреждение, сообщающее нечто вроде "объявленный тип имеет имя, совпадающее со встроенным ключевым словом 'var'". Но на самом деле это вовсе не обязательно, и я покажу скоро — почему. Фактически даже лучше, чтобы компилятор не выдавал такого предупреждения. Хорошие группы разработчиков используют опцию компилятора /WARNASERRORS+, чтобы прекратить компиляцию в случае появления предупреждений в коде. Так что если компилятор выдает предупреждение, ваше приложение не будет компилироваться при переходе на C# 3.0, и Microsoft должна будет подвергнуться критике за то, что игнорирует принципы обратной совместимости.

Основной результат состоит в том, что если вы имеете тип, определенный с именем var, вы просто не можете использовать явно типизированную переменную в любом коде C#, где пространство имен этого типа импортировано в локальный контекст. Если вы это сделаете, то обычно получите ошибку компилятора CS0029, которая говорит по сути о том, что тип, который вы пытаетесь назначить неявно типизированной переменной, не может быть неявно преобразован в ваш пользовательский тип var. Вот так-так! Что за ерунда? Например, следующий код демонстрирует описанное поведение:

```
using System;
using System.Collections.Generic;
public class var
{
}
public class EntryPoint
{
  static void Main() {
      var myList = new List<int>(); // Не компилируется! Ошибка CS0029
  }
}
```

Разработчики компиляторов обычно относятся к подобным проблемам чрезвычайно серьезно, и иногда даже отказываются от ввода нового ключевого слова, если существует вероятность разрушения существующего кода. Однако у разработчиков компилятора C# имеется козырь. Если вы следуете рекомендуемым соглашениям по именованию типов .NET, с начальной заглавной буквой, вы никогда не столкнетесь с подобной ситуацией. Вдобавок, если вы используете анализатор кода Visual Studio или FxCop (отдельная версия анализатора кода) во время разработки приложения, то вы также никогда не столкнетесь с этой проблемой. Все эти правила и руководства также раскрываются во всех подробностях в книге Кшиштофа Квалины (Krzysztof Cwalina) и Брэда Абрамса (Brad Abrams) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Boston, MA: Addison-Wesley Professional, 2005 г.). Я настоятельно рекомендую прочесть эту книгу, если вы еще этого не сделали.

Преобразования типов

Очень часто возникает необходимость преобразовывать экземпляры одного типа в другой. В некоторых случаях компилятор выполняет такое преобразование неявно — когда значение одного типа присваивается переменной другого типа, и

при этом не теряется точность и значение. В случае, если точность может быть утеряна, требуется явное преобразование. Для ссылочных типов правила преобразования аналогичны правилам преобразования указателей в C++.

Семантика преобразования типов подобна той, которая реализована в C++ и Java. Явные преобразования выполняются с применением знакомого синтаксиса приведения, который все они унаследовали от C, т.е. тип, к которому нужно преобразовать, помещается в скобках перед преобразуемым типом:

```
int defaultValue = 12345678;
long value = defaultValue;
int smallerValue = (int) value;
```

В этом коде приведение (int) должно быть явным, поскольку размер int меньше, чем long. То есть существует возможность того, что значение типа long не уместится в пространство, отведенное под int. Присваивание переменной defaultValue переменной value не требует приведения, поскольку long имеет больше места для хранения, чем int. Если при преобразовании теряется значение, возможно, что эта операция сгенерирует исключение во время выполнения. Общее правило гласит, что неявные преобразования гарантированно никогда не сгенерируют исключений, в то время как явные — наоборот, могут.

Язык C# предоставляет средства для определения пользовательских операций явного и неявного приведения определенных пользователем типов к различным другим типам. Эта тема более детально раскрывается в главе 6. Требования исключений для встроенных операторов преобразования применимы к операциям приведения, определенных пользователем. В частности, операции неявных приведений гарантированно никогда не генерируют исключений.

Преобразования к ссылочным типам и обратно моделируют аналогичные операции в Java, а также операции преобразования между указателями в C++. Например, ссылка на тип DerivedType может быть неявно преобразована к ссылке на тип BaseType, если DerivedType наследуется от BaseType. Однако преобразование в обратном направлении должно осуществляться явно. К тому же явное приведение может сгенерировать исключение System.InvalidCastException во время выполнения программы, если CLR не сможет выполнить такое приведение.

Есть один тип неявного приведения, доступного в C#, которое реализовать в C++ не просто — главным образом, из-за семантики значений по умолчанию в C++. Можно неявно конвертировать массив одного ссылочного типа в массив другого ссылочного типа, пока целевой ссылочный тип таков, что может быть неявно преобразован от исходного ссылочного типа, и массивы имеют одинаковую размерность. Например, следующее преобразование совершенно законно:

```
public class EntryPoint
{
    static void Main() {
        string[] names = new string[4];
        object[] objects = names;  // оператор неявного преобразования
        string[] originalNames =
           (string[]) objects;     // оператор явного преобразования
    }
}
```

Поскольку System.String наследуется от System.Object, и, следовательно, неявно преобразуем к System.Object, такое неявное преобразование массива string по имени names в массив object по имени objects возможно. Однако для того, чтобы пойти в обратном направлении, как показано, необходимо явное приведение, которое может сгенерировать исключение в случае неудачи.

Имейте в виду, что неявные преобразования аргументов могут происходить при вызове метода, если аргументы должны быть преобразованы для соответствия типам параметров. Если вы не можете выполнить преобразование неявно, придется выполнить явное приведение.

И, наконец, еще один распространенный вид приведений — *упаковывающее преобразование* (boxing conversion). Такое преобразование требуется, когда вы должны передать тип значения как параметр ссылочного типа в метод, или же присвоить его переменной ссылочного типа. При этом происходит динамическое размещение в куче объекта, содержащего поле типа значения. Значение затем копируется в это поле. Такого рода упаковка подробно описывается в главе 4. В следующем коде демонстрируется идея.

```
public class EntryPoint
{
    static void Main() {
        int employeeID = 303;
        object boxedID = employeeID;
        employeeID = 404;
        int unboxedID = (int) boxedID;
        System.Console.WriteLine( employeeID.ToString() );
        System.Console.WriteLine( unboxedID.ToString() );
    }
}
```

В точке, где переменной boxedID типа object присваивается значение переменной employeeID типа int, происходит упаковка. Создается объект, базирующийся в куче, и в него копируется значение employeeID. Это заполняет разрыв между мирами типов значения и ссылочных типов внутри CLR. Объект boxedID в действительности содержит копию значения employeeID. Я демонстрирую этот факт, изменяя оригинальное значение employeeID после операции упаковки. Прежде чем напечатать значения, я распаковываю значение и копирую значение, содержащееся в объекте, из кучи обратно в другой int в стеке. Распаковка в C# требует явного приведения, поскольку здесь возможно возникновение исключения неправильного приведения.

Операции as и is

Поскольку явное приведение может потерпеть неудачу, сгенерировав исключение, бывает так, что вы хотите проверить тип переменной без выполнения приведения и наблюдения, получится оно или нет. Проверка такого рода утомительна и неэффективна, к тому же исключения дорого обходятся во время выполнения. По этой причине в C# предусмотрены операции, которые приходят на помощь в таких ситуациях, и применение которых гарантированно не приведет к исключениям:

- is
- as

Операция is дает в результате булевское значение, говорящее о том, можете ли вы преобразовать данное выражение в указанный тип, как посредством приведения ссылки, так и посредством операций упаковки и распаковки. Например, рассмотрим следующий код:

```
using System;
public class EntryPoint
{
    static void Main() {
        String derivedObj = "Dummy";
        Object baseObj1 = new Object();
        Object baseObj2 = derivedObj;
        Console.WriteLine( "baseObj2 {0} String",
           baseObj2 is String ? "is" : "isnot" );
        Console.WriteLine( "baseObj1 {0} String",
           baseObj1 is String ? "is" : "isnot" );
        Console.WriteLine( "derivedObj {0} Object",
           derivedObj is Object ? "is" : "isnot" );
        int j = 123;
        object boxed = j;
        object obj = new Object();
        Console.WriteLine( "boxed {0} int",
           boxed is int ? "is" : "isnot" );
        Console.WriteLine( "obj {0} int",
           obj is int ? "is" : "isnot" );
        Console.WriteLine( "boxed {0} System.ValueType",
           boxed is ValueType ? "is" : "isnot" );
    }
}
```

Вывод этого кода будет таким:

```
baseObj2 is String
baseObj1 isnot String
derivedObj is Object
boxed is int
obj isnot int
boxed is System.ValueType
```

Как уже упоминалось, операция is рассматривает только преобразование ссылок. Это значит, что она не может проверять никаких определенных пользователем преобразований, имеющихся в типах.

Операция as подобна is за исключением того, что она возвращает ссылку на целевой тип. Поскольку гарантируется, что она никогда не сгенерирует исключения, здесь просто возвращается null-ссылка, если данное преобразование невозможно. Подобно is, операция as рассматривает только преобразования ссылок или преобразования с упаковкой. Например, взгляните на следующий код:

```
using System;
public class BaseType {}
public class DerivedType : BaseType {}
public class EntryPoint {
```

```
    static void Main() {
        DerivedType derivedObj = new DerivedType();
        BaseType baseObj1 = new BaseType();
        BaseType baseObj2 = derivedObj;
        DerivedType derivedObj2 = baseObj2 as DerivedType;
        if( derivedObj2 != null ) {
          Console.WriteLine( "Преобразование успешно" );
        } else {
          Console.WriteLine( "Преобразование не удалось" );
        }
        derivedObj2 = baseObj1 as DerivedType;
        if( derivedObj2 != null ) {
          Console.WriteLine( "Преобразование успешно" );
        } else {
          Console.WriteLine( "Преобразование не удалось" );
        }
        BaseType baseObj3 = derivedObj as BaseType;
        if( baseObj3 != null ) {
          Console.WriteLine( "Преобразование успешно" );
        } else {
          Console.WriteLine( "Преобразование не удалось" );
        }
    }
}
```

Вывод этого кода будет таким:

```
Преобразование успешно
Преобразование не удалось
Преобразование успешно
```

Иногда вам понадобится проверить, относится ли переменная к определенному типу и если да, то выполнить какую-то операцию над нужным типом. Вы можете проверить переменную на принадлежность к типу, применив операцию `is`, а затем, если она вернет `true`, привести переменную к этому типу. Однако это будет не эффективно. Более удачный подход заключается в том, чтобы следовать идиоме применения операции `as` для получения ссылки на переменную с нужным типом, а затем проверить ее на неравенство `null`, что будет означать, что преобразование успешно. Таким образом, вы выполните только одну операцию поиска вместо двух.

Обобщения

Поддержка обобщений (generics) — одно из наиболее впечатляющих нововведений в языке C#. Используя синтаксис обобщений, вы можете определить тип, зависящий от другого типа, который не специфицирован в точке определения, а вместо этого определен в точке использования обобщенного типа. Например, возьмем тип коллекции. Типы коллекций обычно воплощают в себе такие вещи, как списки, очереди и стеки. Типы коллекций, существующие со времен .NET 1.0, предназначены для хранения любых типов в CLR, поскольку они содержат экземпляры `Object` и все, что наследуется от `Object`. Однако при этом вся информация о конкретном типе элементов коллекции отбрасывается, и все умение компилятора отлавливать

ошибки типа оказывается непригодным. Вы можете приводить ссылку типа, полученную из этой коллекции, к любому другому типу, который, по вашему мнению, должен иметь элемент, а это может привести к сбою во время выполнения. К тому же оригинальные типы коллекций могут содержать произвольную смесь типов элементов вместо того, чтобы принуждать пользователя в коллекцию вставлять только экземпляры определенного типа. Вы можете попробовать решить эту проблему, разработав свои типы, такие как `ListOfIntegers` и `ListOfStrings`, для каждого типа, который собираетесь хранить в коллекции. Однако вы скоро обнаружите, что большая часть управляемого кода таких списков будет похожей, или обобщенной, независимо от типа хранимых элементов. Ключевое слово здесь — *обобщенной*. Применяя обобщенные типы, вы можете объявлять открытый (или обобщенный) тип и написать общий код лишь однажды. Пользователь вашего типа может затем специфицировать, какой тип элементов будет содержать коллекция, прямо в точке, где он решит использовать ее.

Вдобавок применение обобщений сулит выигрыш в эффективности. Концепция обобщений настолько обширна, что их объявлению и использованию посвящена целая глава 11. Однако я считаю, что полезно дать вам почувствовать вкус к применению обобщенных типов уже сейчас, поскольку до главы 11 они будут упоминаться еще не раз.

На заметку! Синтаксис обобщений покажется знакомым тем из вас, кто применял шаблоны C++. Однако важно заметить, что между ними и обобщениями C# есть существенные отличия в поведении, о чем рассказывается в главе 11.

Наиболее распространенное применение обобщений — при объявлении типов коллекций. Например, рассмотрим следующий код:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
class EntryPoint
{
    static void Main() {
        Collection<int> numbers =
          new Collection<int>();
        numbers.Add( 42 );
        numbers.Add( 409 );
        Collection<string> strings =
          new Collection<string>();
        strings.Add( "Joe" );
        strings.Add( "Bob" );
        Collection< Collection<int> > colNumbers
          = new Collection<Collection<int>>();
        colNumbers.Add( numbers );
        IList<int> theNumbers = numbers;
        foreach( int i in theNumbers ) {
          Console.WriteLine( i );
        }
    }
}
```

Этот пример демонстрирует применение обобщенного типа `Collection`. На использование обобщенного типа указывают угловые скобки, окружающие имя типа. В этом случае я объявил коллекцию целых чисел, коллекцию строк и, чтобы продемонстрировать более сложное применение обобщений — коллекцию коллекций целых чисел. К тому же обратите внимание на объявление обобщенного интерфейса, а именно — `IList<>`.

Когда вы специфицируете аргумент-тип для обобщенного типа, указывая его в фигурных скобках, как в случае `Collection<int>`, вы тем самым объявляете закрытый тип. В этом случае `Collection<int>` принимает только один параметр типа, но если бы он принимал больше, то их следовало бы разделить запятыми. Когда CLR впервые встречает такое объявление типа, то генерирует закрытый тип на основе данного обобщенного типа и применяет аргументы типа. Использование закрытого типа, сформированного подобным образом, ничем не отличается от использования любого другого типа, за исключением того, что в объявлении типа применяется специальный синтаксис угловых скобок для формирования закрытого типа.

Теперь, познакомившись в общих чертах с тем, как выглядят обобщения, вы должны подготовиться к восприятию эпизодических применений обобщений, которые будут встречаться до главы 11.

Пространства имен

Язык C#, подобно C++ и аналогам — пакетам Java, поддерживает пространства имен для логического группирования компонентов. Пространства имен помогают избежать конфликтов имен между вашими идентификаторами.

Используя пространства имен, вы можете определять все свои типы так, чтобы их идентификаторы квалифицировались пространством имен, к которому они принадлежат. Вы уже видели пространства имен в действии во многих примерах, приведенных до сих пор. Так, в примере "Hello World!" из главы 1 вы видели использование класса `Console`, который находится в пространстве имен `System` библиотеки классов .NET Framework, чье полное квалифицированное имя выглядит как `System.Console`. Вы можете создавать собственные пространства имен для организации компонентов. Общая рекомендация — используйте при этом идентификатор некоторого рода, например, наименование вашей организации, в качестве пространства имен верхнего уровня, и затем более специфичные идентификаторы библиотек в качестве вложенных пространств имен.

Пространства имен обеспечивают отличный механизм, с помощью которого вы можете сделать ваши типы более осмысленными, особенно, если вы разрабатываете библиотеки, предназначенные для использования другими людьми. Например, вы можете создать общее пространство имен, такое как `MyCompany.Widgets`, куда вы поместите наиболее широко применяемые типы виджетов (графических элементов управления). Затем вы можете создать пространство имен `MyCompany.Widgets.Advanced`, куда поместить менее часто используемые, но более сложные типы. Конечно, вы можете поместить их все в одно пространство имен. Однако пользователи могут запутаться, просматривая типы и обнаружив, что редко применяемые типы перемешаны с часто используемыми.

На заметку! При выборе названия для вашего пространства имен общая рекомендация состоит в том, чтобы это название следовало формуле `<ИмяКомпании>.<Технология>.*`, т.е., чтобы первые две части имени, разделенные точкой, начинались с названия вашей компании, за которым следует название технологии. Тогда вы сможете развивать классификацию пространств имен дальше. Примеры этого можно найти в .NET Framework — там есть, скажем, пространство имен `Microsoft.Win32`.

Определение пространств имен

Синтаксис объявления пространства имен прост. В следующем коде демонстрируется объявление пространства имен `Acme`:

```
namespace Acme
{
    class Utility {}
}
```

Пространства имен не обязательно должны ограничиваться единственной единицей компиляции (т.е. файлом исходного кода C#). Другими словами, одно и то же объявление пространства имен может существовать во многих файлах C#. Когда все скомпилировано, набор идентификаторов, включенных в пространство имен, является объединением всех идентификаторов в каждом из объявлений этого пространства имен. Фактически это объединение пересекает границы сборок. Если множество сборок содержат типы, определенные в одном и том же пространстве имен, то общее пространство имен состоит из всех идентификаторов во всех сборках, определяющих эти типы.

Объявления пространств имен можно вкладывать друг в друга. Вы можете делать это одним из двух способов. Первый способ очевиден:

```
namespace Acme
{
    namespace Utilities
    {
            class SomeUtility {}
    }
}
```

Имея такой пример, для того, чтобы обратиться к классу `SomeUtility`, используя его полное квалифицированное имя, вы должны идентифицировать его, как `Acme.Utilities.SomeUtility`. Следующий пример демонстрирует альтернативный способ определения вложенных пространств имен:

```
namespace Acme
{
}
namespace Acme.Utilities
{
    class SomeUtility {}
}
```

Эффект от этого кода будет точно таким же, как от предыдущего. Фактически вы можете пропустить первое пустое объявление пространства имен `Acme`. Я ос-

тавил ее только в демонстрационных целях, чтобы указать, что объявление пространства Utilities не является физически вложенным в объявление пространства имен Acme.

Любые типы, которые вы объявляете вне пространства имен, становятся частью глобального пространства имен.

На заметку! Вы всегда должны избегать определения типов в глобальном пространстве имен. Такая практика известна как "засорение глобального пространства имен" и считается дурным тоном в программировании. Это должно быть очевидным, поскольку в этом случае нет возможности защитить типы, определенные в разных местах, от потенциальных коллизий имен.

Использование пространств имен

В примере "Hello World!" из главы 1 я слегка коснулся опций, доступных при использовании пространств имен. Давайте рассмотрим некоторый код, использующий класс SomeUtility, определенный в предыдущем разделе:

```
public class EntryPoint
{
    static void Main() {
            Acme.Utilities.SomeUtility util =
            new Acme.Utilities.SomeUtility();
    }
}
```

Такая практика — всегда полностью квалифицировать имена — довольно многословна и в конечном итоге может привести к тяжелому случаю заболевания кистевым туннельным синдромом. Директива using пространств имен позволяет избежать этого. Она сообщает компилятору, что вы используете все пространство имен единицы компиляции или другое пространство имен. Что именно делает ключевое слово using — так это эффективно импортирует все имена из заданного пространства имен в окружающее пространство имен, которым может быть глобальное пространство данной единицы компиляции. Это демонстрируется в следующем примере:

```
using Acme.Utilities;
public class EntryPoint
{
    static void Main() {
            SomeUtility util = new SomeUtility();
    }
}
```

Теперь иметь дело с кодом намного легче, и в некотором отношении его легче читать. Директива using, находясь на уровне глобального пространства имен, импортирует имена типов из Acme.Utilities в глобальное пространство. Иногда, когда вы импортируете имена из нескольких пространств имен, могут возникать конфликты имен в импортированных пространствах, содержащих одноименные типы. В таком случае вы можете импортировать индивидуальные типы из пространства, создавая псевдонимы имен. Такая техника доступна благодаря средству псевдонимов в C#. Давайте модифицируем использование класса SomeUtility та-

ким образом, чтобы снабдить псевдонимом только этот класс, а не все содержимое пространства имен Acme.Utilities:

```
namespace Acme.Utilities
{
    class AnotherUtility() {}
}
using SomeUtility = Acme.Utilities.SomeUtility;
public class EntryPoint
{
    static void Main() {
        SomeUtility util = new SomeUtility();
        Acme.Utilities.AnotherUtility =
           new Acme.Utilities.AnotherUtility();
    }
}
```

В этом коде идентификатор SomeUtility является псевдонимом Acme.Utilities.SomeUtility. Чтобы проиллюстрировать идею, я пополнил пространство имен Acme.Utilities, добавив новый класс по имени AnotherUtility. Этот класс должен быть полностью квалифицирован при обращении к нему, поскольку для него никакого псевдонима не объявлено.

Кстати, совершенно допустимо дать совершенно другое имя псевдониму, отличающееся от SomeUtility.

Хотя присвоение псевдониму другого имени может быть полезно для предотвращения конфликтов имен, все же лучше использовать псевдоним, совпадающий с исходным именем класса, чтобы избежать в будущем путаницы при сопровождении.

На заметку! Если вы следуете четким принципам разделения при определении ваших пространств имен, то вы не столкнетесь с этой проблемой. Считается дурным тоном при проектировании создавать пространства имен, содержащие множество разнообразных типов, относящихся к различным группам функциональности. Вместо этого вы должны создавать пространства имен с интуитивно взаимосвязанными типами внутри. Фактически в .NET Framework вы не раз встретите пространства имен с некоторыми общими типами, включенными в них, и с более "развитыми" типами, включенными во вложенное пространство имен под названием Advanced. Во многих отношениях при создании библиотек такие рекомендации отражают принцип открытости, применяемый при создании интуитивных пользовательских интерфейсов. Другими словами, именование и группировка ваших типов интуитивно понятны и делают их легко постижимыми.

Поток управления

Подобно C, C++ и Java, язык C# включает полный обычный набор структур, управляющих потоком управления. В C# реализован даже такой непопулярный оператор, как goto.

if-else, while, do-while и for

Конструкция if-else в C# идентична той же конструкции в C++ и Java. В качестве стилистической рекомендации: я предпочитаю всегда использовать блоки в операторах if, как и в любых других операторах управления потоком, как показано в следующих разделах, даже если эти блоки содержат всего один оператор, как в примере ниже:

```
if( <test condition> ) {
    Console.WriteLine( "Выполнение находится в этой точке." );
}
```

Операторы while, do и for идентичны используемым в C++ и Java.

switch

Синтаксис оператора switch в C# очень похож на синтаксис switch в C++ и Java. Основное отличие состоит в том, что switch в C# не допускает прохода в следующий раздел. Он требует наличия оператора break (или другой передачи управления) в конце каждого раздела. Я считаю, что это — отличное требование. В C++ и Java присутствует масс трудно находимых ошибок, являющихся следствием того, что разработчики забывают вставить оператор break, или меняют порядок разделов внутри switch, когда один из них позволяет проходить потоку управления в другой. В C# компилятор немедленно сообщит об ошибке, если обнаружит проход из одного раздела в следующий. Единственное исключение из этого правила состоит в том, что вы можете иметь несколько меток (используя ключевое слово case) на один раздел switch, как показано в следующем фрагменте кода. Вы также можете эмулировать прохождение сквозь раздел оператором goto:

```
switch( k ) {
    case 0:
          Console.WriteLine( "case 0" );
          goto case 1;
    case 1:
    case 2:
          Console.WriteLine( "case 1 or 2" );
          break;
}
```

Обратите внимание, что каждый раздел имеет форму оператора перехода, завершающий его. Даже последний раздел должен содержать его. Многие разработчики C++ и Java пропускают оператор break в последнем разделе, поскольку при этом все равно просто происходит выход потока управления за пределы switch. Опять же красота принципа "никаких проходов" проявляется в том, что даже если разработчик, сопровождающий код, впоследствии решит поменять местами порядок меток, он не внесет никаких ошибок, в отличие от C++ и Java. Обычно вы используете оператор break для завершения разделов switch, но можете применять любой оператор для выхода из этого раздела. Сюда входят throw и return, наряду с continue, если switch встроен в цикл, где применение continue имеет смысл.

foreach

Оператор foreach позволяет выполнять итерацию по коллекции объектов в синтаксически естественной манере. Следует отметить, что ту же функциональность можно реализовать с использованием цикла while. Однако это может выглядеть некрасиво, к тому же итерация по элементам коллекции — настолько распространенная задача, что синтаксис foreach — полезное дополнение к языку. Если, к примеру, у вас есть массив (или коллекция любого другого типа) строк, вы можете пройти по всем строкам, используя следующий код:

```
static void Main() {
    string[] strings = new string[5];
    strings[0] = "Bob";
    strings[1] = "Joe";
    foreach( string item in strings ) {
          Console.WriteLine( "{0}", item );
    }
}
```

Внутри скобок цикла foreach вы объявляете тип переменной итератора. В данном примере это строка. За объявлением типа итератора идет идентификатор коллекции, по которой нужно выполнить итерацию. Вы можете использовать любой объект, реализующий шаблон Collection. В главе 9 раскрывается тему коллекций во всех подробностях, включая вопрос о том, какие вещи должен реализовать тип, чтобы рассматриваться в качестве коллекции. Естественно, элементы внутри коллекции, используемые в операторе foreach, должны быть конвертируемы с использованием явного преобразования, к типу итератора. Если это не так, то оператор foreach сгенерирует исключение InvalidCastException во время выполнения. Если вы хотите испытать это на собственном опыте, попробуйте запустить следующую модификацию предыдущего примера:

```
static void Main() {
    object[] strings = new object[5];
    strings[0] = 1;
    strings[1] = 2;
    foreach( string item in strings ) {
          Console.WriteLine( "{0}", item );
    }
}
```

Обратите внимание, однако, что для кода, вставленного в оператор foreach, вообще недопустимо модифицировать переменную итератора. Она должна трактоваться как доступная только для чтения. Это означает, что вы не можете передать переменную итератора в качестве выходного (out) или ссылочного (ref) параметра в метод. Если вы попытаетесь сделать что-либо подобное, то компилятор немедленно сообщит вам об ошибке.

break, continue, goto, return и throw

Язык C# включает набор знакомых операторов, которые безусловно передают управление в другое место кода. Сюда входят break, continue, goto, return

и `throw`. Их синтаксис должен быть знаком любому разработчику C++ или Java (хотя в Java нет оператора `goto`). Их применение, по сути, идентично во всех трех языках.

Резюме

В настоящей главе был представлен синтаксис C#, при этом подчеркивалось, что C#, подобно другим похожим объектно-ориентированным языкам, является строго типизированным языком. Для этих языков вы используете механизм проверки типов компилятора, чтобы найти как можно больше ошибок во время компиляции, вместо того, чтобы находить их позднее, во время выполнения. В CLR все типы классифицируются как типы значений или ссылочные типы, и каждой категории присущи собственные ограничения, которые я буду подчеркивать на протяжении всей книги. Также здесь я представил пространства имен и показал, как они помогают избежать засорения глобального пространства имен слишком большим количеством типов, чьи имена могут вступать в конфликты. И, наконец, я показал, как работают управляющие операторы в C#, в чем сходство и отличие их работы в C++ и Java.

В следующей главе мы погрузимся глубже в устройство классов и структур, и высветим отличия в поведении их экземпляров.

ГЛАВА 4

Классы, структуры и объекты

Все, что угодно, является объектом! По крайней мере, если смотреть с точки зрения CLR и языка программирования C#. Это не удивительно, потому что C# — в конце концов, объектно-ориентированный язык. Создаваемые вами через определение классов объекты C# обладают теми же возможностями, что и другие предопределенные в системе объекты. Фактически, ключевые слова языка C# вроде `int` и `bool` — это просто псевдонимы для предопределенных типов значений из пространства имен `System`, в данном случае — `System.Int32` и `System.Boolean` соответственно.

На заметку! Настоящая глава довольно длинна, но пусть это вам не пугает. Для того чтобы удовлетворить запросы широкой аудитории, я постарался осветить насколько возможно много базового материала по C#. Если вы — профессионал в области C++ или Java, то можете пропустить эту главу и обращаться к ней по мере чтения последующих глав. Некоторые из затронутых здесь тем более подробно раскрываются в последующих главах.

Первый раздел этой главы посвящен определению классов, а за ним следует дискуссия об определениях типов значений. Это две фундаментальных группы типов в исполняющей системе .NET. Затем вы узнаете о `System.Object` (базовом типе для всех прочих типов), изучите нюансы создания и уничтожения экземпляров объектов, выражения инициализации объектов, также тему упаковки и распаковки. Анонимные типы являются новым средством в C# 3.0, и я решил посвятить им отдельный раздел. И, наконец, я расскажу о наследовании и полиморфизме, а также раскрою отличие между наследованием и включением с точки зрения повторного использования кода.

В возможности создания ваших собственных типов заключается мощь объектно-ориентированных систем. Замечательно, что даже несмотря на то, что встроенные типы языка являются простыми старыми объектами CLR, создаваемые вами объекты обращаются со встроенными типами на уровне полей. Другими словами, встроенные типы не обладают никакими специальными возможностями, которые вы не могли бы реализовать в типах, определяемых пользователем. Краеугольный камень создания этих типов — *определение класса*. Определения классов с использованием ключевого слова C# `class` определяет внутреннее состояние и поведение, ассоциируемое с объектами типа этого класса. Внутреннее состояние объекта представлено полями, которые вы объявляете в классе, среди которых могут быть

ссылки на другие объекты либо просто значения. Иногда, хотя и редко, вы можете услышать, что люди описывают "форму" (shape) объекта, поскольку определения полей экземпляра внутри класса формируют "отпечаток" объекта в куче.

Объекты, создаваемые на основе классов, инкапсулируют поля данных, представляющие внутреннее состояние объектов, и объекты могут тонко управлять доступом к этим полям. Поведение объектов определяется реализуемыми методами, которые вы объявляете и определяете внутри определения класса. Вызывая один из методов на экземпляре объекта, вы инициируете единицу работы этого объекта. Эта работа может модифицировать внутреннее состояние объекта, проверить состояние объекта либо сделать что-то другое в том же духе.

Вы можете определять конструкторы, которые система выполняет при каждом создании нового объекта. Вы можете также определить метод, называемый финализатором, работающий в момент уборки объекта сборщиком мусора. Как будет показано в главе 13, финализаторов следует по возможности избегать. Эта глава подробно описывает конструирование и уничтожение, включая детальное описание последовательности событий, происходящих во время создания объекта.

Объекты поддерживают концепцию *наследования*, посредством которой *производный класс* наследует поля и методы *базового класса*. Наследование также позволяет трактовать объекты производного класса как объекты его базового типа. Например, дизайн, в котором объект типа `Dog` унаследован от типа `Animal`, говорит о том, что это модель отношения "является" (is-a) (т.е. собака (`Dog`) является животным (`Animal`)). Таким образом, вы можете просто неявно конвертировать ссылку на тип `Dog` в ссылку на тип `Animal`. Здесь *"неявно"* означает, что преобразование имеет форму простого выражения присваивания. В противоположность этому, вы можете *явно* конвертировать ссылку на тип `Animal` через операцию приведения, чтобы она ссылалась на тип `Dog`, если определенный объект, на который ссылается тип `Animal`, фактически является объектом, созданным на основе класса `Dog`. Эта концепция, называемая *полиморфизмом* — когда вы можете манипулировать объектами взаимосвязанных типов, как если они относятся к одному общему типу — должна быть вам знакомой. Компьютерные фанаты всегда пытаются придумать забавные и мудреные слова для обозначения вещей, подобных этой, и *полиморфизм* — не исключение, хотя он означает всего-навсего то, что объект может принимать несколько идентичностей типов. В этой главе рассматривается наследование, со всеми его подводными камнями.

CLR отслеживает ссылки на объекты. Это значит, что каждая переменная *ссылочного типа* в действительности содержит ссылку на объект в куче (или `null`, если в данный момент не ссылается ни на какой объект). Когда вы копируете значение переменной ссылочного типа в другую переменную ссылочного типа, то создается другая ссылка на тот же самый объект — другими словами, копируется ссылка. Таким образом, вы получаете две переменных, ссылающихся на один объект. В CLR для создания полноценных копий объектов нужно выполнить дополнительную работу, например, реализовать интерфейс `ICloneable` или подобный ему шаблон.

Все объекты, созданные на основе определения класса C#, располагаются в системной куче, которой управляет сборщик мусора (Garbage Collector — GC) CLR. GC избавляет вас от необходимости очистки памяти, занятой вашим объектом. Вы можете выделять память для новых объектов хоть целый день, не заботясь о том, кто освободит потом эту память, ассоциированную с ними.

GC достаточно интеллектуален, чтобы отслеживать ссылки на объекты, и когда он обнаруживает, что на объект не осталось никаких ссылок, он помечает его для удаления. Затем, когда в следующий раз GC уплотняет кучу, он уничтожает такой объект и освобождает занятую им память.

На заметку! В действительности процесс намного сложнее описанного. Существует много скрытых нюансов в том, как GC освобождает память неиспользуемых объектов. Я расскажу об этом в разделе "Уничтожение объектов" далее в этой главе. Имейте в виду, что GC сокращает сложность в этой области, но привносит массу новых сложностей в другие.

Наряду с классами, язык C# также поддерживает определение новых *типов значений* через ключевое слово `struct`. Типы значений — это легковесные объекты, которые обычно не находятся в куче, а вместо этого располагаются в стеке. Чтобы быть совершенно точным, следует сказать, что тип значений может находиться в куче, но только в том случае, если он являются полем внутри объекта, находящегося в куче. Типы значений не могут быть определены так, чтобы наследоваться от другого класса или другого типа значений, как и никакой другой класс или тип значений не может наследоваться от них.

Типы значений могут иметь конструкторы, но не могут иметь финализатора. По умолчанию, когда вы передаете тип значения в метод в качестве параметра, то метод принимает копию этого значения. В настоящей главе и в главе 13 я раскрою множество подробностей о типах значений, наряду с их отличиями от ссылочных типов.

А теперь давайте обратимся к подробностям. Не бойтесь, если они покажутся поначалу несколько громоздкими. Дело в том, что вы можете начать разрабатывать полезные приложения на C#, даже не зная всех тонкостей поведения языка. Это хорошо, потому что C# вместе с IDE-средой Visual Studio предназначен облегчить быструю разработку приложений. Однако чем больше подробностей вы знаете о языке и CLR, тем более эффективными и устойчивыми будут ваши приложения на C#.

Определения классов

Определения классов в C# похожи на определения классов в C++ и Java. Рассмотрим простейший класс, чтобы вы получили представление об этом. В следующем коде показаны базовые части, из которых состоит определение класса.

```
// ПРИМЕЧАНИЕ: Этот код не предназначен для компиляции "как есть"
[Serializable]
public class Derived : Base, ICloneable
{
    private Derived( Derived other ) {
            this.x = other.x;
    }
    public object Clone() { // реализует интерфейс IClonable.Clone
            return new Derived( this );
    }
    private int x;
}
```

Это объявление класса определяет класс `Derived`, унаследованный от класса `Base` и также реализующий интерфейс `ICloneable`.

На заметку! Если вы впервые сталкиваетесь с концепцией интерфейса, не беспокойтесь. Глава 5 полностью посвящена теме интерфейсов и программирования на основе контрактов.

Модификатор доступа перед ключевым словом `class` управляет видимостью типа извне сборки (о сборках рассказывалось в главе 2). Класс `Derived` является общедоступным, а это означает, что пользователи сборки, содержащей этот класс, могут создавать его экземпляры. Этот тип содержит приватный (`private`) конструктор, используемый общедоступным (`public`) методом `Clone`, который реализует интерфейс `ICloneable`. Когда класс реализует интерфейс, это значит, что вы должны реализовать все методы этого интерфейса.

Почти к любой именованной сущности внутри системы типов CLR можно применять *атрибуты*. В данном случае я применил атрибут `Serializable` к классу, чтобы показать пример использования синтаксиса атрибутов. Эти атрибуты становятся частью метаданных, описывающих тип его потребителям. Вдобавок вы можете создавать собственные атрибуты для прикрепления их к различным сущностям, таким как классы, параметры, возвращаемые значения и поля, что позволяет легко реализовать возможности аспектно-ориентированного программирования (АОР).

Поля

Поля (field) — это "хлеб и масло", представляющие состояние объектов. Обычно вы объявляете новый класс только тогда, когда вам нужно смоделировать новый тип объекта, с его собственным внутренним состоянием, представленным его полями экземпляра.

Вы объявляете поля с типом, подобно тому, как это делается со всеми прочими переменными в С. Ниже перечислены допустимые модификаторы полей:

```
new
public
protected
internal
private
static
readonly
volatile
```

Многие из них являются взаимно исключающими. Взаимоисключающие модификаторы управляют доступностью поля и к ним относятся `public`, `protected`, `internal` и `private`. Я расскажу о них подробнее в разделе "Доступность". А пока рассмотрим остальные модификаторы.

Модификатор `static` управляет тем, является поле членом типа или членом объектов, созданных из этого типа. В отсутствии модификатора `static` поле является полем *экземпляра*, и потому каждый объект, созданный из этого класса, получает свою собственную копию этого поля. Так принято по умолчанию. Когда же поле снабжено модификатором `static`, то одно поле разделяется всеми экземплярами класса в пределах домена приложения.

Обратите внимание, что статические поля не включаются в "отпечаток памяти" экземпляров объекта. Другими словами, объекты не инкапсулируют статических полей; вместо этого статические поля инкапсулируют типы. Было бы неэффективно, чтобы все экземпляры объекта содержали копию одной и той же статической переменной в своем отпечатке памяти. Хуже того, компилятору тогда пришлось бы генерировать некоторый скрытый код, чтобы гарантировать синхронизацию изменений статического поля в одном экземпляре с этим полем во всех других экземплярах. По этой причине статические поля в действительности относятся к классу, а не к экземплярам объектов. Фактически, когда статическое поле доступно вне класса, вы используете имя класса, а не экземпляра объекта, чтобы обратиться к этому полю.

На заметку! Статические поля обладают еще одним важным качеством: они глобальны по отношению к домену приложения, внутри которого загружены содержащие их типы. Домен приложения — это абстракция, подобная абстракции процесса внутри операционной системы, но это — более легковесный механизм. Вы можете иметь множество доменов приложения в одном процессе операционной системы. Если процесс вашей CLR содержит несколько доменов приложений, каждый из них имеет собственную копию статических полей класса. Значения статических полей одного домена приложения могут отличаться от значений тех же статических полей в другом домене приложения. Если только вы не создаете дополнительные домены приложения сами, то ваше приложение имеет только один такой домен — домен приложения по умолчанию. Однако важно отметить это отличие при работе в таких средах, как ASP.NET, где концепция домена приложения используется в качестве механизма изоляции между двумя приложениями ASP.NET. Фактически вы легко можете прийти к заключению, что ASP.NET было главной причиной создания концепции домена приложения.

Вы можете инициализировать поля во время создания объекта различными способами. Простейший способ сделать это — прибегнуть к помощи *инициализаторов*. Вы применяете эти инициализаторы в точке определения поля, и они могут быть использованы как для статических полей, так и для полей экземпляра, например:

```
private int x = 789;
private int y;
private int z = A.InitZ();
```

Поле x инициализируется инициализатором. Система обозначений достаточно удобна. Обратите внимание, что эта инициализация происходит во время выполнения, а не во время компиляции. Поэтому данный оператор инициализации может использовать и кое-что другое, помимо констант. Например, переменная z инициализируется вызовом метода `A.InitZ()`. Поначалу такое обозначение инициализации поля может показаться значительным сокращением, избавляющим вас от необходимости инициализировать все поля внутри тела конструктора. Однако все-таки я советую инициализировать поля экземпляра внутри тела конструктора экземпляра. Я расскажу о статических инициализациях и инициализациях экземпляра во всех подробностях далее, в разделе "Создание объектов" настоящей главы, и тогда вы убедитесь в том, что инициализация полей в конструкторе может облегчить создание кода, который проще сопровождать и отлаживать.

Другой модификатор полей, который время от времени оказывается весьма кстати — это `readonly`. Как и можно было ожидать, он позволяет определить поле, которое доступно только для чтения. Вы можете записывать в него только при создании объекта. То же поведение можно эмулировать с большей гибкостью, исполь-

зуя доступное только для чтения *свойство*, о чем пойдет речь в разделе "Свойства". Статические readonly-поля инициализируются в статическом конструкторе, в то время как readonly-поля экземпляра инициализируется в конструкторе экземпляра. Альтернативно вы можете инициализировать такие поля посредством инициализаторов в точке их объявления в классе, как вы делаете это с другими полями. Внутри конструктора вы можете присваивать значения полям readonly столько, сколько нужно. Только внутри конструктора вы можете передать другой функции поле readonly, как параметр ref или out. Рассмотрим пример:

```
public class A
{
    public A()
    {
          this.y = 456;
          // Можно даже еще раз установить y.
          this.y = 654;
          // Можно использовать y как параметр ref.
          SetField( ref this.y );
    }
    private void SetField( ref int val )
    {
          val = 888;
    }
    private readonly int x = 123;
    private readonly int y;
    public const int z = 555;
    static void Main()
    {
        A obj = new A();
        System.Console.WriteLine( "x = {0}, y = {1}, z = {2}",
                                   obj.x, obj.y, A.z );
    }
}
```

Здесь следует отметить один важный нюанс: поле z объявлено с ключевым словом const. Поначалу может показаться, что эффект от этого будет тем же, что и от readonly, но на самом деле это не так. Во-первых, поле const известно и используется во время компиляции. Это значит, что код, сгенерированный компилятором в процедуре Main, может быть оптимизирован заменой всех случаев использования этой переменной непосредственным константным значением. Компилятор вправе предпринять такой трюк для повышения производительности — просто потому, что значение данного поля известно на момент компиляции. К тому же заметьте, что доступ к полю const осуществляется с указанием имени класса, а не имени экземпляра. Это потому, что значения const неявно статические и не влияют на отпечаток памяти или форму экземпляров объекта. Опять же это имеет смысл, поскольку компилятор оптимизирует доступ к участку памяти в экземпляре объекта, поскольку это поле будет все равно одинаковым у всех экземпляров объекта.

Но здесь наблюдается еще одна деталь, касающаяся отличий между полями readonly и const. Поля readonly гарантированно вычисляются во время выполнения. Поэтому предположим, что у вас есть один класс с полем readonly и полем

const, который находится в сборке A, а код из сборки B создает и использует экземпляр класса из сборки A. Теперь представим, что позднее вы перестроили сборку A и модифицировали инициализаторы полей readonly и const. Потребитель из сборки B увидит изменения в поле const только после перекомпиляции кода сборки B. Этого поведение следовало ожидать, поскольку когда сборка B была построена, ссылаясь на начальную версию сборки A, то компилятор оптимизировал использование значений const, подставив литеральное значение в сгенерированный код IL. По этой причине вы должны быть осторожными, когда принимаете решение об использовании поля readonly или значения const, и если отдали предпочтение readonly, тщательно выбирать между полем readonly и доступным только для чтения свойством (свойства рассматриваются далее в разделе "Свойства"). Свойства обеспечивают более высокую гибкость во время проектирования и во время сопровождения, чем поля readonly.

И, наконец, модификатор volatile говорит о том, как следует из его имени, что поле чувствительно ко времени чтения и записи. Технически модификатор указывает компилятору на то, что поле может быть в любой момент доступно и модифицировано операционной системой или аппаратным обеспечением, работающим на этой системе, или, что более вероятно, другим потоком управления. Последний случай наиболее типичен. Обычно доступ к полю из многих потоков представляет собой проблему, только когда вы не применяете технологий синхронизации, таких как использование ключевого слова C# lock или объектов синхронизации операционной системы. Когда поле помечено как volatile, это говорит реализации, а следовательно и JIT-компилятору CLR, что он не должен применять оптимизацию доступа к этому полю. Поскольку доступ к полю из многих потоков без применения приемов синхронизации сомнителен и чреват ошибками, я не стану дальше тратить время на описание применений модификатора volatile. На самом деле он вообще редко нужен, и вы редко будете сталкиваться с ним, если только не станете разрабатывать какой-то необычный способ взаимодействия с устройством или чем-то подобным.

Я уже рассказывал о некоторых способах инициализации полей, которые могут происходить в экземпляре во время инициализации класса. О дополнительных нюансах инициализации полей я еще расскажу в разделе "Инициализация полей". Однако обратите внимание, что C# имеет определенные правила инициализации полей по умолчанию, которые применяются перед выполнением кода инициализации, встречающегося в блоке кода методов конструкторов. По умолчанию C# создает верифицируемый безопасный к типам код, который гарантированно не использует неинициализированные переменные и поля. Компилятор очень тщательно следит за выполнением этого требования. Так, например, он инициализирует все поля — будь то поля экземпляра или статические — значениями по умолчанию, причем это делается до запуска любых ваших инициализаторов. Значение по умолчанию почти для чего угодно может быть представлено либо 0, либо null. Например, вы можете инициализировать целое число или любой другой подобный тип значений установкой всех битов в области его хранения в 0. Для ссылочных типов начальное значение по умолчанию устанавливается в null. Опять-таки обычно это является результатом того, что реализация устанавливает все биты ссылки в 0. Такая инициализация по умолчанию происходит перед выполнением любого кода на экземпляре класса. Таким образом, никак невозможно опросить неинициализированные значения объекта или класса во время начального конструирования.

Конструкторы

Конструкторы вызываются при первоначальной загрузке класса CLR или создании объекта. Существуют два типа конструкторов: статические конструкторы и конструкторы экземпляра. Класс может иметь только один статический конструктор, не имеющий параметров. Имя статического конструктора должно совпадать с именем класса, к которому он принадлежит. Как и к любому другому члену класса, к статическому конструктору вы можете присоединить атрибуты метаданных.

Конструкторы экземпляров, с другой стороны, вызываются при создании экземпляра класса. Обычно они устанавливают состояние объекта посредством инициализации полей в желательное предопределенное состояние. Вы можете также выполнить любую другую работу по инициализации, такую как подключение к базе данных и открытие файла. У класса может быть несколько конструкторов экземпляра, которые могут быть *перегружены* (т.е. иметь разные типы параметров). Как и статический конструктор, конструкторы экземпляра именуются по названию определяющего их класса. Одна заслуживающая упоминания особенность конструкторов экземпляра — необязательное выражение инициализатора. Применяя инициализатор, который следует за двоеточием после списка параметров, вы можете вызвать конструктор базового класса или другой конструктор того же класса через ключевые слова `base` и `this` соответственно. О ключевом слове `base` еще будет говориться в разделе "Ключевое слово `base`" далее в главе. Рассмотрим следующий пример кода с двумя комментариями:

```
class Base
{
    public int x = InitX();
    public Base( int x )
    {
        this.x = x; // устраняется неоднозначность
                       "параметр/переменная экземпляра"
    }
}
class Derived : Base
{
    public Derived( int a )
        :base( a ) // вызов конструктора базового класса
    {
    }
}
```

Методы

Метод определяет процедуру, которую вы можете выполнить над объектом или классом. Если метод является *методом экземпляра*, вы можете вызывать его на объекте. Если же метод *статический*, вы можете вызывать его только на классе. Отличие между ними в том, что метод экземпляра имеет доступ к полям экземпляра объекта, в то время как статический метод не имеет доступа к полям и методам экземпляра. Статические методы могут иметь доступ только к статическим членам класса.

Методы могут иметь атрибуты метаданных, присоединенные к ним, и также могут иметь необязательные модификаторы. Я буду говорить о них на протяжении всей главы. Эти модификаторы контролируют доступность методов, а также их участие в наследовании. Каждый метод либо имеет, либо не имеет типа возврата. Если метод не имеет типа возврата, то в его объявлении в качестве типа возврата должно указываться void. Методы могут иметь или не иметь параметров.

Статические методы

Вы вызываете статические методы на классе в целом, а не на экземплярах этого класса. Статические методы имеют доступ только к статическим членам класса. Вы объявляете статические методы посредством применения модификатора static, как показано в следующем примере:

```
public class A
{
    public static void SomeFunction()
    {
        System.Console.WriteLine( "SomeFunction() called" );
    }
    static void Main()
    {
        A.SomeFunction();
        SomeFunction();
    }
}
```

Обратите внимание, что оба метода в этом примере являются статическими. В методе Main я сначала обращаюсь к методу SomeFunction с использованием имени класса. Затем вызываю статический метод, не квалифицируя его. Это потому, что методы Main и SomeFunction определены в одном и том же классе и оба являются статическими. Если бы метод SomeFunction относился к другому классу, скажем, к классу B, мне пришлось бы ссылаться на этот метод как B.SomeFunction.

Методы экземпляров

Методы экземпляров работают с объектами. Для того чтобы вызвать метод экземпляра, вы должны сослаться на экземпляр класса, определяющего этот метод. В следующем примере показано применение метода экземпляра:

```
public class A
{
    private void SomeOperation()
    {
        x = 1;
        this.y = 2;
        z = 3;
        // присваивание this в объектах является ошибкой.
        // A newinstance = new A();
        // this = newinstance;
    }
    private int x;
```

```
        private int y;
        private static int z;
        static void Main()
        {
            A obj = new A();
            obj.SomeOperation();
            System.Console.WriteLine( "x = {0}, y = {1}, z= {2}",
                                      obj.x, obj.y, A.z );
        }
    }
```

В методе Main вы можете видеть, что я создаю новый экземпляр класса A и затем вызываю метод SomeOperation через экземпляр этого класса. Внутри тела метода SomeOperation я имею доступ к полям экземпляра и статическим полям класса и могу присваивать им значения, просто используя их идентификаторы. Даже несмотря на то, что, как уже поминалось, метод SomeOperation может присваивать значение статическому члену z, не квалифицируя его, я считаю, что для лучшей читабельности кода следует квалифицировать статические поля, присваивая им значения, даже если метод относится к тому же классу. Поступив так, вы поможете тем, кто придет после вас, и кому придется сопровождать ваш код, даже если это будете вы сами!

Обратите внимание, что при присваивании значения полю y я снабжаю его префиксом — идентификатором this. Применяя this, вы можете обращаться к полям экземпляра, как это делал я, когда присваивал значение y в предыдущем примере кода. Поскольку значение this доступно только для чтения, вы не можете ничего присвоить ему такого, чтобы заставить ссылаться на другой экземпляр. Если вы попытаетесь сделать это, то компилятор сообщит об ошибке и не сможет откомпилировать ваш код.

Свойства

Свойства — один из самых замечательных механизмов C# и CLR, который позволяет вам достичь лучшей инкапсуляции. Если говорить коротко, вы используете свойства для ужесточения контроля доступа к внутреннему состоянию объекта.

С точки зрения клиента объекта свойство выглядит и ведет себя так же как общедоступное поле. Нотация доступа к свойству такая же, как при доступе к общедоступному полю экземпляра. Однако свойство не имеет никакого ассоциированного с ним места хранения в объекте, как это присуще полям. Вместо этого свойство представляет собой сокращенную нотацию для определения *средств доступа* (accessors) для чтения и записи полей. Типичный шаблон предусматривает обеспечение доступа к приватному полю класса через общедоступное свойство. C# 3.0 еще более облегчает эту задачу за счет поддержки автореализуемых (auto-implemented) свойств.

Свойства существенно расширяют ваши возможности как проектировщика класса. Например, если свойство представляет количество строк в объекте — таблице базы данных, то объект таблицы может отложить вычисление этого значения до тех пор, пока оно не будет опрошено свойством. Он узнает, когда следует вычислять значение, потому что клиент вызовет средство доступа, чтобы обратиться к свойству.

Объявление свойств

Синтаксис объявления свойств прост. Как и большинство членов класса, свойства допускают присоединение к ним атрибутов метаданных. Некоторые модификаторы, применимые к свойствам, подобны модификаторам методов. Другие модификаторы включают способность объявлять свойство как virtual, sealed, override, abstract и т.д. Все это я раскрою в подробностях далее в разделе "Наследование и виртуальные методы" настоящей главы.

Следующий код определяет свойство Temperature в классе A:

```
public class A
{
    private int temperature;
    public int Temperature
    {
        get
        {
           System.Console.WriteLine( "Извлечение значения temperature" );
           return temperature;
        }
        set
        {
           System.Console.WriteLine( "Установка значения temperature" );
           temperature = value;
        }
    }
}
public class MainClass
{
    static void Main()
    {
        A obj = new A();
        obj.Temperature = 1;
        System.Console.WriteLine( "obj.Temperature = {0}",
                                  obj.Temperature );
    }
}
```

Сначала я определяю свойство по имени Temperature, имеющее тип int. Каждое объявление свойства должно определять тип, представляемый этим свойством. Этот тип должен быть видимым компилятору в точке, где свойство объявлено в классе, и он должен иметь как минимум, ту же видимость, что и определяемое свойство. То есть я имею в виду, что если свойство общедоступное (public), то тип значения, представленного свойством, также должен быть объявлен как public в сборке, в которой он определен. В данном примере тип int — это псевдоним для System.Int32. Этот класс определен в пространстве имен System, и он объявлен как public. Поэтому вы можете использовать int в качестве типа свойства в этом общедоступном классе A.

Также вы можете присвоить этому свойству имя Temperature. Это имя, по которому клиенты будут ссылаться на свойство, как если бы оно было полем. В данном примере я просто возвращаю приватное поле temperature из внутреннего состояния экземпляра объекта. Это — общепринятое соглашение. Вы называете приватное поле именем, совпадающим с именем свойства, но начиная с прописной бук-

вы, в то время как имя свойства начинается с заглавной. Конечно, вы не обязаны следовать этому соглашению, но нет причин от него отказываться, к тому же другие программисты C# ожидают этого от вас.

Средства доступа

В предыдущем примере вы можете видеть, что внутри блока свойства находится еще два блока кода. Это — средства доступа (accessors) к свойству, и внутри их блоков вы помещаете код, который читает и записывет значение свойства. Как видите, один из них называется `get`, а другой — `set`. Из их названий очевидно, какой за что отвечает.

Блок `get` вызывается, когда клиент объекта читает свойство. Как и можно было ожидать, это средство доступа должно возвращать значение или ссылку на объект, соответствующий типу объявления свойства. Также он возвращает объект, неявно конвертируемый в объявленный тип свойства. Например, если тип свойства — `long`, и `get` возвращает `int`, то `int` будет неявно преобразован к `long` без потери точности. В противном случае код в этом блоке подобен параметризованному методу, возвращающему значение или ссылку на тип свойства.

Блок `set` вызывается, когда клиент пытается записывать свойство. Обратите внимание, что возвращаемого значения здесь нет. Также обратите внимание на специальное значение по имени `value`, которое доступно коду внутри этого блока, и которое имеет тот же тип, что объявлен в качестве типа свойства. Когда вы записываете значение в свойство, то переменная `value` устанавливается в значение или ссылку на объект, которые клиент пытается присвоить свойству. Если вы попытаетесь объявить локальную переменную по имени `value` в блоке `set`, то получите ошибку компиляции. Средство доступа `set` подобно методу, принимающему один параметр того же типа, что у свойства, и возвращающего `void`.

Свойства, доступные только для чтения и только для записи

Если вы определяете свойство только со средством доступа `get`, то такое свойство будет доступно только для чтения. Аналогично если вы определяете свойство только со средством доступа `set`, то получите свойство, доступное только для записи. И, наконец, свойство, имеющее оба средства доступа, допускает и чтение, и запись.

Вы можете спросить, чем свойство, доступное только для чтения, лучше обычного общедоступного поля с модификатором `readonly`? На первый взгляд может показаться, что доступное лишь для чтения свойство менее эффективно, чем доступное для чтения общедоступное свойство. Однако, учитывая тот факт, что CLR может встраивать (inline) код для доступа к свойству в том случае, когда свойство просто возвращает поле, этот аргумент о неэффективности отпадает. Теперь, конечно, запись кода не так эффективна. Однако поскольку программисты не так уж ленивы, а автореализуемые свойства C# 3.0 очень облегчают эту задачу, этот аргумент также не слишком убедителен.

Фактически в 99% случаев свойство, доступное только для записи, более гибко, чем общедоступное `readonly`-поле. Одна причина состоит в том, что вы можете отложить вычисление свойства, доступного только для чтения, до того момента, когда оно вам понадобится (прием, известный как *"ленивое" вычисление* или *отложенное выполнение*). Поэтому в действительности это может привести к более эффективному коду, когда свойство предназначено для представления чего-то такого, на вы-

числение чего требуется существенное время. Если вы используете для этой цели общедоступное readonly-поле, вам придется выполнить вычисление в блоке конструктора. Все необходимые для вычисления свойства к этому моменту могут быть еще не готовы. Или же вы можете потратить время на вычисление значения в конструкторе, хотя пользователь объекта, возможно, никогда и не обратится к нему.

К тому же свойства, доступные только для чтения, помогают усилить инкапсуляцию. Если перед вами изначально стоит выбор между общедоступным полем только для чтения и доступным только для чтения свойством, то вы можете обеспечить большую гибкость будущих версий класса в отношении выполнения дополнительной работы в точке доступа свойства без какого-либо влияния на клиента. Например, предположим, что вы хотите выполнить некоторое протоколирование в отладочной сборке при каждом обращении к свойству. Клиент сможет неявно вызывать специальные методы свойства при обращении к данным. Гибкость, которой можно достичь подобным образом, практически не ограничена. Но если вы обращаетесь к значению, представленному в общедоступном readonly-поле, вы не вызовете никаких методов и не сможете сделать ничего дополнительного без перехода от поля к свойству и перекомпиляции кода. Эта дискуссия приводит нас непосредственно к разговору об инкапсуляции в одном из последующих разделов, озаглавленном "Инкапсуляция".

Автореализуемые свойства

Очень часто у вас возникает потребность в типе — скажем, классе, содержащем несколько полей, объединенных в одну сущность. Например, представьте тип Employee, содержащий полное имя и идентификационный номер, но для примера, манипулирующий этими данными в виде строк, как показано ниже:

```
public class Employee
{
    string fullName;
    string id;
}
```

В том виде, как он написан, данный класс, по сути, бесполезен. Его два поля — приватные, и к ним нужно как-то открыть доступ. Для обеспечения инкапсуляции мы не хотим просто сделать эти поля общедоступными. Однако для такого простого маленького типа было бы утомительно кодировать свойства, как показано ниже:

```
public class Employee
{
    public string FullName {
           get { return fullName; }
           set { fullName = value; }
    }
    public string Id {
           get { return id; }
           set { id = value; }
    }
    string fullName;
    string id;
}
```

Как много кода — и только для того, чтобы получить тип с парой свойств!

На заметку! Готов поспорить, что очень многие разработчики просто избегают свойств и используют общедоступные поля в таких вспомогательных типах, причем только из-за нежелания набирать длинный код. Проблема такого упрощенного подхода заключается в том, что вы не можете выполнять никакой проверки достоверности при установке значений полей или выполнять какие-либо "ленивые" вычисления при обращении к полям.

К счастью, C# 3.0 обладает новым средством, называемым автореализуемые свойства (auto-implemented properties), которые существенно сокращают эту утомительную работу. Посмотрите, как изменится приведенный выше тип Employee после использования автореализуемых свойств:

```
public class Employee
{
    public string FullName { get; set; }
    public string Id { get; set; }
}
```

И все! На самом деле вы сообщаете компилятору вот что: "Мне нужно строковое свойство `FullName` с поддержкой для него `get` и `set`". "За кулисами" компилятор генерирует в классе приватное поле для хранения значения и реализует для вас средства доступа к нему. Красота этого решения заключается в том, что здесь требуется ненамного больше ручного ввода кода, чем для объявления простых общедоступных полей, но в то же время, поскольку речь идет о свойствах, вы можете изменять лежащую в их основе реализацию, не модифицируя общедоступного интерфейса данного типа. То есть, если вы решите позднее как-то настроить средство доступа к `Id`, то сможете сделать это, не требуя перекомпиляции клиентов `Employee`.

На заметку! Если вас интересует приватное поле, которое компилятор объявляет в вашем типе для автореализуемых свойств, вы всегда можете увидеть его с помощью ILDASM. В моей текущей реализации приватное поле, хранящее значение свойства `FullName` класса `Employee`, называется `<>k__AutomaticallyGeneratedPropertyField0` и имеет тип `string`. Обратите внимание, что имя поля "непроизносимое", т.е. вы не сможете ввести его в коде, не получив сообщение о синтаксической ошибке. Разработчики компилятора C# сделали это для того, чтобы никто не мог использовать имя этого поля напрямую. В конце концов, имя этого поля — деталь реализации компилятора, которая может измениться в будущем.

Вы можете также создать автореализуемое свойство, доступное только для чтения, вставив ключевое слово `private`, как показано ниже:

```
public class Employee
{
    public string FullName { get; private set; }
    public string Id { get; set; }
}
```

Здесь вы можете спросить, как же тогда вообще устанавливать значение поля `FullName`? В конце концов, это приватное поле, доступное только для чтения и представляющее внутреннее хранилище, имеет сгенерированное компилятором имя, которое мы не можем использовать в конструкторе для присваивания ему

значений. Решение заключается в использовании другого нового средства C# 3.0 под названием *инициализаторы объектов*, как показано в следующем примере:

```
using System;
public class Employee
{
    public string FullName { get; private set; }
    public string Id { get; set; }
}
public class AutoProps
{
    static void Main() {
        Employee emp = new Employee {
            FullName = "John Doe",
            Id = "111-11-1111"
        };
    }
}
```

Инициализаторы объектов дополняют автореализуемые свойства. Они не только предоставляют вам средства установки автореализуемых полей, доступных только для чтения, но также предлагают средство создания экземпляра, чей синтаксис подобен синтаксису, обычно присущему конструкторам, когда переданные конструктору значения во время создания экземпляра используются для соответствующей инициализации типа. Далее в настоящей главе, в разделе "Инициализаторы объектов" вы найдете дополнительную информацию по этой теме.

Инкапсуляция

Вероятно, одной из наиболее важных концепций объектно-ориентированного программирования является инкапсуляция. Инкапсуляция — это дисциплина тщательного контроля доступа к внутренним данным и процедурам объектов. Ни один язык, не поддерживающий инкапсуляцию, не может претендовать на звание объектно-ориентированного.

Вы всегда стараетесь следовать базовой концепции: никогда не определять поля данных ваших объектов с открытым доступом. Это просто. Однако вы удивитесь, узнав, как много программистов все еще объявляют свои поля данных общедоступными. Обычно это случается, когда определяется маленький служебный объект, и его разработчику лень должным образом обдумать его, либо же он просто спешит. Однако есть вещи, которых вы никогда не должны делать; о подобное "срезание углов" — одна из них.

Вы хотите, чтобы клиенты вашего объекта общались с ним только по контролируемым каналам. Обычно это означает контроль взаимодействия с вашим объектом через методы этого объекта (или свойства, которые, по сути, являются вызовами методов). Таким образом, вы обращаетесь с внутренностями объекта как с черным ящиком. Ничего из внутреннего хозяйства не видно внешнему миру, и все коммуникации, которые могут модифицировать эти внутренности, осуществляются по контролируемым каналам. Посредством инкапсуляции вы можете спроектировать такой дизайн, который гарантирует, что внутреннее состояние объекта никогда не будет повреждено.

Простой иллюстрацией к сказанному может быть следующий пример. Я создал фиктивный вспомогательный объект, представляющий прямоугольник. Сам этот пример — детская выдумка, но именно тем он и хорош в качестве аргумента, что обладает минимальной сложностью:

```
class MyRectangle
{
    public uint width;
    public uint height;
}
```

Здесь вы видите самый примитивный пример пользовательского класса прямоугольника. В данный момент меня интересует только ширина и длина прямоугольника. Конечно, полезный класс прямоугольника, предназначенный для использования в графическом механизме, также имеет и координаты начальной точки, но для простоты примера ограничимся шириной и длиной. Поэтому я объявил два поля — width и height — общедоступными. Возможно, я поступил так из-за спешки при проектировании этого базового маленького класса. Но как вы вскоре убедитесь, лишь немногим больше дополнительной работы обеспечили бы существенно большую гибкость.

Теперь предположим, что прошло время, и я использовал этот маленький класс во многих местах. Не стоит забывать, что этот маленький класс прямоугольника в том виде, как он есть, не слишком полезен, и потому представим, что я решил сделать его более полезным. Предположим, что у меня есть некоторый клиентский код, который использует этот класс прямоугольника и которому нужно вычислить площадь прямоугольника. Принципы объектно-ориентированного программирования склоняют меня к выводу, что лучший способ сделать это — позволить экземпляру MyRectangle самому сообщать клиенту о своей площади. Во времена ANSI C и других простых процедурных императивных языков вам нужно было бы создать функцию с именем вроде ComputeArea, которая принимала бы параметр — указатель на экземпляр MyRectangle. К счастью, те времена остались в прошлом, и я использую объектно-ориентированный подход. Сделаем вот как:

```
class MyRectangle
{
    public uint width;
    public uint height;
    public uint GetArea()
    {
        return width * height;
    }
}
```

Как видите, я добавил новый член — метод GetArea. При вызове на экземпляре, заслуживающий доверия MyRectangle вычислит площадь прямоугольника и вернет результат. Теперь у меня есть базовый маленький класс прямоугольника, имеющий одну вспомогательную функцию, определенную в нем, призванную немного облегчить жизнь клиентам, если они пожелают узнать площадь прямоугольника. Но предположим, что у меня есть причина предварительно вычислить площадь прямоугольника, чтобы каждый раз при вызове метода GetArea не пришлось вычислять ее заново. Может, я хочу сделать так потому, что знаю, что GetArea бу-

дет вызываться многократно с одним и тем же экземпляром на протяжении всего времени его существования. Если не обращать внимания на то, что ранняя оптимизация неразумна, представим, что я сделал это. Теперь мой класс MyRectangle выглядит примерно так:

```
class MyRectangle
{
    public uint width;
    public uint height;
    public uint area;
    public uint GetArea()
    {
        return area;
    }
}
```

Присмотревшись внимательнее, вы заметите мои ошибки. Обратите внимание, что все поля — общедоступны. Это позволяет потребителю экземпляров моего класса MyRectangle иметь прямой доступ к его внутренностям. Какой смысл в предоставлении метода GetArea, если потребитель может напрямую обратиться к полю area? Ладно, вы можете сказать, что мне стоило сделать поле area приватным. Таким образом, клиенты были бы вынуждены вызывать GetArea, чтобы получить площадь прямоугольника. Это определенно шаг в правильном направлении. Так и поступим:

```
class MyRectangle
{
    public uint width;
    public uint height;
    private uint area;
    public uint GetArea()
    {
        if( area == 0 ) {
            area = width * height;
        }
        return area;
    }
}
```

Я сделал поле area приватным, вынуждая потребителя вызывать метод GetArea, чтобы получить площадь прямоугольника. Однако по ходу я понял, что должен в какой-то момент вычислить площадь прямоугольника. Но по причине лени я решил проверять значение поля area перед его возвратом, и если оно равно 0, то это значит, что его нужно вычислить перед возвратом. Это — грубая попытка оптимизации, но теперь я вычисляю площадь только при необходимости. Предположим, что потребитель экземпляра моего прямоугольника никогда не захочет узнать его площадь. Тогда, при условии использования предыдущего кода, ему никогда не придется тратить время на вычисление этой площади. Конечно, в этом моем надуманном примере такая оптимизация будет весьма незначительной. Но если вы немного задумаетесь, то я уверен, что сможете придумать пример, который существенно выигрывает от применения "ленивого" вычисления. Подумайте о доступе

к базе данных по медленной сети, когда во время выполнения понадобятся лишь несколько полей таблицы. Или же, для того же объекта доступа к данным может оказаться слишком дорогой операция подсчета строк в таблице. Поэтому вы должны использовать подобную технику лишь при необходимости.

В моем классе прямоугольника все еще присутствует вопиющая проблема. Поскольку поля width и height являются общедоступными, что произойдет, если пользователь изменит одно из значений после вызова GetArea на экземпляре? При этом я получу наихудший пример несогласованности внутренностей объекта. Целостность состояния моего объекта будет нарушена. Это определенно нехорошая ситуация. Поэтому вы можете видеть, что в моем дизайне по-прежнему присутствуют ошибки. Я должен также сделать приватными поля width и height:

```
class MyRectangle
{
    private uint width;
    private uint height;
    private uint area;
    public uint Width
    {
        get
        {
            return width;
        }
        set
        {
            width = value;
            ComputeArea();
        }
    }
    public uint Height
    {
        get
        {
            return height;
        }
        set
        {
            height = value;
            ComputeArea();
        }
    }
    public uint Area
    {
        get
        {
            return area;
        }
    }
    private void ComputeArea()
    {
        area = width * height;
    }
}
```

Наконец, в моей последней версии MyRectangle я поумнел. После того, как поля width и height стали приватными, я понял, что потребитель объектов нуждается в каком-то способе установки и получения значений ширины и высоты. И здесь мне пригодятся свойства C#. Теперь я обработаю изменения внутреннего состояния в теле метода, и методы, вызываемые во время установки значений, относятся к набору специально именованных методов класса. Следует больше сказать о специальных, иногда называемых *зарезервированными*, именах членов, и я сделаю это в разделе "Зарезервированные имена членов". Теперь у меня есть возможности более тонкого контроля доступа к внутренностям объекта. И вместе с этим контролем пришла более высокая степень инкапсуляции. Я могу эффективно управлять внутренним состоянием объекта, так что оно никогда не станет несогласованным. Невозможно гарантировать целостность состояния объекта, когда внешние сущности имеют доступ к внутреннему состоянию "с черного хода".

В данном примере мой объект точно знает, когда изменяются поля width и height. Поэтому он может предпринять необходимые действия для вычисления новой площади. Если объект использует подход "ленивого" вычисления, сохраняя кэшированное значение площади, вычисленное при первом чтении свойства Area, то я должен объявить недействительным значение кэша, как только будет вызван блок set любого из свойств — Width или Height.

Итак, мораль этой истории в том, что совсем немного дополнительной работы по обеспечению инкапсуляции со временем окупится многократно. Одно из самых замечательных свойств инкапсуляции, которое вам следует хорошенько отпечатать в своем сознании — это то, что при использовании свойств внутреннее состояние объекта может изменяться для поддержки другого алгоритма, что никак не затронет его потребителей. Шаблоны проектирования на базе интерфейсов также помогают в этом отношении. Так, например, в финальной реализации класса MyRectangle площадь вычисляется заново, как только устанавливается новое значение свойства Width или Height. Может быть позднее, когда мое программное обеспечение будет почти готово, я запущу профилировщик и обнаружу, что предварительное вычисление площади действительно сильно загружает процессор при работе моего приложения. Никаких проблем! Я могу изменить модель, ориентируя ее на применение кэшированного значения площади, которое вычисляется только при первом обращении, и поскольку я следовал принципам инкапсуляции, потребителям моих объектов даже не нужно знать об этом. Они не будут знать о том, что внутренняя реализация объекта изменилась. В этом-то и состоит мощь инкапсуляции. Когда меняется внутренняя реализация объекта, а использующие его клиенты могут не меняться — значит, инкапсуляция работает так, как должна.

На заметку! Инкапсуляция помогает вам достичь четкого взаимодействия объектов при слабой зависимости между ними.

Доступность

До сих пор я уже несколько раз упоминал о модификаторах доступа. Их применение может выглядеть интуитивно понятным, если у вас есть какой-нибудь опыт работы с любым другим объектно-ориентированным языком программирования вроде C++ или Java. Однако некоторые нюансы доступа к членам в C# и CLI также заслуживают упоминания. Прежде чем рассматривать различные типы модификаторов, давайте немного поговорим о том, где вы можете применять их.

По сути, вы можете использовать модификаторы доступа почти с любой определенной сущностью в программе на C#, включая классы и любые члены внутри этих классов. Модификаторы доступа, применяемые к классу, касаются его видимости извне содержащей этот класс сборки. Модификаторы доступа, примененные к членам класса, включая методы, поля, свойства, события и индексаторы, влияют на видимость члена извне данного класса. В табл. 4.1 описаны различные модификаторы доступа, имеющиеся в C#.

Таблица 4.1. Модификаторы доступа в C#

Модификатор доступа	Значение
`public`	Член полностью видим как извне области определения, так и внутри этой области. Другими словами, доступ к общедоступному члену вообще не ограничен.
`protected`	Член видим только определяющему его классу и любому классу-наследнику данного класса.
`internal`	Член видим везде в пределах содержащей его сборки. Сюда входит определяющий его класс и любая область внутри сборки, но вне данного класса.
`protected internal`	Член видим внутри определяющего его класса и везде внутри сборки. Этот модификатор комбинирует `protected` и `internal`, используя логическую операцию "ИЛИ". Член также видим любому классу-наследнику определяющего его класса, независимо от того, находится он в той же сборке или нет.
`private`	Член видим только в определяющем классе, без исключений. Это — наиболее строгая форма доступа, и она принята по умолчанию для членов класса.

Обратите внимание, что CLR поддерживает еще одну форму доступности, которую проектировщики языка C# сочли необязательной для реализации. Внутри CLR она известна как доступность *семейства и сборки*. В терминологии C# это соответствует `protected` И `internal`. Если по какой-то причине вам совершенно необходим такой модификатор доступа, то вам следует использовать другой язык, такой как C++/CLI или "сырой" IL.

Теперь давайте рассмотрим допустимое применение этих модификаторов к разным определенным сущностям внутри C#. Члены класса могут использовать пять вариантов модификаторов доступа C#. Доступ по умолчанию к членам класса в отсутствие всяких модификаторов — это `private`. Классы, определенные внутри или вне пространства имен, могут иметь только два модификатора доступа: `public` или `internal`. По умолчанию для них принято `internal`.

Вы можете применять `public`, `private` и `internal` к определению членов `struct`. Далее, в разделе "Определения типов значений" настоящей главы, я расскажу во всех подробностях об определении `struct`. Обратите внимание на отсутствие модификаторов `protected` и `protected internal`. Здесь они не нужны, поскольку `struct` по умолчанию герметизированы (`sealed`) — в том смысле, что не могут служить базовыми классами для наследования. Модификатор `sealed` будет рассмотрен подробно в разделе "Герметизированные классы".

На заметку! Еще одно важное замечание для тех, кто работал на C++: члены `struct` являются `private` по умолчанию — как в определении класса, в то время как в C++ они по умолчанию `public`.

И, наконец, члены интерфейсов, которые я опишу полностью в главе 5, а также `enum`, о которых речь пойдет в главе 3, по своей природе являются `public`. Интерфейсы предназначены для определения набора операций, или контракта, который класс может реализовать. Для интерфейса не имеет смысла предусматривать какие-то члены с ограниченным доступом, поскольку члены с ограниченным доступом обычно ассоциируются с реализацией класса, а интерфейсы по определению не содержат реализации. Перечисления, с другой стороны, обычно используются в качестве именованных коллекций констант. Они также не имеют внутренней реализации, так что ограничение доступа для них также не имеет смысла. Фактически, вы можете получить ошибку, если специфицируете модификатор доступа вроде `public` для члена интерфейса или члена перечисления.

Как видите, почти всегда доступом по умолчанию является наиболее ограниченный доступ, имеющий смысл для данной сущности. Другими словами, вам следует поработать, чтобы открыть другой доступ к классам или членам класса. Единственное исключение — доступ к пространству имен, принятый по умолчанию как `public` и не допускающий указания каких-либо модификаторов доступа.

Интерфейсы

Даже несмотря на то, что я посвятил большую часть главы 5 теме интерфейсов, интерфейсы стоит представить уже здесь, поскольку они понадобятся на протяжении оставшейся части этой главы.. Вообще говоря, *интерфейс* — это определение контракта. Классы могут реализовывать разные интерфейсы и, делая это, гарантируют выполнение требований контракта. Когда класс наследуется от интерфейса, он обязан реализовать все члены этого интерфейса. Класс может реализовывать столько интерфейсов, сколько нужно, перечисляя их в списке базовых классов своего определения.

В общих чертах синтаксис интерфейса очень похож на синтаксис класса. Однако каждый его член неявно имеет модификатор `public`. Фактически вы получите ошибку времени компиляции, если объявите любой член интерфейса с каким-либо явным модификатором. Интерфейсы могут содержать только методы экземпляра; т.е. вы не можете включать никаких статических методов в их определения. Интерфейсы не включают реализации; т.е. они по природе своей семантически абстрактны. Если вы знакомы с C++, то знаете, что там можно создать подобную конструкцию, объявляя класс, содержащий только общедоступные, чистые (pure) виртуальные методы, не имеющие реализаций по умолчанию.

Член интерфейса может состоять только из членов, которые в конечном итоге становятся методами в CLR. Сюда относятся методы, свойства, события и индексаторы. Об индексаторах я расскажу в одноименном разделе главы 10.

На заметку! Если вы — сторонник строгой терминологии, то для вас скажу, что спецификация C# на самом деле называет свойства, события, индексаторы, операции, конструкторы и деструкторы *функциональными членами*. На самом деле было бы неправильно называть их *методами*. Методы содержат исполняемый код, так что они также считаются функциональными членами.

Следующий код показывает пример интерфейса и класса, реализующего интерфейс:

```
public interface IMusician
// Примечание: общепринятая практика заключается
// в предварении имен интерфейсов заглавной "I"
{
    void PlayMusic();
}

public class TalentedPerson : IMusician
{
    public void PlayMusic() {}
    public void DoALittleDance() {}
}

public class EntryPoint
{
    static void Main()
    {
        TalentedPerson dude = new TalentedPerson();
        IMusician musician = dude;
        musician.PlayMusic();
        dude.PlayMusic();
        dude.DoALittleDance();
    }
}
```

В данном примере определен интерфейс по имени `IMusican`. Класс `TalentedPerson` указывает, что он желает поддерживать интерфейс `IMusican`. Объявление класса на самом деле говорит: "Я хотел бы вступить в контракт для поддержки интерфейса `IMusican`, и я гарантирую поддержку всех методов этого интерфейса". Требование интерфейса заключается просто в поддержке метода `PlayMusic`, что исправно исполняет класс `TalentedPerson`. В качестве заключительного замечания скажу, что обычно принято называть типы интерфейсов, начиная с заглавной буквы "I". При чтении кода это служит маркером, указывающим, что тип с таким именем на самом деле является интерфейсом.

Теперь клиенты могут обращаться к методу `PlayMusic` одним из двух способов. Они могут либо вызывать его непосредственно через экземпляр объекта, либо получить интерфейсную ссылку на экземпляр объекта и вызвать метод через нее. Поскольку класс `TalentedPerson` поддерживает интерфейс `IMusican`, ссылки на объекты этого класса являются неявно преобразуемыми к ссылкам на `IMusican`. Код внутри метода `Main` в предыдущем примере показывает, как вызывать методы обоими способами.

Тема интерфейсов достаточно обширна, чтобы ей можно было посвятить целую главу, что я и сделал в главе 5. Однако информации об интерфейсах, которую я привел здесь, будет достаточно для облегчения дискуссии в оставшейся части настоящей главы.

Наследование

Если вы спросите, то многие разработчики скажут вам, что наследование — это краеугольный камень объектно-ориентированного программирования. Хотя наследование — вполне ясная концепция для тех, кто впервые сталкивается с ним, я бы оспорил утверждение о том, что наследование — краеугольный камень. Я склонен полагать, что инкапсуляция — более строгое свойство объектно-ориентированного программирования. Наследование — важная концепция и полезный инструмент. Однако, как и множество других мощных инструментов, оно может представлять опасность, будучи использованным неправильно. Моя цель в настоящем разделе — представить вам наследование таким образом, чтобы вы могли оценить его мощь, и при этом помочь вам избежать злоупотребления им.

Ранее я уже описывал синтаксис определения класса. Вы специфицируете базовый класс после двоеточия, следующего за именем класса. В C# класс имеет только один базовый класс (некоторые языки, такие как C++, поддерживают множественное наследование).

Доступность членов

Доступность членов является важным аспектом наследования, особенно, когда речь идет о доступности членов базового класса из производного класса. Любые общедоступные члены базового класса становятся общедоступными и в производном классе.

Любые члены, помеченные модификатором `protected`, доступны только внутри объявляющего их класса и его наследников. Защищенные члены никогда не доступны извне определяющего их класса или его наследников. Приватные (`private`) члены не доступны нигде, кроме определяющего их класса. Так что даже несмотря на то, что производный класс наследует все члены базового класса, включая и приватные, код в производном классе не имеет доступа к приватным членам, унаследованным от базового класса. Вдобавок защищенные внутренние (`protected internal`) члены также видимы всем типам, определенным внутри одной сборки, и классам-наследникам определившего их класса. Реальность состоит в том, что производный класс наследует все члены базового класса за исключением конструкторов экземпляра, статических конструкторов и деструкторов.

Как было показано, вы можете контролировать доступность всего класса в целом при его определении. Единственные возможности доступа к типу класса — это `internal` и `public`. При использовании наследования действует правило, что тип базового класса должен быть доступен как минимум настолько же, как и производный класс. Рассмотрим следующий код:

```
class A
{
    protected int x;
}
public class B : A
{
}
```

Этот код не компилируется, потому что класс `A` объявлен как `internal`, и не является настолько, как минимум, доступным, как производный от него класс `B`.

Напомню, что в отсутствие модификатора доступа класс имеет доступ internal, поэтому класс A на самом деле является internal. Для того чтобы этот код компилировался, вы должны либо продвинуть класс A до уровня доступа public, либо ограничить класс B доступом internal. К тому же заметьте, что для класса A допустимо быть public, а для класса B — internal.

Неявные преобразования и полиморфизм

Вы можете представлять наследование и то, что оно делает, несколькими способами. Первый и наиболее очевидный — наследование позволяет позаимствовать реализацию. Другими словами, вы можете наследовать класс D от класса A и повторно использовать реализацию класса A в классе D. Потенциально это позволит сэкономить некоторую часть работы при определении класса D. Другое применение наследования — *специализация*, когда класс D становится специализированной формой класса A. Например, рассмотрим иерархию классов, представленную на рис. 4.1.

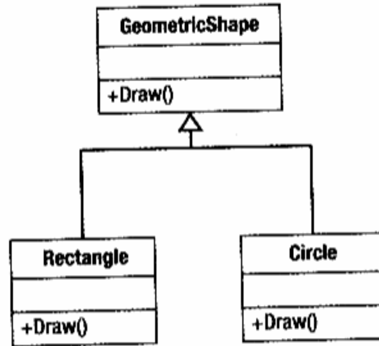

Рис. 4.1. Специализация при наследовании

Как видите, классы Rectangle и Circle наследуются от класса GeometricShape. Другими словами, они являются специализациями класса GeometricShape. Специализация бессмысленна без полиморфизма и виртуальных методов. Боле подробно я раскрою тему полиморфизма в разделе "Наследование и виртуальные методы" настоящей главы. А пока определю в общих чертах, что означает это понятие.

Полиморфизм описывает ситуацию, в которой тип, на который ссылается определенная переменная, может вести себя как (и в действительности быть) экземпляр другого (более специализированного) типа. В главе 5 рассматриваются отличия и сходства между интерфейсами и контрактами. На рис. 4.1 показан метод класса GeometricShape по имени Draw. Этот же метод появляется и в Rectangle, и в Circle. Вы можете реализовать эту модель в следующем коде:

```
public class GeometricShape
{
    public virtual void Draw()
    {
        // Выполнить некоторое рисование по умолчанию
    }
}
```

```
public class Rectangle : GeometricShape
{
    public override void Draw()
    {
        // Нарисовать прямоугольник
    }
}
public class Circle : GeometricShape
{
    public override void Draw()
    {
        // Нарисовать круг
    }
}
public class EntryPoint
{
    private static void DrawShape( GeometricShape shape )
    {
        shape.Draw();
    }
    static void Main()
    {
        Circle circle = new Circle();
        GeometricShape shape = circle;
        DrawShape( shape );
        DrawShape( circle );
    }
}
```

Вы создаете новый экземпляр `Circle` в методе `Main`. Сразу после этого вы получаете ссылку типа `GeometricShape` на тот же объект. Это важный момент. Компилятор здесь неявно преобразует эту ссылку в ссылку на тип `GeometricShape`, позволяя вам использовать простое выражение присваивания. На самом деле, однако, она продолжает ссылаться на тот же объект `Circle`. В этом суть специализации типа и автоматического преобразования, сопровождающего ее.

Теперь рассмотрим остальную часть кода метода `Main`. После того, как вы получили ссылку `GeometricShape` на экземпляр `Circle`, вы можете передать ее методу `DrawShape`, который не делает ничего кроме вызова метода `Draw` переданной ему фигуры. Однако ссылка на объект фигуры на самом деле указывает на `Circle`, метод `Draw` определен как виртуальный, а класс `Circle` переопределяет виртуальный метод, так что вызов `Draw` на ссылке `GeometricShape` на самом деле вызывает `Circle.Draw`. Это и есть полиморфизм в действии. Метод `DrawShape` не интересует, какой именно конкретный тип фигура представляет переданный ему объект. То, с чем он имеет дело — это `GeometricShape`. И `Circle` является `GeometricShape`. Вот почему наследование иногда называют отношением "is-a" ("является"). В данном примере `Rectangle` является `GeometricShape`, и `Circle` является `GeometricShape`. Ключ к ответу на вопрос, когда наследование имеет смысл, а когда нет, лежит в применении отношения "is-a" к вашему дизайну. Если класс `D` наследуется от класса `B`, и класс `D` семантически не является классом `B`, то для данного отношения наследование — неподходящий инструмент.

Еще одно последнее важное замечание о наследовании и возможности преобразования. Я говорил, что компилятор неявно конвертирует ссылку на экземпляр `Circle` в ссылку на экземпляр `GeometricShape`. Неявно в данном случае означает, что код не должен ничего специального делать для выполнения такого преобразования, а под "чем-то специальным" я обычно имею в виду операцию приведения. Поскольку компилятор обладает способностью делать это на основе знания иерархии наследования, то может показаться, что можно и обойтись без получения ссылки на `GeometricShape` перед вызовом `DrawShape` с экземпляром `Circle`. На самом деле так оно и есть! Последняя строка метода `Main` доказывает это. Вы можете просто передать ссылку на экземпляр `Circle` непосредственно в метод `DrawShape`, и поскольку компилятор может неявно преобразовать ее в ссылку на тип `GeometricShape` исходя из отношений наследования, он выполнит всю работу за вас. Опять-таки вы можете видеть всю мощь этого механизма.

Теперь вы можете передавать любой экземпляр объекта, производного от `GeometricShape`. После того, как ваше программное обеспечение будет упаковано в коробку и помечено наклейкой "версия 1", некто может потом заняться версией 2 и определить новые фигуры, унаследованные от `GeometricShape`, причем код `DrawShape` не потребует никаких изменений. Ему даже не нужно будет ничего знать от новой специализации. Это могут быть `Trapezoid`, `Square` (специализации `Rectangle`) или же `Ellipse`. Это не имеет значения до тех пор, пока они наследуются от `GeometricShape`.

Сокрытие членов

Исходя из приведенной в предыдущем разделе дискуссии, вы можете видеть, как, несмотря на свою мощность, концепция наследования может быть использована неправильно. Когда программисты впервые узнают о наследовании, они склонны применять его слишком часто, создавая проекты и иерархические структуры, которые трудно сопровождать. Важно отметить, что у наследования есть альтернативы, которые во многих случаях более оправданы. Среди различного рода ассоциаций, возможных между классами в дизайне программной системы, наследование — самая жесткая из всех. Ближе к концу главы я еще расскажу о многих других проблемах, связанных с наследованием. Однако давайте немного забежим вперед и рассмотрим некоторые основные эффекты от наследования.

Обратите внимание, что наследование расширяет функциональность, но не может ее исключать. Например, общедоступные методы базового класса доступны через экземпляры производного класса и классов, унаследованных от него. Вы не можете удалить эти возможности из производного класса. Рассмотрим следующий код:

```
public class A
{
    public void DoSomething()
    {
        System.Console.WriteLine( "A.DoSomething" );
    }
}
public class B : A
{
    public void DoSomethingElse()
    {
```

```
            System.Console.WriteLine( "B.DoSomethingElse" );
        }
}
public class EntryPoint
{
    static void Main()
    {
        B b = new B();
        b.DoSomething();
        b.DoSomethingElse();
    }
}
```

Здесь в Main вы создаете новый экземпляр класса B, который наследуется от класса A. Класс B получает объединение членов обоих классов — A и B. Вот почему вы можете вызвать и DoSomething, и DoSomethingElse на экземпляре класса B. Это вполне очевидно, поскольку наследование расширяет функциональность.

Но что, если вы хотите наследовать от класса A, но скрыть метод DoSomething? Другими словами, что если вы хотите расширить лишь часть функциональности A? С помощью наследования это невозможно. Однако у вас есть возможность *сокрытия* члена, как показано в следующем коде, который является модифицированной формой предыдущего примера:

```
public class A
{
    public void DoSomething()
    {
        System.Console.WriteLine( "A.DoSomething" );
    }
}
public class B : A
{
    public void DoSomethingElse()
    {
        System.Console.WriteLine( "B.DoSomethingElse" );
    }
    public new void DoSomething()
    {
        System.Console.WriteLine( "B.DoSomething" );
    }
}
public class EntryPoint
{
    static void Main()
    {
        B b = new B();
        b.DoSomething();
        b.DoSomethingElse();
        A a = b;
        a.DoSomething();
    }
}
```

Как видите, в этой версии я ввел в класс B новый метод по имени DoSomething. Также обратите внимание на добавление ключевого слова new к объявлению B.DoSomething. Если не добавить это ключевое слово, то компилятор выдаст предупреждение. Это его способ сообщить вам, чтобы вы выражались яснее относительно сокрытия метода базового класса. Возможно, компилятор делает так потому, что подобное сокрытие членов обычно трактуется как плохой дизайн. Давайте разберемся, почему. Вывод предыдущего кода выглядит так:

```
B.DoSomething
B.DoSomethingElse
A.DoSomething
```

Первое замечание: какой именно метод DoSomething будет вызван, зависит от типа ссылки, через которую он будет вызван. Это достаточно не очевидно, поскольку B является A, а вы знаете, что наследование моделирует отношение "является". В данном случае должен ли весь общедоступный интерфейс A быть доступным потребителям экземпляра класса B? Короткий ответ — нет. Если вы действительно хотите, чтобы метод вел себя по-разному в подклассах, тогда в точке определения класса A вы должны объявлять метод DoSomething виртуальным. Подобным образом вы могли бы утилизировать полиморфизм, чтобы делать правильную вещь. Тогда должна вызываться наиболее последняя версия (версия наследника) метода DoSomething, независимо от того, через какой тип ссылки он был вызван.

Позже я еще вернусь к теме виртуальных методов, но давайте еще раз задумаемся о них здесь. Чтобы объявлять DoSomething как виртуальный метод, вы должны думать о будущем, когда определяете его. То есть вы должны предвидеть, что кто-нибудь может захотеть переопределить его функциональность. Это только одна причина того, почему наследование может усложниться в процессе проектирования по сравнению с тем, что кажется поначалу. Как только вы используете наследование, вам придется задуматься об очень многих подобных вещах. А мы знаем, что никто не в состоянии предсказать будущее.

Даже несмотря на то, что класс B теперь скрывает реализацию DoSomething класса A, помните, что он не удаляет ее. Он скрывает ее при вызове этого метода через ссылку типа B на объект. Однако в методе Main вы можете видеть, что это легко обойти, применив неявное преобразование ссылки на экземпляр типа B в ссылку на экземпляр типа A с последующим вызовом через нее реализации A.DoSomething. То есть старая реализация не исчезла, она просто скрыта. И чтобы добраться до нее, вам понадобится просто выполнить немножко больше работы.

Предположим, что вы передаете ссылку на экземпляр B методу, принимающему ссылку на экземпляр A, подобно тому, как это делается в примере с DrawShape. Ссылка на экземпляр B должна быть неявно преобразована в ссылку на экземпляр A, и если за этим последует вызов метод DoSomething этого экземпляра A, то получим A.DoSomething вместо B.SomeThing. Наверно, это не то, чего ожидал тот, кто вызвал данный метод.

Это — классический пример того, что если язык позволяет вам делать что-то подобное, это не означает, что поступая так, вы создадите хороший дизайн. Почти любые существующие языки, включая C++, обладают такими средствами, которые будучи использованными (у тому же неправильно), лишь добавят излишнюю сложность.

Ключевое слово base

Когда вы наследуете класс, часто в методе производного класса возникает необходимость вызова метода или доступа к полю, свойству или индексатору базового класса. Для этой цели предусмотрено ключевое слово base. Вы можете применять это ключевое слово как любую другую переменную экземпляра, но его можно применять только внутри блока конструктора экземпляра, метода экземпляра или средства доступа к свойству. Вы не можете применять его в статических методах. Это совершенно оправдано, потому что base открывает доступ к реализациям экземпляра базового класса — подобно тому, как this разрешает доступ к экземпляру — владельцу текущего метода. Рассмотрим следующий блок кода:

```
public class A
{
    public A( int var )
    {
        this.x = var;
    }
    public virtual void DoSomething()
    {
        System.Console.WriteLine( "A.DoSomething" );
    }
    private int x;
}
public class B : A
{
    public B()
        : base( 123 )
    {
    }
    public override void DoSomething()
    {
        System.Console.WriteLine( "B.DoSomething" );
        base.DoSomething();
    }
}
public class EntryPoint
{
    static void Main()
    {
        B b = new B();
        b.DoSomething();
    }
}
```

В этом примере вы можете видеть два применения ключевого слова base; первое из них — в конструкторе класса B. Напомним, что класс не наследует конструкторов экземпляра. Однако при инициализации объекта иногда бывает нужно вызвать явно один из конструкторов базового класса во время инициализации производного класса. Это объясняет нотацию в конструкторе экземпляра класса B. Инициализация базового класса происходит после объявления списка параметров

конструктора производного класса, но перед блоком кода конструктора производного класса. Порядок вызовов конструкторов и инициализацию объектов будет рассмотрен во всех подробностях позднее, в разделе "Создание объектов".

Второе применение ключевого слова `base` содержится в реализации `B.DoSomething`. Я решил, что в моей реализации класса `B`, реализуя метод `B.DoSomething`, я хочу позаимствовать реализацию `DoSomething` из класса `A`. Я могу вызвать реализацию `A.DoSomething` непосредственно из реализации `B.DoSomething`, снабдив его префиксом — ключевым словом `base`.

Если вы знакомы с виртуальными методами, то здесь, возможно, недоуменно пожмете плечами. Если метод `DoSomething` виртуальный, а ключевое слово `base` ведет себя, как переменная экземпляра базового класса, не случится ли так, что `base.DoSomething` на самом деле превратится в вызов `B.DoSomething`? В конце концов, так работает полиморфизм, и `base.DoSomething` — эквивалент вызова `((B)this).DoSomething`, что является просто приведением ссылки `this` к ссылке на класс `B`, и потому получится вызов `B.DoSomething`, не правда ли? Да, если бы это было так, то код `B.DoSomething` запустил бы бесконечный цикл (точнее, бесконечную рекурсию — *Примеч. пер.*).

На самом деле никакого бесконечного цикла не запускается. Ключевое слово `base` трактуется специальным образом, когда встречается внутри члена экземпляра для вызова виртуального метода. Обычно вызов виртуального метода на экземпляре означает вызов его наиболее "унаследованной" версии — в данном случае `B.SomeThing`. Однако когда он вызывается через ключевое слово `base`, то вызывается наиболее "унаследованная" реализация метода базового класса. Таким образом, вы можете реализовать переопределенный метод, заимствуя реализацию базового класса. Если вас интересуют подробности, то скажу, что в сгенерированном коде IL вызов через ссылку `base` происходит по инструкции `call`, а не `callvirt`.

Герметизированные классы

Я уже упоминал, что наследование — это такой мощный инструмент, которым легко злоупотребить. Фактически это настолько верно, что я посвятил целый раздел настоящей главы "Наследование, включение и делегирование" теме подводных камней, связанных с наследованием. Когда вы создаете новый класс, иногда вы намереваетесь сделать его базовым классом — заготовкой для дальнейшей специализации. Однако часто бывает и так, что классы проектируются, не принимая во внимание то, будут они использоваться в качестве базовых или нет. На самом деле весьма вероятно, что класс, который вы проектируете сегодня, будет использован в качестве базового завтра, даже если вы не собирались этого допускать.

C# предлагает ключевое слово `sealed` для тех случаев, когда вы не хотите, чтобы клиент наследовал свой класс от вашего. Примененное к целому классу, ключевое слово `sealed` указывает на то, что данный класс — *листовой* в дереве наследования. Под этим я имею в виду запрет наследования от этого класса. Если вы визуализируете вашу диаграмму наследования в виде деревьев, то имеет смысл называть `sealed`-классы листовыми. Поначалу может показаться, что ключевое слово `sealed` вам придется использовать редко. Однако я уверен, что на самом деле должно быть наоборот. При проектировании новых классов вы должны применять это ключевое слово настолько часто, насколько возможно. Я бы даже посоветовал использовать его по умолчанию.

Наследование — такая хитрая штука, что для того, чтобы класс был хорошим базовым классом, вы должны сразу проектировать его с таким прицелом. Если это не так, сразу помечайте его как sealed. Это очень просто. Теперь вы можете подумать "Почему бы мне не оставить его не sealed, чтобы кто-нибудь в будущем мог наследоваться от него, обеспечивая максимальную гибкость?". Ответ для хорошего дизайна: нет. Еще раз подчеркну, что класс, предназначенный для того, чтобы служить базовым классом, должен быть спроектирован с учетом этого с самого начала. Если это не так, то весьма вероятно, что вы столкнетесь с ловушками, пытаясь эффективно наследовать производный класс от него.

На заметку! Во многих случаях классы, предназначенные для того, чтобы быть расширяемыми базовыми классами, находятся в библиотеках. Создание библиотек — кропотливое дело, которому нужно посвятить много времени, чтобы обеспечить максимальную применимость этих библиотек. Вдобавок, однажды опубликовав библиотеку, вы обречены сопровождать ее в течение длительного времени. Если вы планируете разрабатывать библиотеки, я советую обратиться к руководству Кшиштофа Квалины (Krzysztof Cwalina) и Брэда Абрамса (Brad Abrams) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Boston, MA: Addison-Wesley Professional, 2005 г.). Эта книга основана на внутренних руководствах, использованных командой разработчиков библиотеки базовых классов .NET при разработке каркаса.

Абстрактные классы

На противоположном конце спектра от sealed-классов находятся абстрактные классы. Иногда вам нужно спроектировать класс, чье единственное назначение — служить базовым классом. Вы можете пометить классы, подобные этому, ключевым словом abstract.

Ключевое слово abstract сообщает компилятору, что назначение данного класса — служить базовым, и потому коду не разрешается создавать экземпляры этого класса. Вернемся к примеру GeometricShape, приведенному ранее в этой главе:

```
public abstract class GeometricShape
{
    public abstract void Draw();
}
public class Circle : GeometricShape
{
    public override void Draw()
    {
        // Выполнить какое-то рисование
    }
}
public class EntryPoint
{
    static void Main()
    {
        Circle shape = new Circle();
        // Это не будет работать!
        // GeometricShape shape2 = new GeometricShape();
        shape.Draw();
    }
}
```

Не имеет смысла создавать объект GeometricShape сам по себе, поэтому я сделал класс GeometricShape абстрактным. Таким образом, если код в Main попытается создать экземпляр GeometricShape, будет выдана ошибка компиляции. Вы можете также отметить применение ключевого слова abstract в методе GeometricShape.Draw. Это применение данного ключевого слова будет подробно анализироваться в разделе "Виртуальные и абстрактные методы". Если говорить кратко, такое использование ключевого слова — это способ сообщить компилятору, что производные классы должны переопределить этот метод. Поскольку метод должен быть переопределен в производных классах, не имеет смысла иметь реализацию GeometricShape.Draw, раз уж все равно нельзя создать экземпляр GeometricShape. Поэтому абстрактные методы не должны иметь реализации. Если вы пришли из мира C++, то вы можете заметить, что C++ допускает реализацию абстрактных методов. Это верно, но проектировщики C# сочли идею ненужной. По моему опыту, я редко сталкивался с необходимостью реализации по умолчанию абстрактного метода, кроме как в отладочных сборках.

Как видите, бывают случаи в дизайне, когда вы используете базовый класс для определения некоторого шаблона или поведения, предоставляя реализацию наследникам. Листовые классы могут наследоваться от этого базового шаблона, уточняя подробности реализации.

Вложенные классы

Вы определяете вложенные классы внутри области определения другого класса. Классы, определенные внутри контекста пространства имен или вне пространства имен, но не внутри контекста другого класса, называются *не вложенными*. Вложенные классы обладают некоторыми специальными возможностями, которые удобны, когда вам нужен вспомогательный класс, работающий внутри содержащего его класса.

Например, контейнерный класс может содержать коллекцию объектов. Предположим, что вам нужно некоторое средство для выполнения итерации по всем содержащимся объектам, чтобы позволить внешним пользователям, выполняющим итерацию, поддерживать маркер, или итератор такого рода, который запоминает свое текущее место во время итерации. Это распространенная техника проектирования. Избавление пользователей от необходимости хранить прямые ссылки на содержащиеся в коллекции объекты обеспечивает намного больше гибкости для изменения внутреннего поведения контейнерного класса без разрушения кода, использующего этот контейнерный класс. Вложенные классы по нескольким причинам представляют отличное решение такой проблемы.

Для начала, вложенные классы имеют доступ ко всем членам, видимым содержащему их классу, даже если эти члены приватные. Рассмотрим следующий код, который представляет контейнерный класс, включающий экземпляры GeometricShape:

```
using System.Collections;
public abstract class GeometricShape
{
    public abstract void Draw();
}
```

```
public class Rectangle : GeometricShape
{
    public override void Draw()
    {
        System.Console.WriteLine( "Rectangle.Draw" );
    }
}
public class Circle : GeometricShape
{
    public override void Draw()
    {
        System.Console.WriteLine( "Circle.Draw" );
    }
}
public class Drawing : IEnumerable
{
    private ArrayList shapes;
    private class Iterator : IEnumerator
    {
        public Iterator( Drawing drawing )
        {
           this.drawing = drawing;
           this.current = -1;
        }
        public void Reset()
        {
           current = -1;
        }
        public bool MoveNext()
        {
           ++current;
           if( current < drawing.shapes.Count ) {
           return true;
           } else {
           return false;
           }
        }
        public object Current
        {
           get
           {
              return drawing.shapes[ current ];
           }
        }
        private Drawing drawing;
        private int current;
    }
    public Drawing()
    {
        shapes = new ArrayList();
    }
```

```
        public IEnumerator GetEnumerator()
        {
            return new Iterator( this );
        }
        public void Add( GeometricShape shape )
        {
            shapes.Add( shape );
        }
    }
    public class EntryPoint
    {
        static void Main()
        {
            Rectangle rectangle = new Rectangle();
            Circle circle = new Circle();
            Drawing drawing = new Drawing();
            drawing.Add( rectangle );
            drawing.Add( circle );
            foreach( GeometricShape shape in drawing ) {
               shape.Draw();
            }
        }
    }
```

Этот пример демонстрирует несколько новых концепций, таких как интерфейсы `IEnumerable` и `IEnumerator`, которые детально рассматриваются в главе 9. Но пока мы сосредоточимся в первую очередь на использовании вложенного класса. Как вы можете видеть, класс `Drawing` поддерживает метод `GetEnumerator`, являющийся частью реализации `IEnumerable`. Он создает экземпляр вложенного класса `Iterator` и возвращает его.

Но вот что интересно. Класс `Iterator` принимает ссылку на экземпляр содержащего его класса `Drawing` в виде параметра конструктора. Затем он сохраняет этот экземпляр для последующего использования, чтобы можно было добраться до коллекции `shapes` внутри объекта `drawing`. Однако заметьте, что коллекция `shapes` в классе `Drawing` объявлена как `private`. Это не имеет значения, потому что вложенные классы имеют доступ к приватным членам охватывающего их класса.

Также обратите внимание, что класс `Iterator` сам по себе объявлен как `private`. Не вложенные классы могут объявляться только как `public` или `internal` и по умолчанию являются `internal`. Вы можете применять к вложенным классом те же модификаторы доступа, что и к любым другим членам класса. В данном случае класс `Iterator` объявлен как `private`, так что внешний код вроде процедуры `Main` не может создавать экземпляры `Iterator` непосредственно. Только сам класс `Drawing` может создавать экземпляры `Iterator`. Ни для кого другого, кроме `Drawing.GetEnumerator`, не имеет смысла создавать экземпляры `Iterator`.

Вложенные классы, объявленные как `public`, позволяют создавать экземпляры коду, внешнему по отношению к охватывающему классу. Нотация обращения к вложенному классу подобна квалификации пространства имен. В следующем примере вы можете видеть создание экземпляра вложенного класса:

```
public class A
{
    public class B
    {
    }
}
public class EntryPoint
{
    static void Main()
    {
        A.B b = new A.B();
    }
}
```

Иногда, когда вы вводите вложенный класс, его имя скрывает имя члена внутри базового класса с применением ключевого слова new, подобно тому, как работает сокрытие методов. Это бывает очень редко, и в большинстве случаев такого можно избежать. Рассмотрим следующий пример:

```
public class A
{
    public void Foo()
    {
    }
}
public class B : A
{
    public new class Foo
    {
    }
}
```

В этом случае вы определяете вложенный класс Foo внутри определения класса B. Поскольку его имя совпадает с именем метода Foo в классе A, вы обязаны применить ключевое слово new, в противном случае компилятор сообщит вам о конфликте имен. Опять-таки, если вы попадаете в подобную ситуацию, то это означает, скорее всего, что наступило время переосмыслить ваш дизайн или просто переименовать вложенный класс, если только в ваши намерения не входит действительное сокрытие базового члена. Сокрытие базовых членов подобным образом — сомнительное решение, которого обычно не стоит придерживаться лишь по той причине, что язык позволяет это.

Индексаторы

Индексаторы позволяют вам трактовать экземпляр объекта так, будто он является массивом. Это открывает возможность более естественного использования объектов, которые должны вести себя подобно коллекциям, таких как экземпляры класса Drawing из предыдущего раздела.

Вообще индексаторы немного похожи на метод, чье имя — this. Почти как с любой сущностью системы типов C#, вы можете применять атрибуты метаданных и к индексаторам. Вы также можете применять к ним те же модификаторы, что и

почти к любым другим членам класса, за исключением одного — static, т.к. индексаторы не бывают статическими. Таким образом, индексаторы всегда относятся к экземпляру и работают с определенным экземпляром объекта определяющего их класса. За модификаторами в объявлении следует тип индексатора. Индексатор возвратит этот тип объекта вызывающему коду. Затем вы помещаете ключевое слово this, за которым следует список параметров в квадратных скобках, что я продемонстрирую в следующем примере.

По сути, индексатор ведет себя как некий гибрид свойства и метода. В конце концов, "за кулисами" он представляет собой один из специальных методов, определяемых компилятором при объявлении индексатора. Концептуально индексатор подобен методу в том, что он может принимать набор параметров. Однако он также ведет себя и как свойство, поскольку вы объявляете средства доступа к нему с применением аналогичного синтаксиса. К индексаторам вы можете применить многие из тех же модификаторов, что применяются к методам. Например, индексаторы могут быть виртуальными, они могут переопределять базовый индексатор, или же могут быть перегружены в зависимости от списка параметров — точно так же, как методы. За списком параметров следует блок кода индексатора, который по синтаксису похож на блок кода свойства. Главное отличие в том, что средства доступа индексатора могут принимать списки переменных-параметров, в то время как средства доступа свойств не используют определяемых пользователем параметров. Давайте добавим индексатор к объекту Drawing, чтобы посмотреть, как его можно использовать:

```
using System.Collections;
public abstract class GeometricShape
{
    public abstract void Draw();
}
public class Rectangle : GeometricShape
{
    public override void Draw()
    {
        System.Console.WriteLine( "Rectangle.Draw" );
    }
}
public class Circle : GeometricShape
{
    public override void Draw()
    {
        System.Console.WriteLine( "Circle.Draw" );
    }
}
public class Drawing
{
    private ArrayList shapes;
    public Drawing()
    {
        shapes = new ArrayList();
    }
```

```
    public int Count
    {
        get
        {
           return shapes.Count;
        }
    }
    public GeometricShape this[ int index ]
    {
        get
        {
           return (GeometricShape) shapes[index];
        }
    }
    public void Add( GeometricShape shape )
    {
        shapes.Add( shape );
    }
}
public class EntryPoint
{
    static void Main()
    {
        Rectangle rectangle = new Rectangle();
        Circle circle = new Circle();
        Drawing drawing = new Drawing();
        drawing.Add( rectangle );
        drawing.Add( circle );
        for( int i = 0; i < drawing.Count; ++i ) {
           GeometricShape shape = drawing[i];
           shape.Draw();
        }
    }
}
```

Как видите, вы можете обращаться к элементам объекта `Drawing` в методе `Main`, как если бы они находились в обычном массиве. Большинство типов коллекций поддерживают некоторого рода индексатор, подобный этому. К тому же, поскольку индексаторы имеют лишь средство доступа `get`, они доступны только для чтения. Однако имейте в виду, что если коллекция поддерживает ссылки на объекты, то клиентский код может изменять состояние содержащихся в ней объектов через ссылку. Но поскольку индексаторы доступны только для чтения, клиентский код не может заменить содержащийся объект через ссылку на него на какой-то совершенно другой объект.

Следует отметить одно различие между реальным массивом и объектом, представленным индексатором. Вы не можете передавать результат вызова индексатора на объекте в качестве `out`- или `ref`-параметра методу, как это можно делать с реальным массивом. Это ограничение подобно аналогичному ограничению, касающемуся свойств.

Частичные классы

Классы, объявленные как partial (частичные), были нововведением C# 2.0. До сих пор я показывал вам, как определять классы в одном единственном файле. Это было обязательно в C# 1.0. Тогда было невозможно разбить определение класса на несколько файлов.

Поначалу такое соглашение казалось неизбежным. В конце концов, если класс получается настолько большим, что становится трудно управляться с содержащим его файлом, это может служить признаком плохого дизайна. Но на самом деле главной причиной появления частичных классов стала необходимость поддержки инструментов генерации кода.

Обычно, когда вы работаете в среде IDE, она пытается помочь, генерируя для вас некоторый код. Например, мастер генерирует полезные классы-наследники DataSet при использовании средств ADO.NET. Классической проблемой при этом всегда было ручное редактирование сгенерированного инструментом кода. Редактировать такой код всегда опасно, поскольку в любой момент при изменении входных параметров для инструмента этот инструмент регенерирует код, уничтожая все изменения, проведенные вручную. Понятно, что это нежелательно. Раньше единственным способом обходить эту проблему было применение некоторой формы повторного использования, такого как наследование или включение, т.е. приходилось наследовать класс от класса, созданного инструментом генерации кода. Во многих случаях это было противоестественным решением проблемы. И часто код, сгенерированный этими инструментами, не был спроектирован так, чтобы принимать во внимание наследование.

Теперь вы можете положиться на ключевое слово partial в определении класса, которое помещается непосредственно перед ключевым словом class — вы можете разносить определение класса по нескольким файлам. К каждому файлу, содержащему часть определения класса, предъявляются следующие требования: использовать ключевое слово partial, и чтобы все части были определены внутри одного и того же пространства имен, если вы вообще объявляете их внутри пространства имен. Теперь, с добавлением ключевого слова partial, код, сгенерированный инструментальными средствами, может находиться в отдельном файле от ручных добавлений к этому генерированному классу, и когда инструмент перегенерирует код, вы не теряете своих изменений.

Вы должны кое-что знать о процессе, через который проходит компилятор при сборке частичного класса в одно целое. Необходимо компилировать все части класса вместе, чтобы компилятор мог найти все эти части. В основном все члены и аспекты класса сливаются вместе посредством операции объединения. Поэтому они должны сосуществовать вместе, как если бы вы объявили и определили их в одном файле. Списки базовых интерфейсов объединяются вместе. Однако поскольку каждый класс может иметь максимум один базовый класс, если части класса, разбитого на несколько файлов, ссылаются на базовый класс, это должен быть один и тот же базовый класс. Если не обращать внимания на эти совершенно очевидные ограничения, я думаю, вы согласитесь, что частичные классы — весьма полезное добавление к языку C#.

Частичные методы

C# 3.0 вводит ключевое слово partial для методов в дополнение к частичным классам. Частичный метод — это просто метод, чья сигнатура объявляется без тела в одной части частичного класса, а определяется в другой его части. Подобно частичным классам, частичные методы особенно полезны при потреблении кода, созданного мастерами и генераторами кода. Но красота частичных методов состоит в том, что если генератор создает объявление частичного метода в одной части класса, а вы не реализуете этот метод в другой его части, то такой метод просто не включается в результирующий собранный класс. Более того, любой код в сгенерированной части, вызывающий частичный метод, ничего не разрушит. Он просто вообще не будет вызывать такой метод. На частичные методы накладываются некоторые ограничения, необходимые для обеспечения упомянутого поведения.

- Частичные методы должны иметь тип возврата void.
- Частичные методы не могут принимать параметры out, но допускают параметры ref.
- Частичные методы также не могут быть external.
- Частичные методы не могут быть помечены как virtual и не могут быть декорированы модификаторами доступа, поскольку они неявно являются приватными.
- Частичные методы не могут быть помечены как static или unsafe.
- Частичные методы могут быть обобщенными и могут быть декорированы ограничениями, хотя повторение ограничений в объявлении реализации не обязательно.
- Делегаты не могут вызывать частичные методы, поскольку не гарантируется их существование в конечном скомпилированном продукте.

Имея в виду все эти ограничения, давайте рассмотрим краткий пример частичных методов. Представим один частичный класс, который в данном примере является результатом работы некоторого генератора кода:

```
public partial class DataSource
{
    // Некоторые полезные методы
    // ...
    partial void ResetSource();
}
```

Предположим, что этот класс DataSource, созданный генератором, представляет некоторую часть лежащего в основе хранилища данных, которое, чтобы удовлетворить некоторые требования дизайна, должно иметь возможность сбрасываться время от времени. Более того, предположим, что шаги, необходимые для этого сброса источника данных, известны только тому, кто дополнит и использует потом этот частичный класс, и реализует частичный метод. С учетом сказанного, возможное дополнение этого частичного класса его потребителем может выглядеть следующим образом:

```
using System;
public partial class DataSource
{
    partial void ResetSource() {
        Console.WriteLine( "Источник сброшен" );
    }
    public void Reset() {
        ResetSource();
    }
}
public class PartialMethods
{
    static void Main() {
        DataSource ds = new DataSource();
        ds.Reset();
    }
}
```

Вы можете видеть, что я добавил общедоступный метод по имени `Reset`, чтобы `Main` имел возможность сбрасывать экземпляры `DataSource`. Это связано с тем, что метод `ResetSource` неявно приватный. Если вы просмотрите полученный в результате исполняемый код с помощью ILDASM, то увидите там приватный метод `DataSource.ResetSource`, а если заглянете в `DataSource.Reset`, то увидите там вызов `ResetSource`. Если вы закомментируете или удалите частичную реализацию `ResetSource` и перекомпилируете код, то ILDASM покажет, что метод `DataSource.ResetSource` вообще отсутствует, и вызов `ResetSource` внутри метода `Reset` просто удален.

Статические классы

В C# 2.0 появился новый модификатор классов, позволяющий вам указать, что данный класс — не что иное, как коллекция статических членов, и не позволяет создавать его экземпляры. Способ добиться этого заключается просто в применении к классу модификатора `static`. Когда вы делаете это, то на этот класс накладываются следующие ограничения.

- Класс не может наследоваться ни от чего кроме `System.Object`, и если вы не специфицируете этот базовый тип, то такое наследование подразумевается.
- Класс не может служить базовым классом при создании другого класса.
- Класс может содержать только статические члены, которые могут быть общедоступными или приватными. Однако они не могут помечаться как `protected` или `protected internal`, поскольку этот класс не может служить базовым.
- Класс не может иметь никаких операций, поскольку их определение не имеет смысла, если невозможно создавать экземпляры данного класса.

Даже несмотря на то, что весь класс помечен как `static`, вы также помечаете каждый его индивидуальный член как `ststic`. Хотя компилятору было бы просто предположить, что любой член внутри статического класса также является статическим, это внесло бы излишнюю сложность в и без того сложный компилятор. Однако ограничения и вложенные типы, объявленные внутри статического класса

являются статическими по умолчанию. Однако если вы примените модификатор `static` к вложенному классу, он также будет статическим, как и содержащий его класс, а вложенные классы, не снабженные модификатором `static`, допускают создание их экземпляров.

На заметку! По сути, объявление класса как `static` — это то же самое, что объявление его как `sealed` и `abstract` одновременно, но компилятор не позволит вам сделать это. Однако если вы посмотрите на код IL, сгенерированный для статического класса, то увидите, что именно это компилятор и делает — т.е. статический класс в IL декорирован модификаторами `abstract` и `sealed`.

В следующем коде показан пример статического класса:

```
using System;
public static class StaticClass
{
    public static void DoWork() {
        ++callCount;
        Console.WriteLine( "StaticClass.DoWork()" );
    }
    public class NestedClass {
        public NestedClass() {
           Console.WriteLine( "NestedClass.NestedClass()" );
        }
    }
    private static long callCount = 0;
    public static long CallCount {
        get {
           return callCount;
        }
    }
}

public static class EntryPoint
{
    static void Main() {
        StaticClass.DoWork();
        // Это сделать невозможно!
        // StaticClass obj = new StaticClass();
        StaticClass.NestedClass nested =
           new StaticClass.NestedClass();
        Console.WriteLine( "CallCount = {0}",
           StaticClass.CallCount );
    }
}
```

Тип `StaticClass` содержит один метод, поле, свойство и вложенный класс. Обратите внимание, что поскольку `NestedClass` не объявлен как `static`, вы можете создавать его экземпляры, как и любого другого класса. К тому же, поскольку класс `EntryPoint` просто содержит метод `Main`, он также помечен как `static`, чтобы предотвратить непреднамеренное создание его экземпляров кем-либо.

Статические классы удобны, когда вам нужен логический механизм организации набора методов. Примером статического класса из библиотеки базовых классов (Base Class Library) может служить знаменитый класс `System.Console`. Он поддерживает статические методы, свойства и события, и все они являются статическими, поскольку только одна консоль может быть прикреплена к процессу в единицу времени.

Шаблон Singleton

Возможно, наиболее популярным шаблоном проектирования является Singleton ("Одиночный"), который обычно моделирует ситуацию, когда вы можете иметь только один экземпляр класса в единицу времени. Исторически сложилось так, что шаблон Singleton реализуется с помощью приватного конструктора и статического метода под названием что-нибудь вроде `GetInstance`, возвращающего ссылку на единственный допустимый работающий экземпляр. Хотя вы можете применять эту технику в C#, все же статический класс представляет собой отличное средство реализации шаблона Singleton, что доказывает пример `System.Console`.

Если не обязательно, чтобы ваш Singleton представлял собой экземпляр класса, то статический класс — наилучший способ его реализации. Например, если вам никогда не нужно уничтожать и воссоздавать ваш класс, и если вы не хотите использовать ваш Singleton вместе с .NET Remoting, то каждый домен приложения имеет свой собственный экземпляр Singleton, поскольку статические поля являются специфичными для домена приложения. Фактически такой Singleton не располагается в куче, и код, связанный с управлением единственным экземпляром, оказывается не нужным. Но что еще лучше — так это то, что поскольку это на самом деле не экземпляр объекта, вы можете использовать статический класс эффективно и безопасно внутри тела финализаторов объектов. В главе 13 я объясню, чем опасно использование объектов в финализаторах, и каким образом вы можете гарантировать порядок вызова финализаторов множества объектов.

Зарезервированные имена членов

Некоторые возможности, представленные в языке C# — это на самом деле "синтетический сахар", который превращается в вызовы методов в коде IL, которых вы никогда не видите, если только не открываете сгенерированную сборку с помощью инструмента, подобного ILDASM. Об этом нужно помнить, для того чтобы случайно не попытаться объявить метод, чье имя вступит в конфликт с одним из зарезервированных имен методов. К таким синтетическим сокращениям относятся свойства, события и индексаторы. Если вы попытаетесь объявлять метод с одним из этих специальных внутренних имен, имея одновременно свойство, событие или индексатор, объявляющее тот же метод внутри, то компилятор пожалуется на дублирование идентификаторов.

На заметку! Если вы следуете соглашениям, изложенным в руководстве Кшиштофа Квалины (Krzysztof Cwalina) и Брэда Абрамса (Brad Abrams) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Boston, MA: Addison-Wesley Professional, 2005 г.), либо применяете FxCop для регулярного анализа вашего кода, то вы никогда не столкнетесь с конфликтом имен между вашими именами членов классов и зарезервированными именами членов.

Зарезервированные имена для свойств

Для свойства по имени `Prop` типа `T` резервируются следующие сигнатуры для реализации свойства:

```
T get_Prop();
void set_Prop( T value );
```

Зарезервированные имена для индексаторов

Если класс содержит индексатор типа `T` и принимает список параметров, представленный `Params`, он будет содержать следующие зарезервированные имена методов:

```
T get_Item( Params );
void set_Item( Params, T value );
```

Зарезервированные имена для деструкторов

Если класс определен с финализатором (с применением синтаксиса деструктора), он будет содержать определение следующего метода:

```
void Finalize();
```

У меня есть еще много чего сказать о деструкторах и методе `Finalize` — как в этой главе, так и в главе 13.

Зарезервированные имена для событий

Если класс включает определение события типа `T` по имени `Event`, то в таком классе зарезервированы следующие методы:

```
void add_Event( T callback );
void remove_Event( T callback );
```

События мы рассмотрим в главе 10, когда я расскажу о делегатах и анонимных методах.

Определения типов значений

Тип значения — это легковесный тип, который обычно не создается в куче. Единственное исключение из этого правила — тип значения, являющийся полем объекта, которые находится в куче. Тип значения — это тип, поведение которого подчиняется семантике значения. То есть когда вы присваиваете переменную типа значения другой переменной типа значения, то содержимое исходной переменной копируется в переменную назначения, причем создается полная копия источника. Это отличает их от ссылочных типов, или экземпляров объектов, где результатом копирования одной переменной ссылочного типа в другую переменную ссылочного типа является новая ссылка на тот же самый объект. Также когда вы передаете тип значения в качестве параметра в метод, то тело метода получает локальную копию значения, если только параметр не был объявлен как `ref` или `out`. Все встроенные типы C# за исключением `string`, массивов и делегатов, являются типами значений. В C# вы объявляете тип значения, используя ключевое слово `struct` вместо `class`.

В целом синтаксис определения структуры точно такой, как у класса, но с некоторыми заметными исключениями, в чем вы вскоре убедитесь. Структура не объявляет базового класса. К тому же структура является неявно sealed. Это значит, что от структуры нельзя ничего унаследовать. Внутренне структура наследуется от класса System.ValueType, который, в свою очередь, расширяет System.Object. Это потому, что ValueType может предоставлять среди прочих реализации Object.Equals и Object.GetHashCode, что важно для типов значений. В разделе, озаглавленном "System.Object", я раскрою нюансы реализации методов, унаследованных типами значений от System.Object. Подобно классам, структуры могут быть объявлены частично, и к их частям применимы те же правила, что и к частичным классам.

Конструкторы

Типы, определенные как структуры, могут иметь статические конструкторы подобно классам. Структуры также могут иметь и конструкторы экземпляров, но с одним существенным исключением. Они не могут иметь определяемых пользователем умолчаний, конструкторов без параметров, а также инициализаторов полей экземпляра в определении структуры. Однако инициализаторы статических полей допускаются. Конструкторы без параметров типам значений не нужны, поскольку система обеспечивает их таковыми, просто устанавливая значения полей в их значения по умолчанию. Во всех случаях все биты таких полей устанавливаются в 0. Поэтому, если структура содержит поле int, его значением по умолчанию будет 0. Если структура имеет ссылочное поле, то его значение по умолчанию — null. Каждая структура получает этот неявный конструктор без параметров, который заботится об инициализации. Все это — часть попыток языка по обеспечению генерации верифицируемого и безопасного в отношении типов кода. Однако вполне возможно для пользователя объявлять структуру, вообще не вызывая конструктора. Если такое случается, программист отвечает за соответствующую установку данных структуры перед вызовом любых ее методов. Рассмотрим следующий код:

```
using System;
public struct Square
{
    // Иметь общедоступные поля - плохая идея, но я использую
    // их здесь только для примера. В реальном коде
    // применяйте вместо них свойства.
    public int width;
    public int height;
}
public class EntryPoint
{
    static void Main()
    {
        Square sq;
        sq.width = 1;
        // Пока этого сделать нельзя.
        // Console.WriteLine( "{0} x {1}", sq.width, sq.height );
        sq.height = 2;
        Console.WriteLine( "{0} x {1}", sq.width, sq.height );
    }
}
```

В Main я выделил пространство в стеке для объекта Square. Однако немедленно после этого я только присваиваю значение полю width. Я закомментировал следующий за этим вызов Console.WriteLine, поскольку он не скомпилируется. Причина в том, что вы не можете вызывать методы структуры прежде, чем она будет полностью инициализирована. На самом деле свойства "за кулисами" являются вызовами методов. После инициализации поля height я могу успешно использовать экземпляр Square для вывода width и height на консоль. Можете ли вы обнаружить недостаток в приведенном ниже коде?

```
using System;
public struct Square
{
    public int Width
    {
        get
        {
           return width;
        }
        set
        {
           width = value;
        }
    }

    public int Height
    {
        get
        {
           return height;
        }
        set
        {
           height = value;
        }
    }
    private int width;
    private int height;
}
public class EntryPoint
{
    static void Main()
    {
        Square sq;
        sq.Width = 1;
        sq.Height = 1;
    }
}
```

Подскажу: проблема кроется в методе Main. Если вы попытаетесь скомпилировать этот код, то компилятор прекратит работу с ошибкой. Вы не можете инициализировать поля, поскольку они являются private. Также вы не можете инициализи-

ровать их через свойства, поскольку свойства — на самом деле методы, а вызывать методы с не полностью инициализированным значением нельзя. Единственный способ разрубить этот гордиев узел — использовать ключевое слово new, когда вы объявляете новый экземпляр Square. Вы можете либо вызвать один из конструкторов структуры, либо конструктор по умолчанию. В данном случае я вызову конструктор по умолчанию, так что метод Main изменится следующим образом:

```
public class EntryPoint
{
    static void Main()
    {
        Square sq = new Square();
        sq.Width = 1;
        sq.Height = 1;
    }
}
```

Поскольку структура не может наследоваться от другой структуры или класса, вызовы каких-либо базовых конструкторов через ключевое слово base внутри блока конструктора не допускается. Даже несмотря на то, что вы знаете, что структура внутренне наследуется от System.ValueType, вы не можете вызывать конструктор базового типа явным образом.

Смысл this

Ранее я уже говорил, что ключевое слово this внутри методов класса ведет себя как константное, доступное только для чтения значение, содержащее ссылку на текущий экземпляр объекта. Другими словами, это доступная только для чтения ссылка на объект в методах класса. Однако с типами значений this ведет себя как обычный параметр ref. В конструкторах экземпляра, где нет списка инициализации, значение this ведет себя как параметр out. Это значит, что вы действительно присваиваете значение this, как показано в следующем примере:

```
public struct ComplexNumber
{
    public ComplexNumber( double real, double imaginary )
    {
        this.real = real;
        this.imaginary = imaginary;
    }
    public ComplexNumber( ComplexNumber other )
    {
        this = other;
    }
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main()
    {
```

```
            ComplexNumber valA = new ComplexNumber( 1, 2 );
            ComplexNumber copyA = new ComplexNumber( valA );
        }
    }
```

Обратите внимание, что второй конструктор принимает в качестве параметра другое значение ComplexNumber. Этот конструктор ведет себя подобно копирующему конструктору в C++. Но вместо присваивания каждого поля индивидуально вы можете просто присвоить его целиком this, таким образом копируя состояние параметра в единственной строке кода. Опять-таки ключевое слово this действует в этом случае как параметр out.

Напомню, что параметры out ведут себя подобно параметрам ref, но с одним специальным отличием. Когда параметр помечается как out, компилятор знает, что значение не инициализировано в точке, где начинается выполнение тела метода. Поэтому компилятор должен удостовериться, что каждое поле значения инициализировано перед тем, как произойдет выход из конструктора. Например, рассмотрим следующий код, который не компилируется:

```
public struct ComplexNumber
{
    public ComplexNumber( double real, double imaginary )
    {
        this.real = real;
        this.imaginary = imaginary;
    }
    public ComplexNumber( double real )
    {
        this.real = real;
    }
    private double real;
    private double imaginary;
}
```

Проблема с этим кодом заключается во втором конструкторе. Поскольку типы значений обычно создаются в стеке, выделение таких значений просто требует изменения указателя стека. Конечно, выделение такого рода ничего не говорит о состоянии памяти. Дело в том, что память, зарезервированная в стеке для значения, содержит случайный мусор. CLR мог бы инициализировать эти блоки памяти нулями, но это свело бы на нет половину преимуществ типов значений. Типы значений задуманы как легковесные и быстрые. Если CLR придется инициализировать нулями стековую память для типов значений при каждом резервировании памяти для них, это будет достаточно длительной операцией. Конечно, конструктор по умолчанию без параметров, сгенерированный системой, именно это и делает. Но вы должны вызывать его явно, создавая экземпляр ключевым словом new. Поскольку ключевое слово this трактуется как out-параметр в конструкторах экземпляра, конструктор экземпляра должен инициализировать каждое поле значения перед тем, как завершить свою работу. И это обязанность компилятора C#, от которого ожидается, что он сгенерирует верифицируемый и безопасный в отношении типов код, чтобы убедиться в том, что вы это сделали. Вот почему предыдущий пример кода приводит к ошибке компиляции.

Несмотря на то что конструкторы экземпляров типов значений не могут использовать ключевое слово `base` для вызова конструкторов базового класса, они имеют инициализатор. Инициализатору разрешено использовать ключевое слово `this`, чтобы вызывать другие конструкторы той же структуры во время инициализации. Поэтому вы можете внести одно небольшое изменения в код предыдущего примера, чтобы он начал компилироваться:

```
public struct ComplexNumber
{
    public ComplexNumber( double real, double imaginary )
    {
        this.real = real;
        this.imaginary = imaginary;
    }
    public ComplexNumber( double real )
        :this( real, 0 )
    {
        this.real = real;
    }
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main()
    {
        ComplexNumber valA = new ComplexNumber( 1, 2 );
    }
}
```

Обратите внимание на отличие во втором конструкторе. Я поместил туда инициализатор, вызывающий первый конструктор из второго. Даже несмотря на то, что во втором конструкторе одна строка кода лишня, я оставил ее, чтобы подтвердить мысль. Заметьте, что она только присваивает значение `real`, как и в предыдущем примере, но компилятор уже не жалуется. Это объясняется тем, что когда конструктор экземпляра содержит инициализатор, ключевое слово `this` ведет себя в теле конструктора как параметр `ref`, а не как параметр `out`. И, поскольку это `ref`-параметр, компилятор может предположить, что значение было инициализировано правильно перед входом в блок кода метода. По сути, бремя инициализации возлагается на первый конструктор, чья обязанность — убедиться в том, что он инициализировал все поля значения.

И последнее замечание, которое нужно принять во внимание — что даже невзирая на то, что система генерирует "под ковром" стандартный, лишенный параметров инициализатор, вы не можете вызвать его через ключевое слово `this`. Например, следующий код не скомпилируется:

```
public struct ComplexNumber
{
    public ComplexNumber( double real, double imaginary )
    {
        this.real = real;
```

```
            this.imaginary = imaginary;
        }
        public ComplexNumber( double real )
               :this()
        {
            this.real = real;
        }
        private double real;
        private double imaginary;
    }
```

Если у вас была бы структура, имеющая относительно немного полей, и вы хотели бы инициализировать все кроме одного из них нулями или null-ссылками, это позволило бы вам немного сэкономить на вводе кода. Но, увы, компилятор этого не позволяет.

Финализаторы

Типам значений не разрешено иметь финализатор. Концепция финализации, или недетерминированного уничтожения, зарезервирована для экземпляров классов, или объектов. Если бы структуры имели финализаторы, то исполняющая система должна была бы управлять их вызовом при каждом выходе значения из области определения.

Имейте в виду, что нужно быть аккуратным с инициализацией ресурсов внутри конструкторов типов значений. Просто выполняйте ее. Рассмотрим тип значения, имеющий поле — дескриптор определенного рода низкоуровневого системного ресурса. Предположим, что этот низкоуровневый ресурс выделен, или захвачен, в специальном конструкторе, принимающем параметры. Теперь у вас есть пара проблем, которые нужно решить. Поскольку вы не можете создать конструктор по умолчанию, не имеющий параметров, как можно захватить ресурс, когда пользователь создает экземпляр этого значения, не используя специальных конструкторов? Ответ — никак! Вторая проблема состоит в том, что у вас нет автоматического триггера для очистки и освобождения ресурса, поскольку нет деструктора. Вам нужно как-то заставить потребителя значения вызвать некоторый специальный метод для очистки, прежде чем значение выйдет из контекста. Требование к потребителю не забыть сделать что-то подобное свидетельствует о плохом дизайне.

Интерфейсы

Хотя структуре не разрешается наследоваться от некоторого класса, но она может реализовывать интерфейсы. Поддерживаемые интерфейсы перечисляются точно так же, как у классов, в списке базовых интерфейсов после идентификатора структуры. Обычно поддержка интерфейсов для структур — это то же самое, что поддержка интерфейсов для классов. Тема интерфейсов подробно раскрывается в главе 5. Реализация интерфейсов структурами влияет на производительность, поскольку подразумевает операции упаковки при вызове методов через интерфейсную ссылку на экземплярах структур.

Анонимные типы

Насколько часто у вас возникала потребность в легковесном классе, хранящем несколько взаимосвязанных значений для использования внутри определенного метода, и вы стонали от необходимости вводить целое определение типа, с приватными полями и общедоступными свойствами? Так обратитесь к анонимным типам! C# 3.0 позволяет вам вводить такие типы, используя расширенный синтаксис операции new. Посмотрим, на что это похоже:

```
using System;
public class EntryPoint
{
    static void Main() {
        var employeeInfo = new { Name = "Joe", Id = 42 };
        var customerInfo = new { Name = "Jane", Id = "AB123" };
        Console.WriteLine( "Name: {0}, Id: {1}",
           employeeInfo.Name,
           employeeInfo.Id );
        Console.WriteLine( "Типом employeeInfo сейчас является: {0}",
           employeeInfo.GetType() );
        Console.WriteLine( "Типом customerInfo сейчас является: {0}",
           customerInfo.GetType() );
    }
}
```


```
using System;
public class EntryPoint
{
    static void Main() {
        var employeeInfo = new { Name = "Joe", Id = 42 };
        var customerInfo = new { Name = "Jane", Id = "AB123" };
        Console.WriteLine( "Name: {0}, Id: {1}",
           employeeInfo.Name,
           employeeInfo.Id );
        Console.WriteLine( "Типом employeeInfo сейчас является: {0}",
           employeeInfo.GetType() );
        Console.WriteLine( "Типом customerInfo сейчас является: {0}",
           customerInfo.GetType() );
    }
}
```


Обратите внимание на новый интересный синтаксис внутри скобок после ключевого слова new внутри объявления employeeInfo. Пары имя/значение объявляют имя свойства внутри анонимного типа и инициализируют его заданным значением. В данном случае два создается два анонимных типа с двумя свойствами. В первом анонимном типе первое свойство типа System.String по имени Name, а второе — типа System.Int32 под названием Id. Важно отметить, что лежащий в основе тип экземпляра созданного таким образом является строгим типом, но его генерирует компилятор, и вы не знаете его имени. Но как видно из следующего вывода, полученного от приведенного выше кода, вы можете узнать имя этого типа:

```
Name: Joe, Id: 42
Типом employeeInfo сейчас является: <>f__AnonymousType0`2[System.
String,System.Int32]
Типом customerInfo сейчас является: <>f__AnonymousType0`2[System.
String,System.String]
```

На заметку! Имена сгенерированных компилятором типов специфичны для реализации, поэтому вы никогда не должны полагаться на них. Вдобавок, как вы заметили, они "непроизносимы" для компилятора; если вы попытаетесь объявлять экземпляр, используя такое имя типа, то компилятор пожалуется на синтаксическую ошибку.

Поскольку вы не знаете сгенерированного компилятором имени типа, вы вынуждены объявлять экземпляр переменной как неявно типизированную локальную переменную, с применением ключевого слова var, как это делалось в коде примера.

К тому же обратите внимание, что сгенерированный компилятором тип является обобщенным, принимающим два параметра типа. Было бы неэффективно для компилятора генерировать новый тип для каждого анонимного типа, содержащего два типа с одними и теми же именами полей. Вывод, приведенный выше, указывает на то, что действительный тип employeeInfo выглядит так:

```
<>f__AnonymousType0<System.String, System.Int32>
```

И поскольку анонимный тип для customerInfo содержит столько же полей с теми же именами, сгенерированный обобщенный тип используется повторно, и тип customerInfo выглядит следующим образом:

```
<>f__AnonymousType0<System.String, System.String>
```

Если бы анонимный тип для customerInfo включал поля с именами, отличными от используемых в employeeInfo, то был бы объявлен другой обобщенный анонимный тип.

Теперь, когда вы ознакомились с основами анонимных типов, я хочу показать сокращенный синтаксис для их объявления. Обратите внимание на операторы, выделенные полужирным в следующем примере:

```
using System;
public class ConventionalEmployeeInfo
{
    public ConventionalEmployeeInfo( string Name, int Id ) {
        this.name = Name;
        this.id = Id;
    }
    public string Name {
        get {
           return name;
        }
        set {
           name = value;
        }
    }
    public int Id {
        get {
           return id;
        }
```

```
            set {
               id = value;
            }
        }
        private string name;
        private int id;
    }
    public class EntryPoint
    {
        static void Main() {
            ConventionalEmployeeInfo oldEmployee =
               new ConventionalEmployeeInfo( "Joe", 42 );
            var employeeInfo = new { oldEmployee.Name,
                                     oldEmployee.Id };
            string Name = "Jane";
            int Id = 1234;
            var customerInfo = new { Name, Id };
            Console.WriteLine( "employeeInfo Name: {0}, Id: {1}",
                               employeeInfo.Name,
                               employeeInfo.Id );
            Console.WriteLine( "customerInfo Name: {0}, Id: {1}",
                               customerInfo.Name,
                               customerInfo.Id );
            Console.WriteLine( "Анонимным типом сейчас является: {0}",
                               employeeInfo.GetType() );
        }
    }
```

В целях иллюстрации я объявил тип по имени ConventionalEmployeeInfo, который не является анонимным. Обратите внимание, что в точке, где создается экземпляр анонимного типа для employeeInfo, я не указал имен полей, как раньше. В данном случае компилятор использует имена свойств типа ConventionalEmployeeInfo, который является источником данных. Этот же прием работает с применением локальных переменных, как вы можете видеть в объявлении экземпляра customerInfo. В этом случае customerInfo — анонимный тип, реализующий два свойства, доступных для чтения/записи по имени Name и Id. Деклараторы членов анонимных типов, использующих такой сокращенный стиль, называются инициализаторами проекции (projection initializers)[1].

Если вы просмотрите скомпилированную сборку в ILDASM, то заметите, что сгенерированные типы для анонимных типов — это классы. Каждый такой класс также помечен как private и sealed. Однако класс этот исключительно примитивен, и не реализует ничего подобного финализатору или IDisposable.

На заметку! Анонимные типы, даже будучи классами, не реализуют интерфейса IDisposable. Как будет упомянуто в главе 13, общее требование для типов, содержащих одноразовые (disposable) типы, заключается в том, что они тоже должны быть одноразовыми. Но поскольку анонимные типы таковыми не являются, вы должны избегать помещения экземпляров одноразовых типов внутри них.

---

[1] Инициализаторы проекции очень удобны, когда они используются с LINQ, о чем будет рассказано в главе 16.

Не вскрывайте тип анонимных типов. Например, если вы помещаете экземпляры анонимных типов в System.List, каким образом вы собираетесь выполнить приведение этих экземпляров обратно к анонимному типу, когда обратитесь к ним позднее? Помните, что System.List хранит ссылки на System.Object. И даже несмотря на то, что анонимные типы наследуются от System.Object, как привести их обратно к конкретным типам, чтобы обратиться к их свойствам? Вы можете попытаться применить рефлексию, чтобы преодолеть это. Но это потребует настолько много работы, что вы потеряете весь выигрыш от использования анонимных типов. Аналогично, если вы хотите передавать экземпляры анонимных типов из функций через выходные параметры или оператором return, вы должны передавать их как ссылки на System.Object, таким образом, избавляя переменные от полезной информации о типе. Если вы должны передавать экземпляры из метода, то вам следует применять вместо анонимного типа явно определенный тип, такой как ConventionalEmployeeInfo.

Узнав обо всех этих ограничениях, касающихся анонимных типов, вы можете удивиться — в чем вообще их польза, за исключением тех редких случаев внутри локального контекста? Однако выясняется, что они чрезвычайно полезны при использовании с операциями проекций в языке LINQ, что и будет продемонстрировано в главе 16.

Инициализаторы объектов

В C# 3.0 появилось сокращение, которое вы можете использовать при инициализации новых экземпляров объектов. Часто ли вам приходилось писать код, подобный приведенному ниже?

```
Employee developer = new Employee();
developer.Name = "Fred Blaze";
developer.OfficeLocation = "B1";
```

Сразу после создания экземпляра Employee вы немедленно начинаете инициализацию доступных свойств экземпляра. Не правда ли, было бы здорово, если бы можно было делать все это в одном операторе? Конечно, вы всегда можете создать специализированную перегрузку конструктора, принимающего параметры для инициализации нового экземпляра. Однако бывают случаи, когда удобнее этого не делать.

На заметку! Если ваш тип содержит одно или более доступных только для чтения автореализуемых свойств, вы должны использовать инициализатор объекта, чтобы инициализировать свойства во время создания экземпляра.

Новый синтаксис инициализатора объекта показан ниже:

```
using System;
public class Employee
{
    public string Name {
        get; set;
    }
```

```
    public string OfficeLocation {
        get; set;
    }
}
public class InitExample
{
    static void Main() {
        Employee developer = new Employee {
           Name = "Fred Blaze",
           OfficeLocation = "B1"
        };
    }
}
```

Обратите внимание на то, как инициализируется экземпляр developer в методе Main. "За кулисами" компилятор генерирует тот же код, какой получился бы в случае, если бы вы инициализировали свойства вручную после создания экземпляра Employee. Поэтому такая техника работает только в случае доступности свойств — в данном случае Name и OfficeLocation — в точке инициализации.

Инициализаторы объектов можно даже вкладывать друг в друга, как показано в следующем примере:

```
using System;
public class Employee
{
    public string Name { get; set; }
    public string OfficeLocation { get; set; }
}
public class FeatureDevPair
{
    public Employee Developer { get; set; }
    public Employee QaEngineer { get; set; }
}
public class InitExample
{
    static void Main() {
        FeatureDevPair spellCheckerTeam = new FeatureDevPair {
           Developer = new Employee {
              Name = "Fred Blaze",
              OfficeLocation = "B1"
           },
           QaEngineer = new Employee {
              Name = "Marisa Bozza",
              OfficeLocation = "L42"
           }
        };
    }
}
```

Обратите внимание, как инициализируются два свойства spellCheckerTeam с использованием нового синтаксиса. Каждый экземпляр Employee, присваиваемый свойствам, сам инициализируется посредством объектного инициализатора. И, на-

конец, посмотрите, как выглядит еще более сокращенный способ инициализации объектов, позволяющий сократить объем клавишного ввода за счет скрытого усложнения:

```
using System;
public class Employee
{
    public string Name { get; set; }
    public string OfficeLocation { get; set; }
}
public class FeatureDevPair
{
    private Employee developer = new Employee();
    private Employee qaEngineer = new Employee();
    public Employee Developer {
        get { return developer; }
        set { developer = value; }
    }
    public Employee QaEngineer {
        get { return qaEngineer; }
        set { qaEngineer = value; }
    }
}
public class InitExample
{
    static void Main() {
        FeatureDevPair spellCheckerTeam = new FeatureDevPair {
            Developer = {
                Name = "Fred Blaze",
                OfficeLocation = "B1"
            },
            QaEngineer = {
                Name = "Marisa Bozza",
                OfficeLocation = "L42"
            }
        };
    }
}
```

Обратите внимание на то, что я смог пропустить выражения `new` при инициализации свойств `Developer` и `QaEngineer` объекта `spellCheckerTeam`. Однако такой сокращенный синтаксис требует, чтобы поля `spellCheckerTeam` уже существовали перед установкой свойств, т.е. эти поля не должны иметь значение `null`. Поэтому вы видите, что мне пришлось изменить определение `FeatureDevPair`, чтобы создать содержащиеся экземпляры `Employee` в точке инициализации.

На заметку! Если вы не инициализируете поля, представляемые свойствами, во время инициализации объекта, а затем позже напишете код, инициализирующий экземпляры этих объектов с использованием показанного выше сокращенного синтаксиса, то во время выполнения столкнетесь с неприятным сюрпризом. Вы можете ожидать, что в таких случаях код должен генерировать исключение `NullReferenceException`. К сожалению, компилятор не может обнаружить

эту потенциальную проблему во время компиляции. Поэтому вам следует быть очень осторожными при использовании сокращенного синтаксиса, показанного выше. Например, если вы используете этот синтаксис для инициализации экземпляров объектов, которые писали не вы, то нужно быть еще осторожней, потому что, если только вы не просмотрите реализацию этого чужого класса в ILDASM, у вас нет никакой возможности узнать, инициализируются поля во время инициализации объекта или нет.

Упаковка и распаковка

Теперь давайте рассмотрим упаковку и распаковку. Все типы внутри CLR относятся к двум категориям: ссылочные типы (объекты) и типы значений (значения). Вы определяете объекты с помощью классов, а значения — с помощью структур. Между этими двумя группами есть четкое разделение. Объекты находятся в памяти кучи и управляются сборщиком мусора. Значения обычно располагаются в пространстве временного хранения, таком как стек. Единственное достойное упоминания исключение, о котором я уже говорил, состоит в том, что тип значения может находиться в куче, если он хранится в виде поля объекта. Однако он не является автономным, и GC не контролирует время его существования напрямую. Рассмотрим следующий код:

```
public class EntryPoint
{
    static void Print( object obj )
    {
        System.Console.WriteLine( "{0}", obj.ToString() );
    }
    static void Main()
    {
        int x = 42;
        Print( x );
    }
}
```

Выглядит достаточно просто. В `Main` есть `int`, который в C# является псевдонимом для `System.Int32` и представляет собой тип значения. Вы могли бы также объявлять переменную `x` как относящуюся к типу `System.Int32`. Место, выделенное для `x`, находится в локальном стеке. Вы передаете ее в виде параметра методу `Print`. Метод `Print` принимает ссылку на `object` и просто посылает результат вызова `ToString` на этом объекте на консоль. Давайте проанализируем детали. `Print` принимает ссылку на расположенный в куче объект. Но вы передаете методу тип значения. Что здесь же происходит? Как такое возможно?

Ключ лежит в концепции, называемой *упаковкой* (boxing). В точке, где определяется тип значения, CLR создает во время выполнения класс-оболочку для помещения в него типа значения. Экземпляры этой оболочки находятся в куче и обычно называются упаковочными объектами. Это способ, которым CLR преодолевает зазор между типами значений и ссылочными типами. Фактически, если вы используете ILDASM, чтобы заглянуть в код IL, сгенерированный для метода `Main`, то увидите там следующее:

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 15 (0xf)
    .maxstack 1
    .locals init (int32 V_0)
    IL_0000: ldc.i4.s 42
    IL_0002: stloc.0
    IL_0003: ldloc.0
    IL_0004: box [mscorlib]System.Int32
    IL_0009: call void EntryPoint::Print(object)
    IL_000e: ret
} // end of method EntryPoint::Main
```

Обратите внимание на IL-инструкцию box, которая выполняет операцию упаковки перед вызовом метода Print. При этом создается объект, показанный на рис. 4.2.

На рис. 4.2 отображено действие копирования типа значения в упаковочный объект, находящийся в куче. Упаковочный объект ведет себя подобно любому другому ссылочному типу в CLR. К тому же отметьте, что упаковочный тип реализует интерфейсы содержащегося типа значений. Упаковочный тип — это тип класса, сгенерированный внутри виртуальной исполняющей системы CLR в точке определения содержащегося типа значений. Затем CLR использует этот внутренний тип класса при выполнении необходимых операций упаковки.

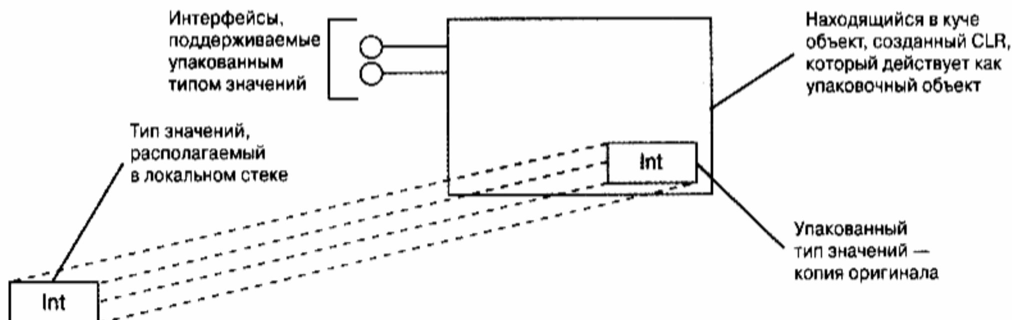

Рис. 4.2. Результат операции упаковки

Наиболее важный момент, касающийся упаковки, который следует иметь в виду — это то, что упакованное значение является копией оригинала. Поэтому любые изменения значения внутри упаковки не распространяются обратно на исходное значение. Например, рассмотрим следующую небольшую модификацию предыдущего кода:

```
public class EntryPoint
{
    static void PrintAndModify( object obj )
    {
        System.Console.WriteLine( "{0}", obj.ToString() );
        int x = (int) obj;
        x = 21;
    }
```

```
        static void Main()
        {
            int x = 42;
            PrintAndModify( x );
            PrintAndModify( x );
        }
    }
```

Вывод этого кода может вас удивить:

```
42
42
```

Фактически оригинальное значение переменной x, объявленной и инициализированной в Main, никогда не изменяется. Когда вы передаете его методу PrintAndModify, оно упаковывается, поскольку метод PrintAndModify принимает object в качестве параметра. Несмотря на то что метод PrintAndModify принимает ссылку на объект, который вы можете модифицировать, принятый им объект является упаковочным, содержащим копию исходного значения. Код также вводит другую операцию, называемую *распаковкой* (unboxing), в метод PrintAndModify. Поскольку значение упаковывается внутри экземпляра объекта в куче, вы не можете изменить это значение, поскольку методы, поддерживаемые объектом — это только те методы, которые реализует System.Object. Технически он также поддерживает те же интерфейсы, что и System.Int32. Поэтому вам нужен какой-то способ получения значения из упаковки. В C# вы можете сделать это синтаксически — посредством приведения. Обратите внимание, что вы приводите экземпляр объекта обратно к int, и компилятор достаточно интеллектуален, чтобы понять, что на самом деле вы хотите распаковать тип значения, и применяет IL-инструкцию unbox, что видно в следующем коде IL метода PrintAndModify:

```
.method private hidebysig static void PrintAndModify(object obj) cil managed
{
    // Code size 28 (0x1c)
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: ldstr "{0}"
    IL_0005: ldarg.0
    IL_0006: callvirt instance string [mscorlib]System.Object::ToString()
    IL_000b: call void [mscorlib]System.Console::WriteLine(string, object)
    IL_0010: ldarg.0
    IL_0011: unbox [mscorlib]System.Int32
    IL_0016: ldind.i4
    IL_0017: stloc.0
    IL_0018: ldc.i4.s 21
    IL_001a: stloc.0
    IL_001b: ret
} // end of method EntryPoint::PrintAndModify
```

Позвольте мне внести ясность относительно того, что происходит во время распаковки в C#. Операция распаковки значения в точности противоположна упаковке. Значение из упаковки копируется в экземпляр значения в локальном стеке. Опять-таки любые изменения, внесенные в эту распакованную копию, не распро-

страняются обратно на значение, содержащееся в упаковке. Как видите, упаковка и распаковка может показаться довольно запутанной. Как было показано, поведение кода не очевидно для случайного наблюдателя, не знакомого с тем фактом, что упаковка и распаковка происходит внутренне. Что еще хуже, так это то, что две копии int создаются между моментом инициации вызова PrintAndModify и моментом манипулирования этим int в методе. Первая копия — та, что помещается в упаковку. Вторая копия — та, что создается, когда упакованное значение копируется из упаковки.

Технически можно модифицировать значение, содержащееся в упаковке. Однако вы должны делать это через интерфейс. Сгенерированная во время выполнения упаковка, содержащая значение, также реализует интерфейс, реализуемый типом значения, и передает вызовы содержащемуся значению. Поэтому вы могли бы поступить следующим образом:

```
public interface IModifyMyValue
{
    int X
    {
        get;
        set;
    }
}
public struct MyValue : IModifyMyValue
{
    public int x;
    public int X
    {
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }
    public override string ToString()
    {
        System.Text.StringBuilder output =
        new System.Text.StringBuilder();
        output.AppendFormat( "{0}", x );
        return output.ToString();
    }
}
public class EntryPoint
{
    static void Main()
    {
        // создать значение
        MyValue myval = new MyValue();
        myval.x = 123;
```

```
        // упаковать его
        object obj = myval;
        System.Console.WriteLine( "{0}", obj.ToString() );
        // модифицировать содержимое упаковки
        IModifyMyValue iface = (IModifyMyValue) obj;
        iface.X = 456;
        System.Console.WriteLine( "{0}", obj.ToString() );
        // распаковать и посмотреть — что получилось.
        MyValue newval = (MyValue) obj;
        System.Console.WriteLine( "{0}", newval.ToString() );
    }
}
```

Запустив этот код на выполнение, вы получите следующий вывод:

```
123
456
456
```

Как и ожидалось, вы можете модифицировать значение внутри упаковки, используя интерфейс по имени `IModifyMyValue`. Однако это не самый прямолинейный процесс. К тому же имейте в виду, что перед тем, как получить ссылку на интерфейс для типа значения, оно должно быть упаковано. Это имеет смысл, если вспомнить, что ссылки на интерфейсы — это также ссылочные типы.

Когда происходит упаковка

Поскольку C# обрабатывает упаковку для вас неявно, важно знать случаи, когда C# упаковывает значение. В основном значение упаковывается при следующих преобразованиях:

- преобразование типа значения в объектную ссылку;
- преобразование типа значения в ссылку на `System.ValueType`;
- преобразование типа значения в ссылку на интерфейс, реализованный этим типом значения;
- преобразование типа `enum` в ссылку на `System.Enum`.

В каждом случае преобразование обычно принимает форму выражения присваивания. Первые два случая довольно очевидны, поскольку CLR здесь заполняет зазор между двумя категориями типов, превращая экземпляр типа значения в ссылочный тип. Третий случай может показаться несколько неожиданным. Всякий раз, когда вы явно приводите ваше значение к поддерживаемому им интерфейсу, происходит упаковка. Рассмотрим следующий код:

```
public interface IPrint
{
    void Print();
}
public struct MyValue : IPrint
{
    public int x;
```

```
        public void Print()
        {
            System.Console.WriteLine( "{0}", x );
        }
    }
    public class EntryPoint
    {
        static void Main()
        {
            MyValue myval = new MyValue();
            myval.x = 123;
            // нет упаковки
            myval.Print();
            // нужно упаковать значение
            IPrint printer = myval;
            printer.Print();
        }
    }
```

Первый вызов Print происходит через ссылку на значение, что не требует упаковки. Однако второй вызов Print происходит через интерфейс. В точке получения интерфейса происходит упаковка. Первым делом это все выглядит так, будто без упаковки можно было бы обойтись, если применять явной ссылки на тип интерфейса. В данном случае это верно, поскольку Print также является частью общедоступного контракта MyValue. Однако если бы вы реализовали метод Print как явный интерфейс, о чем речь пойдет в главе 5, то тогда единственный способ вызова метода был бы через ссылку на интерфейсный тип. Поэтому важно отметить, что всякий раз, когда вы реализуете интерфейс на типе значений явно, то тем самым вынуждаете клиентов вашего типа значений упаковывать его перед вызовом через этот интерфейс. Следующий пример демонстрирует это:

```
public interface IPrint
{
    void Print();
}
public struct MyValue : IPrint
{
    public int x;
    void IPrint.Print()
    {
        System.Console.WriteLine( "{0}", x );
    }
}
public class EntryPoint
{
    static void Main()
    {
        MyValue myval = new MyValue();
        myval.x = 123;
        // нужно упаковать значение
        IPrint printer = myval;
        printer.Print();
    }
}
```

В качестве другого примера рассмотрим поддержку типом `System.Int32` интерфейса `IConvertable`. Однако большинство методов интерфейса `IConvertable` реализовано явно. Поэтому, даже если вы хотите вызвать метод `IConvertable`, такой как `Iconvertible.ToBoolean`, на простом `int`, то сначала должны упаковать его.

На заметку! Обычно для выполнения преобразований, подобных упомянутым ранее, вы захотите полагаться на внешний класс `System.Convert`. Я описал вызов через `IConvertable` только в качестве примера.

Эффективность и путаница

Как вы могли бы ожидать, упаковка и распаковка — не самые эффективные операции в мире. Еще хуже то, что компилятор C# молча выполняет эти операции за вас. Вам действительно нужно знать о том, когда происходит упаковка. Распаковка обычно более явна, поскольку обычно вам приходится выполнять операцию приведения для извлечения значения из упаковки, хотя бывают и случаи неявной распаковки, которые будут описаны далее. В любом случае вы должны уделить внимание аспекту эффективности. Например, рассмотрим контейнерный тип, подобный `System.Collection.ArrayList`. Все значения он хранит в виде ссылок на тип `object`. Если вам нужно вставить в него множество типов значений, то все они будут упакованы! К счастью, обобщения, которые появились в C# 2.0 и .NET 2.0 и описаны в главе 10, помогут преодолеть эту неэффективность. Однако вы всегда должны помнить, что упаковка — неэффективная операция, которой следует избегать, где только возможно. К сожалению, поскольку упаковка — неявная операция в C#, ее обнаружение требует острого глаза. Наилучший инструмент, который можно для этого применить, когда вы сомневаетесь в ее наличии или отсутствии — это ILDASM. Применяя ILDASM, вы можете проверить код IL, сгенерированный для ваших методов, и легко обнаружить там операции упаковки. Утилиту `ILDASM.exe` вы найдете в папке `\bin` .NET SDK.

Как упоминалось ранее, распаковка — обычно явная операция, происходящая во время приведения от ссылки на объект упаковки к типу упакованного значения. Тем не менее, в одном случае распаковка все-таки бывает неявной. Помните, что я говорил об отличии в поведении ссылки `this` внутри методов классов и внутри методов структур? Главное отличие в том, что для типов значений ссылка `this` действует как параметр `ref` или `out` — в зависимости от ситуации. Так что когда вы вызываете метод на типе значения, то скрытый параметр `this` внутри метода должен быть управляемым указателем, а не ссылкой. Компилятор легко справляется с этим, когда вы осуществляете вызов непосредственно через экземпляр типа значения. Однако при вызове виртуального метода или метода интерфейса через упакованный экземпляр, т.е. через объект, то CLR приходится распаковывать экземпляр значения, чтобы получить управляемый указатель на тип значения, содержащийся в упаковке. Не забывайте о скрытых операциях распаковки, если вызываете методы значения через упаковочный объект.

На заметку! Операции распаковки в CLR неэффективны сами по себе. Эта неэффективность происходит из того факта, что C# обычно комбинирует операцию распаковки с копированием значения.

System.Object

Каждый объект в CLR наследуется от System.Object. Object — базового типа для всех других типов. В C# ключевое слово object — это псевдоним System.Object. То, что каждый тип в CLR и C# наследуется от Object, может оказаться удобным. Например, вы можете трактовать коллекцию экземпляров разных типов как однородную, приведя их все к ссылкам на Object.

Даже System.ValueType наследуется от Object. Однако получение ссылки на Object регулируется некоторыми специальными правилами. На ссылочных типах вы можете преобразовывать ссылку на класс A в ссылку на класс Object простым неявным преобразованием. Проход в обратном направлении требует проверки типа во время выполнения и явного приведения с использованием знакомого синтаксиса — предваряя преобразуемый экземпляр именем нового типа в скобках. Прямое получение ссылки Object для типа значения технически невозможно. Семантически это оправдано, поскольку типы значений расположены в стеке. Было бы опасно получить ссылку на кратковременно существующий экземпляр значения и сохранить ее для последующего использования, когда этот экземпляр значения уже, вероятно, исчезнет. По этой причине получение ссылки на Object для типов значений сопряжено с операцией упаковки, как было описано в предыдущем разделе.

Определения класса System.Object выглядит так:

```
public class Object
{
    public Object();
    public virtual void Finalize();
    public virtual bool Equals( object obj );
    public static bool Equals( object obj1, object obj2 );
    public virtual int GetHashCode();
    public Type GetType();
    protected object MemberwiseClone();
    public static bool ReferenceEquals( object obj1, object obj2 );
    public virtual string ToString();
}
```

Object предоставляет несколько методов, которые проектировщики CLI/CLR сочли важными и подходящими к каждому объекту. Методы, имеющие дело с эквивалентностью, требуют отдельного рассмотрения; они будут описаны в следующем разделе. Object предусматривает метод GetType для получения типа времени выполнения любого объекта, работающего в CLR. Такая возможность чрезвычайно удобна в сочетании с рефлексией — определять типы в системе во время выполнения. GetType возвращает объект типа Type, который представляет реальный, или конкретный, тип объекта. Используя этот возвращенный объект, вы можете узнать все о типе объекта, с которым был вызван метод GetType. К тому же, имея две ссылки на тип Object, вы можете сравнивать результат вызова GetType для них обоих, чтобы узнать, являются ли они экземплярами одного и того же типа.

System.Object содержит метод по имени MemberwiseClone, возвращающий неглубокую копию объекта. У меня еще будет, что сказать об этом методе в главе 13. Когда метод MemberwiseClone создает копию, все поля типов значений копируют-

ся бит за битом, в то время как все поля ссылочного типа просто копируются, так что и оригинал, и копия содержат ссылки на один и тот же объект. Когда вы хотите получить копию объекта, такое поведение не обязательно вас устроит. Поэтому если объекты поддерживают копирование, вы должны рассмотреть поддержку `ICloneable` и сделать то, что надо, в реализации интерфейса. К тому же заметьте, что этот метод объявлен как `protected`. Главная причина состоит в том, что только класс копируемого объекта может вызвать его, поскольку `MemberwiseClone` может создать объект без вызова его конструктора экземпляра. Такое поведение потенциально могло бы стать дестабилизирующим, если сделать его `public`.

На заметку! Прежде чем решать реализовывать интерфейс `ICloneable`, узнайте о нем больше в главе 13.

Четыре из методов `Object` являются виртуальными, и если их реализация в `Object` по умолчанию не подходит, вы можете переопределить их. Метод `ToString` удобен для генерации текстового, читабельного для человека вывода и строкового представления объекта. Например, во время разработки вам может понадобиться возможность трассировки объекта в отладочном выводе во время выполнения. В таких случаях имеет смысл переопределить `ToString`, чтобы он предоставлял детальную информацию об объекте и его внутреннем состоянии. Версия `ToString` по умолчанию просто вызывает реализацию `ToString` объекта `Type`, возвращенного `GetType`, таким образом возвращая только имя типа объекта. Это лучше, чем ничего, но наверно для вас будет недостаточно полезно, если вы хотите вызывать `ToString` на объекте[2]. Постарайтесь избежать добавления сторонних эффектов в реализацию `ToString`, поскольку отладчик Visual Studio может вызывать ее для отображения информации во время отладки. Фактически `ToString` — наиболее полезный метод при отладке и мало применимый где-либо еще.

Метод `Finalize` имеет специальное назначение. C# не позволяет явно переопределять этот метод. Также он не позволяет вызывать его на объекте. Если вам нужно переопределить этот метод в классе, вы можете использовать синтаксис деструктора C#. Я еще много расскажу о деструкторах и финализаторах в главе 13.

Эквивалентность и ее смысл

Эквивалентность между ссылочными типами, унаследованными от `System.Object` — непростой вопрос. По умолчанию семантика эквивалентности, представленная `Object.Equals`, подразумевает семантику идентичности. Это значит, что проверка эквивалентности вернет `true`, если две ссылки указывают на один и тот же экземпляр объекта. Однако вы можете изменить семантику `Object.Equals` для эквивалентности значений. Это значит, что две ссылки на два совершенно разных экземпляра объектов могут при сравнении давать `true`, если внутреннее состояние двух экземпляров совпадает. Переопределения `Object.Equals` — настолько распространенная задача, что я посвятил ей несколько разделов главы 13.

---

[2] Обязательно прочтите главу 8, где объясняется, почему `Object.ToString` — не то, что вам нужно при создании программного обеспечения, настраиваемого на разные локали и культуры.

Интерфейс IComparable

System.IComparable — определенный в системе интерфейс, который объекты могут реализовывать, если они поддерживают упорядочивание. Если вашим объектам имеет смысл поддерживать упорядочивание в классах коллекций, предоставляющих возможности сортировки, вы должны реализовать этот интерфейс. Например, хоть это может показаться очевидным, но System.Int32, имеющий в C# псевдоним int, реализует IComparable. В главе 13 будет показано, как можно эффективно реализовать этот интерфейс и его обобщенного родственника IComparable<T>.

Создание объектов

Тема создания объектов может показаться на первый взгляд простой, но в действительности внутренняя "кухня" ее достаточно сложна. Вы должны хорошо понимать, какие операции имеют место во время создания нового экземпляра объекта или экземпляра значения, чтобы эффективно написать код конструктора и эффективно использовать инициализаторы полей. К тому же в CLR конструкторы имеют не только экземпляры объектов, но также и типы, на которых они основаны. Я хочу сказать, что даже структуры и типы классов имеют конструктор, который представлен определением статического конструктора. Конструкторы позволяют вам выполнить всю необходимую инициализацию при загрузке типа в домен приложения.

Ключевое слово new

Ключевое слово new позволяет создавать новые экземпляры объектов или значений. Однако оно ведет себя немного по-разному, когда применяется с типами значений и со ссылочными типами. Например, в C# new не всегда выделяет место в куче. Для начала поговорим о том, что оно делает с типами значений.

Использование new с типами значений

Ключевое слово new для типов значений необходимо только тогда, когда вам нужно вызвать один из конструкторов такого типа. В противном случае типы значений просто получают место, зарезервированное для них в стеке, и клиентский код должен их полностью инициализировать перед использованием. Я расскажу об этом в разделе "Определения типов значений", когда пойдет речь о конструкторах типов значений.

Использование new с типами классов

Операция new необходима для создания объектов типа классов. В этом случае операция new выделяет место в куче для создаваемого объекта. Если ему не удается найти достаточно места, он генерирует исключение типа System.OutOfMemoryException, прерывая весь остальной процесс создания объекта.

После того, как выделено место, все поля объекта инициализируются их значениями по умолчанию. Это подобно тому, что делает сгенерированный компилятором конструктор для типов значений. Для полей ссылочных типов значением по

умолчанию является null. Для полей ссылочных типов участки памяти, занятые их объектами, заполняются нулями. Таким образом, совокупный эффект заключается в том, что все поля в новом объекте инициализируются либо null, либо 0. Как только это сделано, CLR вызывает соответствующий конструктор для экземпляра объекта. Конструктор выбирается на основе переданных параметров, при этом соответствие находится в соответствии с алгоритмом сравнения параметров методов языка C#. Операция new также устанавливает в конструкторе скрытый параметр this, который представляет собой ссылку, доступную только для чтения, которая указывает на новый объект, созданный в куче, и тип этой ссылки совпадает с типом данного класса. Рассмотрим следующий пример:

```
public class MyClass
{
    public MyClass( int x, int y )
    {
        this.x = x;
        this.y = y;
    }
    public int x;
    public int y;
}
public class EntryPoint
{
    static void Main()
    {
        // Это сделать невозможно!
        // MyClass objA = new MyClass();
        MyClass objA = new MyClass( 1, 2 );
        System.Console.WriteLine( "objA.x = {0}, objA.y = {1}",
                                  objA.x, objA.y );
    }
}
```

Обратите внимание, что в методе Main вы не можете создать новый экземпляр MyClass вызовом конструктора по умолчанию. Компилятор C# не создает конструктора по умолчанию для класса, если никакой другой конструктор не определен. Остальная часть кода достаточно очевидна. Я создаю новый экземпляр MyClass и затем вывожу его значения на консоль. Чуть дальше, в разделе "Конструктор экземпляра и порядок создания" я еще расскажу о конструкторах и подробностях создания экземпляра.

Инициализация полей

При определении класса иногда удобно присваивать полям значения в точке их объявления. Фактически вы можете присваивать полям любые литеральные значения или результаты вызовов любых методов до тех пор, пока эти методы не вызываются на экземпляре создаваемого объекта. Например, вы можете инициализировать поля возвращаемым значением статического метода того же класса. Рассмотрим следующий пример:

```
using System;
public class A
{
    private static int InitX()
    {
        Console.WriteLine( "A.InitX()" );
        return 1;
    }
    private static int InitY()
    {
        Console.WriteLine( "A.InitY()" );
        return 2;
    }
    private static int InitA()
    {
        Console.WriteLine( "A.InitA()" );
        return 3;
    }
    private static int InitB()
    {
        Console.WriteLine( "A.InitB()" );
        return 4;
    }
    private int y = InitY();
    private int x = InitX();
    private static int a = InitA();
    private static int b = InitB();
}
public class EntryPoint
{
    static void Main()
    {
        A a = new A();
    }
}
```

Обратите внимание, что вы присваиваете значения всем полям посредством инициализаторов полей и присваиванием возвращаемых значений от вызова методов. Все методы, вызываемые во время инициализации, статические, что позволяет подчеркнуть пару важных моментов относительно инициализации полей. Вывод приведенного выше кода выглядит так:

```
A.InitA()
A.InitB()
A.InitY()
A.InitX()
```

Следует отметить, что два из этих полей — a и b — статические, в то время как x и y — поля экземпляра. Исполняющая система инициализирует статические поля прежде, чем тип класса используется первый раз в данном домене приложения. В следующем разделе "Статические конструкторы (класса)" я покажу, как вы можете изменить время инициализации CLR статических полей.

Во время конструирования экземпляра вызываются инициализаторы полей экземпляров. Как и ожидалось, доказательство этого появляется в консольном выводе после выполнения инициализаторов статических полей. Отметьте один важный момент: порядок вывода инициализаторов экземпляра и сравните его с порядком объявления полей в самом классе. Вы увидите, что инициализация полей — статических или экземпляра — происходит в том порядке, в котором поля перечислены в определении класса. Иногда этот порядок может быть важен, если ваши статические поля основаны на выражениях или методах, которые полагаются на то, что другие поля того же класса должны быть инициализированы раньше. Любой ценой следует избегать написания такого кода. Фактически любой код, требующий от вас учета порядка объявления полей — это плохой код. Если последовательность инициализации существенна, вы должны рассмотреть возможность инициализации всех ваших полей в теле статического конструктора. В результате люди, сопровождающие ваш код впоследствии, не будут неприятно удивлены результатом изменения порядка полей в определении класса, если это понадобится по каким-то причинам.

Статические конструкторы (класса)

Я уже касался темы статических конструкторов в разделе "Поля", но давайте рассмотрим их еще раз повнимательней. Класс может иметь максимум один статический конструктор, и этот статический конструктор не может принимать параметров. Статические конструкторы никогда не вызываются напрямую. Вместо этого CLR вызывает их, когда нужно инициализировать тип для данного домена приложения. Статический конструктор вызывается перед первым созданием экземпляра данного класса или перед первым обращением к любому из статических полей этого класса. Давайте модифицируем предыдущий пример, добавив статический конструктор, и посмотрим, как изменится вывод:

```
using System;
public class A
{
    static A()
    {
        Console.WriteLine( "static A::A()" );
    }
    private static int InitX()
    {
        Console.WriteLine( "A.InitX()" );
        return 1;
    }
    private static int InitY()
    {
        Console.WriteLine( "A.InitY()" );
        return 2;
    }
    private static int InitA()
    {
        Console.WriteLine( "A.InitA()" );
        return 3;
    }
```

```
        private static int InitB()
        {
            Console.WriteLine( "A.InitB()" );
            return 4;
        }
        private int y = InitY();
        private int x = InitX();
        private static int a = InitA();
        private static int b = InitB();
    }
    public class EntryPoint
    {
        static void Main()
        {
            A a = new A();
        }
    }
```

Я добавил статический конструктор и хочу увидеть в выводе, будет ли он вызван. Вот что получилось в результате:

```
A.InitA()
A.InitB()
static A::A()
A.InitY()
A.InitX()
```

Разумеется, статические конструкторы были вызваны перед созданием экземпляра класса. Однако обратите внимание на последовательность вызовов. Инициализаторы статических полей выполняются перед выполнением тела статического конструктора. Это гарантирует правильную инициализацию полей перед тем, как произойдет обращение к ним из тела статического конструктора.

Вызов инициализатора типа перед обращением к любому из членов этого типа является стандартным поведением CLR. Я имею в виду, что инициализатор типа будет выполнен перед тем, как код обратится к полю или методу класса или перед созданием объекта этого класса. Однако вы можете применить определенный в CLR атрибут метаданных `beforefieldinit`, чтобы несколько ослабить строгость этих правил. В отсутствие атрибута `beforefieldinit` CLR вызывает инициализатор типа перед тем, как будет затронут любой член класса. С атрибутом `beforefieldinit` CLR разрешается отложить инициализацию типа до момента, непосредственно предшествующего первому обращению к статическому полю, но не раньше. Это значит, что если для класса установлен атрибут `beforefieldinit`, то вы можете хоть целый день вызывать конструкторы экземпляра и методы, не требуя предварительного выполнения инициализатора. Но как только кто-нибудь попытается обратиться к статическому полю класса, то CLR перед этим вызовет инициализатор типа. Имейте в виду, что атрибут `beforefieldinit` разрешает CLR отложить инициализацию типа на более позднее время, но все же CLR все равно может инициализировать тип задолго до первого обращения к статическому полю.

Компилятор C# устанавливает атрибут `beforefieldinit` для всех классов, которые не определяют специально статического конструктора. Чтобы убедиться в этом, вы можете использовать ILDASM для просмотра сгенерированного кода IL

для предыдущих двух примеров. Для примера из предыдущего раздела, где я не определил статического конструктора, метаданные класса A выглядят так:

```
.class public auto ansi beforefieldinit A
extends [mscorlib]System.Object
{
} // end of class A
```

Метаданные класса A из примера настоящего раздела выглядят иначе:

```
.class public auto ansi A
extends [mscorlib]System.Object
{
} // end of class A
```

Такое поведение компилятора C# имеет большой смысл. Когда вы явно определяете инициализатор типа, то обычно хотите гарантировать, что он будет выполнен перед использованием чего-либо из класса или перед созданием любого экземпляра этого класса. Однако если вы не предусмотрите явного инициализатора типа, но у вас будут инициализаторы статических полей, то компилятор C# создаст такой инициализатор типа, который просто инициализирует все статические поля. Поскольку вы не предоставляете пользовательского кода для инициализатора типа, то компилятор C# может позволить классу отложить инициализацию статического поля до того момента, когда к этому полю произойдет первое обращение.

После всей этой дискуссии о beforefieldinit вы должны отметить одну важную вещь. Предположим, у вас есть класс, подобный тем, что приведены в примерах, где статическое поле инициализируется результатом вызова метода. Если ваш класс не имеет явного инициализатора типа, будет ошибкой предположить, что код, вызванный во время инициализации статического поля, будет вызван перед созданием объекта этого класса. Например, рассмотрим следующий код:

```
using System;
public class A
{
    public A()
    {
        Console.WriteLine( "A.A()" );
    }
    static int InitX()
    {
        Console.WriteLine( "A.InitX()" );
        return 1;
    }
    public int x = InitX();
}
public class EntryPoint
{
    static void Main()
    {
        // Нет гарантий вызова A.InitX() перед этим!
        A a = new A();
    }
}
```

Если ваша реализация InitX содержит некоторые сторонние эффекты, которые должны быть выполнены перед созданием экземпляра объекта данного класса, то вам лучше поместить такой код в статический конструктор, чтобы компилятор не применял атрибут метаданных beforefieldinit к вашему классу. В противном случае нет никаких гарантий, что ваш код со сторонним эффектом в нем будет запущен до создания экземпляра класса.

Конструктор экземпляра и порядок создания

Конструкторы экземпляра следуют многим таким же правилам, что и статические конструкторы, за исключением того, что они более гибкие и более мощные, поэтому подчиняются еще некоторым дополнительным собственным правилам. Рассмотрим эти правила.

Конструкторы экземпляра могут иметь то, что называется *выражением инициализатора*. Выражение инициализатора позволяет конструкторам экземпляра во время инициализации объекта поручать некоторую часть своей работы другим конструкторам экземпляра внутри класса, или, что более важно, конструкторам базового класса. Это важно, если вы возлагаете на конструкторы экземпляра базового класса обязанность инициализации унаследованных членов. Помните, что конструкторы никогда не наследуются, поэтому при инициализации вам нужно прибегнуть к явным мерам — таким как вызовы базовых конструкторов, когда это нужно.

Если класс вообще не реализует конструктора экземпляра, то компилятор генерирует для вас конструктор экземпляра по умолчанию, лишенный параметров, который в действительности будет делать только одно — вызывать конструктор по умолчанию базового класса через ключевое слово base. Если базовый конструктор не имеет доступного конструктора экземпляра, генерируется ошибка компилятора. Например, следующий код компилироваться не будет:

```
public class A
{
    public A(int x) {
        this.x = x;
    }
    private int x;
}
public class B : A
{
}
public class EntryPoint
{
    static void Main()
    {
        B b = new B();
    }
}
```

Можете ли вы понять, почему он не компилируется? Проблема состоит в том, что класс, не имеющий явных конструкторов, получает от компилятора конструктор по умолчанию без параметров; этот класс просто вызывает конструктор по умолчанию базового класса, также не имеющий параметров. Именно это компилятор пытается сделать с классом B. Однако проблема в том, что поскольку класс A не имеет явно

определенного конструктора экземпляра, то компилятор не генерирует и конструктора по умолчанию для класса A. Поэтому в классе A нет доступного конструктора по умолчанию, чтобы он мог вызвать предоставленный компилятором конструктор по умолчанию класса B. И здесь мы сталкиваемся с еще одним нюансом, связанным с наследованием. Чтобы приведенный пример компилировался, вы должны либо явно представить конструктор по умолчанию для класса A, либо в конструкторе B должен быть явный конструктор. Теперь рассмотрим пример, демонстрирующий порядок событий, происходящих во время инициализации экземпляра:

```
using System;
class Base
{
    public Base( int x )
    {
        Console.WriteLine( "Base.Base(int)" );
        this.x = x;
    }
    private static int InitX()
    {
        Console.WriteLine( "Base.InitX()" );
        return 1;
    }
    public int x = InitX();
}
class Derived : Base
{
    public Derived( int a )
           :base( a )
    {
        Console.WriteLine( "Derived.Derived(int)" );
        this.a = a;
    }
    public Derived( int a, int b )
           :this( a )
    {
        Console.WriteLine( "Derived.Derived(int, int)" );
        this.a = a;
        this.b = b;
    }
    private static int InitA()
    {
        Console.WriteLine( "Derived.InitA()" );
        return 3;
    }
    private static int InitB()
    {
        Console.WriteLine( "Derived.InitB()" );
        return 4;
    }
    public int a = InitA();
    public int b = InitB();
}
```

```
public class EntryPoint
{
    static void Main()
    {
        Derived b = new Derived( 1, 2 );
    }
}
```

Прежде чем я начну детальное описание порядка событий, взглянем на вывод этого кода:

```
Derived.InitA()
Derived.InitB()
Base.InitX()
Base.Base(int)
Derived.Derived(int)
Derived.Derived(int, int)
```

Можете ли вы объяснить, почему порядок именно таков, как он есть? На первый взгляд он может показаться довольно путанным, поэтому задержимся на минутку и разберемся, что здесь происходит. Первая строка метода `Main` создает новый экземпляр класса `Derived`. Как видно из вывода, вызван конструктор. Но он вызывается в последней строке вывода! Понятно, что множество вещей происходят перед тем, как будет выполнено тело конструктора класса `Derived`.

Внизу вы видите вызов конструктора `Derived`, принимающий два параметра типа `int`. Обратите внимание, что этот конструктор имеет инициализатор, использующий ключевое слово `this`. Это делегирует работу конструктора другому конструктору `Derived`, имеющему один параметр `int`.

Конструктор `Derived`, принимающий один параметр `int`, также имеет список инициализации, но использует ключевое слово `base`, тем самым вызывая конструктор базового класса `Base`, который принимает один параметр `int`. Однако если конструктор имеет инициализатор, использующий ключевое слово `base`, то прежде чем передать управление конструктору базового класса, такой конструктор вызовет инициализаторы полей, определенные в классе. И помните, что порядок инициализаторов определяется порядком указания полей в определении класса. Такое поведение объясняет первые два элемента вывода программы. Вывод демонстрирует, что инициализаторы полей в `Derived` вызываются раньше инициализаторов `Base`.

После выполнения инициализаторов `Derived` управление передается конструктору `Base`, принимающему один параметр `int`. Обратите внимание, что класс `Base` также содержит поля экземпляра с инициализаторами. В `Base` имеет место то же поведение, что и в `Derived`, поэтому перед выполнением тела конструктора `Base` этот конструктор неявно вызывает инициализаторы класса. Далее в разделе я еще много чего скажу о том, почему поведение организовано именно таким образом, и это касается виртуальных методов. Вот почему третий элемент вывода — `Base.InitX`.

После того, как инициализаторы `Base` выполнены, мы попадаем в блок конструктора `Base`. Как только выполнение тела конструктора подходит к концу, управление возвращается конструктору `Derived`, принимающему один параметр `int`, и завершается, наконец, в блоке кода этого конструктора. После этого приходит очередь выполнения того конструктора, который был вызван в точке, где код создает

экземпляр Derived — в методе Main. Ясно, что большой объем работы по инициализации происходит "за кулисами", когда создается экземпляр объекта.

Как и обещал, теперь я объясню, почему инициализаторы полей производного класса вызываются перед вызовом конструктора базового класса через инициализатор в производном конструкторе. Причина этого довольно-таки тонкая. Виртуальные методы, о которых я расскажу подробнее в разделе "Наследование и виртуальные методы", работают внутри конструкторов в CLR и в C#.

На заметку! Если вы пришли из среды программирования C++, то заметите, что такое поведение при вызове виртуальных методов здесь полностью отличается. В C++ вы никогда не полагаетесь на вызовы виртуальных методов в конструкторе, поскольку когда выполняется тело конструктора таблица виртуальных вызовов еще не готова.

Рассмотрим пример:

```
using System;
public class A
{
    public virtual void DoSomething()
    {
        Console.WriteLine( "A.DoSomething()" );
    }
    public A()
    {
        DoSomething();
    }
}
public class B : A
{
    public override void DoSomething()
    {
        Console.WriteLine( "B.DoSomething()" );
        Console.WriteLine( "x = {0}", x );
    }
    public B()
        :base()
    {
    }
    private int x = 123;
}
public class EntryPoint
{
    static void Main()
    {
        B b = new B();
    }
}
```

Вывод этого кода выглядит так:

```
B.DoSomething()
x = 123
```

Как видите, виртуальные вызовы работают прекрасно и из конструктора класса A. Обратите внимание, что B.DoSomething использует поле x. Теперь, если бы инициализаторы полей не запускались перед вызовом base, можете представить себе неприятности, которые имели бы место, когда виртуальный метод вызывается из конструктора класса A. Вот поэтому инициализаторы полей запускаются перед вызовом базового конструктора, если конструктор содержит инициализатор. Инициализаторы полей также запускаются перед входом в тело конструктора, если в конструкторе не определен инициализатор.

Уничтожение объектов

Если создание объектов показалось вам сложным, приготовьтесь к еще худшему. Как вы знаете, среда CLR содержит в себе сборщик мусора, управляющий памятью от вашего имени. Вы можете создавать столько новых объектов, сколько хотите, но вам никогда не приходится беспокоиться о явном освобождении памяти от них. Огромное большинство ошибок в "родных" приложениях вызваны несоответствиями между выделением/освобождением памяти, известными так же, как утечки памяти. Сборка мусора — это техника, предназначенная для исключения ошибок подобного рода, поскольку исполняющая среда теперь отслеживает ссылки на объекты и уничтожает их экземпляры, когда они более не используются.

CLR отслеживает каждую ссылку на управляемый объект в системе. Как только CLR обнаруживает, что объект становится недоступным ни через одну ссылку, он помечает его к удалению. При следующем проходе сборщика мусора по куче с целью ее сжатия все объекты, помеченные к удалению, либо освобождают занятую ими память, либо помещаются в очередь для удаления, если у них есть финализатор. За это отвечает другой поток — поток финализатора, который проходит по очереди объектов и вызывает их финализаторы перед освобождением памяти. По завершении работы финализатора память объекта освобождается при следующем проходе сборщика, и объект окончательно "умирает", уже безвозвратно.

Финализаторы

Есть много причин, почему вам придется редко писать финализаторы. При чрезмерном применении финализаторы могут снизить производительность CLR, поскольку финализируемые объекты живут дольше, чем их не финализируемые собратья. Даже размещение финализируемых объектов обходится дороже. Вдобавок финализаторы трудно писать, потому что вы не можете делать никаких предположений о состоянии других объектов системы.

Когда поток финализации проходит по объектам в очереди финализации, он вызывает метод Finalize каждого из этих объектов. Метод Finalize представляет собой переопределение виртуального метода из System.Object; однако с C# не разрешается явно переопределять этот метод. Вместо этого вы пишете деструктор, который выглядит как метод без типа возврата, без модификаторов, без параметров, чьим идентификатором служит имя класса, которому предшествует знак тильды. Деструкторы не могут вызываться явно в C#, они не наследуются, как не наследуются конструкторы. Класс может иметь только один деструктор.

Когда вызывается финализатор объекта, то вызывается каждый финализатор в цепочке наследования — от последнего к первому. Рассмотрим следующий пример:

```
using System;
public class Base
{
    ~Base()
    {
        Console.WriteLine( "Base.~Base()" );
    }
}
public class Derived : Base
{
    ~Derived()
    {
        Console.WriteLine( "Derived.~Derived()" );
    }
}
public class EntryPoint
{
    static void Main()
    {
        Derived derived = new Derived();
    }
}
```

Как и можно было ожидать, результат выполнения кода будет следующим:

```
Derived.~Derived()
Base.~Base()
```

Хотя сборщик мусора теперь выполняет задачу по очистке памяти и вам не приходится об этом беспокоиться, у вас появляется целая куча новых забот, когда заходит речь об уничтожении объектов. Я уже упоминал, что финализаторы выполняются в отдельном потоке CLR. Поэтому то, что ваши объекты используют внутри деструктора, должно быть безопасным в отношении потоков, и к тому же вы никогда не должны использовать других объектов в финализаторе, поскольку они могут быть к тому моменту уже финализированы или уничтожены. Сюда относятся объекты, являющиеся полями класса, которому принадлежит финализатор. У вас нет никакого гарантированного способа узнать точно, когда будет вызван ваш финализатор, или в каком порядке будут вызваны финализаторы двух зависимых или не зависимых объектов. Это еще одна причина того, почему вы не должны вводит зависимостей между объектами в блоке кода деструктора. После всего сказанного становится ясно, что вы не должны делать в финализаторе ничего сложнее простой уборки, если она необходима.

По сути, вы должны писать финализатор только когда ваш объект управляет некоторого рода неуправляемыми ресурсами. Однако если эти ресурсы управляются через стандартный дескриптор Win32, я настоятельно рекомендую использовать для управления ими `SafeHandler`. Написания такой оболочки вроде `SafeHandler` — сложная задача, главным образом из-за финализатора и всех тех вещей, вызов которых нужно гарантировать во всех ситуациях (даже в совершенно дьявольских), таких как при отсутствии памяти или при возникновении неожиданных исключений. И, наконец, любой объект, имеющий финализатор, должен реализовать шаблон Disposable, а котором я расскажу далее в разделе "Одноразовые (disposable) объекты".

Детерминированное уничтожение

До сих пор все, что вы видели, касающееся уничтожения объектов в среде со сборщиком мусора CLR, можно назвать недетерминированным уничтожением. Это значит, что вы не можете предсказать время выполнения кода деструктора для объекта. Если кто-то пришел из мира "родного" C++, он согласится с тем, что там все совершенно иначе.

В C++ деструкторы объектов кучи вызываются, когда пользователь явно удаляет объект. В CLR этим занимается сборщик мусора, так что вам не нужно беспокоиться о том, чтобы нечаянно не забыть об удалении ненужного объекта. Однако для объекта C++, находящегося в стеке, деструктор вызывается, как только происходит выход управления из области определения этого объекта. Это называют детерминированным уничтожением, которое чрезвычайно удобно для управления ресурсами.

Рассмотрим случай объекта, хранящего системный дескриптор файла. Вы можете использовать такой находящийся в стеке объект C++ для управления жизненным циклом дескриптора файла. Когда объект создается, его конструктор захватывает дескриптор файла, и как только объект выходит из своего контекста (области определения), вызывается деструктор, и его код закрывает дескриптор файла. Это избавляет клиентский код объекта от необходимости явного управления ресурсом. Это также предотвращает утечку ресурсов, поскольку даже если будет сгенерировано исключение в блоке кода, использующем объект, C++ гарантирует вызов деструкторов всех находящихся в стеке объектов — независимо от того, как именно произошел выход из блока.

Такая идиома называется "захват ресурса является инициализацией" (Resource Acquisition Is Initialization — RAII), и она исключительно удобна для управления ресурсами. C# практически полностью утратил эту возможность автоматической очистки. Конечно, если у вас есть объект, удерживающий файл открытым, и закрывает его в деструкторе, вам не нужно беспокоиться о том, закрыт файл или нет, но вы определенно должны задуматься о том, *когда* он будет закрыт. Дело в том, что вы не можете знать точно, когда он будет закрыт, если код его закрытия находится в финализаторе, который вызывается из недетерминированной финализации. По этой причине считается дурным тоном в дизайне помещать код управления ресурсами, такой как закрытие файловых дескрипторов, в финализатор. Что, если объект уже помечен к финализации, но его финализатор еще не вызван, и вы пытаетесь создать новый экземпляр объекта, чей конструктор пытается открыть тот же ресурс? В случае ресурсов с исключительным доступом при создании нового экземпляра произойдет сбой в коде конструктора. Уверен, вы согласитесь, что подобное нежелательно и, определенно, этого не ожидает клиент вашего объекта.

Вернемся еще раз к проблеме порядка финализации, которая была упомянута немного раньше. Если объект содержит в себе другой объект, и внешний объект помещен в очередь финализации, то внутренний объект также может быть помещен в нее. Однако поток финализации просто проходит по очереди и обрабатывает содержащиеся в нем объекты индивидуально. Его не волнует, кто кому приходится внутренним объектом. Поэтому ясно, что существует вероятность того, что код деструктора обратится к ссылке на объект, находящейся в поле другого объекта, последний может быть уже финализирован. Обращение к такому полю приводит к разветвленному *исключению неопределенного поведения*.

Это блестящий пример того, как сборщик мусора, исключая сложность одного рода, заменяет ее усложнением иного рода. В действительности вы должны по возможности избегать финализаторов. Они не только добавляют сложности, но и препятствуют управлению памятью, поскольку вынуждают такие объекты существовать дольше, чем объекты без финализатора. Это связано с тем, что они помещаются в список финализации, за очистку которого отвечает совершенно отдельный поток. В разделе "Одноразовые (disposable) объекты" и в главе 13 я опишу интерфейс `IDisposable`, включенный в библиотеку Framework Class Library, чтобы облегчить форму детерминированного уничтожения.

Обработка исключений

Важно обратить внимание на поведение исключений внутри контекста финализатора. Если вы пришли из мира C++, то вы знаете, что неправильно разрешать исключениям распространяться за пределы деструктора, поскольку в определенных ситуациях это может привести к прерыванию работы вашего приложения. В C# исключение, сгенерированное в финализаторе и покидающее блок не перехваченным, будет трактоваться как необработанное исключение, и по умолчанию процесс будет прерван после извещения вас об исключении.

На заметку! Это поведение было изменено в .NET 2.0 по сравнению с тем, что было в .NET 1.1. До появления .NET 2.0 необработанные исключения в потоке финализации просто "проглатывались", извещая пользователя, после чего процессу было позволено продолжаться. Опасность такого поведения заключается в том, что система могла после этого работать в полуисправном и несогласованном состоянии. Поэтому лучше уничтожить процесс, чем продолжать его работу с риском причинения большего вреда. В главе 7 я покажу, как можно заставить CLR вернуться к старому поведению, если это вам абсолютно необходимо.

Одноразовые (disposable) объекты

В предыдущем разделе о финализаторах я говорил об отличии между детерминированной и недетерминированной финализацией, вы также видели детерминированная финализация сопряжена со значительными неудобствами. В связи с этим был предложен интерфейс `IDisposable`, который на самом деле был добавлен еще на этапе бета-тестирования первого выпуска .NET Framework, когда разработчики страдали от отсутствия какой-либо встроенной формы детерминированной финализации. Это не идеальная замена детерминированной финализации, но она выполняет свою задачу ценой дополнительной сложности для клиентов ваших объектов.

Интерфейс `IDisposable`

Определение `IDisposable` выглядит так:

```
public interface IDisposable
{
    void Dispose();
}
```

Обратите внимание, что здесь есть только один метод — Dispose, в реализации которого должна выполняться вся грязная работа. То есть вы должны полностью очистить ваш объект и освободить все ресурсы внутри этого метода. Несмотря на то что Dispose вызывается клиентским кодом, а не автоматически системой, все же это хороший способ для клиентского кода заявить: "я завершил работу с этим объектом и не намерен использовать его вновь".

Хотя шаблон IDisposable и представляет собой форму детерминированного уничтожения, все же это решение далеко от идеала. При использовании IDisposable ответственность за вызов метода Dispose возлагается на клиента. У клиента нет никакой возможности поручить системе или компилятору его автоматический вызов. C# немного облегчает эту задачу в случае исключений, перегружая ключевое слово using, о чем речь пойдет в следующем разделе.

Когда вы применяете Dispose, то обычно реализуете класс таким образом, что именно код финализатора повторно использует Dispose. Таким образом, если клиентский код никогда не вызовет Dispose, то код финализатора позаботится об этом в свое время. Другой фактор делает реализацию IDisposable болезненной для объектов, и он заключается в том, что вы обязаны по цепочке вызывать IDisposable для ваших объектов, содержащих ссылки на другие объекты, поддерживающие IDisposable. Это несколько затрудняет проектирование классов, поскольку вы должны знать, реализует ли класс, используемый для вашего поля, тот же интерфейс IDisposable, и если да, вы также должны реализовать IDisposable и не забыть вызвать его метод Dispose внутри вашего.

Принимая во внимание все вышесказанное об IDisposable, вы можете убедиться в том, насколько сборщик мусора добавляет сложности в дизайн, хотя и сокращает вероятность утечек памяти. Я не хочу сказать, что сборщик мусора хуже; на самом деле, он очень полезен при правильном применении. Однако, как и в любом дизайне, все инженерные решения обычно имеют свои плюсы и минусы.

Рассмотрим пример реализации IDisposable:

```
using System;
public class A : IDisposable
{
     private bool disposed = false;
     public void Dispose( bool disposing )
     {
          if( !disposed ) {
             if( disposing ) {
                // Здесь безопасно обращаться к другим объектам
             }
             Console.WriteLine( "Cleaning up object" );
             disposed = true;
          }
     }
     public void Dispose()
     {
          Dispose( true );
          GC.SuppressFinalize( this );
     }
     public void DoSomething()
     {
```

```
            Console.WriteLine( "A.SoSomething()" );
        }
        ~A()
        {
            Console.WriteLine( "Finalizing" );
            Dispose( false );
        }
    }
    public class EntryPoint
    {
        static void Main()
        {
            A a = new A();
            try {
               a.DoSomething();
            }
            finally {
               a.Dispose();
            }
        }
    }
```

Разберем детально этот код, чтобы понять, что в нем происходит в действительности. Первое, что следует отметить в классе — внутреннее булевское поле, регистрирующее факт вызова метода Dispose для данного объекта. Оно присутствует потому, что вполне допускается, чтобы клиентский код вызывал Dispose несколько раз. Поэтому вам нужен какой-то способ узнать, что работа уже выполнена.

Также вы должны заменить, что я построил финализатор в терминах реализации Dispose. Обратите внимание, что я предусмотрел две перегрузки Dispose. Это сделано для того, чтобы знать внутри метода Dispose(bool), вызван он через IDisposable.Dispose или через деструктор. Это говорит, могу ли я безопасно обращаться в методе к содержащимся объектам.

Еще один момент: метод Dispose осуществляет вызов GC.SuppressFinalize. Этот метод сборщика мусора позволяет удержать сборщик мусора от финализации объекта. Если клиентский код вызывает Dispose, и этот метод полностью очищает ресурсы, включая всю работу по финализации, то в дополнительной финализации данного объекта нет необходимости. Вы можете вызвать SuppressFinalize, чтобы предотвратить финализацию объекта. Такая полезная оптимизация помогает сборщику мусора своевременно избавиться от вашего объекта, когда все ссылки на него прекращают свое существование.

Теперь давайте посмотрим, как использовать этот одноразовый объект. Обратите внимание на блок try/finally внутри метода Main. Тему исключений я раскрою в главе 7, а пока вам достаточно знать, что конструкция try/finally гарантирует выполнение определенного кода вне зависимости от того, как завершится выполнение блока кода. В данном случае, независимо от того, как поток управления покинет блок try — нормально через оператор return или даже по исключению — код в блоке finally будет выполнен. Рассматривайте блок finally как страховочную сетку. Я говорю о блоке finally, в котором вызывается Dispose объекта. Независимо от обстоятельств Dispose будет вызван обязательно.

Это блестящий пример того, как недетерминированная финализация возлагает ответственность на клиентский код, или на пользователя, за очистку объекта, в то время как детерминированная финализация не требует от пользователя связываться с написанием этих некрасивых блоков try/finally или вызова Dispose. Это определенно усложняет жизнь пользователю, поскольку ему нужно писать устойчивый к исключениям и/или нейтральный к исключениям код. Проектировщики C# попытались облегчить это бремя, перегружая ключевое слово using. Но хотя это и снижает нагрузку, все же не позволяет полностью освободить пользователя от дополнительного кодирования.

На заметку! C++/CLI позволяет вам использовать RAII способом, знакомым разработчикам C++, не требуя явного вызова Dispose или применения блока using. Было бы замечательно, если бы C# позволял то же самое, но, к сожалению, такое кардинальное изменение в языке повлекло бы за собой много неприятностей.

Ключевое слово using

Ключевое слово using было перегружено для поддержки шаблона IDisposable. Общая идея состояла в том, что оператор using должен захватывать ресурсы внутри фигурных скобок, следующих за ключевым словом using, в то время как область видимости этих локальных переменных ограничена областью определения последующих фигурных скобок.

Рассмотрим модифицированную версию предыдущего примера:

```
using System;
public class A : IDisposable
{
    private bool disposed = false;
    public void Dispose( bool disposing )
    {
        if( !disposed ) {
           if( disposing ) {
              // Здесь безопасно обращаться к другим объектам
           }
           Console.WriteLine( "Очистка объекта" );
           disposed = true;
        }
    }
    public void Dispose()
    {
        Dispose( true );
        GC.SuppressFinalize( this );
    }
    public void DoSomething()
    {
        Console.WriteLine( "A.SoSomething()" );
    }
```

```
    ~A()
    {
        Console.WriteLine( "Finalizing" );
        Dispose( false );
    }
}
public class EntryPoint
{
    static void Main()
    {
        using( A a = new A() ) {
           a.DoSomething();
        }
        using( A a = new A(), b = new A() ) {
           a.DoSomething();
           b.DoSomething();
        }
    }
}
```

"Мясо" изменений находится в методе Main. Обратите внимание, что я заменил неуклюжую конструкцию try/finally более ясным оператором using. "За кулисами" оператор using расширяется до конструкции try/finally, которая была раньше. Но теперь этот код намного легче читать и понимать. Однако он не избавляет клиентский код от необходимости помнить, прежде всего, об операторе using.

Оператор using требует, чтобы все ресурсы, захваченные в процессе, были неявно преобразуемы к IDisposable. То есть они должны реализовать IDisposable. Если этого не будет, вы столкнетесь с предупреждениями во время компиляции.

Типы параметров методов

Параметры методов подчиняются тем же общим правилам, что и в C/C++. То есть по умолчанию параметры объявляют идентификатор переменной, который действителен на протяжении и в контексте самого метода. Здесь нет константных параметров, как в C++, и параметрам методов можно присваивать значения. Если только параметры не объявлены определенным образом — как ref или out — такое присваивание остается локальным в пределах метода.

Я обнаружил, что одной из наибольших заминок для разработчиков C++ в C# оказалось обращение с семантикой переменных, передаваемых в методы. Поскольку доминантным типом внутри CLR является ссылка, переменные таких объектов просто указывают на экземпляры в куче, т. е. аргументы передаются в метод в семантике ссылки. Разработчики C++ привыкли к тому, что по умолчанию в методы передаются копии переменных, если только они не переданы по ссылке или в виде указателя. Другими словами, там аргументы передаются в семантике значений.

В C# аргументы в действительности тоже передаются по значению. Однако для ссылок копируемое значение представляет собой саму ссылку, а не объект, на который она ссылается. Изменения в состоянии ссылаемого объекта внутри метода становятся видимыми коду, вызвавшему этот метод.

Поскольку в C# нет нотации константных параметров, вы должны создавать неизменяемые (immutable) объекты, когда хотите передать константный параметр. Я расскажу подробнее о таких объектах в главе 13.

На заметку! Те разработчики на C++, которые привыкли применять идиомы дескриптор/тело (handle/body) для реализации семантики копирования при записи, должны принимать этот факт во внимание. Это не значит, что вы не можете применять эти идиомы в C#; скорее, это просто означает, что вы должны реализовывать их иначе.

Аргументы-значения

В действительности все параметры, передаваемые в методы, являются аргументами-значениями, если предполагается, что они — нормальные, простые, не декорированные параметры методов. Под не декорируемыми я понимаю отсутствие специальных ключевых слов, таких как `out`, `ref` и `params`, присоединенных к ним. Однако они могут иметь прикрепленные к ним атрибуты, как почти все что угодно в системе типов CLR. Как и все параметры, идентификатор находится в области внутри блока метода, следующего за списком параметров (т.е. внутри фигурных скобок), и метод получает копию переданной переменной в момент его вызова. Однако будьте осторожны с тем, что это означает. Если переданная переменная — это структура, или тип значения, то метод получает копию этого значения. Любые изменения, проведенные локально в этой переменной, не видны вызывающему коду. Если же передаваемая переменная — ссылка на объект в куче, как любая переменная — экземпляр класса, то метод получает копию *ссылки*. То есть любые изменения, проведенные в объекте через такую ссылку, становятся видимы коду, вызвавшему метод.

Аргументы `ref`

Передача параметров по ссылке отмечается помещением модификатора `ref` перед типом параметра в списке параметров метода. Когда переменная передается по ссылке, новая копия этой переменной не создается, и эту переменную из вызывающего метода напрямую затрагивают все действия внутри метода. Как обычно бывает в CLR, это означает две слегка отличающиеся вещи, в зависимости от того, является ли переменная экземпляром типа значения (структуры) или же объекта (класса).

Когда экземпляр значения передается по ссылке, то копия значения из вызывающего кода не создается. Это подобно передаче параметра как указателя в C++, даже несмотря на то, что вы обращаетесь к методам и полям переменной точно так же, как с аргументами-значениями. Когда экземпляр объекта (ссылки) передается по ссылке, никакой копии переменной не создается. Фактически переменная ведет себя как указатель C++ на ссылочную переменную, что в C++ выглядит, как указатель на указатель. Вдобавок верификатор проверяет, чтобы переменная, на которую ссылается параметр `ref`, имела определенное присвоенное значение перед вызовом метода. Рассмотрим некоторые примеры применения полной нотации `ref`-параметров:

```
using System;
public struct MyStruct
{
    public int val;
}
public class EntryPoint
{
    static void Main() {
           MyStruct myValue = new MyStruct();
           myValue.val = 10;
           PassByValue( myValue );
           Console.WriteLine( "Результат PassByValue: myValue.val = {0}",
                                     myValue.val );
           PassByRef( ref myValue );
           Console.WriteLine( "Результат PassByRef: myValue.val = {0}",
                                     myValue.val );
    }
    static void PassByValue( MyStruct myValue ) {
           myValue.val = 50;
    }
    static void PassByRef( ref MyStruct myValue ) {
           myValue.val = 42;
    }
}
```

Этот пример содержит два метода: PassByValue и PassByRef. Оба метода модифицируют поле экземпляра типа значения, переданного им. Однако, как доказывает приведенный ниже вывод, метод PassByValue модифицирует локальную копию, в то время как метод PassByRef модифицирует экземпляр из вызывающего кода:

```
Результат PassByValue: myValue.val = 10
Результат PassByRef: myValue.val = 42
```

К тому же обратите внимание на тот факт, что ключевое слово ref требуется в точке вызова метода PassByRef. Это необходимо, потому что метод может быть перегружен на основе присутствия или отсутствия ключевого слова ref. Другими словами, другой метод PassByRef мог бы также принимать MyStruct по значению, а не по ссылке. Плюс к этому тот факт, что вы обязаны применить ключевое слово ref в точке вызова, на мой взгляд, облегчает чтение кода.

Теперь рассмотрим пример, использующий объект вместо типа значения:

```
using System;
public class EntryPoint
{
    static void Main() {
           object myObject = new Object();
           Console.WriteLine( "myObject.GetHashCode() == {0}",
                  myObject.GetHashCode() );
           PassByRef( ref myObject );
           Console.WriteLine( "myObject.GetHashCode() == {0}",
                  myObject.GetHashCode() );
    }
```

```
    static void PassByRef( ref object myObject ) {
          // Присвоить новый экземпляр этой переменной
          myObject = new Object();
    }
}
```

В этом случае переданная по ссылке переменная является объектом. Но как я сказал, вместо того, чтобы в метод передать копию ссылки, тем самым создавая новую ссылку на тот же объект, здесь передается сама исходная ссылка. Да, звучит запутанно. В предыдущем методе PassByRef переданная ссылка переустанавливается на новый экземпляр объекта. Исходный объект остается без ссылок, и потому готов к тому, чтобы его подобрал сборщик мусора. Чтобы доказать, что новая переменная myObject ссылается на два разных экземпляра в точке вызова и в точке, следующей за вызовом, я отправляют результат вызова myObject.GetHashCode на консоль.

Параметры out

Параметры out (выходные) почти идентичны параметрам ref, но с двумя существенными отличиями. Во-первых, вместо применения ключевого слова ref, вы используете ключевое слово out, и также вы обязаны применить его в точке вызова, как это делали с ref. Однако переменная, на которую ссылается переменная out, не обязана иметь присвоенное значение перед вызовом метода, как в случае ref-параметра. Это потому, что методу не позволено каким-либо полезным способом использовать переменную до тех пор, пока ей не будет присвоено значение. Например, следующий код совершенно корректен:

```
public class EntryPoint
{
    static void Main() {
        object obj;
        PassAsOutParam( out obj );
    }
    static void PassAsOutParam( out object obj ) {
        obj = new object();
    }
}
```

Обратите внимание, что переменной obj в методе Main напрямую не присваивается никакого значения перед вызовом PassAsOutParam. И это правильно, потому что она помечена, как out-параметр. Метод PassAsOutParam не обращается к этой переменной, пока не присвоит ей значение. Если вы попробуете заменить два вхождения out на ref в приведенном выше коде, то увидите ошибку компилятора, подобную следующей:

```
error CS0165: Use of unassigned local variable 'obj'
ошибка CS0165: Использование неприсвоенной локальной переменной 'obj'
```

Массивы params

C# позволяет легко передавать переменный список параметров. Для этого просто объявите последний параметр в вашем списке параметров как тип массива, предварив его ключевым словом `params`. Теперь, если метод вызывается с переменным числом параметров, то эти параметры передаются ему в форме массива, по которому вы легко выполните итерацию, и тип этого массива может базироваться на любом корректном типе. Ниже приведен краткий пример:

```
using System;
public class EntryPoint
{
     static void Main() {
         VarArgs( 42 );
         VarArgs( 42, 43, 44 );
         VarArgs( 44, 56, 23, 234, 45, 123 );
     }
     static void VarArgs( int val1, params int[] vals ) {
         Console.WriteLine( "val1: {0}", val1 );
         foreach( int i in vals ) {
            Console.WriteLine( "vals[]: {0}", i );
         }
         Console.WriteLine();
     }
}
```

В каждом случае `VarArgs` вызывается успешно, но в каждом случае массив, переданный в `vals`, отличается. Как видите, передача переменного числа параметров в C# замечательно проста. Вы можете закодировать эффективный метод `Add` для контейнерного типа, применяя массивы параметров, когда необходим только один вызов для добавления переменного числа элементов.

Перегрузка методов

Перегрузка в C# — прием времени компиляции, при котором в точке вызова компилятор выбирает метод из набора одноименных методов с разной сигнатурой. Компилятор использует список аргументов метода для выбора наиболее подходящего. Типы аргументов, а также модификаторы параметров `ref`, `out` и `params` играют роль в перегрузке методов, поскольку все они составляют часть сигнатуры метода. Методы без параметров-массивов переменной длины получают преимущества перед методами, имеющими такие параметры. Подобно C++, тип возврата метода не является частью сигнатуры (за исключением одного редкого случая операций преобразования, о котором я расскажу в главе 6). Поэтому вы не можете иметь внутри класса перегруженных методов, отличающихся только типом возврата. И, наконец, если компилятор доходит до точки, где обнаруживается неоднозначность в выборе версии перегруженного метода, он останавливается с ошибкой.

А вообще между перегрузкой методов в C# и перегрузкой методов в C++ нет разницы. Она не может послужить причиной исключений времени выполнения, поскольку весь алгоритм выбора применяется во время компиляции. Когда компилятор не может найти точно подходящий метод на основе переданных параметров, он

начинает подбор наиболее подходящего соответствия на основе неявной конвертируемости экземпляров в списке параметров. Поэтому если метод с одним параметром принимает объект типа A, а вы передаете ему объект типа B, унаследованный от A, то в отсутствии версии этого метода, принимающей тип B, компилятор неявно преобразует ваш переданный экземпляр к ссылке типа A, чтобы удовлетворить такой вызов. В зависимости от ситуации и величины набора перегруженных методов процесс выбора может оказаться довольно сложным. Я обнаружил, что лучше минимизировать число неоднозначных перегрузок, когда для удовлетворительного решения компилятору нужно выполнять неявные преобразования. Слишком много неявных преобразований могут затруднить понимание кода, вызывая необходимость применения отладчика, чтобы разобраться, что же именно происходит. Это усложняет задачу персонала сопровождения, которому придется следовать за вами и разбираться в ваших действиях. Я не говорю, что неявные преобразования — плохая вещь при разрешении перегрузки, но лучше использовать их благоразумно и сдержанно, чтобы минимизировать неприятные сюрпризы в будущем.

Наследование и виртуальные методы

C# реализует понятие виртуальных методов так же, как это делают языки C++ и Java. Здесь вообще нет никаких новшеств, поскольку C# — объектно-ориентированный язык, а виртуальные методы — главный механизм реализации динамического полиморфизма. Тем не менее, некоторые заметные отличия между этими языками требуют специального упоминания.

Виртуальные и абстрактные методы

Вы объявляете виртуальный метод, используя в месте его объявления модификаторы — либо `virtual`, либо `abstract`. Оба вводят метод в пространство объявления, как таковой, что может быть переопределен в производных классах. Отличие между этими двумя модификаторами в том, что абстрактные методы обязательно должны быть переопределены, в то время как просто виртуальные — нет. Абстрактные методы подобны чистым (pure) виртуальным методам C++, за исключением того, что чистые виртуальные методы C++ могут иметь ассоциированную с ними реализацию, в то время как абстрактные методы C# — нет. Виртуальные методы, в отличие от абстрактных, обязаны иметь ассоциированную с ними реализацию. Виртуальные методы, наряду с их интерфейсами — единственные средства реализации полиморфизма в C#.

На заметку! "За кулисами" CLR реализует виртуальные методы иначе, чем в C++. В то время как C++ может создавать множество таблиц vtable (динамических таблиц, содержащих указатели на виртуальные методы) для индивидуального объекта класс, в зависимости от их статической иерархической структуры, объекты CLR имеют лишь одну таблицу методов, содержащую как виртуальные, так и не виртуальные методы. Вдобавок таблица в CLR строится на ранней стадии жизненного цикла объекта. И порядок создания объектов не только влияет на порядок вызова статических инициализаторов и конструкторов в иерархии, но он также обеспечивает C# такую возможность, которой нет в C++. Подробнее о том, как CLR управляет таблицами методов для экземпляров объектов, читайте в книге Дона Бокса (Don Box) и Криса Селлса (Chris Sells) *Essential .NET, Volume 1: The Common Language Runtime* (Boston, MA: Addison-Wesley Professional, 2002 г.).

Методы new и override

Чтобы переопределить метод в производном классе, вы должны снабдить его модификатором override. Если этого не сделать, то компилятор предупредит вам о том, что вы должны указать либо модификатор new, либо override в объявлении производного метода. По умолчанию компилятор подразумевает использование модификатора new, что, вероятно, даст эффект, противоположный тому, что вы ожидали. Это поведение отличается от C++, поскольку в C++, если метод помечен как virtual, то любой производный метод с тем же именем и сигнатурой автоматически переопределяет этот виртуальный метод, и модификатор virtual в этих производных методах совершенно не обязателен. Лично мне больше нравится тот факт, что C# требует от вас пометки переопределяющего метода — просто из-за улучшения читабельности кода. Не могу даже подсчитать, насколько много случаев плохо написанного кода C++ с глубокой иерархией мне попадались, где разработчики ленились снабжать виртуальные переопределяемые методы ключевым словом virtual. Поэтому не было никакой возможность узнать, что конкретный метод переопределяет виртуальный метод базового класса, не заглянув в объявление базового класса. И такой жутко спроектированный код имел столь глубокую иерархию, что мне приходилось продираться через горы файлов, чтобы найти ответ. В C# эта проблема решена. Взгляните на следующий код:

```
using System;
public class A
{
    public virtual void SomeMethod() {
        Console.WriteLine( "A.SomeMethod" );
    }
}
public class B : A
{
    public void SomeMethod() {
        Console.WriteLine( "B.SomeMethod" );
    }
}
public class EntryPoint
{
    static void Main() {
        B b = new B();
        A a = b;
        a.SomeMethod();
    }
}
```

Этот код компилируется, но выдается следующее предупреждение:

```
test.cs(12,17): warning CS0114: 'B.SomeMethod()' hides inherited
member 'A.SomeMethod()'.
To make the current member override that implementation, add the
override keyword.
Otherwise add the new keyword.
```

```
test.cs(12,17): предупреждение CS0114: 'B.SomeMethod()' скрывает
унаследованный член 'A.SomeMethod()'.
Чтобы текущий член переопределил эту реализацию, добавьте ключевое
слово override.
В противном случае добавьте ключевое слово new.
```

При выполнении кода вызывается метод `A.SomeMethod`. Так что же делает ключевое слово `new`? Оно разбивает виртуальную цепочку в данной точке иерархии. Когда виртуальный метод вызывается через ссылку на объект, то конкретный вызываемый метод определяется по таблице методов во время выполнения. Если метод виртуальный, то исполняющая система движется по иерархии в поисках наиболее удаленной производной версии метода, и затем вызывает ее. Однако во время поиска, если она встречает метод, помеченный модификатором `new`, то возвращается к методу из предыдущего класса в иерархии и использует его. Таким образом, вызывается именно `A.SomeMethod`. Если бы `B.SomeMethod` был помечен словом `override`, то был бы вызван именно он. Поскольку в C# модификатор `new` применяется по умолчанию, когда не указан никакой другой, компилятор выдает предупреждение, чтобы привлечь внимание тех, кто привык к синтаксису C++. И, наконец, модификатор `new` ортогонален по смыслу модификатору `virtual` — в том плане, что метод, помеченный как `new`, также может быть или не быть виртуальным. В предыдущем примере я не указал модификатор `virtual` для метода `B.SomeMethod`, так что не может быть такого, чтобы класс `C`, производный от `B`, переопределил `B.SomeMethod`, поскольку он не виртуален. Таким образом, ключевое слово `new` не только разрушает виртуальную цепочку, но также переопределяет то, что данный класс и классы-наследники `B` получат виртуальный `SomeMethod`.

Другая сложность, которую нужно упомянуть в отношении переопределения методов — как и когда следует вызывать версию метода базового класса. В C# вы можете вызвать версию базового класса, используя идентификатор `base`, как показано ниже:

```
using System;
public class A
{
    public virtual void SomeMethod() {
        Console.WriteLine( "A.SomeMethod" );
    }
}
public class B : A
{
    public override void SomeMethod() {
        Console.WriteLine( "B.SomeMethod" );
        base.SomeMethod();
    }
}
public class EntryPoint
{
    static void Main() {
        B b = new B();
        A a = b;
        a.SomeMethod();
    }
}
```

Как можно ожидать, вывод приведенного кода напечатает `A.SomeMethod` в строке, следующей после вывода `B.SomeMethod`. Является ли такой порядок событий правильным? Не должно ли быть все наоборот? Не должен ли `B.SomeMethod` вызвать версию базового класса перед тем, как выполнить свою работу? Дело в том, что у вас нет достаточной информации, чтобы ответить на этот вопрос. Здесь есть проблема с наследованием и переопределением виртуальных методов. Как вы можете знать, когда следует вызывать метод базового класса, и нужно ли это делать? Ответ заключается в том, что метод должен быть как следует документирован, чтобы вы могли принять правильное решение. Таким образом, наследование с виртуальными методами повышает вашу нагрузку за счет обязательного документирования, поскольку вы должны снабдить потребителей вашего класса информацией, выходящей за рамки простого общедоступного интерфейса. Например, если вы следуете шаблону не виртуального интерфейса (Non-Virtual Interface — NVI), который я опишу в главе 13, то виртуальный метод, находящийся под вопросом, будет объявлен как `protected`, и тогда вы должны документировать как общедоступные (`public`) методы, так и некоторые защищенные, и виртуальные методы должны ясно устанавливать, должен ли базовый класс вызывать их, и когда.

Методы sealed

По причинам, установленным ранее, я уверен, что вы должны по умолчанию герметизировать (sealed) ваши классы, и разрешать их наследование только в хорошо продуманных случаях. Сколько раз я видел иерархии, создавая которые разработчик думал: "Сделаю я, пожалуй, все методы виртуальными, чтобы обеспечить максимальную гибкость производным классам". Все, чего он добивался — это создание целого выводка вложенных ошибок, которые проявлялись позднее. Такой образ мышления характерен для менее опытных проектировщиков, захваченных сложностью иерархии и виртуальных методов. Тот факт, что наследование сопровождается виртуальными методами, настолько неожиданно сложен, что лучше явно выключать эту возможность, чем открывать ее для злоупотребления. Таким образом, при проектировании классов вы должны отдавать предпочтение созданию герметизированных, ненаследуемых классов, и тщательно документировать общедоступный интерфейс. Потребители, которым понадобится расширить функциональность, смогут это сделать, но не через наследование, а через включение. Расширение включением (containement) в сопровождении изощренных определений интерфейсов намного мощнее, чем наследованием классов.

В редких случаях вы наследуетесь от класса с виртуальными методами и хотите завершить виртуальную цепочку вашим переопределением. Другими словами, вы хотите запретить производным классам дальнейшее переопределение виртуального метода. Чтобы сделать это, вы также помечаете метод модификатором `sealed`. Как следует из его имени, это означает, что ни один производный класс не сможет переопределить данный метод. Можно, однако, представить метод с той же сигнатурой, если он помечен модификатором `new`, как говорилось в предыдущем разделе. Фактически, вы можете пометить новый метод `virtual`, тем самым начав новую виртуальную цепочку в иерархии. Это не то же самое, что герметизация всего класса, которая вообще запрещает дальнейшее наследование от этого класса. Таким образом, если производный класс помечен как `sealed`, то снабжение его методов модификатором `sealed` уже излишне.

Завершающие несколько слов о виртуальных методах C#

Ясно, что C# предусматривает множество гибких ключевых слов, позволяющих делать некоторые интересные вещи с наследованием и виртуальными методами. Однако тот факт, что язык предоставляет их, не означает, что ими нужно злоупотреблять. За последнее десятилетие многие эксперты опубликовали многочисленные книги, посвященные эффективному и безопасному проектированию приложений на C++ и Java. В этих работах чаще говорится о вещах, которых вы не должны делать, чем о вещах, которые делать нужно. Это потому, что C++, наряду с C#, предоставляет в ваше распоряжение арсенал средств, применение которых не обязательно укладываются в рамки требований хорошего дизайна. В конце концов, вы должны стараться создавать классы и конструкции, интуитивные в применении и свободные от скрытых сюрпризов.

Сообразительный читатель, возможно, отметит, что модификатор `new` — кратчайший путь к появлению некоторых серьезных сюрпризов в иерархии классов. Если вы обнаруживаете, что не можете обойтись без модификатора в определении метода, то, скорее всего, это свидетельствует о том, что вы используете класс таким образом, для которого он не предназначен. Может быть, вы наследуетесь от класса, который должен был быть помечен как `sealed`. Вы можете ругать разработчика этого класса за то, что он не пометил определенный метод модификатором `virtual`, чтобы вы легко могли переопределить его. И тогда вы прибегаете к применению модификатора `new`. Только на основании его существования не следует предполагать, что его нужно широко использовать. Проектировщик класса, от которого вы наследуетесь, возможно, просто не намеревался позволять вам выполнять наследование, и просто забыл указать `sealed` в объявлении своего класса. И даже хотя такой проектировщик нечаянно оставил свой класс не герметизированным, он, вероятно, не собирался позволять вам заменять метод, который вы пытаетесь переопределить. Таким образом, всегда старайтесь применять проверенные временем приемы проектирования и избегать этих "великолепных" средств языка, которые противоречат принципам хорошего дизайна.

Наследование, включение и делегирование

Когда несколько лет назад многие люди начали программировать на объектно-ориентированных языках, они считали наследование самым великолепным изобретением со времен бутерброда. Фактически, многие люди рассматривали его как неотъемлемую, важную часть объектно-ориентированного программирования. Некоторые утверждали, что язык, не поддерживающий наследования, вообще не является объектно-ориентированным. В споры на эту тему были вовлечены многие люди в течение многих лет, и периодически они принимали форму религиозных войн. Однако с истечением времени некоторые умные проектировщики стали замечать ловушки, присущие наследованию.

Выбор между интерфейсом и наследованием класса

Когда вы впервые открываете для себя наследование, то поначалу склонны злоупотреблять им. Сделать это легко. И такие злоупотребления затрудняют понимание и сопровождение программных проектов, особенно на таких языках, как C++, поддерживающих множественное наследование. Такой дизайн трудно адаптировать для будущих потребностей, что заставляет отбрасывать его и заменять совершенно новым. В языках, поддерживающих только одиночное наследование, таких как C# и Java, разработчик вынужден более тщательно подходить к применению наследования.

Например, моделируя систему управления кадровыми ресурсами в компании XYZ, наивный дизайнер может завести классы вроде `Payee` (получатель платежа), `BenefitsRecipient` (получатель дохода) и `Developer` (разработчик). Затем, применив множественное наследование, он может представить составной класс наемного разработчика `FulltimeDeveloper`, унаследовав его от всех трех, как показано на рис. 4.3.

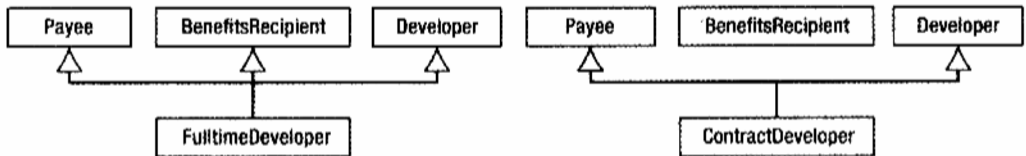

Рис. 4.3. Пример плохого наследования

Как видите, это заставит нашего горе-дизайнера создать новый класс для представления разработчиков по контракту, который не наследует `BenefitsRecipient`. По мере роста системы вы быстро обнаружите недостатки в дизайне, когда сетка наследования станет достаточно сложной и глубокой. Теперь у него получится два класса для разных типов разработчиков, что затруднит управление таким дизайном. А теперь рассмотрим неудачную попытку решить ту же проблему на языке, поддерживающем только одиночное наследование. На рис. 4.4 доказано, что такое решение не лучше предыдущего.

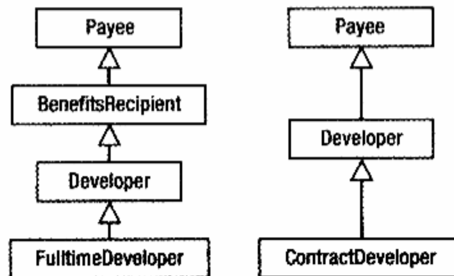

Рис. 4.4. Пример неудачной иерархии с одиночным наследованием

Если присмотреться, вы обнаружите присутствие некоторой неоднозначности. Невозможно, чтобы класс `Developer` мог наследоваться одновременно от `Payee` и `BenefitsRecipient` в среде, где разрешено только одиночное наследование. По этой причине не может быть двух иерархий в одном и том же дизайне. Вам при-

дется создать два разных варианта класса Developer — один для наследования от него FulltimeDeveloper, а другой — для наследования ContractDeveloper. Однако это будет пустой тратой времени. Что более важно, при этом пропадает возможность многократного использования кода — главное преимущество наследования, если вы должны создавать две версии по существу одного и того же класса.

Более удачный подход заключается в том, чтобы иметь класс Developer, содержащий различные свойства, которые представляют определенные характеристики разработчиков внутри компании. Например, поддержка специфического интерфейса может характеризоваться поддержкой определенного свойства. Иерархия наследования с большим количеством уровней глубины — явный признак того, что дизайн нуждается в переосмыслении.

Чтобы увидеть, что же в действительности происходит здесь, давайте задержимся на минуту и проанализируем, что именно означает для вас наследование. В действительности оно позволяет вам бесплатно сэкономить часть работы за счет наследования реализации. Есть существенное отличие между простым наследованием и наследованием реализации. Хотя объектно-ориентированные языки, включая C#, обычно используют сходный синтаксис для этих двух видов наследования, важно отметить, что классы, реализующие интерфейс, не получают никакой готовой реализации вообще. При использовании наследования, однако, вы не только наследуете общедоступный контракт класса, но также наследуете компоновку, или все его внутренности.

Хорошее эмпирическое правило заключается в том, что если вашей целью является, прежде всего, наследование контракта, выберите реализацию интерфейса вместо наследования. Это гарантирует вашему дизайну большую гибкость. Чтобы понять, почему это так, давайте исследуем подводные камни, присущие наследованию.

Сравнение делегирования и композиции и наследования

Другой важный аспект наследования заключается в том, что оно может быть вредным: *наследование может разрушить инкапсуляцию и всегда повышает связность*. Уверен, что все согласны с тем, что инкапсуляция — наиболее фундаментальная и важная объектно-ориентированная концепция. Если это так, то зачем разрушать ее? Еще всякий раз, используя инкапсуляцию, когда базовый тип содержит защищенные поля, вы нарушаете целостность оболочки инкапсуляции и открываете внутренности базового класса. Это не может быть хорошо. Разрешите мне объяснить, почему это не так, и какие альтернативы есть в вашем распоряжении, которые позволят разработать лучший дизайн.

Многие описывают наследование как повторное использование "белого ящика". Лучшая форма повторного использования — "черный ящик", когда внутренности объекта не открываются вам. Вы можете достичь этого, применив отношение включения (containment). Да, это правильно. Вместо наследования нового класса от другого, вы можете включить экземпляр этого другого класса в ваш, тем самым повторно используя класс включенного типа, без нарушения инкапсуляции. Недостаток такого приема состоит в том, что в большинстве языков, включая C#, это потребует чуть больше работы по кодированию, хотя и не намного больше. Зато в результате вы получите более адаптируемый дизайн.

В качестве простого примера того, о чем я говорю, рассмотрим проблемную область, где класс обслуживает некоторого рода специальные сетевые коммуникации. Назовем этот класс NetworkCommunicator, и представим, что он будет выглядеть следующим образом:

```
public class NetworkCommunicator
{
    public void SendData( DataObject obj )
    {
        // Отправить данные по проводам
    }
    public DataObject ReceiveData()
    {
        // Принять данные по проводам
    }
}
```

Теперь предположим, что вы вернулись к нему позже и решили, что было бы неплохо иметь объект EncryptedNetworkCommunicator, в котором данные шифруются перед отправкой. Распространенный подход — унаследовать EncryptedNetworkCommunicator от NetworkCommunicator. Тогда реализация должна выглядеть так:

```
public class EncryptedNetworkCommunicator : NetworkCommunicator
{
    public override void SendData( DataObject obj )
    {
        // Зашифровать данные
        base.SendData( obj );
    }
    public override DataObject ReceiveData()
    {
        DataObject obj = base.ReceiveData();
        // Расшифровать данные
        return obj;
    }
}
```

У такого варианта есть большой недостаток. Прежде всего, хороший дизайн требует, что когда вы собираетесь модифицировать функциональность методов базового класса, то должны переопределить их. Чтобы переопределить их правильно, вы сразу должны объявить их как virtual. Это потребует от вас предвидения будущего при проектировании класса NetworkCommunicator и пометки методов модификатором virtual. Да, вы можете скрыть их в C#, используя ключевое слово new при определении методов производного класса. Но если вы сделаете это, то нарушите принцип, гласящий, что наследование моделирует отношение "является" (is-a). Теперь рассмотрим ситуацию с включением:

```
public class EncryptedNetworkCommunicator
{
    public EncryptedNetworkCommunicator()
    {
        contained = new NetworkCommunicator();
    }
```

```
    public void SendData( DataObject obj )
    {
            // Зашифровать данные
            contained.SendData( obj );
    }
    public DataObject ReceiveData()
    {
            DataObject obj = contained.ReceiveData();
            // Расшифровать данные
            return obj;
    }
    private NetworkCommunicator contained;
}
```

Как видите, работы лишь ненамного больше. Но плюс состоит в том, что вы можете повторно использовать NetworkCommunicator, как если бы он был черным ящиком. Разработчики NetworkCommunicator могли сделать его sealed, а вы все равно смогли бы повторно его использовать. Будь он sealed, вы по определению не смогли бы наследовать от него.

Другой недостаток применения наследования заключается в том, что оно не динамично. Наследование статично в силу того факта, что определяется на этапе компиляции. Такое положение дел, как минимум, может оказаться весьма стесняющим. Применяя включение, вы устраняете это ограничение. Однако чтобы сделать это, вы также должны заручиться поддержкой хорошего друга — полиморфизма. При этом включающий тип может быть, скажем, интерфейсным типом. Тогда включаемый объект просто должен поддерживать контракт этого интерфейса, чтобы повторно использоваться контейнером. Более того, вы можете изменить этот объект во время выполнения. Задумайтесь об этом на минутку. Предположим, что есть объект, представляющий собой контейнер сортируемых объектов. Предположим, что контейнерный тип предусматривает алгоритм сортировки по умолчанию. Если вы реализуете этот алгоритм по умолчанию как включаемый тип, который можно подменить во время выполнения, тогда, если того требует предметная область, вы всегда сможете заменить его собственным специальным алгоритмом сортировки — до тех пор, пока объект нового алгоритма сортировки реализует необходимый интерфейс, ожидаемый контейнерным типом. Такая техника известна как шаблон проектирования Strategy (Стратегия). Блестящий пример использования этого шаблона проектирования вы только что видели.

В итоге вы можете видеть, что дизайн с динамическими конструкциями является намного более гибким, чем при использовании конструкций статических. Сюда относится и предпочтение включения наследованию во многих случаях повторного использования. Такой тип повторного использования известен как делегирование, поскольку работа делегируется включаемому типу. Включение также предохраняет инкапсуляцию, в то время как наследование ее разрушает. Однако, тем не менее, нужно сделать одно предупреждение. Как почти с любым техническим приемом, вы можете переусердствовать с включением. Для мелких служебных классов может быть и не стоит предпринимать слишком много усилий для предпочтения включения. И в некоторых случаях вам нужно использовать наследование для реализации специализации. Но если говорить в целом, дизайн, предпочитающий наследованию включение в качестве механизма повторного использования, дает гораздо больше

гибкости и гораздо лучше выдерживает испытание временем. Всегда учитывайте мощь наследования, включая возможные неприятные последствия от злоупотребления им.

Резюме

В этой очень длинной главе я осветил важные моменты, относящиеся к системе типов C#, которая позволяет вам создавать новые типы, оснащенные всеми возможностями встроенных типов, определенных исполняющей системой. Я начал с описания определений классов, применяемых для определения новых ссылочных типов, затем я продолжил тему, описав определения структур, используемых для создания экземпляров новых типов значений внутри CLR, и описал основные отличия между этими двумя категориями типов. В непосредственной связи с темой типов значений находится тема упаковки и распаковки — операций, которые могут привести к нежелательным последствиям, если вы не знаете, где именно упаковка может быть вставлена компилятором. (В главе 11, посвященной обобщениям, вы увидите, как в некоторых случаях можно вообще избежать упаковки и распаковки.)

Затем я обратился к сложной теме создания и инициализации объектов, а также их уничтожения. Уничтожение — довольно сложная тема в рамках CLR, поскольку ваши ссылочные типы могут поддерживать как детерминированное, так и недетерминированное уничтожение. (Более подробно я расскажу об уничтожении в главе 13, где приведу больше примеров.) Далее я вкратце рассказал о перегрузке методов в C# и различных модификаторах, которыми можно снабдить методы, чтобы управлять их модификацией: `virtual`, `override` или `sealed`. И, наконец, я потратил некоторое время на обсуждение наследования, полиморфизма и включения, а также привел несколько советов относительно выбора между ними.

Последний раздел настоящей главы ведет нас прямо к следующей, где я раскрою важнейшую тему интерфейсно-ориентированного, или контрактно-ориентированного, программирования, а также способов его применения в CLR.

ГЛАВА 5

Интерфейсы и контракты

В те времена, когда вы начинали заниматься разработкой программного обеспечения, наверняка вы встречали упоминание интерфейс-ориентированного программирования. Если вы знакомы с фундаментальной книгой Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) (известных, как "банда четырех") Design Patterns: Elements of Reusable Object-Oriented Software[1], то вы знаете, что многие шаблоны проектирования используют "контракты" в стиле интерфейсов. Если вы не знакомы с этой книгой и изложенных в ней концепциях, я рекомендую прочесть ее. Моей целью в настоящей главе является показать, как моделировать хорошо определенные, поддерживающие версии контракты с использованием интерфейсов. В этом контексте контракт — это соглашение типа на поддержку набора функциональности.

Если вам случалось вести разработку с использованием COM и CORBA в течение последних лет, то вы наверняка занимались именно интерфейсно-ориентированной разработкой. Фактически интерфейс — единственная форма взаимодействия между компонентами в COM. Поэтому большая часть сложности дизайна заключается в разработке надежных интерфейсов, прежде чем вы напишете хоть одну строку кода реализации. Отказ от следования этой парадигме становится источником многих проблем. Например, Visual Studio 2003 предоставляет удобную среду для создания Web-служб. Просто аннотируя методы класса определенным образом, вы можете представить эти методы как методы Web-службы. Однако при этом IDE-среда воспитывает подход, который подразумевает, что интерфейс — это результат аннотации методов класса, а не наоборот. Таким образом, телега ставится впереди лошади. На самом деле вместо этого вы должны четко определить интерфейс Web-службы перед тем, как начать какое-либо кодирование, а только потом кодировать реализацию этого интерфейса. Чтобы привести пример хоть одного преимущества такого подхода, можно заметить, что при этом вы можете кодировать параллельно серверную и клиентскую части, а не одну за другой. Другая часть проблемы состоит в том, что, однажды опубликовав интерфейс для всего мира, вы не можете изменить его. Подобный шаг немедленно разрушит работу всех реализаций, основанных на нем. К сожалению, среда Visual Studio поощряет нарушение этого пра-

[1] Эта книга упоминается в библиографии в конце настоящей книги.

вила, облегчая возможность добавления новых методов в класс и аннотирования их как методов Web-службы.

В хорошо спроектированной интерфейсно-ориентированной системе, такой как системы архитектуры, ориентированной на службы (service-oriented architecture — SOA), вы всегда должны сначала разрабатывать интерфейс как контракт между компонентами. Контракт управляет реализацией, а не наоборот — реализация управляет, или определяет, контракт. К сожалению, слишком много инструментов в прошлом и даже в настоящем стимулируют такую разработку "задом наперед". Но это не значит, что вы должны безропотно следовать их ошибочному пути. В конце концов, контракт, примененный к типу, определяет набор требований к этому типу. Не имеет смысла в том, чтобы сами типы определяли требования, предъявляемые к ним. В среде .NET интерфейсы являются типами.

Интерфейсы определяют типы

Объявление интерфейса определяет ссылочный тип. В переменных этого типа вы можете хранить ссылки на объект, реализующий контракт типа интерфейса. Каждая переменная в CLR хранится в определенном месте памяти, будь то куча или стек. Каждое место хранения имеет ассоциированный с ним тип. Когда переменная — скажем, ссылка на объект — находится в этом месте, она должна быть того же типа, что и это местоположение, либо допускать преобразование к типу, ассоциированному с местоположением. Если она может быть преобразована автоматически в тип местоположения, тогда она является неявно преобразуемой к типу этого места хранения.

Многие примеры используют воображаемый каркас графического интерфейса (GUI) в качестве основы для демонстрации, так что я поступлю точно также. Взгляните на следующий фрагмент кода:

```
public interface IUIControl
{
    void Paint();
}

public class Button : IUIControl
{
    public void Paint() {
    // Нарисовать кнопку
    }
}

public class ListBox : IUIControl
{
    public void Paint() {
    // Нарисовать окно списка
    }
}
```

Этот пример объявляет интерфейс `IUIControl`, который просто предоставляет один метод — `Paint`. Данный интерфейс определяет контракт, который устанавливает, что любой тип, реализующий этот интерфейс, должен реализовать

метод Paint. Конечно, весьма желательно наличие некоторой документации, описывающей семантическое значение того, что должен делать Paint. Например, вы можете представить, что интерфейс по имени IArtist может иметь метод Paint, но его смысл будет совершенно другим, чем в предыдущем примере, т.е. IUIControl.Paint, вероятно, попросит элемент управления нарисовать себя, в то время, как IArtist.Paint, скорее всего, означает, что художник должен рисовать что-нибудь.

На заметку! Я обнаружил, что удобно называть методы в соответствие как с действием, которое они выполняют, так и с направлением этого действия. Например, предположим, что метод IUIControl.Paint принимает в качестве параметра объект Graphics, сообщающий ему, где он должен рисовать себя. По моему мнению, код будет более читабельным, если метод назвать IUIControl.PaintSelfTo. Таким образом, вызов метода читается как предложение на естественном языке — в том смысле, что вызов метода вроде control.PaintSelfTo(myGraphicsObject) говорит: "control, пожалуйста, нарисуй себя на myGraphicsObject".

Как только классы ListBox и Button из предыдущего примера реализуют интерфейс, оба они могут трактоваться как относящиеся к типу IUIControl. Полезно рассмотреть, как CLR управляет этой ситуацией. Если вы попытаетесь сохранить любой экземпляр ListBox или Button в переменной типа IUIControl, такая операция удастся. Ссылки на эти конкретные типы неявно преобразуемы в интерфейсный тип IUIControl, поскольку оба реализуют этот интерфейс. Однако чтобы привести ссылку на IUIControl обратно к ListBox или Button, потребуется явное приведение, и это приведение может потерпеть неудачу во время выполнения, если на самом деле ссылка IUIControl не будет указывать на экземпляр нужного конкретного типа.

Определение интерфейсов

В предыдущем разделе вы получили представление о том, на что похожи объявления интерфейсов C#. Они выглядят подобно объявлениям классов, где ключевое слово class заменено interface, а методы не имеют тела. Однако обратите внимание еще на некоторые важные моменты. Если вы следуете рекомендованным соглашениям, то имена ваших интерфейсов должны начинаться с буквы I. Таким образом, вы сразу легко сможете заменить интерфейсные типы в коде. Интерфейсы могут иметь модификаторы доступа, прикрепленные к ним. Они указывают на то, видимо ли объявление интерфейса вне сборки. Поскольку большинство интерфейсов представляют собой контракты взаимодействия между поставщиками и потребителями, объявления интерфейсов обычно объявляются public.

Члены интерфейса не могут иметь никаких модификаторов доступа. Однако они могут быть декорированы модификатором new, о котором я расскажу ниже. Члены интерфейсов всегда неявно общедоступны (public). Какой был бы смысл в наличии не общедоступных членов интерфейса, если назначение интерфейса — позволить двум объектам взаимодействовать друг с другом?

Интерфейсы определяют контракты

Чтобы подчеркнуть, что интерфейс только специфицирует контракт, я хочу привести аналогию между объявлением интерфейса и языками IDL (Interface Description Language — язык описания интерфейсов) и WSDL (Web Services Description Language — язык описания Web-служб). И COM, и CORBA используют IDL для определения интерфейсов. Его синтаксис подобен C++. Обычно IDL передается через транслятор, такой как `midl.exe` для COM, чтобы генерировать оболочки и, возможно, прокси и заглушки для любого языка, который вам нужен. Другой пример — WSDL, хотя гораздо более выразительный, чем IDL. Схема XML определяет формат WSDL, а документ WSDL используется для описания контракта, или интерфейса, сетевой службы. Порядок использования подобен IDL. Имея документ WSDL, вы прогоняете его через транслятор для любого языка, который вы используете для реализации или потребления службы. Транслятор помогает вам, генерируя оболочку реализации, или интерфейсы, в виде, понятном используемому вами языку. Объявление и потребление интерфейсов в среде .NET должно следовать одному и тому же шаблону.

На практике обычно имеет смысл помещать объявления ваших интерфейсов в отдельную сборку, содержащую только определения интерфейсов и константы, чтобы потребитель и поставщик могли основывать свои реализации в точности на одной и той же версии интерфейса.

Что может быть интерфейсом?

Объявления интерфейсов могут включать ноль или более методов, свойств, событий и индексаторов. Все они неявно общедоступны и не могут быть статическими. Интерфейсы могут наследоваться от одного или более других интерфейсов. Синтаксис точно такой, как и при наследовании класса. Когда речь идет об интерфейсах, я предпочитаю считать, что если интерфейс `B` наследуется от интерфейса `A`, это значит, что если вы реализовали интерфейс `B`, то также обязаны реализовать и интерфейс `A`. Наследование классов просто подразумевает отношение "является" (is-a), где базовая реализация также наследуется. Хотя наследование интерфейсов заимствует тот же синтаксис, что и у наследования классов, было бы неправильно рассматривать их как одно и то же, поскольку наследование интерфейсов объявляет обобщение, а никакой реализации при этом не наследуется. Таким образом, всякий раз, когда вы говорите о наследовании интерфейса, попробуйте думать об этом в терминах отношения "реализует". Это станет яснее, когда я расскажу о том, как производный класс может заново реализовать интерфейсы и как компилятор осуществляет отображение реализации интерфейса на конкретные типы, реализующие интерфейс. Ниже приведен пример того, как можно объявить интерфейс:

```
public delegate void DBEvent( IMyDatabase sender );[2]
public interface IMyDatabase : ISerializable, IDisposable
{
    void Insert( object element );
    int Count { get; }
    object this[ int index ] { get; set; }
    event DBEvent dbChanged;
}
```

[2] Не беспокойтесь, если вы незнакомы с ключевым словом `delegate`, и тем, как делегаты используются для объявления событий. Исчерпывающее рассмотрение делегатов и событий вы найдете в главе 10.

В данном примере IMyDatabase также реализует ISerializable и IDisposable. Поэтому любой конкретный тип, реализующий IMyDatabase, также должен реализовать ISerializable и IDisposable; в противном случае этот конкретный тип не будет компилироваться. Если вы скомпилируете приведенный фрагмент кода в сборку и заглянете в нее с помощью ILDASM, то увидите, что тип IMyDatabase не содержит ничего помимо объявлений методов. Конечно, некоторые из их будут иметь специальные имена, основанные на том факте, что они являются средствами доступа к свойству, индексатором или событием.

Наследование интерфейсов и сокрытие членов

Как упоминалось ранее, интерфейсы поддерживают множественное наследование от других интерфейсов в синтаксическом смысле. Как и с множественным наследованием в C++, вы можете также строить ромбовидные решеточные иерархии, как в следующем коде:

```
public interface IUIControl
{
    void Paint();
}
public interface IEditBox : IUIControl
{
}
public interface IDropList : IUIControl
{
}
public class ComboBox : IEditBox, IDropList
{
    public void Paint() {
    // рисовать реализацию ComboBox
    }
}
```

В этом примере оба интерфейса — IEditBox и IDropList — реализуют интерфейс IUIControl. И поскольку ComboBox реализует оба эти интерфейса, он должен реализовать объединение всех методов, объявленных в интерфейсах, которые он реализует непосредственно, плюс методы тех интерфейсов, от которых они наследуются, рекурсивно. В данном случае сюда относится только метод IUIControl.Paint.

Достаточно просто: все методы из всех интерфейсов объединяются вместе в одно большое объединение, формируя набор методов, который должен реализовать конкретный класс или структура. Поэтому класс ComboBox получает только одну реализацию метода Paint. Если вы выполните приведение экземпляра ComboBox к ссылке IEditBox и ссылке IdropList, то вызов Paint через любую из них приведет к вызову одной и той же реализации метода.

На заметку! Если вы пришли из мира "родного" C++, то вам должна быть знакома вся сложность множественного наследования в ромбовидных диаграммах наследования, и как оно связано с виртуальным наследованием в C++, а также с множествами сгенерированных компилятором виртуальных таблиц. Чтобы понять, в чем здесь состоит отличие C#, представьте, что C# сливает все эти виртуальные таблицы во время компиляции в одну большую таблицу.

Иногда — хотя и редко — вам понадобится объявить метод в интерфейсе, который скрывает метод унаследованного интерфейса. Вы можете использовать ключевое слово new, если хотите предотвратить выдачу предупреждения компилятором.

На заметку! Традиционно объектно-ориентированный анализ и проектирование (OOA-D) считает примером плохого дизайна сокрытие невиртуальных унаследованных членов. Реализация, которая в действительности вызывается, зависит от типа ссылки, даже если две ссылки указывают на один и тот же экземпляр. Например, если A.DoWork — невиртуальный метод, и B наследуется от A и вводит новый метод B.DoWork, который скрывает базовый, то вызов DoWork на ссылке типа B вызовет B.DoWork, а вызов того же метода на ссылке типа A, полученной приведением ссылки на B к типу A, вызовет A.DoWork. Такое поведение не интуитивно для объектно-ориентированных систем. Только потому, что язык позволяет вам делать что-то, это вовсе не означает, что делать это — правильно. Теперь вы видите, зачем прежде всего необходимы предупреждения компилятора.

В следующем примере IEditBox по той или иной причине должен объявлять метод Paint, чья сигнатура в точности совпадает с сигнатурой метода из IUIControl. Поэтому здесь необходимо использовать ключевое слово new:

```
using System;
public interface IUIControl
{
    void Paint();
}
public interface IEditBox : IUIControl
{
    new void Paint();
}
public interface IDropList : IUIControl
{
}
public class ComboBox : IEditBox, IDropList
{
    public void Paint() {
        Console.WriteLine( "ComboBox.IEditBox.Paint()" );
    }
}
public class EntryPoint
{
    static void Main() {
        ComboBox cb = new ComboBox();
        cb.Paint();
        ((IEditBox)cb).Paint();
        ((IDropList)cb).Paint();
        ((IUIControl)cb).Paint();
    }
}
```

При всех вызовах Paint в методе Main они всегда превращаются в вызовы ComboBox.Paint. Это потому, что все обязательные методы, которые ComboBox должен реализовать, сливаются в один большой набор. Обе сигнатуры Paint — одна

из IEditBox, и одна из IUIControl — сливаются в один элемент обязательного списка. В конце оба они отображаются на ComboBox.Paint. Вы можете изменить такое поведение, применив явную реализацию интерфейса (о которой я расскажу в разделе "Явная реализация интерфейса"), где ComboBox может выбирать между двумя разными версиями Paint — одной из IEditBox, и одной из IUIControl.

Когда интерфейс IEditBox объявляет метод Paint, используя ключевое слово new, тем самым он скрывает метод Paint, объявленный в IUIControl. Когда вы вызываете ComboBox.Paint, то вызывается метод IEditBox.Paint, как если выбран IEditBox-путь в иерархии наследования. По сути, в любой момент, когда любой путь скрывает метод, то он скрывает метод для всех путей. Это станет понятнее, когда я расскажу, как компилятор находит соответствие конкретного метода методу интерфейса, когда вызывается метод интерфейса. Этот процесс называется отображением интерфейса, и я расскажу о нем далее в разделе "Правила сопоставления членов интерфейсов" настоящей главы.

Реализация интерфейсов

При реализации интерфейсов в C# вы должны выбрать его реализацию одним из двух способов. По умолчанию реализации интерфейсов являются неявными реализациями. Реализация метода — часть общедоступного контракта класса, но также она реализует интерфейс неявно. Альтернативно вы можете реализовать интерфейс явным образом, причем реализации метода является приватной по отношению к реализующему классу и не становится частью общедоступного интерфейса. Явная реализация обеспечивает некоторую гибкость, особенно при реализации двух интерфейсов с одноименными методами в них.

Неявная реализация интерфейса

Когда конкретный тип реализует методы наследуемых интерфейсов, и эти методы помечены как public, это известно как неявная реализация интерфейса. Что хорошего, когда конкретный тип реализует контракт определенного интерфейса, а потребитель объектов этого типа не может вызвать методы этого контракта? Например, следующий код некорректен:

```
public interface IUIControl
{
    void Paint();
}

public class StaticText : IUIControl
{
    void Paint(); // !!! НЕ КОМПИЛИРУЕТСЯ !!!
}
```

Если вы попытаетесь скомпилировать это, компилятор немедленно пожалуется на то, что класс StaticText не реализовал все методы унаследованного интерфейса — в данном случае, IUIControl. Чтобы это заработало, вы должны переписать код следующим образом:

```
public interface IUIControl
{
    void Paint();
}
public class StaticText : IUIControl
{
    public void Paint(); //К объявлению метода добавлено 'public'
}
```

Теперь код не только скомпилируется, но при вызове `Paint` через ссылку на `StaticText` или через ссылку на `IUIControl`, будет вызываться метод `StaticText.Paint`. Таким образом, потребители могут трактовать экземпляры `StaticText` полиморфно как экземпляры `IUIControl`.

Явная реализация интерфейса

Когда конкретный тип реализует интерфейс явно, его методы также становятся частью общедоступного контракта самого конкретного типа. Однако не всегда нужно, чтобы методы реализации интерфейса становились частью общедоступного интерфейса класса, реализующего этот интерфейс. Например, класс `System.IO.FileStream` реализует `IDisposable`, но вы не должны вызывать `Dispose` через экземпляр `FileStream`. Вместо этого вы должны сначала выполнить приведение ссылки на объект `FileStream` к интерфейсу `IDisposable`, а затем можно вызывать `Dispose`. Когда вам понадобится такое поведение для ваших собственных типов, вы должны реализовать интерфейсы, используя явную реализацию.

На заметку! Чтобы достичь того же результата, что от `Dispose`, но через ссылку на объект `FileStream`, вы должны вызвать `FileStream.Close`. В реализации `FileStream.Close` вызывается напрямую внутренняя реализация метода `Dispose`. Зачем разработчикам `FileStream` это понадобилось? Скорее всего, потому, что в лингвистическом отношении "close" (закрыть) по отношению к файлу более осмысленно, чем "dispose of" (освободить).

Вы можете также использовать явную реализацию для предоставления отдельных реализаций перекрывающихся методов из унаследованных интерфейсов. Давайте еще раз вернемся к примеру `ComboBox` из предыдущего раздела. Если вы хотите представить отдельные реализации для `IEditBox.Paint` и `IUIControl.Paint` внутри `ComboBox`, то для этого нужно использовать явную реализацию интерфейсов, как показано ниже:

```
using System;
public interface IUIControl
{
    void Paint();
}
public interface IEditBox : IUIControl
{
    new void Paint();
}
public interface IDropList : IUIControl
{
}
```

```
public class ComboBox : IEditBox, IDropList
{
    void IEditBox.Paint() {
        Console.WriteLine( "ComboBox.IEditBox.Paint()" );
    }
    void IUIControl.Paint() {
        Console.WriteLine( "ComboBox.IUIControl.Paint()" );
    }
    public void Paint() {
        ((IUIControl)this).Paint();
    }
}
public class EntryPoint
{
    static void Main() {
        ComboBox cb = new ComboBox();
        cb.Paint();
        ((IEditBox)cb).Paint();
        ((IDropList)cb).Paint();
        ((IUIControl)cb).Paint();
    }
}
```

Обратите внимание на изменение в синтаксисе. Теперь ComboBox содержит три реализации Paint. Одна специфична для интерфейса IEditBox, другая — специфична для интерфейса IUIControl, а последняя предназначена просто для удобства, чтобы предоставить метод Paint общедоступному интерфейсу класса ComboBox. Когда вы реализуете методы интерфейса явно, то не только добавляете имя интерфейса, отделенное точкой, к имени метода, но также удаляете модификатор доступа. Это исключает его из общедоступного контракта ComboBox. Однако явные реализации интерфейсов не является точно приватными в том смысле, что вы можете вызывать их после приведения экземпляра ComboBox к требуемому типу интерфейса. В моей реализации ComboBox.Paint — той, что относится к общедоступному контракту ComboBox — я выбираю, какую версию Paint вызвать. В этом случае я выбрал IUIControl.Paint. Точно также легко я мог бы выбрать реализацию IEditBox.Paint явно и IUIControl.Paint неявно, тогда мне не понадобилась бы третья реализация Paint. Но в этом случае я полагаю, что реализация собственного метода Paint добавляет больше гибкости и более оправдана для ComboBox, чтобы он мог использовать другую реализацию, в то же время добавив ей ценности. Если вы скомпилируете и запустите предыдущий пример, то увидите вывод, подобный следующему:

```
ComboBox.IUIControl.Paint()
ComboBox.IEditBox.Paint()
ComboBox.IUIControl.Paint()
ComboBox.IUIControl.Paint()
```

Конечно, этот пример довольно надуманный, но он предназначен для того, чтобы продемонстрировать сложность явных реализаций интерфейсов и сокрытие членов при множественном наследовании интерфейсов.

Переопределение реализаций интерфейсов в производных классах

Предположим, что у вас есть удобная реализация ComboBox, как в предыдущем разделе, и разработчик решил не герметизировать этот класс, чтобы вы могли наследоваться от него.

На заметку! Я советую вам объявлять все свои классы как sealed, если только вы явно не собираетесь наследоваться от них. В главе 4 я объяснил подробно, почему это желательно.

Теперь представим, что вы создаете новый класс FancyComboBox и хотите, чтобы он как-то иначе себя рисовал — может быть, в некоторой новой психоделической теме. Вы можете попробовать что-нибудь вроде следующего:

```
using System;
public interface IUIControl
{
    void Paint();
    void Show();
}
public interface IEditBox : IUIControl
{
    void SelectText();
}
public interface IDropList : IUIControl
{
    void ShowList();
}
public class ComboBox : IEditBox, IDropList
{
    public void Paint() { }
    public void Show() { }
    public void SelectText() { }
    public void ShowList() { }
}
public class FancyComboBox : ComboBox
{
    public void Paint() { }
}
public class EntryPoint
{
    static void Main() {
        FancyComboBox cb = new FancyComboBox();
    }
}
```

Однако компилятор предупредит вас о том, что FancyComboBox.Paint скрывает ComboBox.Paint, и что вы, возможно, подразумевали использование ключевого слова new. Это покажется неожиданным, если вы предполагаете, что методы, реализующие методы интерфейса, должны быть автоматически виртуальными. В C# это не так.

На заметку! "За кулисами" реализации методов интерфейсов вызываются так, будто они являются виртуальными. Любые реализации метода интерфейса, не помеченные `virtual` в C#, помечаются как `virtual` и `final` (герметизированный) в сгенерированном коде IL. Если же метод помечен как `virtual` в C#, то в сгенерированном коде IL он будет помечен как `virtual` и `newslot` (новый). Это может послужить причиной некоторой путаницы.

Столкнувшись с подобной проблемой, у вас есть два выбора. Один — заново реализовать интерфейс `IUIControl` в классе `FancyComboBox`:

```
using System;
public interface IUIControl
{
    void Paint();
    void Show();
}
public interface IEditBox : IUIControl
{
    void SelectText();
}
public interface IDropList : IUIControl
{
    void ShowList();
}
public class ComboBox : IEditBox, IDropList
{
    public void Paint() {
        Console.WriteLine( "ComboBox.Paint()" );
    }
    public void Show() { }
    public void SelectText() { }
    public void ShowList() { }
}
public class FancyComboBox : ComboBox, IUIControl
{
    public new void Paint() {
        Console.WriteLine( "FancyComboBox.Paint()" );
    }
}
public class EntryPoint
{
    static void Main() {
        FancyComboBox cb = new FancyComboBox();
        cb.Paint();
        ((IUIControl)cb).Paint();
        ((IEditBox)cb).Paint();
    }
}
```

В этом примере следует отметить два момента. Во-первых, `FancyComboBox` перечисляет `IUIControl` в списке наследования. Так вы указываете, что `FancyComboBox` собирается заново реализовать интерфейс `IUIControl`. Если бы `IUIControl` насле-

довался от другого интерфейса, то FancyComboBox пришлось бы повторно реализовать методы унаследованного интерфейса. Я также должен был использовать ключевое слово new для FancyComboBox.Paint, поскольку он скрывает CombBox.Paint. Это не было бы проблемой, если бы ComboBox реализовал метод IUIControl.Paint явно, поскольку он не был бы частью общедоступного контракта. Когда компилятор находит соответствие метода класса методу интерфейса, он также просматривает общедоступные методы базовых классов. В реальности FancyComboBox должен был бы указать, что он заново реализует IUIControl.Paint, но без повторного объявления его методов, так что компилятор должен был бы просто связать интерфейс с методами базового класса. Конечно, это было бы бессмысленно, поскольку причина повторной реализации интерфейса в производном классе — изменение поведения.

На заметку! Возможность заново реализовать интерфейс — мощное средство. Оно высвечивает огромную разницу между способом обработки интерфейсов в C# и CLR и трактовкой в C++ интерфейсов как определения абстрактных классов. Исключается сложность виртуальных таблиц C++, а вместе с ними и вопрос о том, когда вы должны применять виртуальное наследование C++. Как я уже говорил, и не устаю повторять вновь, интерфейсы C#/CLR — это не что иное, чем просто контракт, который гласит: "Вы, мистер Конкретный Класс, согласны реализовать все эти методы, перечисленные в контракте, т. е. интерфейсе".

Когда вы реализуете методы контракта интерфейса неявно, они должны быть общедоступны. Пока они отвечают этим требованиям, они также могут иметь другие атрибуты, включая ключевое слово virtual. Фактически реализация интерфейса IUIControl в классе ComboBox с применением виртуальных методов, в противоположность не виртуальным, несколько облегчает решение предыдущей проблемы, как показано ниже:

```
using System;
public interface IUIControl
{
    void Paint();
    void Show();
}
public interface IEditBox : IUIControl
{
    void SelectText();
}
public interface IDropList : IUIControl
{
    void ShowList();
}
public class ComboBox : IEditBox, IDropList
{
    public virtual void Paint() {
        Console.WriteLine( "ComboBox.Paint()" );
    }
    public void Show() { }
    public void SelectText() { }
    public void ShowList() { }
}
```

```
public class FancyComboBox : ComboBox
{
    public override void Paint() {
        Console.WriteLine( "FancyComboBox.Paint()" );
    }
}
public class EntryPoint
{
    static void Main() {
        FancyComboBox cb = new FancyComboBox();
        cb.Paint();
        ((IUIControl)cb).Paint();
        ((IEditBox)cb).Paint();
    }
}
```

В этом случае FancyComboBox не обязан реализовать IUIControl. Он должен просто переопределить виртуальный метод CombBox.Paint. Намного яснее для CombBox сразу объявлять Paint как virtual. Всякий раз, когда вам приходится использовать ключевое слово new для подавления предупреждений компилятора о сокрытии метода, рассмотрим возможность объявления метода базового класса virtual.

Внимание! Сокрытие методов вызывает путаницу и затрудняет понимание и отладку кода. И снова напомню: вы не должны делать что-либо только потому, что язык это позволяет.

Конечно, разработчику CombBox нужно было бы подумать заранее о том, что кто-то пожелает наследоваться от класса CombBox, и предвидеть эти сложности. По моему мнению, лучше герметизировать класс и избежать любых сюрпризов от людей, которые пытаются наследовать ваш класс, когда вы никак не предназначали его для этого. Представьте, какой шум они поднимут, столкнувшись с проблемой. Случалось ли вам когда-нибудь в прошлом работать с библиотекой Microsoft Foundation Classes (MFC) и попадать в ситуацию, когда вы рвете на себе волосы, пытаясь наследоваться от класса MFC и мечтая о том, чтобы какой-то определенный метод был виртуальным? В таком случае легко проклинать проектировщиков MFC за чудовищную непредусмотрительность, которая проявилась в том, что они не сделали метод виртуальным, когда в действительности дело в том, что у них, вероятно, и в мыслях не было, что кто-то захочет наследоваться от этого класса. В главе 13 описано, как в такой ситуации наследование можно заменить включением.

Берегитесь побочных эффектов от реализации интерфейсов типами значений

Все примеры, приведенные до сих пор, показывали, как классы могут реализовывать методы интерфейсов. На самом деле типы значений также могут реализовывать интерфейсы. Однако при этом проявляется один главный побочный эффект. Если вы приводите тип значений к интерфейсному типу, то это приводит к упаковке. Хуже того, если вы модифицируете значение через ссылку на интерфейс, то тем самым модифицируете упакованную копию, а не оригинал. Учитывая слож-

ности, присущие упаковке, которые я описал в главах 4 и 13, вы можете счесть такое поведение нежелательным.

Возьмем для примера System.Int32. Я уверен, вы согласитесь, что это — один из самых базовых типов CLR. Однако известно это вам или нет, но он реализует несколько интерфейсов: IComparable, IFormattable и IConvertible. Рассмотрим, к примеру, реализацию System.Int32 интерфейса IConvertible. Все методы реализованы явно. Но ни один из них не входит в общедоступный контракт System.Int32. Если вы хотите вызвать один из его методов, то должны сначала привести ваш тип значения Int32 к ссылке на интерфейс IConvertible. Только после этого вы можете вызвать один из методов IConvertible. И, конечно же, поскольку переменные типов интерфейсов являются ссылками, значение Int32 должно быть упаковано.

Отдавайте предпочтение классу Convert вместо интерфейса IConvertible

Даже несмотря на то, что в качестве примера я использую интерфейс IConvertible, реализованный типом значения, документация не рекомендует вызывать метод IConvertible на Int32; вместо этого рекомендуется применять класс Convert. Этот класс представляет коллекцию методов со многими перегрузками для распространенных типов, которые позволяют преобразовать одно значение почти в любое другое, включая пользовательские типы (посредством Convert.ChangeType), что облегчает последующее изменение вашего кода. Например, если у вас есть код:

```
int i = 0;
double d = Int32.ToDouble(i);
```

и вы хотите изменить тип i на long, то также должны заменить тип Int32 на Int64. С другой стороны, если вы напишете следующим образом:

```
int i = 0;
double d = Convert.ToDouble(i);
```

тогда все, что нужно будет сделать — это изменить тип i.

Правила сопоставления членов интерфейсов

Каждый язык, поддерживающий определения интерфейсов, имеет правила сопоставления реализаций методов с методами интерфейсов. Сопоставление членов интерфейсов в C# достаточно прямолинейно и сводится к нескольким простым правилам. Однако чтобы определить, какой именно метод в действительности вызывается во время выполнения, вы должны также учитывать правила CLR. Эти правила действуют только во время компиляции. Предположим, у вас есть иерархия классов и интерфейсов. Чтобы найти реализацию SomeMethod в ISomeInterface, начинайте со дна иерархии и ищите первый тип, реализующий нужный интерфейс. В данном случае этот интерфейс — ISomeInterface. Это уровень, с которого начинается сопоставление метода. Как только вы нашли тип, рекурсивно перемещайтесь вверх по иерархии типов и ищите метод с совпадающей сигнатурой, отдавая предпочтение явным реализациям членов интерфейса. Если ничего не найдено, обратитесь к общедоступным методам экземпляра, соответствующим той же сигнатуре.

Компилятор C# использует этот алгоритм при сопоставлении реализаций методов с реализациями интерфейсов. Выбранный им метод должен быть общедоступным методом экземпляра или явно реализованным методом экземпляра, причем он может быть (а может и не быть) помечен в C# модификатором `virtual`. Однако при генерации кода IL все вызовы методов интерфейса выполняются IL-инструкцией `callvirt`.

Таким образом, даже если метод не обязательно помечен как `virtual` в смысле C#, CLR трактует вызовы интерфейса как виртуальные. Не путайте эти две концепции. Если метод помечен как `virtual` в C# и имеет методы, переопределяющие его в типах, лежащих ниже, то компилятор C# генерирует существенно отличающийся код в точке вызова. Будьте осторожны, поскольку это может вызвать путаницу, как показано в следующем надуманном примере:

```
using System;
public interface I
{
    void Go();
}
public class A : I
{
    public void Go() {
        Console.WriteLine( "A.Go()" );
    }
}
public class B : A
{
}
public class C : B, I
{
    public new void Go() {
        Console.WriteLine( "C.Go()" );
    }
}
public class EntryPoint
{
    static void Main() {
        B b1 = new B();
        C c1 = new C();
        B b2 = c1;
        b1.Go();
        c1.Go();
        b2.Go();
        ((I)b2).Go();
    }
}
```

Вывод этого примера выглядит так:

```
A.Go()
C.Go()
A.Go()
C.Go()
```

Первый вызов на b1 очевиден, как и следующий — на c1. Однако третий вызов — на b2 — не очевиден вовсе. Поскольку метод A.Go не помечен как virtual, компилятор генерирует код, вызывающий A.Go. Четвертый, и последний, вызов почти так же запутан, но только если не учитывать тот факт, что CLR обрабатывает виртуальные вызовы на ссылках типа класса существенно иначе, чем вызовы на интерфейсных ссылках. Сгенерированный IL-код для четвертого вызова выполняет вызов I.Go, который, в данном случае обращается в C.Go, потому что b2 — на самом деле C, а C реализует I.

Вам следует быть осторожными при поиске метода, вызываемого в действительности, поскольку вы должны учитывать, является ли тип вашей ссылки типом класса или типом интерфейса. Компилятор C# генерирует в IL вызовы виртуальных методов, чтобы вызовы шли через интерфейсные методы, а CLR использует внутри таблицы интерфейсов для обеспечения этого.

На заметку! Программисты на C++ должны понимать, что таблицы интерфейсов отличаются от виртуальных таблиц (vtable) C++. Каждый тип CLR имеет только одну таблицу методов, в то время как экземпляр типа C++ может иметь множество виртуальных таблиц.

Содержимое этих интерфейсных таблиц определяется компилятором с использованием его правил сопоставления методов. Более детальную информацию об этих интерфейсных таблицах вы найдете в книге Дона Бокса (Don Box) и Криса Селлса (Chris Sells) *Essential .NET, Volume 1: The Common Language Runtime* (Boston, MA: Addison-Wesley Professional, 2002 г.), а также в документации по стандарту CLR.

Правила сопоставления методов C# объясняют ситуацию, о которой я говорил ранее в разделе "Наследование интерфейсов и сокрытие членов". Сокрытие метода в одном иерархическом пути ромбовидной иерархии скрывает метод во всех путях наследования. Правила устанавливают, что когда вы идете по иерархии, то прекращаете поиск, как только на определенном уровне найдете подходящий метод. Эти простые правила также объясняют, как повторная реализация интерфейса может существенно повлиять на процесс сопоставления методов, тем самым сокращая поиск компилятора при его проходе по иерархии. Рассмотрим пример такого действия:

```
using System;
public interface ISomeInterface
{
    void SomeMethod();
}
public interface IAnotherInterface : ISomeInterface
{
    void AnotherMethod();
}
public class SomeClass : IAnotherInterface
{
    public void SomeMethod() {
        Console.WriteLine( "SomeClass.SomeMethod()" );
    }
    public virtual void AnotherMethod() {
        Console.WriteLine( "SomeClass.AnotherMethod()" );
    }
}
```

```
public class SomeDerivedClass : SomeClass
{
    public new void SomeMethod() {
        Console.WriteLine( "SomeDerivedClass.SomeMethod()" );
    }
    public override void AnotherMethod() {
        Console.WriteLine( "SomeDerivedClass.AnotherMethod()" );
    }
}
public class EntryPoint
{
    static void Main() {
        SomeDerivedClass obj = new SomeDerivedClass();
        ISomeInterface isi = obj;
        IAnotherInterface iai = obj;
        isi.SomeMethod();
        iai.SomeMethod();
        iai.AnotherMethod();
    }
}
```

Давайте применим правило поиска к каждому вызову метода в Main из предыдущего примера. Во всех случаях я неявно преобразую экземпляр SomeDerivedClass в ссылки двух интерфейсов — ISomeInterface и IAnotherInterface. Я выполнил первый вызов SomeMethod через ISomeInterface. Сначала пройдемся по иерархии классов, начиная с конкретного типа ссылки, в поисках первого класса, реализующего этот интерфейс или интерфейс, наследующий его. Это приведет нас к реализации класса SomeClass, поскольку даже несмотря на то, что он не реализует ISomeInterface напрямую, все же он реализует IAnotherInterface, наследующий ISomeInterface. Таким образом, мы приходим к вызову SomeClass.SomeMethod. Вы можете удивиться, почему не был вызван метод SomeDerivedClass.SomeMethod. Но если вы следуете правилам, то должны пропустить SomeDerivedClass, в поисках самого нижнего в иерархии класса, реализующего интерфейс. Чтобы вместо этого был вызван SomeDerivedClass.SomeMethod, класс SomeDerivedClass должен был бы повторно реализовать ISomeInterface. Второй вызов SomeMethod через ссылку IAnotherInterface следует точно по тому же пути в поисках подходящего метода.

Все становится интереснее при третьем вызове в Main, где вызывается AnotherMethod через ссылку на IAnotherInterface. Как и до этого, поиск начинается с самого нижнего в иерархии класса, реализующего этот интерфейс, — внутри SomeClass. Поскольку в SomeClass есть метод с подходящей сигнатурой, ваш поиск закончен. Однако вся штука в том, что метод с сопоставимой сигнатурой объявлен как virtual. Поэтому, когда выполняется вызов, механизм виртуальных методов помещает выполнение внутрь SomeDerivedClass.AnotherMethod. Важно отметить, что AnotherMethod не изменяет правил сопоставления методов интерфейса, несмотря на виртуальность. Не изменяет до тех пор, пока после обнаружения соответствия интерфейсного метода его виртуальная природа не повлияет на выбор реализации для вызова во время выполнения.

На заметку! Сопоставление метода интерфейса применяется статически во время компиляции. Диспетчеризация виртуального метода происходит динамически во время выполнения. Вы должны отметить разницу между этими двумя механизмами, пытаясь определить, какая же именно реализация метода будет вызвана.

Вывод предыдущего примера кода выглядит следующим образом:

```
SomeClass.SomeMethod()
SomeClass.SomeMethod()
SomeDerivedClass.AnotherMethod()
```

Явная реализация интерфейса с типами значений

Вы неоднократно встретите интерфейсы общего применения, принимающие параметры в форме ссылки на `System.Object`. Такие интерфейсы обычно являются широко используемыми, не обобщенными интерфейсами. Например, рассмотрим интерфейс `IComaparable`, который выглядит так:

```
public interface IComparable
{
    int CompareTo( object obj );
}
```

На заметку! В .NET 2.0 добавилась поддержка `IComparable<T>`, применение которого вы всегда должны рассматривать наряду с `IComparable` для большей безопасности типов.

В том, что метод `CompareTo` принимает такой общий тип, есть большой смысл, потому что очень неплохо иметь возможность передавать ему почти все, что угодно, чтобы сравнить с объектом, реализующим этот метод. Когда речь идет исключительно о ссылочных типах, здесь не происходит никакой потери эффективности, поскольку преобразование ссылочных типов к типу `System.Object` и обратно обходится бесплатно во всех случаях. Но все несколько усложняется, когда речь заходит о типах значений. Рассмотрим некоторый код, чтобы увидеть неприглядные детали:

```
using System;
public struct SomeValue : IComparable
{
    public SomeValue( int n ) {
        this.n = n;
    }
    public int CompareTo( object obj ) {
        if( obj is SomeValue ) {
           SomeValue other = (SomeValue) obj;
           return n - other.n;
        } else {
           throw new ArgumentException( "Неверный тип!" );
        }
    }
    private int n;
}
```

```
public class EntryPoint
{
    static void Main() {
        SomeValue val1 = new SomeValue( 1 );
        SomeValue val2 = new SomeValue( 2 );
        Console.WriteLine( val1.CompareTo(val2) );
    }
}
```

В невинном вызове `WriteLine` в `Main` вы видите, что `val1` сравнивается с `val2`. Но давайте посмотрим внимательнее, насколько много здесь понадобится операций упаковки. Во-первых, поскольку `CompareTo` принимает объектную ссылку, `val2` нужно упаковать в точке вызова метода. Если бы вы реализовали метод `CompareTo` явно, то вам пришлось бы выполнить приведение значения `val1` к интерфейсу `IComparable`, что опять означает затраты на упаковку. Но и внутри метода `CompareTo` кошмар упаковки не заканчивается. К счастью, можно воспользоваться оптимизацией при сравнении `SomeValue` с определенными типами. Вы можете предусмотреть безопасную в отношении типов версию метода `CompareTo` для выполнения работы, как показано ниже:

```
using System;
public struct SomeValue : IComparable
{
    public SomeValue( int n ) {
        this.n = n;
    }
    int IComparable.CompareTo( object obj ) {
        if( obj is SomeValue ) {
           SomeValue other = (SomeValue) obj;
           return n - other.n;
        } else {
           throw new ArgumentException( "Неверный тип!" );
        }
    }
    public int CompareTo( SomeValue other ) {
        return n - other.n;
    }
    private int n;
}
public class EntryPoint
{
    static void Main() {
        SomeValue val1 = new SomeValue( 1 );
        SomeValue val2 = new SomeValue( 2 );
        Console.WriteLine( val1.CompareTo(val2) );
    }
}
```

В данном примере упаковка в вызове `CompareTo` полностью исключена. Это объясняется тем, что компилятор выбирает версию, наиболее подходящую для типа. В данном случае, поскольку вы реализуете `IComparable.CompareTo` явно, существует только одна перегрузка `CompareTo` в общедоступном контракте `SomeValue`.

Но даже если бы IComparable.CompareTo не был реализован явно, компилятор все равно выбрал бы безопасную в отношении типов версию. Типичный шаблон поведения включает сокрытие безтиповых версий от нечаянного применения, так что пользователь должен осуществлять явную упаковку. Эта операция преобоазует интерфейсную ссылку, чтобы получить безтиповую версию.

Главный итог состоит в том, что вы определенно захотите следовать этой идиоме при всякой реализации интерфейса в типе значений, где вы определяете, что можно определить перегрузки с лучшей защитой типа, чем те, что перечислены в объявлении интерфейса. Избежание ненужной упаковки — всегда хорошая вещь, и ваши пользователи будут благодарны вам за заботу об эффективности.

Соображения, касающиеся версий

Концепция поддержки версий, по сути, тесно связана с концепцией интерфейсов. Когда вы создаете, определяете и публикуете интерфейс, тем самым вы определяете контракт, или в более строгих терминах — стандарт. Всякий раз, имея стандартную форму коммуникаций, вы должны следовать ей, чтобы не нарушать контракта с клиентами. Например, рассмотрим стандарт 802.11, на использовании которого основаны многие устройства WiFi. Важно, чтобы точки доступа одного поставщика работали с устройствами как можно большего числа других поставщиков. Все это работает до тех пор, пока все поставщики согласны и следуют стандарту. Можете представить хаос, который наступил бы, если бы WiFi-карта только одного-единственного поставщика работала с вашими любимым кафе в северо-тихоокеанском регионе? Это был бы кошмар. Поэтому у нас есть стандарты.

Теперь ничто не мешает расширять стандарты. Некоторые производители именно это и делают. В некоторых случаях, если вы используете точку доступа производителя A с беспроводной картой того же поставщика, то вы можете достичь намного более высокой скорости, чем установлено стандартом. Точно также ничто не мешает пересмотреть стандарт. Обычно стандарты сопровождаются номерами версий, прикрепленными к ним, и когда они изменяются, то увеличивается номер версии. Большей частью устройства, реализующие новую версию, также поддерживают и предыдущую. Хотя это и не обязательно, но лучше ориентироваться на тех поставщиков, которые стараются захватить наибольшую часть рынка. В примере с 802.11, номера 802.11a, 802.11b и 802.11g представляют различные ревизии стандарта.

Мысль, которую я хочу выразить в этом примере, заключается в том, что те же правила следует применять к интерфейсам, когда вы публикуете их. Обычно вы не создаете интерфейсов, если только не намереваетесь разрешить различным сущностям взаимодействовать друг с другом с использованием общего контракта. Поэтому, завершив разработку контракта, снабдите его номером версии. Номер версии можно формировать различными способами. Если это новая ревизия вашего интерфейса — просто присвойте ей новое имя; в этом случае вы никогда не меняете оригинальный интерфейс. Вероятно, такую идиому вы встречали в мире COM. Обычно кто-то, возможно, Microsoft, решает, что у него есть веские причины усовершенствовать поведение интерфейса, то в результате вы обнаруживаете новое определение интерфейса, завершающееся либо суффиксом *Ex*, либо числовым суффиксом. В любом случае это — совершенно другой интерфейс, отличающийся

от предыдущего, даже если контракт нового интерфейса наследует оригинальный интерфейс, и реализация может быть разделена ими обоими.

На заметку! Когда вам нужно создать усовершенствованный интерфейс на базе другого, современные распространенные руководства по дизайну не рекомендуют использовать суффикс Ex, как это делает COM. Вместо этого лучше дополнить имя интерфейса порядковым номером. Поэтому, если оригинальный интерфейс — ISomeContact, то его усовершенствованная версия должна называться ISomeContact2.

В действительности, если ваше определение интерфейса находится внутри сборки, поддерживающей версии, вы можете определить более новую версию того же интерфейса, даже с тем же именем, в сборке с тем же именем, но с более свежим номером версии. Загрузчик сборок разрешит и загрузит правильную версию во время выполнения. Однако такая практика может привести к путанице разработчиков, использующих ваш интерфейс, поскольку им придется явно следить за тем, на какую сборку ссылаться во время выполнения.

Контракты

При проектировании приложения или системы вам часто придется представлять описание контракта. Программный контракт не отличается от любого другого контракта. Обычно вы определяете контракт для облегчения взаимодействия между двумя типами в вашем дизайне. Например, предположим, что у вас есть виртуальный зоопарк, а в нем — животные. Теперь экземпляр вашего ZooKeeper (смотритель) нуждается в способе взаимодействия с коллекцией объектов ZooDweller (обитатель), которые нужно перегнать по воздуху в определенное место. Игнорируя тот факт, что они все должны быть довольно послушными, нужно отметить, что они также должны уметь летать. Однако не все животные умеют летать, так что не все типы в этом виртуальном зоопарке смогут поддерживать этот "летающий" контракт.

Контракты, реализованные классами

Рассмотрим один способ управления сложностью, связанной с необходимостью переноса всех этих обитателей по воздуху из одного места в другое. Во-первых, рассмотрим допущения, которые можно принять здесь. Скажем, Zoo может иметь только один ZooKeeper. Во-вторых, предположим, что вы можете моделировать местоположения внутри Zoo с помощью простой двумерной структуры Point. Если попробовать смоделировать такую систему, получим примерно следующий код:

```
using System;
using System.Collections.ObjectModel;
namespace CityOfShanoo.MyZoo
{
public struct Point
{
    public double X;
    public double Y;
}
```

```
public abstract class ZooDweller
{
    public void EatSomeFood() {
        DoEatTheFood();
    }
    protected abstract void DoEatTheFood();
}
public sealed class ZooKeeper
{
    public void SendFlyCommand( Point to ) {
        // Для простоты реализация опущена
    }
}
public sealed class Zoo
{
    private static Zoo theInstance = new Zoo();
    public static Zoo GetInstance() {
        return theInstance;
    }
    private Zoo() {
        creatures = new Collection<ZooDweller>();[3]
        zooKeeper = new ZooKeeper();
    }
    public ZooKeeper ZooKeeper {
        get {
           return zooKeeper;
        }
    }
    private ZooKeeper zooKeeper;
    private Collection<ZooDweller> creatures;
}
}
```

Поскольку может быть только один зоопарк в `CityOfShanoo`, класс `Zoo` моделируется как одиночный объект (Singleton), и единственный путь получить один и только один экземпляр `Zoo` — вызвать `Zoo.GetInstance`. К тому же вы можете получить ссылку на `ZooKeeper` через свойство `Zoo.ZooKeeper`. Это общепринятая практика в .NET Framework — называть свойства по пользовательскому типу, который оно представляет.

На заметку! Шаблон проектирования Singleton — один из наиболее широко используемых и знаменитых шаблонов. По сути, этот шаблон допускает существование только одного экземпляра его типа в единицу времени. Многие люди продолжают спорить о лучшем способе реализации этого шаблона. Реализации меняются в зависимости от используемого языка. Но обычно один экземпляр `static private` внутри объявления типа "лениво" инициализируется при первом обращении к нему. Именно это делает приведенная реализация класса `Zoo`, где статический инициализатор не вызывается до тех пор, пока к типу не произойдет первое обращение методом `GetInstance`.

---

[3] Если `Collection<ZooDweller>` выглядит не знакомой, не беспокойтесь. Это объявление коллекции, основанное на обобщенном типе. Обобщения подробно рассматриваются в главе 11.

Первоначальный дизайн определяет ZooDweller как абстрактный класс, реализующий метод EatSomeFood. ZooDweller использует шаблон не виртуального интерфейса (Non-Virtual Interface — NVI), описанный в главе 13, где виртуальный метод, переопределяемый конкретным типом, объявлен protected вместо public.

Важно отметить, что тип ZooDweller, по сути, определяет контракт, несмотря на то, что не является интерфейсом. Этот контракт утверждает, что любой тип, унаследованный от ZooDweller, должен реализовать EatSomeFood. Любой код, использующий экземпляр ZooDweller, должен быть уверен в том, что этот метод поддерживается.

На заметку! Обратите внимание, что для определения контракта интерфейс не обязателен.

Пока в предложенном дизайне недостает ключевой операции — той, что отправит наших животных в полет в направлении места назначения внутри зоопарка. Ясно, что вы не можете поместить метод Fly в тип ZooDweller, потому что не все обитатели в зоопарке умеют летать. Придется выразить этот контракт каким-то другим способом.

Интерфейсные контракты

Поскольку не все обитатели в зоопарке умеют летать, интерфейс представляет блестящий механизм определения "летающего" контракта. Рассмотрим следующую модификацию примера из предыдущего раздела:

```
public interface IFly
{
    void FlyTo( Point destination );
}
public class Bird : ZooDweller, IFly
{
    public void FlyTo( Point destination ) {
        Console.WriteLine( "Перелет к ({0}. {1}).", destination );
    }
    protected override void DoEatTheFood() {
        Console.WriteLine( "Прием пищи." );
    }
}
```

Теперь, используя интерфейс IFly, Bird определяется как наследник ZooDweller, реализующий IFly.

На заметку! Если вы собираетесь наследовать несколько разных типов птиц от Bird, причем все эти разные птицы будут иметь разные реализации ToFly, подумайте об использовании шаблона NVI. Вы можете ввести метод protected virtual по имени DoFly, который переопределят базовые типы, причем Bird.DoFly будет вызываться через DoFlyTo. Подробное объяснение того, чем хороша эта идея, читайте в разделе "Использование шаблона NVI" главы 13.

Выбор между интерфейсами и классами

Предыдущий раздел о контрактах показал, что вы можете реализовать контракт различными способами. В среде C# и .NET двумя главными способами являются использование интерфейсов и классов, причем классы могут даже быть абстрактными. В примере с зоопарком вполне ясно, когда вы должны использовать интерфейс вместо абстрактного класса для определения контракта. Однако выбор не всегда очевиден, поэтому давайте рассмотрим условия применения обоих методов.

Поскольку C# поддерживает абстрактные классы, вы можете легко смоделировать контракт посредством абстрактных классов. Но какой метод мощнее? И какой из них больше подходит? Это непростые вопросы, хотя руководства рекомендуют предпочесть класс, если это возможно. Исследуем это.

На заметку! С ростом популярности COM некоторые разработчики сделали ошибочный вывод, что единственный способ определения контракта заключается в определении интерфейса. К такому заключению легко прийти, переходя из среды COM в среду C# — просто потому, что базовый строительный блок COM — интерфейс, а при этом и C#, и .NET поддерживают интерфейсы.

Если вы знакомы с COM и имеете опыт разработки серьезных проектов в этой технологии, то наверняка вы реализовывали объекты COM на языке C++. Возможно, вы даже использовали библиотеку активных шаблонов (Active Template Library — ATL), чтобы оградить себя от сложности низкоуровневых задач разработки COM. Но в основе всего — каким образом C++ моделирует интерфейсы COM? Ответ: абстрактными классами.

Когда вы реализуете контракт посредством определения интерфейса, вы определяете контракт, поддерживающий версии. Это значит, что интерфейс, однажды реализованный, не должен никогда изменяться, как если бы его вытесали из камня. Конечно, вы можете изменить его позже, но вы утратите популярность, когда весь код ваших клиентов перестанет компилироваться с модифицированным интерфейсом. Рассмотрим следующий пример:

```
public interface IMyOperations
{
    void Operation1();
    void Operation2();
}
// Клиентский класс
public class ClientClass : IMyOperations
{
    public void Operation1() { }
    public void Operation2() { }
}
```

Теперь вы представили миру этот чудесный интерфейс `IMyOperations`, и тысячи клиентов ринулись реализовывать его. Затем вы начинаете получать от клиентов запросы на поддержку `Operation3` в вашей библиотеке. Кажется достаточно легким просто добавить `Operation3` к интерфейсу `IMyOperations`, но это будет ужасной ошибкой. Если вы добавите еще одну операцию к `IMyOperations`, то весь код ваших клиентов перестанет компилироваться до тех пор, пока они не реализуют новую операцию. К тому же, код в другой сборке, который знает о новом `IMyOperations`, может попытаться привести экземпляр `ClientClass` к ссылке

IMyOperations и затем вызвать Operation3, что приведет к сбою во время выполнения. Ясно, что вы не должны модифицировать уже опубликованный интерфейс.

Внимание! Никогда не модифицируйте уже опубликованное объявление интерфейса.

Вы могли бы также справиться с этой проблемой, определив полностью новый интерфейс, скажем, ImyOperations2. Однако классу ClientClass пришлось бы реализовывать оба интерфейса, чтобы получить новое поведение, как показано ниже:

```
public interface IMyOperations
{
    void Operation1();
    void Operation2();
}
public interface IMyOperations2
{
    void Operation1();
    void Operation2();
    void Operation3();
}
// Клиентский класс
public class ClientClass : IMyOperations,
    IMyOperations2
{
    public void Operation1() { }
    public void Operation2() { }
    public void Operation3() { }
}
public class AnotherClass
{
    public void DoWork( IMyOperations ops ) {
    }
}
```

Модификация ClientClass для поддержки новой операции из IMyOperations2 не особенно трудна, но как насчет уже существующего кода, вроде показанного в AnotherClass? Проблема в том, что метод DoWork принимает тип IMyOperations. Для того чтобы он мог вызывать новый метод Operation3, нужно изменить прототип DoWork, или же код внутри него должен выполнять приведение параметра к типу IOperations2, чтобы может дать сбой во время выполнения. Раз вы хотите, чтобы компилятор мог перехватывать как можно больше ошибок, связанных с типами, будет лучше все-таки изменить прототип DoWork, чтобы он принимал тип IMyOperations2.

На заметку! Если вы определяете ваш оригинальный интерфейс IMyOperations с полностью поддерживающей версии, строго именованной сборкой, то вам придется создать новый интерфейс с тем же именем в новой сборке, только если версия этой новой сборки будет отличаться. Хотя .NET Framework поддерживает это явно, это не значит, что вы должны поступать так без тщательного взвешивания, поскольку введение двух интерфейсов IMyOperations, отличающихся только номером версии содержащей их сборки, может запутать ваших клиентов.

Вот как много работы нужно выполнить, чтобы предоставить в распоряжение клиентов новую операцию. Рассмотрим ту же ситуацию, но с использованием абстрактного класса:

```
public abstract class MyOperations
{
    public virtual void Operation1() {
    }
    public virtual void Operation2() {
    }
}
// Клиентский класс
public class ClientClass : MyOperations
{
    public override void Operation1() { }
    public override void Operation2() { }
}
public class AnotherClass
{
    public void DoWork( MyOperations ops ) {
    }
}
```

MyOperations — базовый класс для ClientClass. Его преимущество состоит в том, что при необходимости он может содержать реализацию по умолчанию. В противном случае виртуальные методы MyOperations могут быть объявлены как abstract, поскольку для клиентов не имеет смысла создание экземпляров MyOperations. Теперь предположим, что вы хотите добавить новый метод Operation3 в MyOperations, не затрагивая существующих клиентов. Вы можете делать это до тех пор, пока добавленная операция не абстрактна, иначе это потребует изменений в производных типах, как показано ниже:

```
public abstract class MyOperations
{
    public virtual void Operation1() {
    }
    public virtual void Operation2() {
    }
    public virtual void Operation3() {
        // Новая реализация по умолчанию
    }
}
// Клиентский класс
public class ClientClass : MyOperations
{
    public override void Operation1() { }
    public override void Operation2() { }
}
public class AnotherClass
{
    public void DoWork( MyOperations ops ) {
        ops.Operation3();
    }
}
```

Обратите внимание, что добавление MyOperations.Operation3 не потребует никаких изменений в ClientClass, и AnotherClass.DoWork может вызывать Operation3, не внося никаких изменений в объявление метода. Такая техника, правда, не лишена своих недостатков. Вы ограничены тем фактом, что управляющая исполняющая система допускает наследование класса только от одного базового класса. Поскольку ClientClass наследует MyOperations, чтобы получить функциональность, он использует свой единственный билет на наследование. Это может наложить сложные ограничения на код ваших клиентов. Например, что если одному из ваших клиентов нужно создавать объект для использования с .NET Remoting? Чтобы сделать это, класс должен наследоваться от MarshalByRefObject.

Иногда непросто найти золотую середину, выбирая между интерфейсами и классами. Лично я руководствуюсь следующими эмпирическими правилами.

- Если моделируется отношение "is-a" (является), использовать класс. Если имеет смысл назвать ваш контракт именем существительным, тогда, вероятно, его стоит моделировать классом.
- Если моделируется отношение РЕАЛИЗУЕТ, использовать интерфейс. Если имеет смысл назвать контракт именем прилагательным, как если речь идет о качестве, то вероятно, вы должны моделировать его интерфейсом.
- Рассмотреть возможность помещения вашего интерфейса и объявления абстрактного класса в оболочку отдельной сборки. Реализации в других сборках тогда смогут ссылаться на эту отдельную сборку.
- Если возможно, предпочитать классы интерфейсам. Это может способствовать расширяемости.

Вы можете видеть примеры описанных приемов по всей библиотеке базовых классов .NET Framework (Base Class Library — BCL). Рассмотрите возможность применения их и в своем коде.

Резюме

В этой главе вы ознакомились с интерфейсами и способами моделирования хорошо определенных, поддерживающих версии контрактов с применением интерфейсов. Наряду с демонстрацией различных способов реализации интерфейсов классами, я также описал процесс, которому следует компилятор C# при нахождении соответствия методов интерфейса методам реализующего класса. Я описал интерфейсы с точки зрения ссылочных типов и типов значения — в частности, как дорогостоящие операции упаковки могут нанести ущерб производительности при использовании интерфейсов с типами значений. И, наконец, я потратил некоторое время на сравнение и противопоставление применения интерфейсов и классов при моделировании контрактов между типами в вашем дизайне.

В следующей главе я проясню сложности, связанные с перегрузкой операций в языке C#, а также расскажу, почему стоит избегать их при создании кода, используемого другими языками .NET.

ГЛАВА 6

Перегрузка операций

Язык C# позаимствовал возможность перегрузки операций из C++. Точно так же, как перегружаются методы, вы можете перегружать операции, подобные +, -, * и т.д. В дополнение к перегрузке арифметических операций вы можете также создавать собственные операции для преобразования от одного типа к другому. Вы можете перегружать и другие операции, позволяя использовать объекты в булевских выражениях проверки.

Можете — не значит должны

Перегрузка операций может сделать использование некоторых классов и структур более естественным. Однако перегрузка операций, выполненная неаккуратно, может значительно ухудшить читабельность кода и снизить возможность его понимания. Вы должны быть внимательными, рассматривая семантику операций с типами. Не вводите того, что трудно будет расшифровать. Всегда ориентируйтесь на создание более читабельного кода — и не только для той счастливой души, которая будет хлопать глазами, изучая ваш код, но также и для себя. Случалось ли вам когда-нибудь смотреть на код и думать: "Кто в здравом уме мог написать такую чушь?", а потом обнаруживать, что это были вы? Со мной такое было...

Другая причина не перегружать операции заключается в том, что не все языки .NET поддерживают перегруженные операции, потому что они не являются частью CLS. От языков, ориентированных на CLI, не требуется поддержка перегруженных операций. Так, например Visual Basic 2005 был первой .NET-версией языка для поддержки перегрузки операций. Поэтому важно, чтобы ваши перегруженные операции были синтаксическими сокращениями для функциональности, представленной другими методами, выполняющими те же операции, и которые можно было бы вызвать из других CLS-совместимых языков. Фактически, я рекомендую проектировать типы так, как если бы перегруженные операции не существовали. Тогда позже при желании вы сможете добавить их, просто вызывая уже существующие методы того же семантического назначения.

Типы и форматы перегруженных операций

Вы определяете перегруженные операции как общедоступные статические методы в классах, для расширения которых они предназначены. В зависимости от типа перегружаемой операции, такие методы могут принимать один или два параметра, и всегда возвращают значение. Для всех операций за исключением операций преобразования типом одного из параметров должен быть тип включаю-

щего метод класса. Например, не имеет смысла перегружать операцию + в классе Complex, если он будет складывать вместе два значения double. К тому же, как вы вскоре убедитесь, это и невозможно.

Типичная операция + для класса Complex может выглядеть так:

```
public static Complex operator+( Complex lhs, Complex rhs )
```

Даже несмотря на то, что этот метод складывает два экземпляра Complex вместе, чтобы произвести третий экземпляр Complex, ничто не запрещает сделать один из параметров типа double, чтобы, таким образом, прибавлять double к экземпляру Complex. Как именно вы будете прибавлять double к экземпляру Complex — ваше дело. Вообще синтаксис перегрузки операций следует приведенному шаблону, где + заменяется нужной операцией, и конечно, некоторые операции принимают только один параметр.

На заметку! При сравнении операций C# с операциями C++ обратите внимание, что объявление операции C# больше похоже на прием объявления дружественных функций для реализации операций в C++, поскольку операции C# не являются методами экземпляра.

Существуют всего три группы перегружаемых операций. Унарные операции принимают только один параметр. К знакомым вам унарным операциям относятся ++ и --. Бинарные операции, как следует из их названия, принимают два параметра и включают знакомые математические операции, такие как +, -, / и *, а также операции сравнения. И, наконец, операции преобразования определяют пользовательские преобразования типов. Они могут иметь либо операнд, либо возвращаемое значение того же типа, что и класс или структура, в которой они объявлены.

Несмотря на то что операции являются статическими и общедоступными, и как таковые, наследуются производными классами, методы операций должны иметь как минимум, один параметр в их объявлении, совпадающий с *включающим* типом, что делает невозможным точное соответствие методов операций производных типов сигнатуре методов операций базового класса. Например, следующее объявление неправильно:

```
public class Apple
{
    public static Apple operator+( Apple rhs, Apple lhs ) {
        // Метод не делает ничего, существует только для примера.
        return rhs;
    }
}
public class GreenApple : Apple
{
    // НЕВЕРНО!!! Не скомпилируется.
    public static Apple operator+( Apple rhs, Apple lhs ) {
        // Метод не делает ничего, существует только для примера.
        return rhs;
    }
}
```

Если вы попытаетесь скомпилировать предыдущий код, то получите следующую ошибку компиляции:

```
error CS0563: One of the parameters of a binary operator must be the
containing type
ошибка CS0563: Один из параметров бинарной операции должен быть
включающего типа
```

Операции не должны изменять свои операнды

Вы уже знаете, что методы операций являются статическими. Поэтому настоятельно рекомендуется (читай: требуется), чтобы вы не изменяли переданные им параметры. Вместо этого вы должны создавать новый экземпляр возвращаемого типа и возвращать его, как результат операции. Неизменяемые структуры и классы вроде `System.String` — блестящие кандидаты для реализации пользовательских операций. Такое поведение естественно для булевских операций, которые обычно возвращают тип, отличающийся от типов переданных в параметрах.

На заметку! "Минуточку!" — скажут некоторые из вас, принадлежащие к сообществу C++ — "а как же тогда реализовать постфиксные и префиксные операции `++` и `--` без изменения операнда?". Все операции C# являются статическими, и к ним относятся также и постфиксные, и префиксные операции, в то время как в C++ они представлены методами экземпляров, модифицирующими экземпляр объекта через указатель `this`. Красота подхода C# заключается в том, что вам не нужно беспокоиться о реализации двух различных версий операции `++`, чтобы поддерживать префиксную и постфиксную его формы, как в C++. Компилятор выполняет задачу создания временных копий объекта для обработки отличия в поведении между префиксом и постфиксом. Это — еще одна причина того, что операции могут возвращать новые экземпляры, никогда не модифицируя состояния самих операндов. Если вы не будете следовать такой практике, то обречете себя на некоторые серьезные неприятности при отладке.

Имеет ли значение порядок параметров?

Предположим, вы создаете `struct` для представления простых комплексных чисел — скажем, структуру `Complex` — и вам нужно складывать вместе экземпляры `Complex`. Было бы также удобно иметь возможность прибавлять простые значения `double` к экземпляру `Complex`. Добавление такой функциональности — не проблема, поскольку вы можете перегрузить метод операции + так, чтобы один параметр был `Complex`, а другой — `double`. Это объявление могло бы выглядеть примерно так:

```
static public Complex operator+( Complex lhs, double rhs )
```

Имея такую операцию, объявленную и определенную в структуре `Complex`, вы можете писать код вроде следующего:

```
Complex cpx1 = new Complex( 1.0, 2.0 );
Complex cpx2 = cpx1 + 20.0;
```

Это избавляет вас от необходимости создавать дополнительный экземпляр `Complex`, состоящий только из реальной части, равной `20.0`, чтобы добавить ее к `cpx1`. Однако представьте, что вы хотите иметь возможность менять местами операнды и делать что-нибудь вроде следующего:

```
Complex cpx2 = 20.0 + cpx1;
```

Если вы хотите поддерживать разный порядок операндов разных типов, то для этого должны предусмотреть разные перегрузки для операции. Если вы перегру-

жаете бинарную операцию, используя разные типы параметров, то можете создавать зеркальные перегрузки — т.е. другой метод операции, который просто меняет местами параметры.

Перегрузка операции сложения

Давайте рассмотрим краткий пример структуры Complex, которая не претендует на звание исчерпывающей реализации, а просто демонстрирует перегрузку операций. На протяжении всей главы я буду отталкиваться от этого примера, и добавлять к нему дополнительные операции:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    static public Complex Add( Complex lhs,
                               Complex rhs ) {
        return new Complex( lhs.real + rhs.real,
                            lhs.imaginary + rhs.imaginary );
    }
    static public Complex Add( Complex lhs,
                               double rhs ) {
        return new Complex( rhs + lhs.real,
                            lhs.imaginary );
    }
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              imaginary );
    }
    static public Complex operator+( Complex lhs,
                                     Complex rhs ) {
        return Add( lhs, rhs );
    }
    static public Complex operator+( double lhs,
                                     Complex rhs ) {
        return Add( rhs, lhs );
    }
    static public Complex operator+( Complex lhs,
                                     double rhs ) {
        return Add( lhs, rhs );
    }
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        Complex cpx2 = new Complex( 1.0, 2.0 );
```

```
            Complex cpx3 = cpx1 + cpx2;
            Complex cpx4 = 20.0 + cpx1;
            Complex cpx5 = cpx1 + 25.0;
            Console.WriteLine( "cpx1 == {0}", cpx1 );
            Console.WriteLine( "cpx2 == {0}", cpx2 );
            Console.WriteLine( "cpx3 == {0}", cpx3 );
            Console.WriteLine( "cpx4 == {0}", cpx4 );
            Console.WriteLine( "cpx5 == {0}", cpx5 );
        }
    }
```

Обратите внимание, что как рекомендуется, перегруженные методы операции вызывают методы, выполняющие саму операцию. Фактически это очень упрощает поддержку обоих последовательностей операндов операции + для типа `Complex`.

Совет. Если вы абсолютно уверены, что ваш тип будет использоваться только в среде C# или с языком, поддерживающим перегруженные операции, то можете пренебречь этим правилом и просто написать перегруженные операции.

Операции, допускающие перегрузку

Давайте вкратце перечислим операции, которые вы можете перегружать. Унарные операции, бинарные операции и операции преобразования — три главных типа операций. Невозможно перечислить здесь все операции преобразования, поскольку их набор неограничен. Вдобавок вы можете использовать одну тернарную операцию — знакомую вам `?:` — для условных операторов, но не можете перегружать ее напрямую. Далее, в разделе "Булевские операции" я расскажу, как можно "поиграть" с тернарной операцией. В табл. 6.1 перечислены все перегружаемые операции, за исключением операций преобразования.

Таблица 6.1. Унарные и бинарные операции

Унарные операции	Бинарные операции
+	+
-	-
!	*
~	/
++	%
--	&
`true` и `false`	|
	^
	<<
	>>
	== и !=
	> и <
	>= и <=

Операции сравнения

Бинарные операции сравнения == и !=, < и >, а также >= и <= должны быть реализованы парами. Конечно, это имеет совершенно прямой смысл, поскольку я сомневаюсь, чтобы в каком-то случае вы захотели разрешить пользователям применять операцию ==, но не !=. Кроме того, если ваш тип позволяет упорядочивание через реализацию интерфейса IComparable или его обобщенного аналога IComparable<T>, то имеет смысл реализовать все операции сравнения. Их реализация тривиальна, если следовать каноническому руководству, приведенному в главах 4 и 13, соответственно переопределяя Equals и GetHashCode и реализуя IComparable (и, необязательно, IComparable<T>). Учитывая это, перегрузка операций просто требует, чтобы вы вызывали эти реализации. Рассмотрим модифицированную форму примера Complex, следующую этому шаблону для реализации всех операций сравнения:

```
using System;
public struct Complex : IComparable,
           IEquatable<Complex>,
           IComparable<Complex>
{
    public Complex( double real, double img ) {
        this.real = real;
        this.img = img;
    }
    // Перегрузка System.Object
    public override bool Equals( object other ) {
        bool result = false;
        if( other is Complex ) {
           result = Equals( (Complex) other );
        }
        return result;
    }
    // Версия, безопасная к типам
    public bool Equals( Complex that ) {
        return (this.real == that.real &&
                this.img == that.img);
    }
    // Должен быть перегружен, если перегружен Object.Equals()
    public override int GetHashCode() {
        return (int) this.Magnitude;
    }
    // Версия, безопасная к типам
    public int CompareTo( Complex that ) {
        int result;
        if( Equals( that ) ) {
           result = 0;
        } else if( this.Magnitude > that.Magnitude ) {
           result = 1;
        } else {
           result = -1;
        }
        return result;
    }
```

```
    // Реализация IComparable
    int IComparable.CompareTo( object other ) {
        if( !(other is Complex) ) {
           throw new ArgumentException( "Неверное сравнение" );
        }
        return CompareTo( (Complex) other );
    }
    // Перегрузка System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              img );
    }
    public double Magnitude {
        get {
           return Math.Sqrt( Math.Pow(this.real, 2) +
                  Math.Pow(this.img, 2) );
        }
    }
    // Перегруженные операции
    public static bool operator==( Complex lhs, Complex rhs ) {
        return lhs.Equals( rhs );
    }
    public static bool operator!=( Complex lhs, Complex rhs ) {
        return !lhs.Equals( rhs );
    }
    public static bool operator<( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) < 0;
    }
    public static bool operator>( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) > 0;
    }
    public static bool operator<=( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) <= 0;
    }
    public static bool operator>=( Complex lhs, Complex rhs ) {
        return lhs.CompareTo( rhs ) >= 0;
    }
    // Прочие методы пропущены для ясности
    private double real;
    private double img;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        Complex cpx2 = new Complex( 1.0, 2.0 );
        Console.WriteLine( "cpx1 = {0}, cpx1.Magnitude = {1}",
        cpx1, cpx1.Magnitude );
        Console.WriteLine( "cpx2 = {0}, cpx2.Magnitude = {1}\n",
        cpx2, cpx2.Magnitude );
        Console.WriteLine( "cpx1 == cpx2 ? {0}", cpx1 == cpx2 );
```

```
            Console.WriteLine( "cpx1 != cpx2 ? {0}", cpx1 != cpx2 );
            Console.WriteLine( "cpx1 < cpx2 ? {0}", cpx1 < cpx2 );
            Console.WriteLine( "cpx1 > cpx2 ? {0}", cpx1 > cpx2 );
            Console.WriteLine( "cpx1 <= cpx2 ? {0}", cpx1 <= cpx2 );
            Console.WriteLine( "cpx1 >= cpx2 ? {0}", cpx1 >= cpx2 );
        }
    }
```

Обратите внимание, что методы операций просто вызывают методы, реализующие Equals и CompareTo. Кроме того, я следую рекомендации о предоставлении версий, безопасных к типам, двух методов через реализацию IComparable<Complex> и IEquatable<Complex>, поскольку Complex — тип значения, и я хочу по возможности избежать упаковки[1]. Дополнительно я явно реализовал метод IComparable.CompareTo, чтобы предоставить компилятору большой безопасный к типам молоток, чтобы затруднить пользователям возможность нечаянно вызвать неверную версию. Всякий раз, когда у вас есть возможность использовать систему контроля типов компилятора, чтобы выявить ошибки на этапе компиляции, а не на этапе выполнения, вы должны воспользоваться ею. Если бы я не реализовал явно Icomparable.CompareTo, то компилятор спокойно пропустит оператор, где я пытаюсь сравнить экземпляр Apple с экземпляром Complex. Конечно, вы ожидаете исключения InvalidCastException во время выполнения, если попытаетесь сделать нечто настолько глупое, так что всегда предпочитайте ошибки времени компиляции ошибкам времени выполнения.

Операции преобразования

Операции преобразования, как следует из их названия, являются операциями, преобразующими объекты одного типа в объекты другого типа. Операции преобразования могут допускать неявные преобразования наряду с явными. Неявные преобразования выполняются при простом присваивании, в то время как явные требуют знакомого синтаксиса приведений, где целевой тип указывается в скобках, предшествующих экземпляру, из которого присваивается значение.

На неявные операции накладывается одно важное ограничение. Стандарт C# требует, чтобы неявные операции не генерировали исключений, чтобы они всегда гарантированно завершались успешно, без потери информации. Если вы не можете соблюсти это требование, то ваше преобразование должно быть явным. Например, при преобразовании одного типа в другой всегда имеется возможность утери информации, если целевой тип не настолько выразительный, как исходный. Рассмотрим преобразование long в short. Ясно, что информация может быть утеряна, если значение long окажется больше максимально допустимого для типа short. Даже несмотря на то, что по умолчанию исключение в случае усечения не генерируется, иногда стоит сгенерировать исключение во время выполнения. Такое преобразование должно быть явным и требовать от пользователя использования синтаксиса приведения. Теперь представьте, что вы идете в противоположном направлении, преобразуя short в long. Такое преобразование всегда пройдет успешно, поэтому оно может быть неявным.

---

[1] Я подробно описываю эти рекомендации в главе 5, в разделе, озаглавленном "Явная реализация интерфейса с типами значений".

На заметку! Выполнение явных преобразований от типа с большим диапазоном хранимых значений к типу с меньшим диапазоном может дать ошибку усечения, если исходное значение окажется слишком большим, чтобы быть представленным в маленьком типе. Например, если вы выполните явное приведение `long` к `short`, то можете вызвать ситуацию переполнения. По умолчанию ваш компилированный код будет молча выполнять усечение. Если вы откомпилируете код с опцией компилятора `/checked+`, то при попытке явного преобразования `long` к `short` будет генерироваться исключение `System.OverflowException`. Я рекомендую всегда выполнять сборку с включенной опцией `/checked+`.

Посмотрим, какие операции преобразования вы должны предусмотреть для `Complex`. Я могу представить, по крайней мере, один определенный случай — преобразование из `double` в `Complex`. Несомненно, такое преобразование должно быть неявным. Другой вариант — `Complex` в `double` — требует явного преобразования. (Поскольку приведение `Complex` к `double` все равно не имеет смысла и показано здесь только для примера, вы можете возвращать общую величину (magnitude), а не просто реальную часть комплексного числа, выполняя приведение к `double`.) Рассмотрим пример реализации операций приведения:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    // Переопределение System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})", real, imaginary );
    }
    public double Magnitude {
        get {
            return Math.Sqrt( Math.Pow(this.real, 2) +
                    Math.Pow(this.imaginary, 2) );
        }
    }
    public static implicit operator Complex( double d ) {
        return new Complex( d, 0 );
    }
    public static explicit operator double( Complex c ) {
        return c.Magnitude;
    }
    // Прочие методы пропущены для ясности.
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        Complex cpx2 = 2.0;        // Использовать неявную операцию.
        double d = (double) cpx1; // Использовать явную операцию.
        Console.WriteLine( "cpx1 = {0}", cpx1 );
        Console.WriteLine( "cpx2 = {0}", cpx2 );
        Console.WriteLine( "d = {0}", d );
    }
}
```

Синтаксис в методе Main использует операции преобразования. Однако будьте осторожны при реализации операций преобразования, чтобы не создать для пользователей никаких сюрпризов или путаницы с вашими неявными преобразованиями. Сложно внести путаницу с явными операциями, когда пользователи вашего типа вынуждены применять синтаксис приведения для того, чтобы заставить их работать. В конце концов, чем можно удивить пользователя, если он должен указать тип для преобразования в круглых скобках? Но с другой стороны, невнимательное или неправильное использование неявного преобразования может стать причиной большой путаницы. Если вы напишете множество операций неявного преобразования, не имеющих семантического смысла, я гарантирую, что ваши пользователи в один прекрасный день будут весьма удивлены, когда компилятор решит выполнить преобразование, когда они вовсе этого не ожидали. Например, компилятор может выполнять неявное преобразование, пытаясь подогнать аргумент при вызове метода. Даже если операции преобразования имеют семантический смысл, они могут таить в себе ряд сюрпризов, поскольку у компилятора есть возможность молча преобразовывать экземпляры одного типа в другой, когда он сочтет это необходимым.

C# требует, чтобы вы явно писали операции неявного преобразования для определяемых вами типов[2]. Таким образом, вы не сможете нечаянно создать операцию неявного преобразования, не осознавая того, что вы делаете (как это возможно в C++). Однако для того, чтобы предоставить такие преобразования, вы должны всегда подчиняться правилам перегрузки методов. Рассмотрим случай, когда Complex предоставляет другую операцию явного преобразования экземпляра Fraction также в экземпляр double. Это потребует добавления в Complex методов со следующими сигнатурами:

```
public static explicit operator double( Complex d )
public static explicit operator Fraction( Complex f )
```

Эти два метода принимают тот же тип Complex и возвращают другой тип. Однако правила перегрузки четко указывают, что тип возврата не учитывается в сигнатуре метода. Если следовать этому правилу, такие два метода вносят неоднозначность, приводящую к ошибке компиляции. Но на самом деле они не вносят неоднозначности, поскольку существует специальное правило, разрешающее рассматривать тип возврата операций преобразования как часть сигнатуры. Кстати, ключевые слова implicit и explicit не входят в сигнатуру методов операций преобразования. Поэтому невозможно иметь одновременно и явную, и неявную версии одной и той же операции преобразования. Естественно, по крайней мере, один из типов в сигнатуре операции преобразования должен быть включающим типом. Не допускается типу Complex реализовать операцию преобразования из типа Apples в тип Oranges.

Булевские операции

Для некоторых типов имеет смысл участие в булевских выражениях проверки, таких как внутри скобок блока if или внутри тернарной операции ?:. Чтобы это работало, у вас есть две альтернативы. Первая заключается в том, что вы мо-

---

[2] Да, я понимаю последствия моих высказываний и возможную путаницу, вызванную применением слов *явный* и *неявный*. Я явно надеюсь не запутать вас неявно...

жете реализовать две операции преобразования, известные как `operator true` и `operator false`. Вы должны реализовать эти две операции в паре, чтобы позволить комплексному числу (типу `Complex`) участвовать в булевских выражениях проверки. Рассмотрим следующую модификацию типа `Complex`, чтобы можно было использовать его в выражениях, где значение (0, 0) означает `false`, а все остальное — `true`:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    // Переопределение System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              imaginary );
    }
    public double Magnitude {
        get {
           return Math.Sqrt( Math.Pow(this.real, 2) +
                             Math.Pow(this.imaginary, 2) );
        }
    }
    public static bool operator true( Complex c ) {
        return (c.real != 0) || (c.imaginary != 0);
    }
    public static bool operator false( Complex c ) {
        return (c.real == 0) && (c.imaginary == 0);
    }
    // Прочие методы пропущены для ясности.
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        if( cpx1 ) {
           Console.WriteLine( "cpx1 равно true" );
        } else {
           Console.WriteLine( "cpx1 равно false" );
        }
        Complex cpx2 = new Complex( 0, 0 );
        Console.WriteLine( "cpx2 равно {0}", cpx2 ? "true" : "false" );
    }
}
```

Вы можете видеть здесь две операции для применения к типу `Complex` проверок на предмет равенства `true` и `false`. Обратите внимание на синтаксис — он несколько причудлив. Он выглядит почти так же, как в операциях преобразования, за исключением того, что типом возврата является `bool`. Я не совсем понимаю, зачем это нужно, потому что вы все равно не сможете указать никакой другой тип

возврата для этих операций. Если сделать это, то компилятор немедленно сообщит, что единственный допустимый тип возврата для operator true и operator false — это bool. Тем не менее, тип возврата вы должны указывать. Также заметьте, что эти операции нельзя пометить как explicit или implicit, потому что они не являются операциями преобразования. Как только вы определите эти две операции в типе, вы сможете использовать экземпляры Complex в булевских выражениях проверки, как показано в методе Main.

Альтернативно вы можете реализовать преобразование к типу bool, чтобы получить тот же результат. Обычно вы захотите реализовать эту операцию неявно для простоты использования. Рассмотрим модифицированную форму предыдущего примера с использованием неявной операции преобразования к bool вместо operator true и operator false:

```
using System;
public struct Complex
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    // Переопределение System.Object
    public override string ToString() {
        return String.Format( "({0}, {1})",
                              real,
                              imaginary );
    }
    public double Magnitude {
        get {
           return Math.Sqrt( Math.Pow(this.real, 2) +
                  Math.Pow(this.imaginary, 2) );
        }
    }
    public static implicit operator bool( Complex c ) {
        return (c.real != 0) || (c.imaginary != 0);
    }
    // Прочие методы пропущены для ясности.
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex cpx1 = new Complex( 1.0, 3.0 );
        if( cpx1 ) {
           Console.WriteLine( "cpx1 равно true" );
        } else {
           Console.WriteLine( "cpx1 равно false" );
        }
        Complex cpx2 = new Complex( 0, 0 );
        Console.WriteLine( "cpx2 равно {0}", cpx2 ? "true" : "false" );
    }
}
```

Конечный результат — тот же, что и в предыдущем примере. Теперь вы можете недоумевать, зачем вообще нужно реализовывать `operator true` и `operator false` вместо простой неявной операции преобразования к `bool`? Ответ связан с тем, допустимо для вашего типа преобразование в тип `bool` или нет. В последней форме, где вы реализовали неявную операцию преобразования, следующий оператор будет корректным:

```
bool f = cpx1;
```

Такое присваивание должно работать, потому что компилятор найдет операцию неявного преобразования во время компиляции и применит ее. Однако если вы смертельно устали после ночи кодирования этой строки и действительно намерены присваивать `f` значение совершенно отличной переменной, может пройти много времени прежде, чем вы найдете ошибку. Это лишь один пример того, как необдуманное применение операций неявного преобразования может привести к проблеме.

Существует следующее эмпирическое правило: предусматривайте только те операции, что действительно необходимы для выполнения работы, и не более. Если все, что вам нужно от вашего типа — в данном случае `Complex` — это возможность участия в булевских выражениях проверки, реализуйте только `operator true` и `operator false`. Не реализуйте операцию неявного преобразования в `bool`, если только вы действительно не нуждаетесь в ней. Если так случится, что она вам нужна, и вы реализовали неявное преобразование в `bool`, тогда вам не нужно реализовывать `operator true` и `operator false`, потому что они будут избыточны. Если вы представите все три, то компилятор использует операцию неявного преобразования в `bool` вместо `operator true` и `operator false`, потому что вызов первого не более эффективен, чем второго, предполагая, что вы закодировали их одинаково.

Резюме

В настоящей главе я представил некоторые полезные рекомендации по перегрузке операций, включая унарные, бинарные и операции преобразования. Перегрузка операций — одно из тех средств, что делают C# настолько мощным и выразительным языком .NET. Однако только то, что вы можете что-то делать, еще не означает, что вы должны это делать. Неправильное использование операций преобразования и неправильное определение семантики перегрузок других операций вновь и вновь становятся причиной путаницы для пользователей (а среди них — и самого разработчика типа) наряду с нежелательным поведением. Когда доходит до перегрузки операций, предусматривайте их лишь столько, сколько необходимо, и не вторгайтесь в область общей семантики разнообразных операций. Поскольку от CLS не требуется обязательная поддержка перегружаемых операций, не все языки .NET их поддерживают. Поэтому важно всегда предоставлять явно именованные методы, выполняющие ту же функциональность. Иногда такие методы уже определены в системных интерфейсах, таких как `IComparable` или `IComparable<T>`. Никогда не изолируйте функциональность только внутри перегруженных операций, если только вы не уверены на 100%, что ваш код будет использоваться языками .NET, поддерживающими перегружаемые операции.

В следующей главе я расскажу о сложностях и приемах, связанных с созданием безопасного и нейтрального по отношению к исключениям кода в .NET Framework.

ГЛАВА 7

Исключения: безопасность и обработка

CLR включает мощную поддержку исключений. Исключения могут создаваться и генерироваться в точке, где выполнение кода не может быть продолжено, поскольку возникли некоторые исключительные условия (обычно — сбой метода или некорректное состояние). Написание безопасного по отношению к исключениям кода — настоящее искусство, которым следует овладеть. Было бы ошибкой предполагать, что единственной задачей при написании безопасного к исключениям кода является генерация исключения при возникновении ошибки и перехват его в некоторой точке. Такое представление безопасного к исключениям кода недальновидно и является прямым путем к безысходности. На самом деле безопасное к исключениям кодирование означает гарантию целостности системы перед лицом возникающих исключений. Когда исключение генерируется, исполняющая система последовательно "раскручивает" стек, выполняя очистку. Ваша задача, как программиста — структурировать ваш код таким образом, чтобы целостность состояния ваших объектов не нарушалось при раскручивании стека. В этом суть приемов безопасного к исключениям кодирования.

В настоящей главе я покажу вам, как CLR обрабатывает исключения, и механизм, участвующий в обработке исключений. Однако обработка исключений этим не ограничивается. Например, я расскажу, какие части кода должны обрабатывать исключения, а также продемонстрирую способы преодоления ловушек, поджидающих вас на этом пути. Что более важно, я покажу вам, как при написании безопасного к исключениям кода можно вообще обходиться без обработки исключений. Такой код обычно называют *нейтральным по отношению к исключениям.* Это может показаться сюрпризом, но читайте дальше, и вы ознакомитесь с деталями.

Как CLR трактует исключения

Как только исключение сгенерировано, CLR начинает процесс итеративной "раскрутки" стека исключений, фрейм за фреймом[1]. Делая это, она очищает все

[1] Если вы не знакомы с термином *фрейм стека*, вам стоит обратиться к источнику `http://en.wikipedia.org/wiki/Stack_frame`. Говоря коротко, когда каждый метод вызывается в процессе выполнения программы, в стеке строится фрейм, содержащий переданные параметры и все локальные параметры метода. Фрейм удаляется при возврате из метода. Однако когда метод вызывает другой метод и т.д., новые фреймы складываются поверх текущего фрейма, формируя вложенную структуру фреймов вызова стека.

объекты, локальные по отношению к каждому фрейму стека. В некоторой точке фрейм в стеке может иметь обработчик исключения, зарегистрированный именно для типа сгенерированного исключения. Как только CLR достигает этого фрейма, она вызывает этот обработчик, чтобы справиться с ситуацией. Если стек полностью раскручен, а обработчик для сгенерированного исключения не найден, может быть инициировано событие "необработанного исключения" для текущего домена приложения, и выполнение приложения может быть прервано.

Механика обработки исключений в C#

Если вы когда-нибудь имели дело с исключениями в С-образных языках, таких как C++, Java или даже C/C++ с использованием расширений Microsoft, предназначенных для обработки исключений (`__try/__catch/__finally`), то вы уже знакомы с базовым синтаксисом исключений в C#. В таком случае вы можете пропустить несколько следующих разделов или просто бегло просмотреть их, чтобы освежить в памяти. Попробуйте в процессе найти области, которые существенно отличают C# от других языков в стиле C.

Генерация исключений

Акт генерации исключения достаточно прост. Вы просто выполняете оператор `throw`, параметром которого является необходимое для генерации исключение. Например, предположим, что вы написали собственный класс коллекции, позволяющий пользователям обращаться к элементам по индексу, и вы хотите известить пользователя, когда в параметре передается неверный индекс. Вы можете сгенерировать исключение `ArgumentOutOfRange`, как показано в следующем коде:

```
public class MyCollection
{
    public object GetItem( int index ) {
        if( index < 0 || index >= count ) {
           throw new ArgumentOutOfRangeException();
        }

        // Здесь выполнить какую-то полезную работу
    }

    private int count;
}
```

Исполняющая система может также генерировать исключения в качестве побочного эффекта от выполнения кода. Примером сгенерированного системой исключения может служить `NullReferenceException`, которое случается, когда вы пытаетесь обратиться к полю или вызвать метод на объекте, когда фактически ссылка на объект не существует.

Изменения, касающиеся необработанных исключений, которые появились в .NET 2.0

Когда исключение сгенерировано, исполняющая система начинает искать в стеке соответствующий этому исключению блок catch. Проходя по стеку, она раскручивает его, очищая по пути каждый фрейм.

Если поиск завершается в последнем фрейме потока, а обработчик исключения не найден, в этой точке исключение считается необработанным. Что случится дальше — зависит от того, какая версия .NET Framework используется кодом.

На заметку! Вы можете инсталлировать фильтр для необработанных исключений, зарегистрировав делегат с AppDomain.UnhandledException. Когда необработанное исключение проходит сквозь весь стек, этот делегат будет вызван и получит экземпляр UnhandledExceptionEventArgs.

На заметку! CLR транслирует необработанные исключения, проходящие через статические конструкторы. Более подробно я раскрою эту тему в разделе "Исключения, сгенерированные в статических конструкторах" далее в главе.

В .NET 1.1 проектировщики CLR решили "проглатывать" некоторые необработанные исключения в интересах повышенной стабильности. Например, если финализатор генерирует исключение в .NET 1.1, вместо прерывания потока финализатора и всего процесса исключение "проглатывается" и не позволяет уничтожить поток финализатора или прервать процесс. Аналогично, если необработанное исключение проникает в поток, отличающийся от главного потока, этот поток прерывается, не затрагивая остального процесса. В потоке, управляемом пулом потоков, исключение "проглатывается", и поток возвращается в пул — такое поведение аналогично обработке исключения в потоке финализатора. Если необработанное исключение распространяется вверх из главного потока, оно ведет себя, как и ожидалось, и либо прерывается процесс, либо отображается диалог отладки JIT, спрашивающий у пользователя, что он желает предпринять.

Такое поведение выглядит неплохо в принципе, но в реальности дает результат, обратный ожидаемому. Вместо обеспечения повышенной стабильности, система приходит в нестабильное состояние, поскольку код выполняется в недетерминированном состоянии. Например, представим финализатор, выполняющий некоторую ответственную работу. Предположим, что на полпути выполнения этой работы сгенерировано исключение. Вторая половина работы финализатора останется невыполненной. Система находится в потенциально нестабильном, незавершенном состоянии. Внешне все продолжает работать нормально, хотя состояние системы может быть далеким от нормы. На практике это вызывает значительную нестабильность, поскольку источники ошибок трудно найти, раз исключения "проглатываются".

.NET 2.0 решает эту проблему, требуя, чтобы любое необработанное исключение, кроме AppDomainUnloadException и ThreadAbortException, вызывало прерывание потока. Это звучит грубо, но на самом деле это именно то поведение, которое вам нужно от необработанного исключения. В конце концов, это *необработанное* исключение. Теперь, если поток прерывается, как ожидалось, это означает генера-

цию большого красного флага в точке исключения, что позволяет вам немедленно обнаружить проблему и устранить ее. Это всегда хорошо. Вы всегда хотите, чтобы ошибки проявились как можно раньше; никогда не "глотайте" исключения, позволяя системе работать, как ни в чем не бывало.

На заметку! Если все-таки вы хотите эмулировать поведение .NET 1.1 с необработанными исключениями, вы можете потребовать его, добавив следующую опцию в конфигурационный файл вашего приложения:

```
<system>
    <runtime>
        <legacyUnhandledExceptionPolicy enabled="1"/>
    </runtime>
</system>
```

Обзор синтаксиса оператора try

Код внутри блока try защищен от исключений так, что если исключение сгенерировано, то исполняющая система ищет подходящий блок catch, чтобы обработать исключение. Независимо от того, существует или нет подходящий блок catch, если предусмотрен блок finally, он всегда выполняется, независимо от того, как поток управления покидает блок try. Рассмотрим пример оператора try в C#:

```
using System;
using System.Collections;
using System.Runtime.CompilerServices;
// Отключить предупреждение компилятора: CS1058
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = false)]
public class Entrypoint
{
    static void Main() {
        try {
           ArrayList list = new ArrayList();
           list.Add( 1 );
           Console.WriteLine( "Элемент 10 = {0}", list[10] );
        }
        catch( ArgumentOutOfRangeException x ) {
           Console.WriteLine( "=== Обработчик
           ArgumentOutOfRangeException"+
           " ===" );
           Console.WriteLine( x );
           Console.WriteLine( "=== Обработчик
           ArgumentOutOfRangeException"+
           " ===\n\n" );
        }
        catch( Exception x ) {
           Console.WriteLine( "=== Обработчик исключения ===" );
           Console.WriteLine( x );
           Console.WriteLine( "=== Обработчик исключения ===\n\n" );
        }
```

```
        catch {
            Console.WriteLine( "=== Необработанное исключение" +
                    " Handler ===" );
            Console.WriteLine( "Исключение, которое не" +
                    " ожидалось..." );
            Console.WriteLine( "=== Необработанное исключение" +
                    " Handler ===" );
        }
        finally {
            Console.WriteLine( "Очистка..." );
        }
    }
}
```

Увидев код внутри блока try, вы знаете, что он предназначен для генерации исключения ArgumentOutOfRangeException. Как только исключение сгенерировано, исполняющая система начинает поиск подходящей конструкции catch, являющейся частью этого оператора try и максимально соответствующей типу исключения. Ясно, что лучше всего подходит первая конструкция catch. Поэтому исполняющая система немедленно начинает выполнение операторов из первого блока catch. Если бы меня не интересовало действительное содержание исключения, я мог бы опустить объявление переменной исключения x в конструкции catch и ограничится объявлением типа. Но в данном случае я хотел продемонстрировать, что объекты исключения в C# предоставляют наглядную трассировку стека, которая может пригодиться при отладке. Генерируя вывод для настоящей главы, я компилировал примеры, не включая отладочную информацию. Однако если вы включите ее, то заметите, что трассировка стека также включает имена файлов и номера строк кода на различных уровнях стека.

Вторая конструкция catch будет перехватывать исключения общего типа Exception. Если код в блоке try сгенерирует исключение, производное от System.Exception и отличающееся от ArgumentOutOfRangeException, то этот второй блок catch обработает его. В C# множественные конструкции catch, ассоциированные с одним блоком try, должны следовать в таком порядке, чтобы наиболее специфичные исключения обрабатывались первыми. Компилятор C# просто не компилирует код, в котором общие конструкции catch предшествуют более специфичным. Вы можете убедиться в этом, поменяв местами первые две конструкции catch в предыдущем примере.

В C# каждое исключение, которое вы можете сгенерировать, должно наследоваться от System.Exception. Поскольку я объявил конструкцию catch, специально предназначенную для исключений типа System.Exception, как насчет третьей конструкции catch? Даже несмотря на то, что невозможно в C# сгенерировать исключение типа, не унаследованного от System.Exception, это не является невозможным для CLR. (Например, в C++ вы можете сгенерировать исключение любого типа.) Поэтому, если вы напишете ArrayList на языке, который позволяет это, может получиться, что код сгенерирует исключение не очень полезного типа, такого как System.Int32. Звучит странно, но такое может быть. В этом случае вы можете перехватить такое исключение в C#, применив catch без явного типа исключения и без переменной. К сожалению, в этом случае не существует простого способа узнать тип сгенерированного исключения. К тому же оператор try может иметь максимум одну такую общую конструкцию catch без аргументов.

На заметку! Начиная с .NET 2.0, ситуация с общими конструкциями catch немного отличается от .NET 1.1. Там появился новый атрибут — `RuntimeCompatibilityAttribute`, который вы можете прикрепить к своей сборке. Компиляторы C# и Visual Basic, ориентированные на .NET 2.0, применяют это свойство по умолчанию. Это говорит исполняющей системе о том, что нужно поместить исключения, не унаследованные от `System.Exception`, в оболочки исключения типа `RuntimeWrappedException`, которое наследуется от `System.Exception`. Это удобно, поскольку позволяет коду C# получить доступ к сгенерированному исключению. Ранее вы не имели доступа к сгенерированному исключению, поскольку оно перехватывалось общим, лишенным параметра, блоком `catch`. Теперь вы можете получить доступ к действительному типу сгенерированного исключения через свойство `RuntimeWrappedException.WrappedException`. Если ваш код содержит `catch` без параметров, компилятор по умолчанию выдает предупреждение CS1058, если только вы не отключите атрибут, как это сделал я в предыдущем примере.

В самом конце находится блок `finally`. Независимо от того, как произошел выход из блока `try` — по достижении его конечной точки, через генерацию исключения или оператор `return` — блок `finally` выполняется всегда. Если есть подходящий блок `catch` в том же фрейме, что и блок `finally`, он выполняется перед блоком `finally`. Вы можете убедиться в этом, взглянув на вывод предыдущего кода примера, который выглядит следующим образом:

```
=== Обработчик ArgumentOutOfRangeException ===
System.ArgumentOutOfRangeException: Index was out of range. Must be
non-negative and less than the size of the collection.
Parameter name: index
```

System.ArgumentOutOfRangeException: Выход индекса за допустимые пределы. Должен быть неотрицательным и меньше размера коллекции.
Имя параметра: index

```
    at System.Collections.ArrayList.get_Item(Int32 index)
    at Entrypoint.Main()
=== Обработчик ArgumentOutOfRangeException ===

Очистка...
```

Повторная генерация и трансляция исключений

Внутри определенного фрейма стека вы можете решить перехватывать все исключения или же определенное их подмножество, достаточно долго выполняя одну и ту же очистку и затем заново генерируя исключение, чтобы продолжить его распространение по стеку. Чтобы сделать это, вы используете оператор `throw` без параметра:

```
using System;
using System.Collections;
public class Entrypoint
{
    static void Main() {
        try {
            try {
                ArrayList list = new ArrayList();
```

```
                list.Add( 1 );
                Console.WriteLine( "Элемент 10 = {0}", list[10] );
            }
            catch( ArgumentOutOfRangeException ) {
                Console.WriteLine( "Выполнить полезную работу и" +
                                   " повторить исключение" );
                // Заново сгенерировать перехваченное исключение.
                throw;
            }
            finally {
                Console.WriteLine( "Очистка..." );
            }
        }
        catch {
            Console.WriteLine( "Готово" );
        }
    }
}
```

Обратите внимание, что любые блоки finally, ассоциированные с фреймом исключения, с которым ассоциирован блок catch, будут выполнены перед любым выполнением обработчика любого исключения более высокого уровня. Вы можете видеть это в следующем выводе кода:

```
Выполнить полезную работу и повторить исключение
Очистка...
Готово
```

В разделе "Обеспечение нейтральности к исключениям" этой главы я представлю некоторые приемы, которые помогут вам избежать перехвата исключений, выполнения очистки и затем повторной их генерации. Такой стиль работы довольно неуклюж, поскольку вы должны следить за правильной повторной генерацией исключений. Если вдруг вы нечаянно забудете это сделать, все станет плохо, поскольку в этом случае, вероятно, просто не справитесь с исключительной ситуацией. Приемы, которые я покажу, помогут вам достичь цели, чтобы единственным местом, куда помещается блок catch, было то, где может возникнуть действие по исправлению ситуации.

Иногда вы можете счесть необходимым "транслировать" исключение внутри обработчика исключений. В этом случае вы перехватываете исключение одного типа, но затем генерируете исключение другого типа — возможно, более точного — в блоке catch для передачи его на обработку на следующем уровне. Рассмотрим следующий пример:

```
using System;
using System.Collections;
public class MyException : Exception
{
    public MyException( String reason, Exception inner )
        :base( reason, inner ) {
    }
}
```

```
public class Entrypoint
{
    static void Main() {
        try {
           try {
               ArrayList list = new ArrayList();
               list.Add( 1 );
               Console.WriteLine( "Элемент 10 = {0}", list[10] );
           }
           catch( ArgumentOutOfRangeException x ) {
               Console.WriteLine( "Выполнить полезную работу и " +
                                  " повторить исключение" );
                  throw new MyException( "Лучше сгенерирую это", x ) ;
           }
           finally {
               Console.WriteLine( "Очистка..." );
           }
        }
        catch( Exception x ) {
           Console.WriteLine( x );
           Console.WriteLine( "Готово" );
        }
    }
}
```

Одним особым качеством типа `System.Exception` является способность включать в себя ссылку на вложенное исключение через свойство `Exception.InnerException`. Таким образом, когда генерируется новое исключение, вы можете предохранить цепочку исключений для исключений, обрабатывающих их. Я рекомендую использовать это полезное свойство стандартного типа исключений C# для трансляции ваших исключений. Вывод предыдущего кода выглядит следующим образом:

```
Выполнить полезную работу и повторить исключение
Очистка...
MyException: Лучше сгенерирую это --> System.ArgumentOutOfRangeException:
Index was out of range.
Must be non-negative and less than the size of the collection.
Parameter name: index
    at System.Collections.ArrayList.get_Item(Int32 index)
    at Entrypoint.Main()
    -- End of inner exception stack trace --
    -- Конец трассировки стека внутреннего исключения --
    at Entrypoint.Main()
Готово
```

Имейте в виду, что по возможности вы должны избегать трансляции исключений. Чем больше вы перехватываете и генерируете заново в стеке, тем больше изолируете код, обрабатывающий исключение, от кода, его генерирующего. То есть становится трудным сопоставить точку перехвата с исходной точкой генерации исключения. Да, свойство `Exception.InnerException` помогает смягчить пробле-

му, но все равно трудно найти изначальную причину проблемы, если исключения транслируются по пути.

Исключения, сгенерированные в блоке `finally`

Возможно, но крайне нежелательно, генерировать исключения внутри блока `finally`. Следующий код демонстрирует пример:

```
using System;
using System.Collections;
public class Entrypoint
{
    static void Main() {
        try {
            try {
                ArrayList list = new ArrayList();
                list.Add( 1 );
                Console.WriteLine( "Элемент 10 = {0}", list[10] );
            }
            finally {
                Console.WriteLine( "Очистка..." );
                throw new Exception( "Лучше сгенерирую это" );
            }
        }
        catch( ArgumentOutOfRangeException ) {
            Console.WriteLine( "Оп! Аргумент вышел за допустимые пределы!" );
        }
        catch {
            Console.WriteLine( "Готово" );
        }
    }
}
```

Первое исключение просто теряется, и новое исключение распространяется по стеку. Ясно, что это нежелательно. Вы никогда не должны терять след исключений, поскольку тогда становится почти невозможным определить, что именно вызвало исключение в самом начале.

Исключения, сгенерированные в финализаторах

Деструкторы C# не являются действительно детерминированными деструкторами, а финализаторами CLR. Финализаторы исполняются в контексте потока финализатора, который является на самом деле контекстом произвольного потока. Если финализатор должен генерировать исключение, то CLR может не знать, как обработать эту ситуацию и может просто прервать поток (и процесс). Рассмотрим следующий код:

```
using System;
public class Person
{
    ~Person() {
        Console.WriteLine( "Очистка Person..." );
```

```
            Console.WriteLine( "Очистка Person завершена..." );
        }
    }

    public class Employee : Person
    {
        ~Employee() {
            Console.WriteLine( "Очистка Employee..." );
            object obj = null;
            // Следующий код сгенерирует исключение.
            Console.WriteLine( obj.ToString() );
            Console.WriteLine( "Очистка Employee завершена..." );
        }
    }

    public class Entrypoint
    {
        static void Main() {
            Employee emp = new Employee();
            emp = null;
        }
    }
```

Ниже показан вывод этого кода:

```
Очистка Employee...

Unhandled Exception: System.NullReferenceException: Object reference
not set to an instance of an object.
   at Employee.Finalize()
Необработанное исключение: System.NullReferenceException: Объектная
ссылка не установлена в экземпляр объекта.
   at Employee.Finalize()

Очистка Person...
Очистка Person завершена...
```

Вы обнаружите небольшое отличие в поведении этого примера при запуске под .NET 1.1 и .NET 2.0. В .NET 1.1 исключение "проглатывается" вместе с выводом на консоль, и выполнение продолжается дальше. В .NET 2.0 ваша исполняющая среда отобразит вам знакомый отладочный диалог JIT, который спросит, не желаете ли вы отладить приложение. Проблема в том, что у вас мало времени на ответ, прежде чем исполняющая система .NET 2.0 прервет ваше приложение. Если вы еще не сделали этого, прочтите о различиях в трактовке необработанных исключений между .NET 1.1 и .NET 2.0 в разделе "Изменения, касающиеся необработанных исключений, которые появились в .NET 2.0" ранее в настоящей главе.

Следует любой ценой избегать намеренной генерации исключений в финализаторах, поскольку вы можете прервать процесс. В качестве завершающего замечания: прежде всего, прочтите в главе 13 обо всех "за" и "против" создания финализаторов.

Исключения, сгенерированные в статических конструкторах

Если исключение сгенерировано, и в стеке нет обработчика, так что его поиск завершается в статическом конструкторе, то исполняющая система обрабатывает этот случай специальным образом. Она транслирует исключение в System.TypeInitializationException и генерирует его вместо первоначального. Прежде чем генерировать новое исключение, свойство InnerException нового исключения устанавливается в исходное исключение. Таким образом, любой обработчик исключения инициализации типа может легко обнаружить причину сбоя.

Такая трансляция исключения имеет смысл, потому что конструкторы по своей природе не могут возвращать значение, свидетельствующее об успехе или неудаче. Исключения — единственный механизм, имеющийся в вашем распоряжении, который позволяет сигнализировать о сбое конструктора. Что более важно — поскольку система вызывает статические конструкторы в определенное самой системой время[2], для них имеет смысл использование типа TypeInitializationException, чтобы точнее определиться, когда что-то идет не так. Например, предположим, у вас есть статический конструктор, который может потенциально генерировать ArgumentOutOfRangeException. Теперь представьте разочарование пользователей, если ваше исключение распространится во включающем потоке в некоторый случайный момент времени, поскольку точный момент вызова статического конструктора определяется системой. Может показаться, что ArgumentOutOfRange-Exception материализуется буквально "из воздуха". Помещение вашего исключения в оболочку TypeInitializationException немного проясняет ситуацию и предупреждает пользователей, или, можно надеяться, разработчика, о том, что проблема возникла во время инициализации типа.

Следующий код демонстрирует пример того, как выглядит TypeInitialization-Exception с вложенным внутри него исключением:

```
using System;
using System.IO;
class EventLogger
{
    static EventLogger() {
        eventLog = File.CreateText( "logfile.txt" );
        // Следующий оператор сгенерирует исключение.
        strLogName = (string) strLogName.Clone();
    }
    static public void WriteLog( string someText ) {
        eventLog.Write( someText );
    }
    static private StreamWriter eventLog;
    static private string strLogName;
}
public class EntryPoint
{
    static void Main() {
```

[2] Система может вызывать статические конструкторы во время загрузки типа или просто перед обращением к статическому члену, в зависимости от того, как сконфигурирована CLR для текущего процесса.

```
            EventLogger.WriteLog( "Зарегистрировать это!" );
        }
    }
```

Запустив этот пример, вы получите следующий вывод:

```
Unhandled Exception: System.TypeInitializationException:
The type initializer for 'EventLogger' threw
an exception. --> System.NullReferenceException: Object reference not set
to an instance of an object.
    at EventLogger..cctor()
    -- End of inner exception stack trace --
    at EntryPoint.Main()
```

Обратите внимание, что наряду с указанием того, что внешнее исключение имеет тип `TypeInitializationException`, вывод также показывает внутреннее исключение, с которого все началось — это `NullReferenceException`.

Кто должен обрабатывать исключения?

Где вы должны обрабатывать исключения? Вы можете найти ответ в применении варианта шаблона Expert (Эксперт), который устанавливает, что работа должна выполняться сущностью, являющейся экспертом в данной области. Это — окольный путь уведомить, что вы должны перехватывать исключение в точке, где вы действительно можете обработать его с уровнем знаний, достаточным для того, чтобы справиться с исключительной ситуацией наилучшим образом. Иногда перехватывающая сущность может находиться близко к точке генерации исключения внутри фрейма стека. Код может перехватить исключение, затем предпринять некоторые действия по исправлению ситуации, после чего позволить программе нормально продолжить выполнение. В других случаях единственным разумным местом для перехвата исключения может быть метод точки входа — `Main`, где вы можете либо прервать процесс после предоставления некоторых полезных данных, либо сбросить процесс, как если бы приложение было просто только что перезапущено. Главный вывод в том, что вы должны найти лучший способ восстановления после исключения, если это возможно, и лучшее место определяется тем, где наиболее целесообразно сделать это.

Избегайте применения исключений для управления потоком выполнения

Может возникнуть искушение применять исключения для управления потоком выполнения в сложных методах. Это никогда нельзя считать хорошей идеей по двум причинам. Во-первых, генерация и обработка исключений обходится дорого. Поэтому, если вы хотите использовать их для управления потоком выполнения внутри метода, находящегося в ядре вашего приложения, его производительность, скорее всего, снизится. Во-вторых, это нивелирует саму природу исключений. Главное назначение исключений — указать исключительные условия таким образом, чтобы их можно было четко обработать или сообщить о них.

Исторически сложилось так, что программисты достаточно ленивы в отношении обработки исключительных условий. Сколько раз вы видели код, где программист даже не позаботился проверить значение, возвращенное функцией API или вызовом метода? Такие пассивные подходы к обработке ошибок могут быстро привести к серьезным проблемам. Исключения представляют собой синтаксически сжатый способ для описания и обработки ошибочных условий, не засоряя ваш код множеством блоков `if` и других традиционных (не связанных с исключениями) конструкций обработки ошибок. В то же время исполняющая система поддерживает исключения, и выполняет за вас огромную работу, когда исключение сгенерировано. Раскрутка стека — сама по себе нетривиальная задача. В конечном итоге точка, в которой сгенерировано исключение, и точка, где оно обработано, могут находиться далеко друг от друга и не иметь прямой связи друг с другом. Поэтому бывает трудно при чтении кода определить, где исключение будет перехвачено и обработано. Только этих причин достаточно для того, чтобы придерживаться традиционных приемов при управлении нормальным потоком выполнения.

На заметку! На эту тему вы можете найти статью *The Cost of Exceptions* (Цена исключений) в блоге Рико Мариани (Rico Mariani) по адресу `http://blogs.msdn.com/ricom/archive/2003/12/19/44697.aspx`. Рико — эксперт в области проблем производительности в CLR.

Обеспечение нейтральности к исключениям

Когда исключения были впервые добавлены к C++, многие разработчики были впечатлены возможностью генерировать их, перехватывать и обрабатывать. Фактически распространенным заблуждением в то время было то, что обработка исключений состоит из стратегического размещения операторов `try` по всему коду и "подсыпания" `throw` при необходимости. Со временем сообщество разработчиков осознало, что беспорядочное разбрасывание операторов `try` ухудшает читабельность кода, когда в большинстве случаев единственной целью, которой они пытались достичь, было аккуратно убрать за сгенерированным исключением и позволить ему распространяться далее по стеку. Хуже того, это затрудняло написание кода и его дальнейшее сопровождение. Код, не обрабатывающий исключений, но от которого ожидается правильное поведение перед лицом этих исключений, называется *нейтральным к исключениям* кодом.

Ясно, что должен был быть лучший способ написания нейтрального по отношению к исключениям кода, без необходимости полагаться повсюду на операторы `try`. Фактически единственное место, где есть необходимость в операторе `try` — это точка, в которой вы выполняете любого рода восстановление или протоколирование системы в ответ на исключение. Со временем многие начали понимать, что написание операторов `try` было, фактически наименее значимой частью написания безопасного и нейтрального к исключениям кода. Вообще единственный код, который должен перехватывать исключения — это код, знающий, как исправить ситуацию. Такой код может находиться даже в главной точке входа и просто выполнять сброс системы в некоторое известное начальное состояние, по сути, перезапуская ее.

Под нейтральным к исключениям кодом я подразумеваю код, который не предусматривает возможности каким-то специальным образом обрабатывать исключения, но, тем не менее, должен изящно справляться с ними. Обычно этот код находится где-то в стеке — между кодом, сгенерировавшим исключение, и кодом, перехватившим его, — и он не должен зависеть от исключения на его пути в стеке. Здесь некоторые из вас, возможно, подумали об операторе `throw` без параметров, которое позволяет перехватить исключение, выполнить какую-то работу и затем повторно сгенерировать исключение. Однако я хотел бы представить вам намного более ясную технику, позволяющую писать нейтральный к исключениям код без использования единого оператора `try`, создавая при этом более устойчивый и более читабельный код.

Базовая структура нейтрального к исключениям кода

Общая идея, лежащая в основе написания нейтрального к исключениям кода, подобна идее, лежащей в основе кода фиксации/отката (commit/rollback). Вы пишете такой код, который гарантирует, что если его выполнение не завершено, то вся операция отменяется без каких-либо последствий для состояния системы. Изменения в состоянии фиксируются только в том случае, когда код достигает конечной точки выполнения. Вы должны кодировать ваши методы подобным образом для того, чтобы они были нейтральны к исключениям. Если исключение сгенерировано до конца метода, то состояние системы должно оставаться неизменным. Следующий шаблон демонстрирует, как необходимо структурировать ваш метод, чтобы достичь этой цели:

```
void ExceptionNeutralMethod()
{
    //————————————
    // Весь код, который потенциально может сгенерировать исключение,
    // находится в первом разделе. Здесь не применяется никаких
    // изменений в состоянии ни к каким объектам системы, включая данный.
    //————————————
    //————————————
    // Все изменения фиксируются в этой точке с использованием
    // операций, гарантирующих отсутствие генерации исключений.
    //————————————
}
```

Как видите, эта техника не работает, если только у вас нет набора операций, которые гарантированно не генерируют исключений. В противном случае невозможно реализовать проиллюстрированное поведение "фиксации/отката". К счастью, исполняющая система .NET предоставляет несколько операций, спецификация которых гарантирует, что исключения никогда не будут ими сгенерированы.

Начнем с построения примера, описывающего то, что я имею в виду. Предположим, что у вас есть система или приложение, в котором вы управляете сотрудниками. Договоримся, что как только сотрудник зарегистрирован, для него создается объект `Employee`, который должен храниться в одной и только одной коллекции в системе. В данный момент только две коллекции в системе представляют активных сотрудников, а одна представляет уволенных. Вдобавок коллекции существуют внутри объекта `EmployeeDatabase`, как показано в следующем примере:

```
using System.Collections;
class EmployeeDatabase
{
    private ArrayList activeEmployees;
    private ArrayList terminatedEmployees;
}
```

В примере используются коллекции типа `ArrayList` из пространства имен `System.Collections`. Реальная система, очевидно, должна иметь дело с чем-нибудь более удобным, например, с базой данных.

Теперь посмотрим, что случится при увольнении сотрудника. Естественно, вы должны переместить этого сотрудника из коллекции `activeEmployees` в коллекцию `terminatedEmployees`. Наивная реализация такой задачи могла бы выглядеть так:

```
using System.Collections;
class Employee
{
}
class EmployeeDatabase
{
    public void TerminateEmployee( int index ) {
        object employee = activeEmployees[index];
        activeEmployees.RemoveAt( index );
        terminatedEmployees.Add( employee );
    }
    private ArrayList activeEmployees;
    private ArrayList terminatedEmployees;
}
```

Этот код выглядит достаточно адекватным. Метод, выполняющий перемещение, предполагает, что вызывающий его код каким-то образом знает индекс текущего сотрудника в списке `activeEmployees` до вызова `TerminateEmployee`. Он копирует ссылку на указанного сотрудника, удаляет ее из `activeEmployees`, и добавляет в коллекцию `terminatedEmployees`. Так что же плохого в этом методе?

Присмотритесь к методу повнимательней и подумайте, где в нем могут генерироваться исключения. Фактически исключение может быть сгенерировано при любом вызове метода внутри данного метода. Если индекс выйдет за пределы допустимого диапазона, можно ожидать генерации исключения `ArgumentOutOfRange` в первых двух строках. Конечно, если исключение диапазона будет сгенерировано из первой строки, то выполнение никогда не дойдет до второй строки, но идея должна быть вам ясна. И, если памяти недостаточно, может случиться, что вызов `Add` завершиться неудачей с генерацией исключения. Опасность исходит от возможности генерации исключения после модификации состояния системы. Представим, что передан правильный индекс. Первые две строки успешно выполнятся. Однако если исключение будет сгенерировано при попытке добавить сотрудника к `terminatedEmployees`, то сотрудник будет потерян для системы. Что же делать, чтобы решить проблему?

Начальная попытка может использовать операторы `try` для предотвращения повреждения состояния системы. Рассмотрим следующий пример:

```
using System.Collections;
class Employee
{
}
class EmployeeDatabase
{
    public void TerminateEmployee( int index ) {
        object employee = null;
        try {
            employee = activeEmployees[index];
        }
        catch {
            // Оп! Индекс вне диапазона.
        }
        if( employee != null ) {
            activeEmployees.RemoveAt( index );
            try {
                    terminatedEmployees.Add( employee );
            }
            catch {
                    // Оп! Распределить память не удалось.
                    activeEmployees.Add( employee );
            }
        }
    }
    private ArrayList activeEmployees;
    private ArrayList terminatedEmployees;
}
```

Смотрите, как быстро код стало трудно читать и понимать — и все "благодаря" операторам try. Вы должны вынести ссылку employee из оператора и инициализировать ее значением null. Попытавшись получить ссылку на сотрудника, вы должны проверить ее на равенство null, чтобы убедиться, в том, что вы действительно получили ее. В случае успеха вы можете выполнить добавление employee в список terminatedEmployees. Однако если по какой-то причине это не получится, вы должны вернуть employee обратно в список activeEmployees.

Вы сразу можете заметить несколько проблем данного подхода. Прежде всего, что случится, если у вас не получится добавить employee обратно в коллекцию activeEmployees? Можно ли смириться с таким сбоем? Нет! Это неприемлемо, поскольку состояние системы уже изменилось. Во-вторых, вам, вероятно, нужно вернуть код ошибки из этого метода, чтобы сообщить, почему он потерпел неудачу. Я не сделал этого в предыдущем коде. Метод не может просто спокойно вернуть управление, как будто все прошло гладко, когда на самом деле действие завершить не удалось. В-третьих, код получился просто громоздким и трудно читаемым. И последнее: в этом коде по-прежнему остаются проблемы, на описание которых я просто не хочу тратить времени.

Так каково же решение? Подумайте о том, что вы пытаетесь сделать с помощью операторов try. Вы хотите выполнить действия, которые, возможно, сгенерируют исключение, и если это случится, вернуться к предыдущему состоянию. На самом

деле вы можете попробовать разработать вариант кода без операторов `try`, который будет работать следующим образом: сначала попытается выполнить в одном методе все действия, которые могут сгенерировать исключение, и за последним из них зафиксировать изменения, используя операции, не генерирующие исключений.

На заметку! Сообщество C++ признало такую технику, отчасти, благодаря блестящей работе, опубликованной Хербом Саттером (Herb Sutter) в его серии *Exceptional C++* (Boston, MA: Addison-Wesley Professional). Ничто не мешает вам применить эту технику целиком в мире C#.

```
using System.Collections;
class Employee
{
}
class EmployeeDatabase
{
    public void TerminateEmployee( int index ) {
        // Клонировать важнейшие объекты.
        ArrayList tempActiveEmployees =
           (ArrayList) activeEmployees.Clone();
        ArrayList tempTerminatedEmployees =
           (ArrayList) terminatedEmployees.Clone();
        // Выполнить действия над временными объектами.
        object employee = tempActiveEmployees[index];
        tempActiveEmployees.RemoveAt( index );
        tempTerminatedEmployees.Add( employee );
        // Зафиксировать изменения.
        ArrayList tempSpace = null;
        ListSwap( ref activeEmployees,
                  ref tempActiveEmployees,
                  ref tempSpace );
        ListSwap( ref terminatedEmployees,
                  ref tempTerminatedEmployees,
                  ref tempSpace );
    }
    void ListSwap( ref ArrayList first,
                   ref ArrayList second,
                   ref ArrayList temp ) {
        temp = first;
        first = second;
        second = temp;
        temp = null;
    }
    private ArrayList activeEmployees;
    private ArrayList terminatedEmployees;
}
```

Для начала обратите внимание на отсутствие операторов `try`. В их отсутствии замечательно то, что методу не нужен код возврата. Вызывающий код может ожидать, что метод либо работает, как обещано, либо генерирует исключение. Только две строки в методе затрагивают состояние системы — это последние два вызова `ListSwap`.

ListSwap предназначен для того, чтобы позволить вам заменить ссылки на объекты ArrayList в EmployeeDatabase ссылками на временные модифицированные копии.

Как такая техника может быть настолько лучше, если она выглядит намного менее эффективной? Секретов два. Один, очевидный, состоит в том, что независимо от того, где бы в этом методе не было сгенерировано исключение, состояние EmployeeDatabase останется незатронутым. Но что, если исключение произойдет внутри ListSwap? Здесь кроется другой секрет: ListSwap никогда не генерирует исключений. Одним из наиболее важных условий, необходимых для создания нейтрального к исключениям кода является наличие набора операций, которые гарантированно выполняются без сбоев при нормальных условиях. Я не рассматриваю случай, когда какой-нибудь "умелец" выдернет вилку компьютера из розетки во время выполнения ListSwap, либо в этот момент произойдет катастрофическое землетрясение или налетит торнадо. Давайте посмотрим, почему ListSwap не генерирует никаких исключений.

Для того чтобы создать нейтральный по отношении к исключениям код, нужно иметь несколько операций, таких как операции присваивания, которые гарантированно не генерируют исключений. К счастью, CLR предоставляет такие операции. Присваивание ссылок, когда не требуется никаких преобразований типа — один из примеров таких операций. Каждая ссылка указывает на объект, хранящийся в определенном месте, и это место имеет ассоциированный с ним тип. Однако, как только место выделено и существует, копирование ссылок с одного на другое представляет собой простое копирование уже выделенных мест — подобно копированию обычных указателей в C++, которые просто не могут вызвать сбой. Это отлично работает, когда вам нужно скопировать ссылку одного типа на ссылку того же самого типа.

Но что случится, когда потребуется преобразование? Может ли оно сгенерировать исключение? Стандарт C# специфицирует, что выполнение неявных операций преобразования никогда не генерирует исключений. Если ваше присваивание подразумевает неявное преобразование, вы защищены, если исходить из того, что пользовательские операции неявного преобразования следуют стандарту и не генерируют исключений[3]. Если вы найдете пользовательскую операцию неявного преобразования, которая генерирует исключения, я советую немедленно показать спецификацию C# ее разработчику. Однако явные преобразования, в форме приведений, могут генерировать исключения. Подытожим: простое присваивание одной ссылки другой, независимо от того, требует оно неявного преобразования или нет, исключений не сгенерирует.

Простое присваивание одной ссылки на место в памяти другой — это все, что делает ListSwap. После того, как вы привели временные объекты ArrayList в нужное состояние и добрались до точки вызовов ListSwap, значит, вы достигли точки, в которой можно быть уверенным, что никаких исключений в методе TerminateEmployee уже не произойдет. Теперь можно безопасно выполнить замену. Объекты ArrayList в EmployeeDatabase подменяются временными объектами. По завершении метода исходные объекты ArrayList готовы к тому, чтобы их подобрал сборщик мусора.

---

[3] В руководстве по C# четко указано, что пользовательские операции преобразования НЕ ДОЛЖНЫ генерировать исключений.

Еще один момент, который нужно отметить относительно ListSwap — временное место для хранения экземпляра ArrayList во время обмена выделяется вне метода ListSwap и передается ему как параметр ref. Я делаю это для того, чтобы избежать исключения StackOverflowException внутри ListSwap. Существует минимальная вероятность, что при вызове ListSwap стек будет заполнен, и простое выделение очередного места в стеке закончится неудачей и вызовет исключение. Поэтому я выполняю этот шаг вне границ метода ListSwap. При входе в ListSwap вся необходимая память выделена и готова к использованию.

Эта техника, применяемая слишком часто в системе, требующей жесткой стабильности, быстро приводит к появлению чересчур сложных методов, которые нужно разбивать на более мелкие функциональные единицы. По сути, эта идиома повышает сложность метода, к которому она применена. Поэтому, если вы обнаруживаете, что становится трудно и громоздко обеспечить "пуленепробиваемость" метода, вам стоит проанализировать его и убедиться в том, что вы не пытаетесь выполнить слишком много работы сразу, которую можно было бы разбить на более мелкие порции.

Кстати, вы можете обнаружить необходимость в том, чтобы сделать операции обмена, подобные ListSwap, атомарными для многопоточной среды. Можно модифицировать ListSwap для использования некоторого рода блокирующего объекта, такого как семафор (mutex) или объект System.Threading.Monitor. Однако вы можете нечаянно сделать ListSwap способным генерировать исключения, что нарушит предъявляемые к нему требования. К счастью, пространство имен System.Threading предоставляет класс Interlocked для выполнения таких операций обмена в атомарном режиме, причем методы гарантированно не генерируют исключения. Класс Interlocked представляет обобщенную перегрузку всех полезных методов, обеспечивая их высокую эффективность. Обобщенные методы Interlocked ограничены работой только со ссылочными типами.

Резюмирую сказанное: вы должны делать все, что может сгенерировать исключение, перед модификацией состояния объекта, над которым производится операция. Пройдя последнюю точку, где возможны любые исключения, зафиксируйте изменения с использованием операций, гарантированно не генерирующих исключения.

Если перед вами стоит задача создания устойчивой реальной системы, на целостность которой полается множество людей, трудно переоценить ценность этой идиомы. Конечно, она не так эффективна, как наивный подход, и требует больше системных ресурсов для эффективной работы, но ваши клиенты всегда предпочтут неэффективность повреждению данных. Ваши коллеги также поблагодарят вас, поскольку утечки ресурсов и прочие неприятности, являющиеся побочным эффектом генерации исключений, очень трудно находить из-за их "бессвязной" природы.

Ограниченные области выполнения

Пример из предыдущего раздела демонстрирует некоторый уровень паранойи, который вы должны допускать для написания "пуленепробиваемого" нейтрального к исключениям кода. Я был настолько параноиком, что для предотвращения исключения переполнения стека выделил дополнительное место, необходимое ListSwap, перед вызовом этого метода. Вы можете подумать, что позаботились обо всех неприятностях. К сожалению, вы ошибаетесь. В среде CLR могут случаться

другие асинхронные исключения, такие как ThreadAbortException (о котором я расскажу в главе 12), OutOfMemoryException и StackOverflowException.

Например, что если на фазе фиксации изменений метода TerminateEmployee домен приложения будет остановлен, вызвав исключение ThreadAbortException? Или что если во время первого вызова ListSwap компилятор JIT не сможет выделить достаточно памяти для первоначальной компиляции метода? Ясно, что с такой нехорошей ситуацией трудно справиться. Фактически во времена .NET 1.1 вы мало что могли сделать в таких дьявольских ситуациях. Однако, начиная с .NET 2.0, можно применить *ограниченную область выполнения* (Constrained Execution Region — CER), или *критичный финализатор*.

CER представляет собой участок кода, который CLR подготавливает до выполнения, так что когда в нем возникает потребность, все необходимое имеется под рукой и вероятность сбоя минимальна. Более того, CLR откладывает доставку любых асинхронных исключений вроде ThreadAbortException на время выполнения кода CER. Вы можете применить магию CER, используя класс RuntimeHelpers из пространства имен System.Runtime.CompilerServices. Чтобы создать CER, просто вызовите в своем коде RuntimeHelpers.PerpareConstrainedRegions перед оператором try. CLR затем проверит блоки catch и finally, и подготовит их, пройдя по графу вызовов, чтобы убедиться, что все вызываемые в нем методы уже JIT-скомпилированы и необходимое пространство стека доступно[4]. Несмотря на то что вы вызываете PrepareConstrainedRegions до оператора try, действительный код внутри блока try не подготовлен. Поэтому вы можете использовать следующую идиому для подготовки произвольных участком вода, упаковывая их в блок finally внутри CER:

```
RuntimeHelpers.PrepareConstrainedRegions();
try {} finally
{
    // сюда поместите критичный код
}
```

Давайте посмотрим, как можно модифицировать предыдущий пример с использованием CER, чтобы сделать его еще более надежным:

```
using System.Collections;
using System.Runtime.CompilerServices;
using System.Runtime.ConstrainedExecution;
class Employee
{
}

class EmployeeDatabase
{
```

[4] Кстати, виртуальные методы и делегаты представляют проблему, поскольку граф вызовов не может их учесть во время подготовки. Однако если вы знаете цель виртуального метода или делегата, то можете подготовить его явно, вызвав для этого RuntimeHelpers.PrepareDelegate. Я рекомендую почитать статью Стивена Тоуба (Stephen Toub) *Keep Your Code Running with the Reliability Features of the .NET Framework* (Сохраняйте работоспособность своего кода с помощью средств повышения надежности .NET Framework), доступную по адресу http://msdn.microsoft.com/msdnmag/issues/05/10/Reliability/default.aspx.

```
    public void TerminateEmployee( int index ) {
        // Клонировать важные объекты.
        ArrayList tempActiveEmployees =
            (ArrayList) activeEmployees.Clone();
        ArrayList tempTerminatedEmployees =
            (ArrayList) terminatedEmployees.Clone();
        // Выполнить действия над временными объектами.
        object employee = tempActiveEmployees[index];
        tempActiveEmployees.RemoveAt( index );
        tempTerminatedEmployees.Add( employee );
        RuntimeHelpers.PrepareConstrainedRegions();
        try {} finally {
            // Зафиксировать изменения.
            ArrayList tempSpace = null;
            ListSwap( ref activeEmployees,
                      ref tempActiveEmployees,
                      ref tempSpace );
            ListSwap( ref terminatedEmployees,
                      ref tempTerminatedEmployees,
                      ref tempSpace );
        }
    }
    [ReliabilityContract( Consistency.WillNotCorruptState,
                          Cer.Success )]
    void ListSwap( ref ArrayList first,
                   ref ArrayList second,
                   ref ArrayList temp ) {
        temp = first;
        first = second;
        second = temp;
        temp = null;
    }
    private ArrayList activeEmployees;
    private ArrayList terminatedEmployees;
}
```

Обратите внимание, что раздел фиксации в методе TerminateEmployee упакован в CER. Во время выполнения, прежде чем запустить этот код, CLR подготовит его, также подготавливая метод ListSwap и убедившись, что стек может обслужить эту работу. Конечно, такая подготовительная операция может потерпеть неудачу, и это нормально, поскольку на этот момент вы еще не приступили к выполнению кода, фиксирующего изменения. Обратите внимание на добавление Reliability-ContractAttribute к методу ListSwap. Этот атрибут информирует исполняющую систему, какого рода гарантии предоставляет метод ListSwap, чтобы правильно сформировать CER. Вы можете также прикрепить ReliabilityContractAttribute к методу TerminateEmployee, но это полезно только для кода, выполняемого внутри CER. Если вы хотите снабдить этим атрибутом метод TerminateEmployee, чтобы можно было вызывать его внутри CER, определенного где-либо, вы можете добавить следующий атрибут:

```
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
```

Этот ReliabilityContractAttribute выражает цель, которую вы ставите перед TerminateEmployee прежде всего. То есть сбой может произойти, но состояние системы не будет повреждено.

На заметку! Даже несмотря на то, что CLR гарантирует, что асинхронные исключения не вмешаются в выполнение потока, пока он находится внутри CER, это не дает никаких гарантий подавления всех исключений. Подавляются только те, которые находятся вне вашего контроля. Это значит, что если вы создаете объекты внутри CER, то должны быть готовы к обработке исключения OutOfMemoryException или любого другого, которое может быть вызвано вашим кодом.

Критичные финализаторы и SafeHandle

Критичные финализаторы подобны CER в том отношении, что код внутри них защищен от асинхронных исключений и прочих подобных опасностей, исходящих от виртуальной исполняющей системы и находящихся вне вашего влияния. Чтобы указать, что ваш объект имеет критичный финализатор, просто наследуйте его от CriticalFinalizerObject. Поступив так, вы гарантируете вашему объекту наличие критичного финализатора, исполняющегося в контексте CER, и потому подчиняющегося всем правилам, налагаемым CER. Вдобавок CLR будет выполнять критичные финализаторы после завершения работы с остальными некритичными финализируемыми объектами.

В действительности у вас редко возникнет необходимость в создании критично финализируемого объекта. Вместо этого вы можете получить нужное поведение, наследуясь от SafeHandle. SafeHandle — важнейший инструмент при создании "родного" функционально совместимого кода через P/Invoke или COM, поскольку позволяет вам гарантировать, что не произойдет никаких утечек ресурсов изнутри CLR. До появления .NET 2.0 это было невозможно. Во времена .NET 1.1 вы должны были обычно представлять непрозрачный тип "родного" дескриптора посредством управляемого типа IntPtr. Это работало почти хорошо, если не учитывать того, что вы были не гарантированы от очистки лежащего в основе ресурса в случае возникновения асинхронного исключения, такого как ThreadAbortException. Как обычно, добавление дополнительного уровня посредничества[5] в форме SafeHandle позволило смягчить остроту этой проблемы в .NET 2.0.

Внимание! Прежде чем решить, что вам нужно создавать наследника SafeHandle, поищите среди наследников SafeHandle, определенных в .NET Framework, — нет ли там подходящего для вас. Например, если вы разрабатываете код для прямого обращения к драйверу устройства вызовом Win32-функции DeviceIoControl через P/Invoke, то типа SafeHandle вполне достаточно для хранения дескриптора, которым вы открываете драйвер напрямую.

При создании вашего собственного класса-наследника SafeHandle вы должны выполнить короткую последовательность шагов. В качестве примера давайте создадим производный от SafeHandle класс — SafeBluetoothRadioFindHandle, чтобы можно было перечислить подключенные к системе устройства Bluetooth,

[5] Авторитет в C++ Эндрю Кениг (Andrew Koenig) любит называть это теоремой программной инженерии, т.е. любая проблема программной инженерии может быть решена добавлением уровня посредничества.

если таковые есть. Шаблон перечисления устройств Bluetooth в "родном" коде довольно прост и распространен в Win32 API. Вы вызываете Win32-функцию BluetoothFindFirstRadio, и в случае успеха, она возвращает дескриптор первого устройства через выходной параметр и дескриптор перечисления — через возвращаемое значение. Вы можете затем получить любое дополнительное устройство вызовом Win32-функции BluetoothFindNextRadio. По завершении работы вы должны вызвать Win32-функцию BluetoothFindRadioClose на дескрипторе перечисления. Рассмотрим приведенный ниже код:

```
using System;
using System.Runtime.InteropServices;
using System.Runtime.ConstrainedExecution;
using System.Security;
using System.Security.Permissions;
using System.Text;
using Microsoft.Win32.SafeHandles;
//
// Соответствует Win32 BLUETOOTH_FIND_RADIO_PARAMS
//
[StructLayout( LayoutKind.Sequential )]
class BluetoothFindRadioParams
{
    public BluetoothFindRadioParams() {
        dwSize = 4;
    }
    public UInt32 dwSize;
}
//
// Соответствует Win32 BLUETOOTH_RADIO_INFO
//
[StructLayout( LayoutKind.Sequential, CharSet = CharSet.Unicode )]
struct BluetoothRadioInfo
{
    public const int BLUETOOTH_MAX_NAME_SIZE = 248;
    public UInt32 dwSize;
    public UInt64 address;
    [MarshalAs( UnmanagedType.ByValTStr,
                SizeConst = BLUETOOTH_MAX_NAME_SIZE )]
    public string szName;
    public UInt32 ulClassOfDevice;
    public UInt16 lmpSubversion;
    public UInt16 manufacturer;
}
//
// Безопасный обработчик перечисления устройств Bluetooth
//
[SecurityPermission( SecurityAction.Demand, UnmanagedCode = true )]
sealed public class SafeBluetoothRadioFindHandle
    : SafeHandleZeroOrMinusOneIsInvalid
{
    private SafeBluetoothRadioFindHandle() : base( true ) { }
```

```
        override protected bool ReleaseHandle() {
            return BluetoothFindRadioClose( handle );
        }
        [DllImport( "Irprops.cpl" )]
        [ReliabilityContract( Consistency.WillNotCorruptState,
                              Cer.Success )]
        [SuppressUnmanagedCodeSecurity]
        private static extern bool BluetoothFindRadioClose(IntPtr hFind );
    }
    public class EntryPoint
    {
        private const int ERROR_SUCCESS = 0;
        static void Main() {
            SafeFileHandle radioHandle;
            using( SafeBluetoothRadioFindHandle radioFindHandle
               = BluetoothFindFirstRadio(new BluetoothFindRadioParams(),
                                          out radioHandle) ) {
               if( !radioFindHandle.IsInvalid ) {
                   BluetoothRadioInfo radioInfo = new
                   BluetoothRadioInfo();
                   radioInfo.dwSize = 520;
                   UInt32 result = BluetoothGetRadioInfo( radioHandle,
                                                  ref radioInfo );
                   if( result == ERROR_SUCCESS ) {
                      // Пошлем содержимое радиоинформации
                      // на консоль.
                      Console.WriteLine( "address = {0:X}",
                                          radioInfo.address );
                      Console.WriteLine( "szName = {0}",
                                          radioInfo.szName );
                      Console.WriteLine( "ulClassOfDevice = {0}",
                                          radioInfo.ulClassOfDevice );
                      Console.WriteLine( "lmpSubversion = {0}",
                                          radioInfo.lmpSubversion );
                      Console.WriteLine( "manufacturer = {0}",
                                          radioInfo.manufacturer );
                   }
                   radioHandle.Dispose();
               }
            }
        }
        [DllImport( "Irprops.cpl" )]
        private static extern SafeBluetoothRadioFindHandle
            BluetoothFindFirstRadio( [MarshalAs(UnmanagedType.LPStruct)]
                                     BluetoothFindRadioParams pbtfrp,
                                     out SafeFileHandle phRadio );
        [DllImport( "Irprops.cpl" )]
        private static extern UInt32
            BluetoothGetRadioInfo( SafeFileHandle hRadio,
                                   ref BluetoothRadioInfo pRadioInfo );
    }
```

Главная проблема этого примера — `SafeBluetoothRadioFindHandle`. Вы можете наследовать его непосредственно от `SafeHandle`, но исполняющая система предлагает два вспомогательных класса — `SafeHandleZeroOrMinusOneIsInvalid` и `SafeHandleMinusOneIsInvalid` — от которых наследоваться удобнее.

Внимание! Будьте осторожны, имея дело с функциями Win32 через P/Invoke, и всегда тщательно читайте документацию, чтобы знать, чему равно неверное значение дескриптора. Интерфейс Win32 API известен путаницей в этом вопросе. Например, Win32-функция `CreateFile` возвращает `-1`, чтобы обозначить неудачу. Функция `CreateEvent` возвращает дескриптор `NULL` в случае ошибки. В обоих случаях типом возвращаемого значения является `HANDLE`.

При разработке собственного наследника `SafeHandle` нужно помнить о нескольких вещах.

- *Применяйте требование безопасного доступа к классу, которому придется вызывать неуправляемый код.* Конечно, вы не обязаны делать этого, если в действительности не вызываете неуправляемого кода, но вероятность создания наследника `SafeHandle` без вызова неуправляемого кода весьма невелика.
- *Предусматривайте конструктор по умолчанию, который инициализирует наследника `SafeHandle`.* Обратите внимание, что `SafeBluetoothRadioFindHandle` объявляет приватный конструктор по умолчанию. Поскольку слой P/Invoke обладает особыми полномочиями, он может создавать экземпляры объекта даже в случае приватного конструктора. Приватный конструктор удерживает клиентов от создания экземпляров без вызова функций Win32, создающих лежащий в основе ресурс.
- *Переопределяйте виртуальное свойство* `IsInvalid`. В данном случае в этом не было необходимости, поскольку базовый класс `SafeHandleZeroOrMinusOneIsInvalid` делает это за вас.
- *Переопределяйте виртуальный метод* `ReleaseHandle`, *используемый для очистки ресурса.* Обычно в нем вы осуществляете вызов P/Invoke для освобождения неуправляемого ресурса. В данном примере вызывается `BluetoothFindRadioClose`. Обратите внимание, что при объявлении метода для вызова P/Invoke вы применяете контракт достоверности, поскольку метод `ReleaseHandle` вызывается в контексте CER. Вдобавок разумно применить к методу атрибут `SuppressUnmanagedCodeSecurityAttribute`.

Определив наследника `SafeHandle`, вы готовы использовать его через объявления P/Invoke. В предыдущем примере вы объявляли метод `BluetoothFindFirstRadio` для вызова через P/Invoke. Если вы посмотрите описание этой функции в Microsoft Developer Network (MSDN), то увидите, что она возвращает тип `BLUETOOTH_RADIO_FIND`, который представляет собой дескриптор внутреннего объекта перечисления радиоустройств. Во времена .NET 1.1 вам нужно было бы объявить тип возврата метода как `IntPtr`. Начиная с .NET 2.0, вы указываете в качестве типа его возврата `SafeBluetoothRadioFindHandle`, а слой маршализации P/Invoke делает остальное. Теперь дескриптор перечисления защищен от утечки исполняющей системой на случай каких-либо асинхронных исключений, исходящих от виртуальной исполняющей системы.

Внимание! При маршализации между методом COM или функцией Win32, возвращающей дескриптор в структуре, промежуточный слой не предоставляет поддержки работы с наследниками SafeHandle. В таких редких случаях вам нужно вызывать метод SetHandle наследника SafeHandle после получения структуры от функции или метода COM. Однако если вам придется так поступать, вы захотите убедиться, что операции, создающие дескриптор, и последующий вызов SetHandle происходят внутри CER, чтобы ничто не могло прервать процесс выделения ресурса и присвоения дескриптора объекту SafeHandle. В противном случае вы можете столкнуться с утечкой ресурсов.

Создание пользовательских классов исключений

У System.Exception есть три общедоступных конструктора и один защищенный. Первый — конструктор по умолчанию, который на самом деле мало что делает. Второй — конструктор, принимающий ссылку на строковый объект. Строка представляет собой общее, определяемое программистом сообщение, которое вы можете рассматривать как более дружественное к пользователю описание исключения. Третий конструктор также принимает строку сообщения, как и второй, но вдобавок принимает ссылку на другой объект Exception. Ссылка на другое исключение позволяет вам отслеживать исходные исключения, когда одно исключение транслируется в другое внутри блока try.

Хорошим примером может служить ситуация, когда исключение не обрабатывается, а просачивается вверх, во фрейм стека статического конструктора. В этом случае исполняющая система генерирует TypeInitializationException, но только после установки внутреннего исключения в оригинальное исключение, чтобы тот, кто перехватывает TypeInitializationException, по крайней мере, знал, чем вызвано исключение изначально. И, наконец, защищенный конструктор позволяет создавать исключение из объекта SerializationInfo. Вы всегда захотите создавать сериализуемые исключения, чтобы можно было использовать через границы контекстов, например, с .NET Remoting. Это значит, что вам также нужно будет пометить свои пользовательские классы исключений атрибутом SerializableAttribute.

Класс System.Exception очень полезен с этими тремя общедоступными конструкторами. Однако следует считать плохим дизайном простую генерацию объектов типа System.Exception всякий раз, когда в программе что-то идет не так. Вместо этого имеет смысл создать новый, более специфичный тип исключения, наследуя его от System.Exception. Таким образом, тип исключения будет более выразительным в описании проблемы, его вызвавшей. Еще лучше то, что ваш производный класс может содержать данные, которые соответствуют причине генерации данного исключения. И помните, что в C# все исключения должны наследоваться от System.Exception. Давайте посмотрим, что нужно сделать, чтобы эффективно определять пользовательские исключения.

Возьмем предыдущий пример EmployeeDatabase. Предположим, что для того, чтобы добавить сотрудника в базу данных, его данные нужно проверить. Если данные сотрудника по каким-то причинам не подойдут, метод Add сгенерирует исключение типа EmployeeVerificationException. Обратите внимание, что имя нового типа исключения заканчивается на Exception. Полезно выработать у себя такую привычку, поскольку это общепринятое соглашение, позволяющее легко находить

типы исключений внутри вашей системы типов. Это также считается хорошим тоном среди сообщества программистов на C#. Посмотрим, как может выглядеть такое исключение:

```
using System;
using System.Runtime.Serialization;
[Serializable()]
public class EmployeeVerificationException : Exception
{
    public enum Cause {
        InvalidSSN,
        InvalidBirthDate
    }
    public EmployeeVerificationException( Cause reason )
        :base() {
        this.reason = reason;
    }
    public EmployeeVerificationException( Cause reason,
                                          String msg )
        :base( msg ) {
        this.reason = reason;
    }
    public EmployeeVerificationException( Cause reason,
                                          String msg,
        Exception inner )
        :base( msg, inner ) {
        this.reason = reason;
    }
    protected EmployeeVerificationException(
           SerializationInfo info,
           StreamingContext context )
        :base( info, context ) { }
    private Cause reason;
    public Cause Reason {
        get {
           return reason;
        }
    }
}
```

В методе `EmployeeDatabase.Add` вы можете видеть пример вызова `Validate` на объекте `emp`. Это довольно сырой пример, где вы заставляете проверку завершиться неудачей, сгенерировав `EmployeeVerificationException`. Но главная цель этого примера — создание нового типа исключения. Вы не раз еще убедитесь в том, что просто создания нового типа исключения достаточно для доставки дополнительной информации. В данном случае я хотел проиллюстрировать пример, где тип исключения несет в себе больше информации о причинах неудачи проверки, поэтому создал свойство `Reason`, чье поле заднего плана должно быть инициализировано в конструкторе. К тому же заметьте, что `EmployeeVerificationException` наследуется от `System.Exception`. Поначалу существовало представление, что все определенные в .NET Framework типы исключений должны наследоваться от `System.Exception`,

в то время как определяемые пользователями исключения должны наследоваться от ApplicationException, чтобы их можно было легко отличить друг от друга. Эта цель частично была утеряна из-за того, что некоторые исключения, определенные в .NET Framework, наследуются от ApplicationException[6].

Вы можете удивиться, зачем я определил целых четыре конструктора для такого простого типа исключений. Дело в том, что при определении новых типов исключений действует традиционная идиома, требующая определения тех же самых четырех общедоступных конструкторов, которые есть в System.Exception. Если бы я не собирался передавать дополнительные данные о причине исключения, то конструкторы EmployeeVerificationException в точности соответствовали по форме конструкторам System.Exception. Если вы последуете этой идиоме при создании собственных типов исключений, то пользователи смогут трактовать ваш тип исключения точно таким же образом, как любое другое определенное системой исключение. Плюс к этому ваше унаследованное исключение сможет использовать на сообщения и внутреннее исключение, уже инкапсулированные в System.Exception.

Работа с выделенными ресурсами и исключениями

Если вы — опытный профессионал в C++, то одна вещь в мире C# покажется вам наиболее захватывающей — это отсутствие детерминированной деструкции. Разработчики C++ привыкли использовать конструкторы и деструкторы объектов, располагаемых в стеке для управления ценными ресурсами. Эта идиома даже имеет собственное название — RAII (*Resource Acquisition Is Initialization* — захват ресурсов является инициализацией). Это означает, что вы можете создавать объекты в стеке C++, в которых некоторый ценный ресурс выделяется в конструкторе, и если он освобождается в деструкторе, то вы можете положиться на его автоматический вызов для очистки в нужное время. Например, независимо от того, как объекты, расположенные в стеке, выходят из области видимости — через нормальное выполнение при достижении конца контекста либо по исключению — вы всегда имеете гарантию, что деструктор будет выполнен, освободив при этом ценный ресурс.

Когда C# и CLR впервые были представлены разработчикам в виде бета-версии, многие немедленно подняли шум об этом недостатке в исполняющей системе. Считаете вы это недостатком или нет, это явно не касается наиболее полного расширения, которое появилось после замечаний сообщества бета-разработчиков. Проблема происходила отчасти из природы объектов CLR, которыми заведует сборщик мусора, наряду с тем фактом, что дружественный деструктор в синтаксисе C# был повторно использован для реализации финализаторов объекта. Важно также помнить, что финализаторы очень сильно отличаются от деструкторов. Использование синтаксиса деструктора для финализаторов только добавило путаницы. Были также и другие технические причины, некоторые касающиеся эф-

[6] Подробнее об этом, наряду со многими другими полезными рекомендациями, написано в руководстве Кшиштофа Квалины (Krzysztof Cwalina) и Брэда Абрамса (Brad Abrams) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Boston, MA: Addison-Wesley Professional, 2005 г.).

фективности, почему детерминированные деструкторы, как мы их знаем, не были включены в исполняющую систему.

После некоторых размышлений было предложено решение в виде шаблона Disposable, используемого посредством реализации интерфейса IDisposable. За более детальной информацией относительно шаблона Disposable и ваших объектов обращайтесь к главам 4 и 13. По сути, если ваш объект нуждается в детерминированной деструкции, он получает ее, реализуя интерфейс IDisposable. Однако вы должны явно вызывать ваш метод Dispose, чтобы выполнить "уборку" за одноразовым объектом. Если вы забудете это сделать, и ваш объект закодирован правильно, то ресурс не будет утерян — он будет просто очищен тогда, когда сборщик мусора, наконец, вызовет ваш финализатор. Внутри C++ вы должны только не забыть поместить код очистки в деструктор, и вам никогда не придется заботиться об очистке ваших локальных объектов, поскольку такая очистка происходит автоматически при выходе из области определения.

Рассмотрим следующий надуманный пример, иллюстрирующий опасность, с которой вы можете столкнуться:

```
using System;
using System.IO;
using System.Text;
public class EntryPoint
{
    public static void DoSomeStuff() {
        // Открыть файл.
        FileStream fs = File.Open( "log.txt",
                                   FileMode.Append,
                                   FileAccess.Write,
                                   FileShare.None );
        Byte[] msg = new UTF8Encoding(true).GetBytes("Doing Some"+
                                                     " Stuff");
        fs.Write( msg, 0, msg.Length );
    }

    public static void DoSomeMoreStuff() {
        // Открыть файл.
        FileStream fs = File.Open( "log.txt",
                                   FileMode.Append,
                                   FileAccess.Write,
                                   FileShare.None );
        Byte[] msg = new UTF8Encoding(true).GetBytes("Doing Some"+
                                                     " More Stuff");
        fs.Write( msg, 0, msg.Length );
    }
    static void Main() {
        DoSomeStuff();
        DoSomeMoreStuff();
    }
}
```

Этот код выглядит довольно невинно. Однако если вы запустите его, то почти наверняка столкнетесь с исключением IOException. Код в DoSomeStuff созда-

ет объект FileStream с исключительной блокировкой файла. Как только объект FileStream выходит из контекста в конце функции, он помечается для уборки, но когда именно она произойдет, приходится полагаться на сборщик мусора (GC). Таким образом, когда следующий раз вы пытаетесь открыть файл в DoSomeMoreStuff, то получаете исключение, поскольку ценный ресурс все еще заблокирован недоступным объектом FileStream. Ясно, что это весьма неприятная ситуация. Даже не думайте о явном вызове GC.Collect в Main перед вызовом DoSomeMoreStuff. Попытка управлять алгоритмом работы CG, чтобы заставить его подбирать объекты в определенное время — путь к снижению производительности. Вы не можете помочь GC лучше выполнять его работу, поскольку не имеете представления о том, как он реализован.

Как же поступить? Так или иначе, вы должны гарантировать закрытие файла. Однако здесь есть шероховатость: независимо от того, как вы делаете это, вы должны помнить о том, что это нужно сделать. В этом отличие от C++, где вы можете поместить код очистки в деструктор и потом просто быть уверенным, что ресурс будет очищен в надлежащий момент. Одним из вариантов может быть вызов метода Close объекта FileStream в каждом методе, использующем его. Это работает хорошо, но степень автоматизации ниже, и вы всегда должны помнить, что это нужно сделать. Однако даже если вы не забудете делать это, что случится, если будет сгенерировано исключение до того, как будет вызван метод Close? Вы окажетесь в той же ситуации, что и раньше, с утерянным ресурсом, к которому невозможно обратиться даже для того, чтобы освободить его.

Те из вас, кто знаком с обработкой исключений, отметят, что эту проблему можно решить, используя блоки try/finally, как в следующем примере:

```
using System;
using System.IO;
using System.Text;
public class EntryPoint
{
    public static void DoSomeStuff() {
        // Открыть файл.
        FileStream fs = null;
        try {
           fs = File.Open( "log.txt",
                           FileMode.Append,
                           FileAccess.Write,
                           FileShare.None );
           Byte[] msg =
                 new UTF8Encoding(true).GetBytes("Doing Some"+
                                                 " Stuff\n");
           fs.Write( msg, 0, msg.Length );
        }
        finally {
           if( fs != null ) {
               fs.Close();
           }
        }
    }
```

```
    public static void DoSomeMoreStuff() {
        // Открыть файл.
        FileStream fs = null;
        try {
           fs = File.Open( "log.txt",
                           FileMode.Append,
                           FileAccess.Write,
                           FileShare.None );
           Byte[] msg =
                  new UTF8Encoding(true).GetBytes("Doing Some"+
                                                  " More Stuff\n");
           fs.Write( msg, 0, msg.Length );
        }
        finally {
           if( fs != null ) {
               fs.Close();
           }
        }
    }
    static void Main() {
        DoSomeStuff();
        DoSomeMoreStuff();
    }
}
```

Блоки `try/finally` решают проблему. Но насколько уродливым стал код! Плюс к тому, давайте признаем, то многие из нас довольно ленивы, а здесь — огромный объем дополнительного клавиатурного набора. Более того, больше текста означает больше возможностей внести ошибку. И, наконец, это затрудняет чтение кода. Как можно было ожидать, есть путь получше. Многие объекты, такие как `FileStream`, имеющие метод `Close`, также реализуют шаблон `IDisposable`. Обычно вызов `Dispose` на этих объектах — это то же самое, что вызов `Close`. Конечно, вызывать `Close` через `Dispose` или наоборот — это спор о яблоках и апельсинах (что из них лучше), если вы все равно должны явно вызывать один или другой. К счастью, есть хорошая причина того, почему большинство классов, имеющих метод `Close`, реализуют `Dispose` — так вы можете эффективно применять их в операторе `using`, который обычно применяется как часть шаблона Disposable в C#. Поэтому вы должны изменить код следующим образом:

```
using System;
using System.IO;
using System.Text;
public class EntryPoint
{
    public static void DoSomeStuff() {
        // Открыть файл.
        using( FileStream fs = File.Open( "log.txt",
                                          FileMode.Append,
                                          FileAccess.Write,
                                          FileShare.None ) ) {
```

```
            Byte[] msg =
            new UTF8Encoding(true).GetBytes("Doing Some" +
                                            " Stuff\n");
            fs.Write( msg, 0, msg.Length );
        }
    }
    public static void DoSomeMoreStuff() {
        // Открыть файл.
        using( FileStream fs = File.Open( "log.txt",
                                          FileMode.Append,
                                          FileAccess.Write,
                                          FileShare.None ) ) {
            Byte[] msg =
            new UTF8Encoding(true).GetBytes("Doing Some" +
                                            " More Stuff\n");
            fs.Write( msg, 0, msg.Length );
        }
    }
    static void Main() {
        DoSomeStuff();
        DoSomeMoreStuff();
    }
}
```

Как видите, этот код гораздо легче понять, и оператор using позаботится обо всем, что связано с блоками try/finally. Возможно, вы не удивитесь, просмотрев сгенерированный код в ILDASM и увидев там, что компилятор генерирует блоки try/finally вместо оператора using. Вы можете даже вкладывать конструкции using одну внутрь другой — точно так же, как это можно делать с блоками try/finally.

Но даже несмотря на то, что оператор using устраняет симптом "уродливого кода" и сокращает шансы появления дополнительных ошибок, он все-таки требует от вас помнить о необходимости его применения. Это не так удобно, как детерминированная деструкция локальных объектов в C++, но это лучше, чем оснащение вашего кода блоками try/finally, и определенно, это лучше, чем ничего. В результате мы получаем в C# некоторую форму детерминированной деструкции через оператор using, но она остается детерминированной, только если вы не забываете сделать ее таковой.

Обеспечение поведения отката

Разрабатывая нейтральные к исключениям методы, как было описано в разделе "Обеспечение нейтральности к исключениям" настоящей главы, часто вы сочтете удобным механизм, который может выполнять откат любых изменений в случае генерации исключения. Эту проблему можно решить, применив классическую технику ввода дополнительного промежуточного слоя в виде вспомогательного класса. Для иллюстрации давайте используем объект, представляющий соединение с базой данных, и включающий методы по имени Commit (фиксация) и Rollback (откат).

В мире C++ популярное решение этой проблемы подразумевает разработку вспомогательного класса, экземпляр которого создается в стеке. Вспомогательный

класс также имеет метод Commit. При его вызове он просто прогоняет метод объекта базы данных, но перед этим выставляет внутренний флаг. Трюк кроется в деструкторе. Если деструктор выполняется перед тем, как флаг выставлен, есть только две возможности. Первая — когда пользователь просто забыл вызвать Commit. Поскольку это означает ошибку в коде, такую возможность рассматривать не будем. Вторая возможность попасть в деструктор при не выставленном флаге — очистка объекта в процессе раскрутки стека в поисках обработчика исключения. В зависимости от состояния флага в коде деструктора вы знаете, попали вы сюда в результате нормального выполнения или же через исключение. Если вы попали через исключение, все, что вам нужно сделать — это вызвать Rollback на объекте базы данных, и вы получите необходимую функциональность.

Все это прекрасно в мире C++, где вы можете использовать детерминированную деструкцию. Однако того же конечного результата можно достичь с применением формы детерминированной деструкции C#, представляющей собой сочетание IDisposable и ключевого слова using. Напомню, что деструктор в "родном" C++ отображается на интерфейс IDisposable в C#. Все, что вы должны сделать — взять код, который поместили бы в деструктор в C++, и включить его в метод Dispose вспомогательного класса C#. Посмотрим, как может выглядеть такой вспомогательный класс C#:

```
using System;
using System.Diagnostics;
public class Database
{
    public void Commit() {
        Console.WriteLine( "Изменения зафиксированы" );
    }
    public void Rollback() {
        Console.WriteLine( "Изменения отменены" );
    }
}
public class RollbackHelper : IDisposable
{
    public RollbackHelper( Database db ) {
        this.db = db;
    }
    ~RollbackHelper() {
        Dispose( false );
    }
    public void Dispose() {
        Dispose( true );
    }
    public void Commit() {
        db.Commit();
        committed = true;
    }
    private void Dispose( bool disposing ) {
        // Если объект уже disposed, не делать ничего. Помните, что
        // совершенно законно вызывать Dispose() несколько раз на
        // одном объекте.
```

```
        if( !disposed ) {
            // Помните, что мы не хотим ничего делать с db,
            // если попали сюда из финализатора, поскольку
            // поле базы данных может быть уже финализированным!
            // Однако мы хотим освободить неуправляемые ресурсы.
            // В данном случае их нет.
            if( disposing ) {
                if( !committed ) {
                    db.Rollback();
                }
            } else {
                Debug.Assert( false, "Сбой при вызове Dispose()" +
                                     " на RollbackHelper" );
            }
        }
    }
    private Database db;
    private bool disposed = false;
    private bool committed = false;
}

public class EntryPoint
{
    static private void DoSomeWork() {
        using( RollbackHelper guard = new RollbackHelper(db) ) {
            // Здесь выполняем некоторую работу, которая может
            // сгенерировать исключение.
            // Удалите комментарий со следующей строки,
            // чтобы сгенерировать исключение:
            // nullPtr.GetType();
            // Если добрались сюда, фиксируем.
            guard.Commit();
        }
    }
    static void Main() {
        db = new Database();
        DoSomeWork();
    }
    static private Database db;
    static private Object nullPtr = null;
}
```

Внутри метода DoSomeWork выполняется некоторая работа, которая может дать сбой с исключением. Если случится исключение, нам нужно, чтобы все изменения, проведенные в объекте Database, были отменены. Внутри блока using вы создаете новый объект RollbackHelper, содержащий ссылку на объект Database. Если поток управления достигает точки вызова Commit на ссылке guard, значит, все хорошо, если предположить, что метод Commit не сгенерирует исключений. Даже если это случится, вы должны кодировать его таким образом, чтобы Database оставалась в корректном состоянии. Однако если ваш код внутри защищенного блока

сгенерирует исключение, то метод Dispose на RollbackHelper аккуратно откатит вашу базу данных.

Независимо от того, что случилось, метод Dispose будет вызван на экземпляре RollbackHelper, благодаря блоку using. Если вы забудете применить блок using, то финализатор RollbackHelper не сможет ничего для вас сделать, поскольку финализация объектов происходит в случайном порядке, и Database, на которую ссылается RollbackHelper, может быть финализирована до экземпляра RollbackHelper. Чтобы помочь вам найти место, где вы ошиблись, можно закодировать утверждение (assertion) во вспомогательном (helper) объекте, как это выше сделал я. Поскольку применение всего шаблона держится на блоке using, предположим, что вы не забыли о нем.

Как только поток управления благополучно достигнет метода Dispose и попадет туда через явный вызов Dispose, а не через финализатор, он просто проверит флаг фиксации, и если таковой не установлен, вызовет Rollback на экземпляре Database. Вот и все. Получается почти столь же элегантно, как в решении C++, за исключением того, что как мы уже ранее говорили в этой главе, вы должны не забыть о применении ключевого слова using, чтобы вся эта кухня работала. Если вы хотите посмотреть, что произойдет в случае генерации исключения, просто удалите комментарий со строки с попыткой обращения к нулевой ссылке внутри метода DoSomeWork.

Возможно, вы заметили, что я не беспокоюсь о том, что случится, если Rollback сгенерирует исключение. Ясно, что для устойчивости кода оптимально требовать, чтобы все операции, выполняемые RollbackHelper в процессе отката, были гарантированы от генерации исключений. Это возвращает нас к одному из наиболее основополагающих требований для генерации строго безопасного к исключениям и нейтрального к исключениям кода: чтобы создать устойчивый, безопасный к исключениям код, вы должны иметь четко определенный набор операций, которые гарантированно не генерируют исключений.

В мире C++ при раскрутке стека, вызванной исключением, откат происходит в деструкторе. Ветераны C++ знают, что в деструкторе никогда нельзя генерировать исключений, потому что стек находится в процессе раскрутки, и если сгенерировать исключение, он будет прерван очень грубо. А нет ничего хуже, чем исчезновение приложения без каких-либо следов в виде трассировки. Но что, если такое случится в C#? Напомню, что блок using разворачивается в конструкцию try/finally. Вы можете вспомнить, что когда исключение генерируется в блоке finally, который выполняется в результате предыдущего исключения, то предыдущее исключение просто теряется. Что еще хуже, так это то, что исполнявшийся перед этим блок finally не доходит до своего конца. Поэтому с учетом того, что утеря информации об исключении — это всегда плохо, и очень затрудняет поиск причин проблемы, настоятельно рекомендуется никогда не генерировать исключений внутри блока finally. Я уже говорил об этом ранее в этой главе, но никогда не будет лишним повториться. CLR не прервет ваше приложение, но приложение, вероятно, окажется в неопределенном состоянии, если сгенерировать исключение во время выполнения блока finally, и вы будете недоумевать, каким образом попали в такое дурацкое состояние.

Резюме

В этой главе я раскрыл основы обработки исключений, наряду с объяснением применения шаблона Expert для определения наилучшего места обработки конкретного исключения. Я коснулся отличий между .NET 1.1 и более поздними версиями CLR в отношении обращения с необработанными исключениями, а также рассказал, как .NET 2.0 и более поздние версии справляются с ними в более согласованной манере. Основная часть этой главы была посвящена описанию приемов создания "пуленепробиваемого" безопасного к исключениям кода, который гарантирует стабильность системы перед лицом неожиданных исключительных ситуаций. Также я описал ограниченные области выполнения, которые вы можете применять для того, чтобы откладывать асинхронные исключения во время прерывания потока. Создание устойчивого, безопасного и нейтрального к исключениям кода — непростая задача. К сожалению, огромное большинство программных систем, существующих сегодня, просто полностью игнорируют эту проблему. Это совершенно удручающая ситуация, учитывая богатство информации на эту тему, которая появилась с тех пор, как обработка исключений была добавлена в C++.

К сожалению, многих программистов безопасность исключений интересует в последнюю очередь. Они ошибочно предполагают, что могут справиться с любой проблемой исключений во время тестирования, разбрасывая операторы `try` по всему коду. В действительности безопасность исключений — насущная проблема, которую нужно принимать во внимание еще на этапе проектирования программного обеспечения. Пренебрежение этим приводит к появлению нестандартных систем, которые не приносят пользователям ничего, кроме разочарования и потери рынков в пользу тех компаний, разработчики которых тратят немного больше времени на обеспечение безопасности исключений. Более того, по мере того, как компьютеры вторгаются в повседневную жизнь все большего и большего количества людей, возрастает вероятность того, что появятся государственные нормативные акты, требующие строгого тестирования систем, чтобы доказать, что общество может на них положиться. Не думайте, что вас это не коснется. Я могу представить ситуацию, когда какое-нибудь социалистическое правительство может установить такие правила вообще для всего продаваемого в стране коммерческого программного обеспечения (о, ужас!). Вы слыхали когда-нибудь истории о том, например, как интегрированная система управления воздушным движением целой страны или континента была остановлена из-за программной ошибки? Разве не возненавидите вы разработчика, который пренебрег безопасностью исключений, что послужило причиной возникновения такой ситуации?

В следующей главе я раскрою основные аспекты работы со строками в C# и .NET Framework. Вдобавок я подниму важнейшую тему глобализации.

ГЛАВА 8

Работа со строками

Тип `System.String` — почетный "гражданин" базовой библиотеки классов .NET Framework. Он представляет собой идеальный пример того, как можно создать неизменяемый ссылочный тип, который ведет себя как тип значения.

Обзор String

Экземпляры `String` являются неизменяемыми в том смысле, что, однажды создав их, вы не можете их изменять. Хотя поначалу это может показаться неэффективным, такой подход на самом деле повышает эффективность кода. Когда вы активно копируете экземпляры `String` внутри приложения, вы создаете новые экземпляры, указывающие на ту же строковую информацию, что и исходный экземпляр. Даже если вы вызываете на строке метод `ICloneable.Clone`, то получаете при этом экземпляр, указывающий на те же строковые данные, что и источник. Это совершенно безопасно, потому что общедоступный интерфейс `String` не предоставляет никакой возможности модифицировать действительные данные `String`.

Конечно, вы можете обмануть систему, применив небезопасный код, но надеюсь, вы не станете этого делать. Фактически, если вам нужна строка, представляющая собой "глубокую копию" исходной строки, то для этого вы можете вызвать метод `Copy`.

На заметку! Те из вас, кто знаком с общими шаблонами и идиомами проектирования, могут распознать здесь идиому "handle/body" (дескриптор/тело) или "envelope/letter" (конверт/письмо). В C++ обычно вы реализуете эту идиому при проектировании типов, основанных на ссылках, которые вы можете передавать по значению. Многие реализации стандартной библиотеки C++ реализуют стандартные строки именно таким образом. Однако благодаря управляемой сборщиком мусора куче C#, вам не нужно беспокоиться о поддержке счетчика ссылок на лежащие в основе данные.

Во многих средах, среди которых C++ и C, строка обычно вообще не является встроенным типом, а скорее более примитивной, сырой конструкцией, такой как указатель на первый элемент в массиве символов. Обычно процедуры манипуляции со строками не являются частью языка, вместо этого они — часть библиотеки, используемой вместе с языком. Хотя это почти верно и для C#, ситуацию немного затеняет исполняющая система .NET. Проектировщики спецификации CLI могли бы представить строки в виде простых массивов типа `System.Char`, но они предпочли вместо этого включить в коллекцию встроенных типов `System.String`. Фактически `System.String` стоит особняком в коллекции встроенных типов, по-

скольку является типом ссылочным, а большинство встроенных типов — типы значений. Однако это отличие нивелируется тем фактом, что тип String ведет себя в соответствие с семантикой значений.

Возможно, вы уже знаете, что тип System.String представляет строку символов Unicode, а System.Char — 16-битный символ Unicode. Конечно, это облегчает локализацию и переносимость на другие операционные системы. Однако иногда вам может понадобиться интерфейс с внешними системами, в которых используется кодирование строк, отличное от Unicode. Для таких случаев вы можете применить класс System.Text.Encoding, чтобы преобразовывать строки между разными системами кодирования, включая ASCII, UTF-7, UTF-8 и UTF-32. Кстати, исполняющая система использует внутри себя для представления Unicode формат UTF-16[1].

Строковые литералы

Когда вы объявляете строку в вашем коде C#, то компилятор создает объект System.String, который затем помещает во внутреннюю таблицу модуля, именуемого *внутренним пулом*. Идея заключается в том, что всякий раз, когда вы объявляете в своем коде новый строковый литерал, то компилятор сначала проверяет, не объявили ли вы его ранее где-нибудь еще, и если это так, то код просто ссылается на уже имеющийся литерал. Взглянем на пример объявления строкового литерала в C#:

```
using System;
public class EntryPoint
{
    static void Main( string[] args ) {
        string lit1 = "c:\\windows\\system32";
        string lit2 = @"c:\windows\system32";
        string lit3 = @"
Jack and Jill
Went up the hill...
";
        Console.WriteLine( lit3 );
        Console.WriteLine( "Object.RefEq(lit1, lit2): {0}",
                           Object.ReferenceEquals(lit1, lit2) );
        if( args.Length > 0 ) {
            Console.WriteLine( "Полученный параметр: {0}",
                               args[0] );
            string strNew = String.Intern( args[0] );
            Console.WriteLine( "Object.RefEq(lit1, strNew): {0}",
                               Object.ReferenceEquals(lit1, strNew) );
        }
    }
}
```

Во-первых, обратите внимание на объявление двух литеральных строк — lit1 и lit2. Объявленный тип — string, который в C# является псевдонимом для System.String. Первый экземпляр инициализируется обычным строковым литералом, который может содержать знакомые управляющие последовательности, ис-

[1] Подробнее о стандарте Unicode можно почитать на сайте www.unicode.org.

пользуемые в C и C++, такие как \t и \n. Поэтому вы должны защищать сам символ обратного слэша, как обычно, удваивая его. Дополнительную информацию о допустимых управляющих последовательностях вы можете найти в документации MSDN. Однако C# представляет особый тип объявления строки, называемый *дословными строками* (verbatim strings), где все, что находится внутри объявления строки, помещается в нее "как есть". Такое объявление предваряется символом @, как показано в примере. Следует обратить особое внимание на тот факт, что странное объявление lit3 совершенно корректно. Переносы строки внутри кода трактуются как части строки, что доказывает вывод этой программы. Дословные строки могут быть удобны в тех случаях, когда вы создаете строки для подтверждения форм, и вам нужно специальным образом компоновать их внутри кода. Единственная управляющая последовательность, которая допустима внутри дословных строк — это "", и она применяется для вставки в строку символа двойной кавычки.

Ясно, что lit1 и lit2 содержат строки с одним и тем же значением, даже несмотря на то, что вы объявляете их по-разному. Учитывая то, что я сказал в предыдущем разделе, вы можете ожидать, что эти два экземпляра будут ссылаться на один и тот же строковый объект. Фактически так оно и есть, что подтверждает вывод программы, где я проверяю их с помощью Object.ReferenceEquals.

И, наконец, этот пример демонстрирует использование статического метода String.Intern. Иногда может понадобиться, находится ли уже объявленная вами строка во внутреннем пуле. Если да, может оказаться более эффективным сослаться на нее, чем создавать новый экземпляр. Код принимает строку из командной строки и затем создает новый экземпляр с применением метода String.Intern. Этот метод всегда возвращает корректную ссылку на строку, но это будет либо ссылка на экземпляр строки во внутреннем пуле, либо новая копия строки с переданным значением. Если в командной строке передать "c:\windows\system32", то код выдаст следующий вывод:

```
Jack and Jill
Went up the hill...

Object.RefEq(lit1, lit2): True
Полученный параметр: c:\windows\system32
Object.RefEq(lit1, strNew): True
```

Спецификаторы формата и глобализация

Часто возникает необходимость в форматировании особым образом данных, которые приложение отображает пользователям. Например, вам может понадобиться отображать значение с плавающей точкой, представляющее некоторое вещественное значение, в экспоненциальной форме или в форме с фиксированной точкой. В форме с фиксированной точкой вам может понадобиться использовать некоторый, зависящий от культуры, символ в качестве десятичного разделителя. Традиционно обращение с подобного рода сложностями всегда было достаточно болезненным. У программистов на C для форматирования значений есть семейство функций printf, но ему недостает некоторых специфичных для локали возможностей. C++ пошел дальше и представил более устойчивый и расширяемый механизм форматирования в виде стандартных потоков ввода-вывода, но также пренебрегающий локалями. Стандартная библиотека .NET предлагает собствен-

ный мощный механизм для решения этих задач в гибкой и расширяемой манере. Однако прежде чем переходить к самой теме спецификаторов формата, рассмотрим несколько предварительных вопросов.

На заметку! Важно принимать во внимание соображения, связанные с различными культурами, еще на ранней стадии цикла разработки. Многие разработчики склонны откладывать на потом решение проблем, связанных с глобализацией. Но если вы заметили, проектировщики .NET Framework приложили массу усилий к созданию богатой библиотеки, позволяющей решать проблемы глобализации. Богатство и ширина API глобализации указывает на ее сложность. Учитывая соображения глобализации на ранней стадии разработки, вы избавляете себя от неприятностей, которые неизбежно возникнут позже.

Object.ToString, IFormattable и CultureInfo

Каждый объект наследует метод по имени ToString от System.Object. Возможно, вы уже знакомы с ним. Это исключительно удобно — получать строковое представление вашего объекта для вывода, хотя бы только для отладочных целей. Для ваших собственных специальных классов вы увидите, что реализация по умолчанию метода ToString просто возвращает тип самого объекта. Вы должны предусмотреть собственное переопределение этого метода, чтобы он отображал что-то полезное. Как и можно было ожидать, все встроенные типы делают это. Поэтому, если вы вызываете ToString на экземпляре System.Int32, то получаете строковое представление хранимого в нем значения. Но если вам понадобится строковое представление шестнадцатеричного формата, то Object.ToString вам в этом не поможет, поскольку не предусматривает возможности запросить нужный формат. Должен быть другой способ строкового представления объекта. Фактически он есть, и предусматривает реализацию интерфейса IFormattable, который выглядит следующим образом:

```
public interface IFormattable
{
    string ToString( string format, IFormatProvider formatProvider )
}
```

Вы заметите, что встроенные числовые типы, как и типы даты-времени, реализуют этот интерфейс. Используя этот метод, вы можете точно специфицировать, каким образом вы хотите форматировать значение, представив строку спецификатора формата. Прежде чем приступить к описанию того, как выглядят форматные строки, разрешите объяснить некоторые предварительные концепции, начиная со второго параметра метода IFormattable.ToString.

Объект, реализующий интерфейс IFormatProvider, является (как ни странно) поставщиком формата. Общая задача поставщика формата внутри .NET Framework — это предоставление специфичной для данной культуры форматной информации, такой как символ валюты, символ десятичного разделителя и т.д. Когда вы передаете в этом параметре null, то используемым IFormattable.ToString поставщиком формата обычно является экземпляр CultureInfo, возвращенный методом System.Globalization.CultureInfo.CurrentCulture. Этот экземпляр CultureInfo идентифицирует культуру, используемую текущим потоком. Однако у вас есть возможность переопределить ее, создав новый экземпляр CultureInfo и

передав его конструктору строку, описывающую информацию о желаемой локали, как указано в стандарте RFC 1766. Больше информации об именах культур вы найдете в документации MSDN по классу `CultureInfo`. И, наконец, вы можете даже представить нейтральный по отношению к культурам экземпляр `CultureInfo`, передав ему объект, представленный методом `CultureInfo.InvariantCulture`.

На заметку! Экземпляры `CultureInfo` используются в качестве удобного группирующего механизма форматной информации, относящейся к определенной культуре. Например, один экземпляр `CultureInfo` может представлять специфичные для культуры качества американской версии английского языка, в то время как другие — качества, специфичные для английского, на котором говорят в Соединенном Королевстве. Каждый экземпляр `CultureInfo` содержит в себе специфичные экземпляры `DateTimeFormatInfo`, `NumberFormatInfo`, `TextInfo` и `CompareInfo`, присущие представленному языку и региону.

Как только реализация `IFormattable.ToString` получает корректного поставщика формата — передан ли он явно или прикреплен к текущему потоку — она может опрашивать этого поставщика, вызывая метод `IFormatProvider.GetFormat`. Форматеры реализованы .NET Framework в типах `NumberFormatInfo` и `DateTimeFormatInfo`. Этот механизм исключительно расширяем, поскольку вы можете предоставлять собственные типы форматеров, и другие типы, которые знают, как использовать их, могут запрашивать их экземпляры у специального поставщика форматов.

Предположим, что вы хотите конвертировать числовое значение с плавающей точкой в строку. Поток выполнения реализации `IFormattable.ToString` для `System.Double` выполняет следующие основные шаги.

1. Реализация получает ссылку на тип `IFormatProvider`, который является либо переданным, либо прикрепленным к текущему потоку, если передан `null`.

2. Запрашивается поставщик формата для экземпляра типа `NumberFormatInfo` через вызов `IFormatProvider.GetFormat`. Поставщик формата инициализирует свойства экземпляра `NumberFormatInfo` на основе представленной им культуры.

3. Используется экземпляр `NumberFormatInfo` для соответствующего форматирования числа, создавая строковое его представление на основе спецификации форматной строки.

Создание и регистрация пользовательских типов `CultureInfo`

Средства глобализации .NET Framework всегда были его сильной стороной. Однако всегда оставалась возможность для усовершенствований, и большая часть этих усовершенствований появились в .NET 2.0. В частности, в .NET 1.1 всегда был болезненным процесс ввода новой описывающей культуру информации в систему, если каркасу не известна информация об этой культуре и регионе. .NET 2.0 Framework представил в пространстве имен `System.Globalization` новый класс по имени `CultureAndRegionInfoBuilder`.

Используя `CultureAndRegionInfoBuilder`, вы получаете возможность определять и вводить в систему информацию, описывающую совершенно новую культуру

и регион, а также регистрировать ее для глобального использования. Аналогично вы можете модифицировать уже существующую в системе информацию о культурах и регионах. И если это для вас недостаточно гибко, вы можете даже сериализовать информацию в файл LDML (Locale Data Markup Language — язык разметки данных локали), имеющий формат на базе XML. Как только вы зарегистрировали в системе новую культуру и регион, вы можете создавать экземпляры `CultureInfo` и `RegionInfo`, применяя строковое имя, зарегистрированное в системе.

Именую ваши описания новых культур, вы должны следовать стандартному формату именования культур. Здесь принять формат

```
[префикс-]язык[-регион][-суффикс[...]]
```

где только идентификатор `язык` является обязательной частью, а все остальные — не обязательны. Префикс может быть одним из следующих:

- `i` — для имен культур, зарегистрированных через Internet Assigned Numbers Authority (IANA);
- `x` — для всех прочих.

Вдобавок префиксная часть может быть представлена в верхнем или нижнем регистре. Часть языка представлена двухсимвольным кодом нижнего регистра, в стандарте ISO 639-1, в то время как регион описывается двухсимвольным кодом верхнего регистра, соответствующим стандарту ISO 3166. Например, русский язык, на котором говорят в России, обозначается как `ru-RU`. Компонент суффикс используется для дополнительного уточнения культуры на основе некоторых данных. Скажем, сербский, на котором говорят в Сербии, может обозначаться либо `sr-SP-Cyrl` либо `sr-SP-Latn` — один для кириллического алфавита, другой — для латинского. Если вы определяете культуру, специфичную для подразделения вашей компании, вы можете создать ее, используя имя `x-ru-RU-Компания-Подразделение`.

Чтобы убедиться, насколько легко использовать объект `CultureAndRegionInfoBuilder`, попробуем создать фиктивную культуру на базе уже существующей. В Соединенных Штатах основной системой мер является английская. Представим, что в какой-то момент США решили перейти на метрическую систему, и вам нужно модифицировать информацию о культуре на некоторых машинах, чтобы она соответствовала новым веяниям. Посмотрим, как будет выглядеть нужный код:

```
using System;
using System.Globalization;
public class EntryPoint
{
    static void Main() {
        CultureAndRegionInfoBuilder cib = null;
        cib = new CultureAndRegionInfoBuilder(
                    "x-en-US-metric",
                    CultureAndRegionModifiers.None );
        cib.LoadDataFromCultureInfo( new CultureInfo("en-US") );
        cib.LoadDataFromRegionInfo( new RegionInfo("US") );
        // Внести изменения.
        cib.IsMetric = true;
        // Создать файл LDML.
        cib.Save( "x-en-US-metric.ldml" );
```

```
        // Зарегистрировать в системе.
        cib.Register();
    }
}
```

На заметку! Чтобы скомпилировать предыдущий пример, вам понадобится специально сослаться на сборку `sysglobl.dll`. Если вы хотите собрать пример из командной строки, используйте следующую команду:

```
csc /r:sysglobl.dll example.cs
```

Как видите, процесс очень прост, поскольку `CultureAndRegionInfoBuilder` имеет продуманный интерфейс. Для иллюстрации я направил LDML в файл, чтобы вы смогли увидеть, как выглядит описание культуры, хотя оно довольно многословно, чтобы приводить его здесь. Следует отметить один важный момент: для вызова метода `Register` вы должны обладать соответствующими привилегиями. Обычно нужно, чтобы вы были администратором, хотя это требование и можно обойти, если изменить доступность к каталогу `%WINDIR%\Globalization` и к ключу реестра `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CustomLocale`. Как только вы зарегистрировали культуру в системе, вы можете обращаться к ней по имени, которое применяется для указания любой культуры в CLR. Например, чтобы убедиться, что информация о культуре и регионе зарегистрирована правильно, вы можете собрать и запустить следующий тестовый код:

```
using System;
using System.Globalization;
public class EntryPoint
{
    static void Main() {
        RegionInfo ri = new RegionInfo("x-en-US-metric");
        Console.WriteLine( ri.IsMetric );
    }
}
```

Форматные строки

Вы должны знать, как выглядят форматные строки. Встроенные числовые объекты используют стандартные строки числовых форматов или пользовательские форматные строки, определенные .NET Framework, которые вы можете найти в документации MSDN, запустив поиск по "standard numeric format strings". Стандартные форматные строки обычно имеют форму `Axx`, где `A` — запрашиваемый формат, а `xx` — необязательный спецификатор точности. Примерами спецификаторов формата для чисел могут служить: `"C"` — для валюты, `"D"` — для десятичных чисел, `"E"` — для научной нотации, `"F"` — для нотации с фиксированной точкой и `"X"` — для шестнадцатеричной нотации. Каждый тип поддерживает также спецификатор `"G"` для обозначения общего вида, которым является спецификатор формата по умолчанию и который применяется, когда вы вызываете `Object.ToString`, не указывая форматной строки. Если эти форматные строки не отвечают вашим потребностям, вы можете даже использовать одну из пользовательских форматных строк, позволяющих описать то, что вам нужно, в более или менее наглядном виде.

Суть всего этого механизма заключается в том, что каждый тип интерпретирует и определяет форматную строку специфично для контекста его собственных нужд. Другими словами, System.Double вполне может трактовать спецификатор формата G иначе, чем тип System.Int32. Более того, ваш собственный тип — скажем, Employee, волен определять форматные строки по своему усмотрению. Например, форматная строка "SSN" может порождать выходную строку, содержащую номер социального страхования сотрудника.

На заметку! Что даже еще удобнее — можно позволить вашим собственным типам обрабатывать форматную строку "DBG", таким образом создавая детализированную строку, представляющую внутреннее состояние, для отправки в выходной отладочный протокол.

Рассмотрим небольшой пример кода, использующий описанные концепции:

```
using System;
using System.Globalization;
using System.Windows.Forms;
public class EntryPoint
{
    static void Main() {
        CultureInfo current = CultureInfo.CurrentCulture;
        CultureInfo germany = new CultureInfo( "de-DE" );
        CultureInfo russian = new CultureInfo( "ru-RU" );
        double money = 123.45;
        string localMoney = money.ToString( "C", current );
        MessageBox.Show( localMoney, "Локальные деньги" );
        localMoney = money.ToString( "C", germany );
        MessageBox.Show( localMoney, "Деньги Германии" );
        localMoney = money.ToString( "C", russian );
        MessageBox.Show( localMoney, "Деньги России" );
    }
}
```

В этом примере я отображаю строку, используя тип MessageBox, определенный в Window.Forms, поскольку консоль не очень подходит для отображения символов Unicode. Спецификатор формата, который я выбрал — "C" — служит для отображения числа в формате валюты. Для первого отображения я использую экземпляр CultureInfo, прикрепленный к текущему потоку. Для второго — создаю CultureInfo для Германии и России. Обратите внимание, что при формировании строки тип System.Double использовал свойства CurrencyDecimalSeparator, CurrencyDecimalDigits и CurrencySymbol экземпляра NumberFormatInfo, возвращенного методом CultureInfo.GetFormat.

Если бы я отображал экземпляр DateTime, то аналогичным образом реализация DateTime метода IFormattable.ToString использовала бы экземпляр DateTimeFormatInfo возвращенного методом CultureInfo.GetFormat.

Console.WriteLine и String.Format

На протяжении настоящей книги вы видели, что я интенсивно использовал в примерах метод Console.WriteLine. Одна из удобных форм этого метода, иден-

тичных некоторым перегрузкам String.Format, позволяет вам строить составные строки, заменяя дескрипторы формата внутри строки переменным числом переданных параметров. На практике String.Format подобен семейству функций printf из C и C++. Однако он намного гибче и безопаснее, поскольку основан на средствах форматирования .NET Framework, описанных ранее. Взглянем на следующий быстрый пример использования String.Format:

```
using System;
using System.Globalization;
using System.Windows.Forms;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 3 ) {
            Console.WriteLine( "Введите 3 параметра" );
            return;
        }
        string composite =
            String.Format( "{0} + {1} = {2}",
                           args[0],
                           args[1],
                           args[2] );
        Console.WriteLine( composite );
    }
}
```

Здесь вы можете видеть указатели места заполнения (placeholder) в фигурных скобках, и числа внутри них, представляющие индексы в следующем далее списке параметров. Метод String.Format, как и метод Console.WriteLine, имеет перегрузку, принимающую переменное количество параметров для использования в качестве подставляемых значений. В данном примере метод String.Format заменяет каждый указатель места заполнения, используя общее форматирование типа, которое вы получаете вызовом версии ToString, лишенной параметров. Если экземпляр, подставляемый в данное место, поддерживает IFormattable, то вызывается метод IFormattable.ToString со спецификатором формата null, что обычно эквивалентно применению спецификатора "G". Кстати, если вам нужно вставить фигурные скобки внутрь исходной строки, отображаемой в выводе, вы должны дублировать их, применяя {{ или }}.

Точный формат подставляемого элемента:

```
{index[,alignment][:formatString]}
```

где элементы внутри квадратных скобок не обязательны. Значение index отсчитывается от нуля и используется для ссылки на один из завершающих параметров, переданных методу. Значение выравнивания (alignment) представляет ширину, которая отводится элементу внутри составной строки. Например, если вы установите ширину восемь символов, и передадите более короткую строку, то ширина будет дополнена пробелами. И, наконец, часть formatString подставляемого элемента позволяет вам обозначить точный формат.

Форматная строка — это строка в определенном стиле, которую вы должны использовать, если собираетесь вызвать IFormattable.ToString на самом эк-

земпляре, о чем говорилось в предыдущем разделе. К сожалению, вы не можете специфицировать отдельный экземпляр IFormatProvider для каждой из подставляемых строк. Если вам нужно создать составную строку из элементов, используя множество поставщиков форматов или множество культур, вы должны прибегнуть непосредственно к IFormattable.ToString.

Примеры строкового форматирования в пользовательских типах

Давайте рассмотрим другой пример использования знаменитого типа Complex, с которым мы упражняемся на протяжении всей этой книги. На этот раз попробуем реализовать IFormattable, чтобы сделать его немножко более удобным для генерации строковой версии экземпляра.

```
using System;
using System.Text;
using System.Globalization;
public struct Complex : IFormattable
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    // Реализация IFormattable
    public string ToString( string format,
                            IFormatProvider formatProvider ) {
        StringBuilder sb = new StringBuilder();
        if( format == "DBG" ) {
            // Генерация отладочного вывода для данного объекта
            sb.Append( this.GetType().ToString() + "\n" );
            sb.AppendFormat( "\tдействительная:\t{0}\n", real );
            sb.AppendFormat( "\tмнимая:\t{0}\n", imaginary );
        } else {
            sb.Append( "( " );
            sb.Append( real.ToString(format, formatProvider) );
            sb.Append( " : " );
            sb.Append( imaginary.ToString(format, formatProvider) );
            sb.Append( " )" );
        }
        return sb.ToString();
    }
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main() {
        CultureInfo local = CultureInfo.CurrentCulture;
        CultureInfo germany = new CultureInfo( "de-DE" );
        Complex cpx = new Complex( 12.3456, 1234.56 );
```

```
        string strCpx = cpx.ToString( "F", local );
        Console.WriteLine( strCpx );
        strCpx = cpx.ToString( "F", germany );
        Console.WriteLine( strCpx );
        Console.WriteLine( "\nОтладочный вывод:\n{0:DBG}", cpx );
    }
}
```

Все "мясо" этого примера заключается внутри реализации `IFormattable.ToString`. Я реализовал форматную строку `"DBG"` для данного типа, которая создает строку, отображающую внутреннее состояние объекта, и которая может пригодиться в целях отладки. Уверен, что вы можете представить и больше информации об экземпляре для вывода в протокол отладки, но думаю, идею вы поняли. Если форматная строка не равна `"DBG"`, то вы просто обращаетесь к реализации `IFormattble` типа `System.Double`. Обратите внимание на применение `StringBuilder` для создания строки, возвращаемой в конечном итоге. Также я решил использовать метод `Console.WriteLine` и его синтаксис формата экземпляра для отладочного вывода на консоль — просто, чтобы показать разнообразие применения.

ICustomFormatter

`ICustomFormatter` — это интерфейс, позволяющий вам заменять или расширять встроенный или уже существующий интерфейс `IFormattable` объекта. Всякий раз, когда вызывается `String.Format` или `StringBuilder.AppendFormat` для преобразования экземпляра объекта в строку, перед тем, как произойдет вызов через реализацию `IFormattable.ToString` объекта, сначала выполняется проверка, не предоставляет ли переданный `IFormatProvider` собственного средства форматирования. Это делается посредством вызова `IFormatProvider.GetFormat` с передачей типа `ICustomFormatter`. Если возвращается реализация `ICustomFormatter`, то метод использует ее. В противном случае он использует реализацию `IFormattable.TiString` самого объекта или же реализацию `Object.ToString`, если объект не реализует `IFormattable`.

Рассмотрим следующий пример, где я опять обращаюсь к предыдущему примеру `Complex`, но на этот раз расширяю возможности отладочного вывода вне структуры `Complex`. Измененный код выделен полужирным.

```
using System;
using System.Text;
using System.Globalization;

public class ComplexDbgFormatter : ICustomFormatter, IFormatProvider
{
    // Реализация IFormatProvider
    public object GetFormat( Type formatType ) {
        if( formatType == typeof(ICustomFormatter) ) {
            return this;
        } else {
            return CultureInfo.CurrentCulture.
            GetFormat( formatType );
        }
    }
```

```
    // Реализация ICustomFormatter
    public string Format( string format,
    object arg,
    IFormatProvider formatProvider ) {
        if( arg.GetType() == typeof(Complex) &&
        format == "DBG" ) {
            Complex cpx = (Complex) arg;
            // Сгенерировать отладочный вывод для данного объекта
            StringBuilder sb = new StringBuilder();
            sb.Append( arg.GetType().ToString() + "\n" );
            sb.AppendFormat( "\tдействительная:\t{0}\n", cpx.Real );
            sb.AppendFormat( "\tмнимая:\t{0}\n", cpx.Img );
            return sb.ToString();
        } else {
            IFormattable formattable = arg as IFormattable;
            if( formattable != null ) {
                return formattable.ToString( format, formatProvider );
            } else {
                return arg.ToString();
            }
        }
    }
}

public struct Complex : IFormattable
{
    public Complex( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public double Real {
        get { return real; }
    }
    public double Img {
        get { return imaginary; }
    }
    // Реализация IFormattable
    public string ToString( string format,
                IFormatProvider formatProvider ) {
        StringBuilder sb = new StringBuilder();
        sb.Append( "( " );
        sb.Append( real.ToString(format, formatProvider) );
        sb.Append( " : " );
        sb.Append( imaginary.ToString(format, formatProvider) );
        sb.Append( " )" );
        return sb.ToString();
    }
    private double real;
    private double imaginary;
}
```

```
public class EntryPoint
{
   static void Main() {
      CultureInfo local = CultureInfo.CurrentCulture;
      CultureInfo germany = new CultureInfo( "de-DE" );
      Complex cpx = new Complex( 12.3456, 1234.56 );
      string strCpx = cpx.ToString( "F", local );
      Console.WriteLine( strCpx );
      strCpx = cpx.ToString( "F", germany );
      Console.WriteLine( strCpx );
      ComplexDbgFormatter dbgFormatter =
         new ComplexDbgFormatter();
      strCpx = String.Format( dbgFormatter,
                              "{0:DBG}",
                              cpx );
      Console.WriteLine( "\nОтладочный вывод:\n{0}", strCpx );
   }
}
```

Конечно, этот пример немного сложнее. Но если вы не являетесь автором типа Complex, то для вас это может быть единственным способом обеспечить специальное форматирование данного типа. Применяя этот прием, вы можете предоставлять пользовательского форматирование для любого из встроенных типов системы.

Сравнение строк

Когда речь заходит о сравнении строк, здесь .NET Framework обеспечивает достаточную гибкость. Вы можете сравнивать строки как на основе информации о культуре, так и без ее учета. Вы можете также сравнивать строки с учетом регистра или без, причем правила независимого от регистра сравнения варьируются от культуры к культуре. Каркас предусматривает несколько способов сравнения строк — некоторые из них представлены непосредственно в типе System.String — через статический метод String.Compare. Вы можете выбирать среди нескольких его перегрузок, и самая базовая из них использует для сравнений объект CultureInfo, прикрепленный к текущему потоку.

У вас часто будет возникать необходимость в сравнении строк, и вам не придется беспокоиться о накладных расходах, связанных с культурно-зависимым сравнением. Блестящим примером может быть сравнение внутренних строковых данных, скажем, из конфигурационного файла, или сравнение файловых каталогов. Во времена .NET 1.1 главным инструментом, имеющимся в вашем распоряжении, был метод String.Compare с передачей свойства InvariantCulture. В большинстве случаев он прекрасно работает, даже если используемая информация о культуре является нейтральной по отношению ко всем культурам, что обычно влечет за собой излишние накладные расходы при таких сравнениях. В .NET 2.0 Framework было представлено новое перечисление StringComparison, которое позволяет выбрать правильный способ сравнения, независимый от культуры. Перечисление StringComparison выглядит следующим образом:

```
public enum StringComparison
{
    CurrentCulture,
    CurrentCultureIgnoreCase,
    InvariantCulture,
    InvariantCultureIgnoreCase,
    Ordinal,
    OrdinalIgnoreCase
}
```

Последние два элемента в перечислении представляют особый интерес. Сравнение на базе порядка — основной способ сравнения строк; он просто сравнивает числовые значения символов двух строк (т.е., по сути, сравнивает "сырые" двоичные значения каждого символа). Выполнение сравнения подобным образом исключает все нюансы, связанные с культурой из процесса сравнения и значительно повышает его эффективность. На моем компьютере я запустил несколько грубых временных циклов для оценки двух приемов сравнения строк равной длины. Сравнение на основе кодов символов оказалось в девять раз быстрее. Конечно, будь строки более сложными, и если бы в них содержалось что-то кроме латинских символов нижнего регистра, то выигрыш производительности был бы еще большим.

В .NET 2.0 Framework предложен новый класс по имени `StringComparer`, реализующий интерфейс `IComparer`. Такие вещи, как сортированные коллекции, могут использовать его для управления сортировкой. Тип `System.StringComparer` следует тому же шаблону, что локальная поддержка `IFormattable`. Вы можете использовать свойство `StringComparer.CurrentCulture` для получения экземпляра `StringComparer`, специфичного для культуры текущего потока. Вдобавок вы можете получить экземпляр `StringComparer` от `StringComparer.CurrentCultureIgnoreCase` для выполнения независимого от регистра сравнения. К тому же вы можете получить культурно-инвариантные экземпляры, используя свойства `InvariantCulture` и `InvariantCultureIgnoreCase`. И, наконец, вы можете использовать свойства `Ordinal` и `OrdinalIgnoreCase` для получения экземпляров, сравнивающих строки на основе обычных правил порядкового сравнения строк.

Как можно было ожидать, если информация о культуре, присоединенная к текущему потоку, вам не подходит, вы можете создавать экземпляры `StringComparer` на основе явных локалей, просто вызывая метод `StringComparer.Create` и передавая ему экземпляр `CultureInfo`, представляющий нужную вам локаль вместе с флагом чувствительности сравнения к регистру. Строки, используемые для указания используемой локали — те же, что применяется в `CultureInfo`; они описаны в документации MSDN.

При выборе среди разных способов сравнения руководствуйтесь целесообразностью для конкретной задачи. Общее эмпирическое правило заключается в том, что нужно использовать культурно-специфичные и культурно-инвариантные сравнения для всех данных, видимых пользователю, т.е. данных, которые будут представлены конечному пользователю тем или иным образом. В других случаях следует предпочесть порядковое сравнение. Однако вряд ли вам придется когда-либо применять сравнение строк `InvariantCulture` для отображения их пользователю. Применяйте порядковое сравнение, имея дело с данными, имеющими полностью внутренний характер. Фактически, порядковое сравнение с применением `InvariantCulture` почти бесполезно.

На заметку! До появления версии .NET 2.0 Framework существовало общее руководство, гласящее, что если вы сравниваете строки для принятия решений, связанных с безопасностью, то должны использовать `InvariantCulture` вместо базового сравнения на `CultureInfo.CurrentCulture`. При таких сравнениях вам нужно тонко контролируемая среда, в которой можно быть уверенным, что она не изменится по сравнению с вашей тестовой средой. Если вы основываете сравнение на `CurrentCulture`, этого достичь не удастся, поскольку конечные пользователи могут изменить текущую культуру на машине и тем самым запустить в действие непротестированный путь выполнения кода для принятия решения, связанного с безопасностью, поскольку провести тестирование для всех существующих культур практически невозможно. Естественно, в .NET 2.0 и далее рекомендуется выполнять такие ответственные сравнения на основе простого порядка кодов символов, а не через `InvariantCulture`, чтобы повысить уровень эффективности и безопасности.

Работа со строками из внешних источников

В пределах .NET Framework все строки представлены массивами символов Unicode UTF-16. Однако часто возникает необходимость взаимодействия с внешним миром, использующим некоторую другую форму кодирования вроде UTF-8. Иногда даже взаимодействуя с другими сущностями, использующими 16-битные строки Unicode, может случиться, что они применяют порядок следования байтов в 2-байтных символах, противоположный принятому на платформе Intel. .NET Framework облегчает эту работу с помощью класса `System.Text.Encoding`.

В этом разделе я не хотел бы погружаться в детальное описание `System.Text.Encoding`, но настоятельно рекомендую вам обратиться к документации по этому классу в MSDN, где вы найдете все необходимые подробности. Рассмотрим краткий пример преобразования строк между разными кодировками с применением объектов `Encoding`, обеспечиваемыми классом `System.Text.Encoding`.

```
using System;
using System.Text;
public class EntryPoint
{
   static void Main() {
      string leUnicodeStr = "ЗДОРОВО!";
      Encoding leUnicode = Encoding.Unicode;
      Encoding beUnicode = Encoding.BigEndianUnicode;
      Encoding utf8 = Encoding.UTF8;
      byte[] leUnicodeBytes = leUnicode.GetBytes(leUnicodeStr);
      byte[] beUnicodeBytes = Encoding.Convert( leUnicode,
                                                 beUnicode,
                                                 leUnicodeBytes);
      byte[] utf8Bytes = Encoding.Convert( leUnicode,
                                            utf8,
                                            leUnicodeBytes );
      Console.WriteLine( "Исх. строка: {0}\n", leUnicodeStr );
      Console.WriteLine( "Байты Little Endian Unicode:" );
      StringBuilder sb = new StringBuilder();
      foreach( byte b in leUnicodeBytes ) {
         sb.Append( b ).Append(" : ");
      }
```

```
        Console.WriteLine( "{0}\n", sb.ToString() );

        Console.WriteLine( "Байты Big Endian Unicode:" );
        sb = new StringBuilder();
        foreach( byte b in beUnicodeBytes ) {
            sb.Append( b ).Append(" : ");
        }
        Console.WriteLine( "{0}\n", sb.ToString() );

        Console.WriteLine( "Байты UTF:" );
        sb = new StringBuilder();
        foreach( byte b in utf8Bytes ) {
            sb.Append( b ).Append(" : ");
        }
        Console.WriteLine( sb.ToString() );
    }
}
```

Этот пример сначала создает `System.String` с некоторым текстом на русском языке. Как упоминалось, строка содержит строку Unicode, но в каком порядке идут байты символа — вначале старший или младший? Ответ зависит от платформы, на которой вы работаете. В системе Intel обычно сначала идет младший байт. Однако поскольку вы не имеет доступа к лежащему в основе байтовому представлению строк, так как оно скрыто от вас, это не имеет значения. Для того чтобы получить байты строки, вы должны использовать один из объектов `Encoding`, которые вы получаете от `System.Text.Encoding`. В моем примере я получаю локальные ссылки на объекты `Encoding` для обработки строк Unicode с первым старшим байтом (Big Endian) в символе, первым младшим байтом (Little Endian) и UTF-8. Получив их, я могу использовать их для преобразования строк в нужное мне байтовое представление. Как видите, я получаю три представления одной и той же строки и посылаю последовательность байтов в стандартный вывод. В данном примере, поскольку текст основан на кириллическом алфавите, байтовый массив UTF-8 получается длиннее, чем байтовый массив Unicode. Если бы исходная строка базировалась на латинском наборе символов, то массив UTF-8 получился бы короче, чем массив Unicode — обычно наполовину. Главное, что я хочу сказать: вы никогда не должны строить предположений относительно требований по хранению для любой из кодировок. Если вам нужно знать, сколько места потребуется для хранения кодированной строки, вызовите метод `Encoding.GetByteCount`, чтобы получить это значение.

Внимание! Никогда не стройте предположений относительно внутреннего формата представления строк в CLR. Нет никаких гарантий того, что оно не будет варьироваться от одной платформы к другой. Будет весьма неприятно, если вдруг ваш код сделает какое-то предположение, отталкиваясь от платформы Intel, а затем даст сбой при запуске на платформе Sun с работающей исполняющей системой Mono CLR. Microsoft может даже однажды решить запустить Windows на другой платформе — точно так же, как в Apple решили начать использовать процессоры Intel.

Обычно вам нужно двигаться в противоположном направлении и выполнять преобразование массива байтов из внешнего мира в строку, с которой система сможет легко работать. Например, стек протоколов Bluetooth использует кодировку

Unicode с первым старшим байтом для передачи строковых данных. Чтобы преобразовать байты в System.String, вызывайте метод GetString на кодировщике, который вы используете. Вы также должны применять кодировщик, соответствующий источнику кодируемых данных.

Это приводит к важному замечанию, которое всегда надо иметь в виду. При передаче строки в другие системы и обратно в формате "сырых" байт вы всегда должны знать схему кодировки, применяемую в протоколе, с которым имеете дело. Что более важно, вы всегда должны использовать соответствующий кодировке объект Encoding, чтобы преобразовывать байтовый массив в System.String, даже если знаете, что кодировка в протоколе — такая же точно, как применяется внутри System.String на платформе, для которой вы строите приложение. Почему? Представим, что вы разрабатываете приложение на платформе Intel, где протокол кодирования предполагает, что в паре байт символа первым идет младший, и вам известно, что такой же порядок принят на платформе кодирования. Вы решаете сэкономить, и отказываетесь от применения объекта System.Text.Encoding.Unicode для преобразования байтов строки. Позднее вы решаете запустить приложение на платформе, которая внутри использует обратный порядок байтов (старший — первый). Для вас будет большим сюрпризом, когда приложение начнет сбоить — только потому, что вы построили ошибочное предположение относительно внутренней реализации кодирования System.String. Эффективность — не проблема, если всегда использовать кодировщик, потому что платформа, на которой внутреннее кодирование совпадает с внешним, не выполняет лишних преобразований.

В предыдущем примере вы видели использование класс StringBuilder для отправки массива байт на консоль. Давайте теперь посмотрим, что собой представляет тип StringBuilder.

StringBuilder

Поскольку объекты System.String являются неизменными, иногда они становятся узким местом для эффективности, когда вы пытаетесь собирать строки "на лету". Вы можете создать составную строку, используя операцию сложения, как показано ниже:

```
string compound = "Голосуйте" + " за " + "меня";
```

Однако такой метод неэффективен, поскольку этот код создает целых четыре строки для того, чтобы выполнить свою работу. Создание всех этих промежуточных строк может повысить нагрузку на память. Хотя эта строка кода довольно искусственна, вы можете представить, насколько страдает эффективность сложной системы, которая выполняет массу строковых манипуляций, от интенсивного использования памяти. Рассмотрим случай, когда вы реализуете пользовательский кодировщик base64, который последовательно добавляет символы в процессе обработки двоичного файла. Библиотека .NET уже предоставляет эту функциональность в классе System.Convert, но давайте пока для примера мы ее проигнорируем. Если вы многократно в цикле используете операцию + для создания огромной строки base64, ваша производительность очень быстро деградирует по мере увеличения размера исходных данных. В такой ситуации вы можете прибегнуть к классу System.Text.StringBuilder, который реализует изменяемую строку специально для эффективного построения составных строк.

Я не стану сейчас углубляться в детальное описание методов StringBuilder, поскольку при желании вы найдете его в документации MSDN. Однако я раскрою несколько моментов, о которых обычно умалчивается. Внутри себя StringBuilder поддерживает массив символов, которым управляет динамически. Методы — "рабочие лошадки" этого класса — это Append, Insert и AppendFormat. Если вы найдете описание этих методов в MSDN, то увидите, что они имеют богатый набор перегрузок для поддержки добавления и вставки строк от множества распространенных типов. Когда вы создаете экземпляр StringBuilder, в вашем распоряжении имеются разнообразные конструкторы на выбор. Конструктор по умолчанию создает новый экземпляр StringBuilder, с определяемой системой емкостью по умолчанию. Однако эта емкость не ограничивает максимального размера создаваемой строки. Вместо этого она представляет объем строковых данных, которые объект StringBuilder может удерживать перед тем, как ему понадобится увеличить размер внутреннего буфера, а с ним и допустимую емкость. Если вы имеете приблизительное представление о размере строки, которую собираетесь строить, то можете передать этот размер объекту StringBuilder в виде параметра одного из конструкторов, и тогда он инициализирует свой буфер соответствующим образом. Это может помочь экземпляру StringBuilder избежать необходимости в слишком частом повторном размещении буфера в памяти по мере его заполнения.

Также в одной из перегрузок конструктора вы можете определить свойство максимальной емкости. По умолчанию максимальная емкость равна System.Int32.MaxValue, что в настоящее время составляет 2 147 483 647 байт, но точное значение может измениться в процессе эволюции системы. Если вы хотите предохранить буфер StringBuilder от роста свыше некоторой величины, вы можете предусмотреть альтернативную максимальную емкость в одной из перегрузок конструкторов. Если операция вставки или добавления потребует увеличения буфера сверх указанного предела, будет сгенерировано исключение ArgumentOutOfRangeException.

Для удобства все методы, которые добавляют и вставляют данные в экземпляр StringBuilder, возвращают ссылку на this. Поэтому вы можете связывать в цепочки операции одного и того же построителя строк, как показано ниже:

```
using System;
using System.Text;
public class EntryPoint
{
    static void Main() {
        StringBuilder sb = new StringBuilder();

        sb.Append("StringBuilder ").Append("является ").Append("очень... ");

        string built1 = sb.ToString();

        sb.Append("удобным");

        string built2 = sb.ToString();
        Console.WriteLine( built1 );
        Console.WriteLine( built2 );
    }
}
```

В данном примере вы можете видеть, что я преобразую экземпляр `StringBuilder` по имени `sb` в новый экземпляр `System.String` по имени `built1` вызовом `sb.ToString`. Для максимальной эффективности `StringBuilder` просто передает ссылку на символьный буфер экземпляру строки, так что никакого копирования не требуется. Если подумать, вы согласитесь, что часть пользы от `StringBuilder` оказалась бы под вопросом, если бы он не поступал описанным образом. В конце концов, если вы создаете громадную строку, скажем, размером в несколько мегабайт, такую как закодированное в base64 графическое изображение, вы не захотите, чтобы для создания строки пришлось копировать весь буфер. Однако как только вы создали `System.String`, вы получаете ссылку на `System.String` и ссылку `StringBuilder` на тот же массив символов. Поскольку объект `System.String` является неизменным, внутренний символьный массив `StringBuilder` теперь также становится неизменным. После этого `StringBuilder` переключается на идиому "копия по записи" для этого буфера. Важно иметь в виду такое поведение, когда вы работаете с большими строковыми данными посредством использования `StringBuilder`.

Поиск строк с помощью регулярных выражений

Тип `System.String` сам по себе предоставляет некоторые рудиментарные методы поиска, подобные `IndexOf`, `IndexOfAny`, `LastIndexOf`, `LastIndexOfAny` и `StartsWith`. Применяя эти методы, вы можете определить, содержит ли строка определенную подстроку, и где именно. Однако эти методы становятся весьма неуклюжими и слишком примитивными, когда возникает необходимость в эффективном сложном поиске строк. К счастью, библиотека классов .NET Framework содержит классы, реализующие регулярные выражения (regexp). Если вы еще не знакомы с регулярными выражениями, я настоятельно рекомендую изучить их синтаксис и способы их эффективного использования. Синтаксис регулярных выражений сам по себе является языком. Среди замечательных источников информации на эту тему — книга Джеффри Фридла (Jeffrey E. F. Friedl) *Mastering Regular Expressions, Third Edition* (Sebastapol, CA: O'Reilly Media, 2006 г.), а также материал статьи *Regular Expression Language Elements* (Элементы языка регулярных выражений) в составе документации MSDN. Возможности механизма регулярных выражений соответствуют аналогичным механизмам в Perl 5 и Python. Полное описание всех возможностей регулярных выражений вместе с их синтаксисом выходит за рамки настоящей книги. Тем не менее, некоторые особенности использования регулярных выражений в .NET Framework я все-таки опишу.

Существует три главных типа операций, для которых вы можете применить регулярные выражения. Первый — поиск в строке вхождения некоторого специфического шаблона и места такого вхождения. Поисковый шаблон может быть чрезвычайно сложным. Второй тип подобен первому, за исключением того, что в процессе поиска вы исключаете части выражения поиска. Например, если вы ищете в строке дату в определенном формате, вы можете решить разбить дату на три части и разнести их по разным переменным. И, наконец, регулярные выражения часто применяются для операций поиска с заменой. Операции такого рода базируются на предыдущих двух. Давайте посмотрим, как можно достичь этих трех целей с применением реализации регулярных выражений .NET Framework.

Поиск с помощью регулярных выражений

Как и сам System.String, большинство объектов, созданных из классов регулярных выражений, являются неизменяемыми. Главная "рабочая лошадка" всей системы регулярных выражений — класс Regex, находящийся в пространстве имен System.Text.RegularExpressions. Одним из наиболее распространенных способов его применения является создание экземпляра Regex для представления вашего регулярного выражения посредством передачи его конструктору строки шаблона поиска. Затем вы применяете строку для определения наличия соответствия. Вы можете также найти, где именно в строке находятся последовательные вхождения искомого шаблона. Двинемся дальше и рассмотрим сначала пример того, как выглядит базовый поиск с Regex, а потом обратимся к более полезным способам применения Regex.

```
using System;
using System.Text.RegularExpressions;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 1 ) {
            Console.WriteLine( "Вы должны предоставить строку." );
            return;
        }
        // Создать regex для поиска шаблона IP-адреса
        string pattern = @"\d\d?\d?\.\d\d?\d?.\d\d?\d?.\d\d?\d?";
        Regex regex = new Regex( pattern );
        Match match = regex.Match( args[0] );
        while( match.Success ) {
            Console.WriteLine( "IP-адрес найден в позиции {0} со " +
                               "значением {1}",
                               match.Index,
                               match.Value );
            match = match.NextMatch();
        }
    }
}
```

Этот пример выполняет поиск IP-адреса в строке, переданной в виде параметра командной строки. Поиск довольно топорный, но дальше я его усовершенствую. Регулярные выражения могут состоять из литеральных символов, которые нужно найти, а также управляющих символов, имеющих специальное назначение. Знакомый метод обратного слэша применяется для отмены символов в регулярном выражении. В данном примере \d означает десятичную цифру. Символы, снабженные суффиксом — знаком вопроса (?) означают одно или более вхождений предыдущего шаблонного символа регулярного выражения. Обратите внимание на защищенные слэшем точки. Дело в том, что точка сама по себе в регулярном выражении имеет определенный смысл. Незащищенная точка соответствует любому одиночному символу в позиции, где она находится. В конце концов, вы увидите, что намного проще применять синтаксис дословных (verbatime) строк при объявлении регулярных выражений, чтобы избежать чрезмерного применения обратных

слэшей. Если вызвать предыдущий пример, передав ему в командной строке аргумент в кавычках:

```
"Вот такой IP-адрес:123.123.1.123"
```

то вывод будет выглядеть так:

```
IP-адрес найден в позиции 19 со значением 123.123.1.123
```

Предыдущий пример создает новый экземпляр `Regex` по имени `regex` и затем, используя метод `Match`, применяет шаблон к заданной строке. Результат соответствия сохраняется в переменной `match`. Эта переменная содержит первое вхождение шаблона в строку. Вы можете использовать свойство `Match.Success`, чтобы определить, нашло регулярное выражение в строке соответствие или нет. Далее вы увидите код, использующий свойства `Index` и `Value` для того, чтобы узнать больше о найденном вхождении. И, наконец, вы сможете перейти к следующему соответствию в искомой строке, вызвав метод `Match.NextMatch`, и таким образом, пройтись по всей цепочке, пока не переберете все вхождения шаблона в строку.

Альтернативно вместо вызова `Match.NextMatch` в цикле можно вызвать метод `Regex.Matches` для получения коллекции `MatchCollection`, которая даст все вхождения сразу вместо того, чтобы перебирать их по одному. К тому же все примеры использования `Regex` в настоящей главе вызывают методы экземпляра `Regex`. Многие из таких методов, например, `Match` и `Replace`, также предусматривают статические версии, для которых не нужно сначала создавать экземпляра `Regex`, а просто передать шаблон регулярного выражения при вызове метода.

Поиск и группирование

Если посмотреть на предыдущий пример, то все, что в нем происходит — это осуществляется поиск по шаблону групп из десятичных цифр, разделенных точками, причем каждая группа может содержать от одной до трех цифр. Я говорю это потому, чтобы показать, что такой грубый поиск обнаружит соответствие неправильного IP-адреса вроде `999.888.777.666`. Более совершенный поиск IP-адреса должен выглядеть следующим образом:

```
using System;
using System.Text.RegularExpressions;
public class EntryPoint
{
   static void Main( string[] args ) {
      if( args.Length < 1 ) {
         Console.WriteLine( "Вы должны предоставить строку." );
         return;
      }
      // Создать regex для поиска шаблона IP-адреса.
      string pattern = @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                       @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                       @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                       @"([01]?\d\d?|2[0-4]\d|25[0-5])";
      Regex regex = new Regex( pattern );
      Match match = regex.Match( args[0] );
```

```
        while( match.Success ) {
            Console.WriteLine( "IP-адрес найден в позиции {0} со " +
                               "значением {1}",
                               match.Index,
                               match.Value );
            match = match.NextMatch();
        }
    }
}
```

По сути, в приведенном регулярном выражении четыре группы одного и того же поискового шаблона `[01]?\d\d?|2[0-4]\d|25[0-5]` разделены точками, которые, конечно же, отменены. Каждое из этих подвыражений соответствует числу от 0 до 255. Такое регулярное выражение для поиска IP-адреса уже намного лучше, хотя все еще не идеально. Однако вы можете видеть, что это намного ближе, и с небольшой тонкой настройкой его можно применять для верификации IP-адресов, переданных в строке. Таким образом, вы можете применять регулярные выражения эффективной верификации пользовательского ввода, чтобы гарантировать, что он будет соответствовать определенной форме. Например, у вас может быть Цеb-сервер, который ожидает ввода телефонных номеров США в формате `(xxx) xxx-xxxx`. Регулярные выражения позволяют вам легко проверять корректность ввода номеров пользователями.

Возможно, вы заметили добавление скобок в выражение поиска IP-адресов в предыдущем примере. Скобки используются для определения групп, формирующих подвыражения внутри общего регулярного выражения в дискретные кусочки. Группы также могут содержать в себе другие группы. Таким образом, шаблон регулярного выражения, описывающий IP-адрес в предыдущем примере, формирует группу для каждой части IP-адреса. В дополнение вы можете обращаться к каждой индивидуальной группе внутри соответствия. Рассмотрим следующую модифицированную версию того же примера:

```
using System;
using System.Text.RegularExpressions;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 1 ) {
            Console.WriteLine( "Вы должны предоставить строку." );
            return;
        }
        // Создать regex для поиска шаблона IP-адреса.
        string pattern = @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"([01]?\d\d?|2[0-4]\d|25[0-5])";
        Regex regex = new Regex( pattern );
        Match match = regex.Match( args[0] );
        while( match.Success ) {
            Console.WriteLine( "IP-адрес найден в позиции {0} со " +
                               "значением {1}",
                               match.Index,
                               match.Value );
```

```
            Console.WriteLine( "Группы:" );
            foreach( Group g in match.Groups ) {
                Console.WriteLine( "\t{0} в позиции {1}",
                                   g.Value,
                                   g.Index );
            }
            match = match.NextMatch();
        }
    }
}
```

Для каждого соответствия я добавил цикл, выполняющий итерацию по всем индивидуальным группам внутри соответствия. Как можно было ожидать, в коллекции будет как минимум, четыре группы, — по одной для каждой части IP-адреса. Фактически, есть еще пятый элемент группы — все совпадение. Таким образом, одна из групп внутри коллекции групп, возвращенных `Match.Groups` всегда будет содержать само полное соответствие.

Если передать следующий ввод последнему примеру:

```
"Вот такой IP-адрес:123.123.1.123"
```

то результат будет выглядеть следующим образом:

```
IP-адрес найден в позиции 19 со значением 123.123.1.123
Группы:
        123.123.1.123 в позиции 19
        123 в позиции 19
        123 в позиции 23
        1 в позиции 27
        123 в позиции 29
```

Группы представляют собой блестящую возможность указания частей заданной входной строки. Например, в некоторый момент, выполняя верификацию формата пользовательского ввода телефонного номера, вы можете также захватить область — группу кода междугородней связи — для последующего использования. Организация подстрок соответствия в группы весьма удобна. Но что еще удобнее возможность присвоения группам имен. Взгляните на следующий модифицированный пример:

```
using System;
using System.Text.RegularExpressions;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 1 ) {
            Console.WriteLine( "Вы должны предоставить строку." );
            return;
        }
        // Создать regex для поиска шаблона IP-адреса.
        string pattern = @"(?<part1>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"(?<part2>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"(?<part3>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"(?<part4>[01]?\d\d?|2[0-4]\d|25[0-5])";
```

```
        Regex regex = new Regex( pattern );
        Match match = regex.Match( args[0] );
        while( match.Success ) {
           Console.WriteLine( "IP-адрес найден в позиции {0} со " +
                              "значением {1}",
                              match.Index,
                              match.Value );
           Console.WriteLine( "Группы:" );
           Console.WriteLine( "\tЧасть 1: {0}",
                              match.Groups["part1"] );
           Console.WriteLine( "\tЧасть 2: {0}",
                              match.Groups["part2"] );
           Console.WriteLine( "\tЧасть 3: {0}",
                              match.Groups["part3"] );
           Console.WriteLine( "\tЧасть 4: {0}",
                              match.Groups["part4"] );
           match = match.NextMatch();
        }
    }
}
```

В этом варианте я захватываю каждую часть в группу по имени, и когда отправляю результат на консоль, то обращаюсь к группе по имени через индексатор в GroupCollection, возвращенный методом Match.Groups, который принимает строковый аргумент.

С возможностью именования групп появилась возможность ссылаться на них внутри поиска. Например, если вы ищете точное повторение предыдущего соответствия, вы можете обратиться к предыдущей группе с помощью того, что называется обратной ссылкой, включив \k<имя>, где имя — имя группы для обратной ссылки. Например, рассмотрим следующий пример, который ищет IP-адреса, у которых все части совпадают:

```
using System;
using System.Text.RegularExpressions;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 1 ) {
           Console.WriteLine( "Вы должны предоставить строку." );
           return;
        }
        // Создать regex для поиска шаблона IP-адреса.
        string pattern = @"(?<part1>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"\k<part1>\." +
                         @"\k<part1>\." +
                         @"\k<part1>";
        Regex regex = new Regex( pattern );
        Match match = regex.Match( args[0] );
        while( match.Success ) {
           Console.WriteLine( "IP-адрес найден в позиции {0} со " +
                              "значением {1}",
                              match.Index,
                              match.Value );
```

```
            match = match.NextMatch();
        }
    }
}
```

Следующий вывод демонстрирует результат запуска этого кода с параметром "Мой IP-адрес выглядит как 123.123.123.123":

```
IP-адрес найден в позиции 26 со значением 123.123.123.123
```

Замена текста с помощью Regex

Если вы когда-либо использовали язык Perl для выполнения любой обработки текста, то знаете, что без механизма регулярных выражений там не обойтись. Но одним из огромных преимуществ Perl являются возможности подстановки текста. Регулярные выражения в .NET предоставляют в ваше распоряжение те же возможности через перегрузки метода Regex.Replace. Предположим, что вы хотите обработать строку, найдя в ней введенный пользователем IP-адрес, и отобразить строку. Однако из соображений безопасности хотите заменить IP-адрес маской xxx.xxx.xxx.xxx. Следующий пример демонстрирует, как достичь этой цели:

```
using System;
using System.Text.RegularExpressions;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 1 ) {
            Console.WriteLine( "Вы должны предоставить строку." );
            return;
        }
        // Создать regex для поиска шаблона IP-адреса.
        string pattern = @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"([01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"([01]?\d\d?|2[0-4]\d|25[0-5])";
        Regex regex = new Regex( pattern );
        Console.WriteLine( "Полученный ввод -> {0}",
                           regex.Replace(args[0],
                           "xxx.xxx.xxx.xxx") );
    }
}
```

Теперь при таком вводе:

```
"Вот такой IP-адрес:123.123.1.123"
```

вывод будет выглядеть так:

```
Полученный ввод -> Вот такой IP-адрес:xxx.xxx.xxx.xxx
```

Конечно, когда вы находите соответствие внутри строки, то можете пожелать заменить ее чем-то таким, что зависит от найденного соответствия. В предыдущем примере каждое вхождение просто заменяется статической строкой. Для того чтобы выполнять замену на основе экземпляра соответствия, вы можете соз-

дать экземпляр делегата MatchEvaluator и передать его методу Regex.Replace. Затем при каждом обнаружении соответствия он будет вызывать экземпляр делегата MatchEvaluator, передавая ему найденное соответствие. Тогда делегат может создавать заменяющую строку на основе текущего соответствия. Делегат MatchEvaluator имеет следующую сигнатуру:

```
public delegate string MatchEvaluator( Match match );
```

Предположим, что вы хотите поменять порядок индивидуальных частей IP-адреса. Для этого вы можете использовать MatchEvaluator в сочетании с Regex.Replace, чтобы выполнить такую работу, как показано в следующем примере:

```
using System;
using System.Text;
using System.Text.RegularExpressions;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 1 ) {
            Console.WriteLine( "Вы должны предоставить строку." );
            return;
        }
        // Создать regex для поиска шаблона IP-адреса.
        string pattern = @"(?<part1>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"(?<part2>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"(?<part3>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                         @"(?<part4>[01]?\d\d?|2[0-4]\d|25[0-5])";
        Regex regex = new Regex( pattern );
        Match match = regex.Match( args[0] );

        MatchEvaluator eval = new MatchEvaluator(
                                  EntryPoint.IPReverse );
        Console.WriteLine( regex.Replace(args[0],
                               eval) );
    }
    static string IPReverse( Match match ) {
        StringBuilder sb = new StringBuilder();
        sb.Append( match.Groups["part4"] + "." );
        sb.Append( match.Groups["part3"] + "." );
        sb.Append( match.Groups["part2"] + "." );
        sb.Append( match.Groups["part1"] );
        return sb.ToString();
    }
}
```

При каждом обнаружении соответствия вызывается делегат для определения того, какой должна быть заменяющая строка. Однако поскольку все, что мы делаем — это изменение порядка, работа не слишком сложна для того, что называется *подстановкой с помощью регулярного выражения*. Если в примере, предшествующем данному, вы бы решили использовать перегрузку Replace, не использующую делегат MatchEvaluator, вы могли бы достичь того же результата, поскольку regex позволяет обращаться к переменным групп в строке замены. Чтобы сослаться на

одну из именованных групп, вы можете использовать синтаксис, показанный в следующем примере:

```
using System;
using System.Text;
using System.Text.RegularExpressions;
public class EntryPoint
{
   static void Main( string[] args ) {
      if( args.Length < 1 ) {
         Console.WriteLine( "Вы должны предоставить строку." );
         return;
      }
      // Создать regex для поиска шаблона IP-адреса.
      string pattern = @"(?<part1>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                       @"(?<part2>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                       @"(?<part3>[01]?\d\d?|2[0-4]\d|25[0-5])\." +
                       @"(?<part4>[01]?\d\d?|2[0-4]\d|25[0-5])";
      Regex regex = new Regex( pattern );
      Match match = regex.Match( args[0] );
      string replace = @"${part4}.${part3}.${part2}.${part1}" +
                       @" (the reverse of $&)";
      Console.WriteLine( regex.Replace(args[0], replace) );
   }
}
```

Чтобы включить одну из именованных групп, просто используйте синтаксис `${имя}`, где `имя` — имя группы. Вы можете также видеть, что я ссылаюсь на полный текст соответствия, используя `$&`. Доступны и другие строки подстановок, такие как `` $` ``, что означает часть входной строки, предшествующую найденному соответствию, и `$'`, что означает весь текст после соответствия. Есть еще много строк подстановок; все они описаны в документации MSDN.

Как вы можете представить, реализация регулярных выражений .NET обеспечивает столь же богатые возможности сложных замен строк, как и в языке Perl.

Варианты создания Regex

Одна из перегрузок конструкторов `Regex` позволяет вам передавать различные варианты типа `RegexOptions` при создании экземпляра `Regex`. Аналогично, такие методы `Regex`, как `Match` и `Replace`, также имеют статические перегрузки, принимающие флаги `RegexOptions`. В этом разделе я расскажу о некоторых наиболее часто используемых вариантах, а полное описание всех вариантов и их поведения вы найдете в документации по `RegexOptions` в составе MSDN.

По умолчанию регулярные выражения интерпретируются во время выполнения. Сложные регулярные выражения могут потреблять значительное время работы процессора, когда механизм регулярных выражений станет их обрабатывать. Для таких ситуаций рассмотрите возможность применения опции `Compiled`. Эта опция заставит представить регулярное выражение кодом IL, сгенерированным JIT-компилятором. Это замедлит первое применение регулярного выражения, но при частом использовании окупится с лихвой. К тому же не забудьте, что JIT-компилированный код увеличивает рабочий набор приложения.

Часто вы столкнетесь с необходимостью поиска, независимого от регистра. Это можно указать в шаблоне регулярного выражения, но это ухудшит его читабельность. Намного проще передать флаг `IgnoreCase` при создании экземпляра `Regex`. Используя этот флаг, механизм `Regex` также примет во внимание специфичные для культуры проблемы нечувствительности к регистру, обратившись к экземпляру `CultureInfo`, ассоциированному с текущим потоком. Если вам нужно выполнять независимый от регистра поиск независимым от культуры способом, комбинируйте флаг `IgnoreCase` с флагом `CultureInvariant`.

Флаг `IgnorePatternWhitespace` также полезен для сложных регулярных выражений. Этот флаг сообщает механизму регулярных выражений о необходимости игнорировать все пробелы внутри выражения соответствия, а также игнорировать все комментарии в строках, следующие за символом #. Это позволяет удобно комментировать действительно сложные регулярные выражения. Например, вот как можно модифицировать предыдущий пример поиска IP-адреса, используя `IgnorePatternWhitespace`:

```
using System;
using System.Text.RegularExpressions;
public class EntryPoint
{
    static void Main( string[] args ) {
        if( args.Length < 1 ) {
            Console.WriteLine( "Вы должны предоставить строку." );
            return;
        }
        // Создать regex для поиска шаблона IP-адреса.
        string pattern = @"

# Первая часть совпадения
([01]?\d\d? # По крайней мере, одна цифра, возможно,
            # предваренная 0 или 1, за которой,
            # возможно, следует другая цифра

# ИЛИ
|2[0-4]\d # Начинается с 2, затем идет число 0-4
          # и затем любая цифра

# ИЛИ
|25[0-5]) # 25, за которым следует число 0-5
\. # Целая группа, за которой следует точка.

# ПОВТОР
([01]?\d\d?|2[0-4]\d|25[0-5])\.

# ПОВТОР
([01]?\d\d?|2[0-4]\d|25[0-5])\.

# ПОВТОР
([01]?\d\d?|2[0-4]\d|25[0-5])
";
        Regex regex = new Regex( pattern,
                                 RegexOptions.IgnorePatternWhitespace );
        Match match = regex.Match( args[0] );
```

```
            while( match.Success ) {
                Console.WriteLine( "IP-адрес найден в позиции {0} со " +
                                   "значением {1}",
                                   match.Index,
                                   match.Value );
                match = match.NextMatch();
            }
        }
    }
```

Обратите внимание, насколько выразительными могут быть комментарии внутри регулярного выражения. И с учетом того, насколько сложными могут становиться регулярные выражения, они никогда не будут лишними.

Резюме

В этой главе я коснулся лишь верхушки айсберга возможностей обработки строк, имеющихся в .NET Framework и C#. Поскольку строковый тип настолько широко используется, проектировщики CLR вместо того, чтобы просто включать его в базовую библиотеку классов, отнесли его к набору встроенных типов. Это мудрое решение, учитывая, насколько распространено применение строк. Более того, библиотека предусматривает исчерпывающую реализацию специфичных для культуры шаблонов через `CultureInfo`, что обычно требуется при создании глобальных приложений, интенсивно работающих со строками.

Я показал, как вы можете создавать свои собственные культуры, просто применяя для этого класс `CultureAndRegionInfoBuilder`. По сути, любое программное обеспечение, напрямую взаимодействующее с пользователем и предназначенное для глобального применения, должно быть готовым обслуживать специфичные локальные потребности. И, наконец, я провел краткую экскурсию в мир возможностей .NET Framework обрабатывать регулярные выражения, хотя изложение полного руководства по языку регулярных выражений и не входило в мои намерения. Полагаю, вы согласитесь с тем, что средства обработки строк и текстов, встроенные в CLR, .NET Framework и язык C#, отлично продуманы и удобны в применении.

В главе 9 я расскажу о массивах и других развитых типах коллекций, имеющихся в .NET Framework. Также я потрачу изрядное количество времени на описание новой поддержки итераторов в C#.

ГЛАВА 9

Массивы, типы коллекций и итераторы

Типы коллекций окружают нас, начиная с самых истоков программирования. Уверен, что вы помните упражнения со связными списками, которые приходилось выполнять всем, кто учился программированию. В этой главе я представлю краткий обзор массивов, не слишком погружаясь в детали, поскольку массивы не особенно меняются между выпусками .NET.

Однако я потрачу больше времени на объяснение основных обобщенных интерфейсов коллекций и итераторов, вместе с описанием тех замечательных вещей, которые вы можете с ними делать. Традиционно создание перечислителей для типов коллекций всегда было утомительным и скучным делом. Итераторы облегчили эту задачу, при этом значительно повысив читабельность вашего кода.

Представление массивов

Массивы C#, как и массивы CLR, в значительной мере происходят от массивов C/C++. В C/C++ вы обычно обращаетесь к массивам, смещая указатель, который вначале указывает на начало непрерывного диапазона элементов, находящихся в некотором блоке памяти. Массивы C/C++ не имеют встроенных средств контроля диапазона, что является причиной многих ошибок, с которыми вам наверняка приходилось сталкиваться. C# и CLR элегантно решают эту проблему, сделав массивы встроенным, неявным типом исполняющей системы.

Когда вы объявляете тип — будь то класс или структура — исполняющая система резервирует право молча сгенерировать тип массива, основанный на этом новом типе. Сгенерированный тип массива является ссылочным типом, т.е. экземпляры массива — это классы. Этот сгенерированный тип наследуется от `System.Array`, и дальше — от `System.Object`. Таким образом, вы можете трактовать все массивы C# полиморфно — через ссылку на `System.Array`. Конечно, это означает, что каждый массив, независимо от конкретного типа его элементов, реализует все методы и свойства `System.Array`.

Способ объявления массивов в C# подобен C/C++, за исключением того, что проектировщики языка постарались сделать синтаксис немножко более интуитив-

ным на их взгляд — в том, что квадратные скобки в объявлении следуют за типом, а не именем переменной. Следующий пример показывает три способа создания массива целых чисел и вывода их на консоль:

```
using System;
public class EntryPoint
{
    static void Main() {
        int[] array1 = new int[ 10 ];
        for( int i = 0; i < array1.Length; ++i ) {
            array1[i] = i*2;
        }
        int[] array2 = new int[] { 2, 4, 6, 8 };
        int[] array3 = { 1, 3, 5, 7 };
    }
}
```

Обычный способ создания экземпляра массива и наполнения его начальными значениями показан на примере инициализации массива array1. Элементы индексируются с применением индексатора, который обычно больше или равен 0. Может быть, вам уже известно, что массивы в CLR могут иметь нижнюю границу, определяемую пользователем. Однако в C# нижняя граница массива всегда равна 0, чтобы соответствовать ограничению CLR, гласящему, что массивы должны иметь нулевую нижнюю границу. Приемы инициализации, использованные с массивами array2 и array3, показывают сокращенную нотацию выполнения той же задачи. Обратите внимание, что во всех случаях вы должны сначала разместить экземпляры массива в куче, используя операцию new. То же происходит и с экземпляром array3, но здесь компилятор делает это за вас, позволяя сократить нотацию. Интересно отметить, что массив типа object — т.е. System.Object[] — сам имеет тип System.Object.

Одним из удобств массивов .NET является то, что они обеспечивают контроль диапазона. Поэтому, если вы попытаетесь выйти за границу массива, то получите ошибку времени выполнения — исполняющая система сгенерирует исключение IndexOutOfRangeException, вместо обращения к случайному участку памяти, как это происходит в "родном" C/C++. Так что вы можете попрощаться с этими коварными, трудно обнаруживаемыми ошибками, поскольку CLR не позволит им прятаться очень долго, потому они определенно не останутся незамеченными в течение длительного времени.

И, наконец, заметьте, что вы можете удобно выполнять итерацию по элементам массива, используя оператор C# foreach. Это работает потому, что System.Array реализует IEnumerable. Мне еще будет, что сказать о IEnumerable и его родственнике IEnumerator в разделе "IEnumerable<T>, IEnumerator<T>, IEnumerable и IEnumerator".

Неявно типизированные массивы

C# 3.0 представил сокращенный способ инициализации массивов, когда конкретный тип массива может быть выведен во время выполнения. Взглянем на новый синтаксис на примере следующего фрагмента кода:

```
using System;
public class EntryPoint
{
    static void Main() {
        // Традиционный массив
        int[] conventionalArray = new int[] { 1, 2, 3 };
        // Неявно типизированный массив
        var implicitlyTypedArray = new [] { 4, 5, 6 };
        Console.WriteLine( implicitlyTypedArray.GetType() );
        // Массив double
        var someNumbers = new [] { 3.1415, 1, 6 };
        Console.WriteLine( someNumbers.GetType() );
        // Не компилируется!
        // var someStrings = new [] { "int",
        // someNumbers.GetType() };
    }
}
```

Здесь для сравнения первая переменная массива по имени `conventionalArray` использует один из вариантов традиционного синтаксиса объявления и инициализации массива. Однако следующая переменная — `implicitlyTypedArray` — применяет новый сокращенный синтаксис, лишенный информации о типе.

Вместо того чтобы предоставить компилятору информацию о типе, я просто предоставляю ему возможность догадаться, что каждый элемент в массиве будет иметь тип `int`. И чтобы еще сэкономить нажатия клавиш и облегчить мне жизнь, `implicitlyTypedArray` объявляется как неявно типизированная локальная переменная. Если вы выполните этот код, то обнаружите, что вызов метода `WriteLine()`, следующий за ней, показывает, что неявно типизированная переменная имеет тип `System.Int32[]`. Фактически, вы могли бы выразить ту же строку кода следующим образом:

```
int[] implicitlyTypedArray = new [] { 4, 5, 6 };
```

Однако, поскольку вы уже предоставили компилятору возможность догадаться о типе элементов массива, вы можете также пойти немного дальше и позволить ему вывести весь тип массива, особенно если переменная остается локальной по отношению к контексту метода. Но что случится, если вы объявите массив, используя множество типов в списке инициализации?

Когда компилятор встречает множество типов внутри списка инициализации неявно типизированного массива, он определяет тип, к которому все элементу могут быть неявно приведены. И, конечно, для практических целей любой тип экземпляра может конвертироваться к своему собственному типу. Поэтому в объявлении экземпляра `someNumbers` в данном примере компилятор определит, что все типы внутри скобок являются преобразуемыми к `System.Double[]`. Не удивительно, что следующий вызов метода `WriteLine()` подтверждает это. Но что, если типы элементов не могут быть неявно преобразованы к одному общему типу?

Когда компилятор не может найти подходящий общий тип, к которому можно преобразовать все элементы массива, он выдает предупреждение CS0826:

```
No best type found for implicitly typed array
Не найдено подходящего типа для неявно типизированного массива
```

Если вы удали комментарий со строки, где объявляется переменная someStrings, то столкнетесь с таким поведением, поскольку экземпляры System.Type не могут быть неявно преобразованы к System.String.

Но что случится, если вы объявите массив с двумя элементами, и оба типа окажутся преобразуемыми друг к другу? Такая ситуация случается очень редко, и обычно является следствием наличия двух пользовательских типов, у которых определены операции неявного преобразования друг к другу[1]. Чтобы увидеть, что произойдет, я так и сделал. Попытавшись объявить неявно типизированный массив с элементами каждого из типов, я столкнулся с ошибкой компиляции CS0826, как и следовало ожидать.

Наверно, вы уже думаете о множестве полезных приложений для неявно типизированных массивов. Но в большинстве случаев они просто экономят вам клавиатурный набор[2]. Это удобно, если ваш массив содержит закрытые обобщенные типы, которые требуют значительного клавиатурного набора для ввода их типа. Потому в этом отношении неявно типизированные массивы позволяют написать более читабельный код. Но в других случаях они, наоборот, могут затруднить понимание кода для инженеров поддержки, если знание типа массива в точке его объявления существенно для понимания работы кода. Однако, поскольку неявно типизированные массивы в действительности являются неявно типизированными локальными переменными, то если только метод, который вы читаете, не слишком сложный и большой, у вас не должно быть проблем в определении действительного типа такой переменной. Если же вы не можете сразу догадаться о типе, читая код, это может означать, что функция слишком сложна и нуждается в рефакторизации.

С учетом всего сказанного, можно согласиться, что неявно типизированные массивы отлично подходят для создания экземпляров n-кратных элементов. Например, следующий фрагмент кода демонстрирует сокращенный способ объявления матрицы целых чисел:

```
using System;
public class EntryPoint {
    static void Main() {
        var threeByThree = new [] {
            new [] { 1, 2, 3 },
            new [] { 4, 5, 6 },
            new [] { 7, 8, 9 }
        };
        foreach( var i in threeByThree ) {
            foreach( var j in i ) {
                Console.Write( "{0}, ", j );
            }
            Console.Write( "\n" );
        }
    }
}
```

[1] В главе 6 я показал, как можно определять собственные операции явного и неявного преобразования типов.

[2] Неявно типизированные массивы также очень полезны при использовании с LINQ. Фактически, то же можно сказать о большинстве новых средств C# 3.0. Взятые по отдельности, они обладают минимальной ценностью, но в целом при использовании с LINQ они обеспечивают возможность создание исключительно выразительных конструкций. Я расскажу о LINQ в главе 16.

Конвертируемость и ковариантность

Когда вы объявляете массив, способный содержать экземпляры определенного типа, то экземпляры, которые вы можете поместить в такой массив, в действительности могут быть экземплярами производного типа. Например, если вы создаете массив экземпляров типа `Animal`, то в него с успехом можно поместить экземпляры `Dog` или `Cat`, если оба они наследуются от `Animal`.

На заметку! В C/C++ сохранение экземпляров `Dog` (собака) или `Cat` (кошка) в массивы типа `Animal` категорически не одобряется, поскольку объекты, если они хранятся по значению, усекаются, и поэтому "кошачесть" или "собачесть" животного окажутся отброшенными, а останется только то, что делает их животными. В C# все не так, поскольку массив содержит ссылки на объекты в куче. Если проводить аналогию с массивами C/C++, то массивы C# подобны массивам C/C++, содержащим указатели на `Cat` и `Dog` в виде указателей на объекты `Animal`.

Вы можете объявлять типы массивов другим, еще более интересным способом:

```
using System;
public class Animal { }
public class Dog : Animal { }
public class Cat : Animal { }
public class EntryPoint
{
    static void Main() {
        Dog[] dogs = new Dog[ 3 ];
        Cat[] cats = new Cat[ 2 ];
        Animal[] animals = dogs;
        Animal[] moreAnimals = cats;
    }
}
```

Присваивание `dogs` и `cats` переменной `animals` — это нечто такое, что вы определенно не можете делать в "родном" C/C++. Массивы являются присваиваемыми до тех пор, пока их размерность совпадает, а типы элементов являются конвертируемыми друг в друга. Такая возможность присвоения массивов в CLR обеспечивается тем фактом, что массивы ковариантны, а не инвариантны. Поскольку оба массива в предыдущем примере имеют размерность 1, а типы `Dog` и `Cat` являются конвертируемыми в `Animal`, такое присвоение работает. Создатели C# включили поддержку ковариантных массивов в CLR прежде всего для того, чтобы обеспечить поддержку языка Java.

На заметку! Полная информация о типе массива включает его ранг (количество измерений) и тип содержимого.

Возможности сортировки и поиска

Если взглянуть на полный интерфейс `System.Array`, как он описан в документации MSDN, вы заметите, что несколько методов позволяют сортировать элементы в массиве. Эти методы применимы, если содержащийся тип реализует `IComparable` — стандартный интерфейс сравнения элементов определенного

типа[3]. Естественно, вы не можете сортировать многомерные массивы, и если попытаетесь, то будьте готовы перехватить исключение RankException.

К тому же, если вы попытаетесь сортировать массив, в котором один или более типов не поддерживают IComparable, то можете ожидать исключения типа InvalidOperationException. Поэтому всегда представляйте, что может пойти не так, когда вы вызываете методы массивов, которые могут выполнять операции, недопустимые в некоторых случаях.

Используя статические методы Index() и LastIndexOf(), вы можете искать определенные значения внутри массива. Если такой метод не может найти запрошенное значение, он возвращает -1. Эти методы не предполагают никакого определенного алгоритма, кроме того, что первый из них начинает поиск с начала, а второй — с конца массива. Если вам нужно выполнять ускоренный поиск, вы можете применить статический метод BinarySearch. Однако прежде чем сделать это, массив необходимо отсортировать, и, конечно, это потребует от элементов массива реализации интерфейса IComaparable.

Синхронизация

Не раз и не два вы столкнетесь с необходимостью синхронизировать доступ к массиву или типу коллекции, реализующим ICollection[4]. Тип System.Array реализует как ICollection, так и IList. Одним из свойств ICollection является IsSynchronized, которое всегда возвращает false для обычных массивов. Это объясняется тем, что обычные массивы не синхронизированы по умолчанию, поскольку это потребовало бы излишних расходов в тех случаях, когда синхронизация не нужна. Поэтому вы должны управлять синхронизацией самостоятельно.

Простейший способ управления синхронизацией заключается в применении класса System.Monitor, который вы обычно используете через ключевое слово C# lock. Этот класс позволяет вам запросить встроенную блокировку синхронизации на объекте[5]. Однако вместо запроса блокировки на самом объекте массива вы должны пользоваться блокировкой на объекте ICollection.SyncRoot.

На заметку! Вы можете запросить блокировку на любом объекте, имеющем ссылку в CLR. Каждый объект оснащен "лениво" создаваемым блоком синхронизации, который содержит переменную блокировки, которой CLR управляет внутренне, когда System.Monitor пытается запросить блокировку.

Многие массивы и реализации коллекций могли бы возвращать ссылку на действительный контейнер через свойство ICollection.SyncRoot, но не делают этого по разным причинам. ICollection.SyncRoot представляет общий способ для синхронизированного доступа как к массивам, так и к коллекциям. Мне еще будет, что сказать о синхронизации, когда речь пойдет об интерфейсе ICollection в разделе "Синхронизация коллекций".

[3] Также доступен интерфейс IComparable<T> — обобщенная форма IComparable.

[4] В главе 12 раскрывается тема синхронизации и параллелизма в подробностях, вместе с темой многопоточности .NET Framework.

[5] Подробнее о System.Monitor и прочих приемах синхронизации читайте в главе 12.

Векторы против массивов

Интересно отметить, что CLR поддерживает два специальных типа для обращения с массивами в коде C#. Если ваш массив одномерный, и его нижняя граница равна 0, что обычно верно для массивов C#[6], то CLR использует специальный встроенный тип под названием `vector`, который на самом деле является подтипом `System.Array`. CLR поддерживает специальные инструкции IL, предназначенные для прямого обращения с векторами. Если ваш массив многомерный, то тип вектора CLR не используется, а вместо него применяется обычный объект массива. Чтобы продемонстрировать это, взглянем на некоторый код IL, сгенерированный из следующего короткого примера:

```
public class EntryPoint
{
    static void Main() {
        int val = 123;
        int newVal;
        int[] vector = new int[1];
        int[,] array = new int[1,1];
        vector[0] = val;
        array[0,0] = val;
        newVal = vector[0];
        newVal = array[0,0];
    }
}
```

Посмотрим на сгенерированный код IL для метода `Main`:

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 46 (0x2e)
    .maxstack 4
    .locals init ([0] int32 val,
                  [1] int32 newVal,
                  [2] int32[] 'vector',
                  [3] int32[0...,0...] 'array')
    IL_0000: nop
    IL_0001: ldc.i4.s 123
    IL_0003: stloc.0
    IL_0004: ldc.i4.1
    IL_0005: newarr [mscorlib]System.Int32
    IL_000a: stloc.2
    IL_000b: ldc.i4.1
    IL_000c: ldc.i4.1
    IL_000d: newobj instance void int32[0...,0...]::.ctor(int32,int32)
    IL_0012: stloc.3
    IL_0013: ldloc.2
    IL_0014: ldc.i4.0
```

[6] Массивы, объявленные в синтаксисе массивов C#, всегда имеют нижнюю границу 0. Если вам нужен массив с ненулевым начальным индексом, вы должны создать его экземпляр посредством метода `System.Array.CreateInstance()`.

```
    IL_0015: ldloc.0
    IL_0016: stelem.i4
    IL_0017: ldloc.3
    IL_0018: ldc.i4.0
    IL_0019: ldc.i4.0
    IL_001a: ldloc.0
    IL_001b: call instance void int32[0...,0...]::Set(int32,
                                                      int32,
                                                      int32)

    IL_0020: ldloc.2
    IL_0021: ldc.i4.0
    IL_0022: ldelem.i4
    IL_0023: stloc.1
    IL_0024: ldloc.3
    IL_0025: ldc.i4.0
    IL_0026: ldc.i4.0
    IL_0027: call instance int32 int32[0...,0...]::Get(int32,int32)
    IL_002c: stloc.1
    IL_002d: ret
} // end of method EntryPoint::Main
```

Обратите внимание на различие в использовании двух массивов C#. В строке `IL_0005` инструкция `newarr` создает экземпляр, представленный переменной `vector`. Многомерный массив, хранящийся в переменной `array`, создается в строке `IL_000d`. В первом случае "родная" инструкция IL выполняет операцию, в то время как вызов обычного конструктора обрабатывает операцию во втором случае. Аналогично при обращении к элементам IL-инструкции `stelem` и `ldelem` используются для вектора, тогда как обычные вызовы методов обрабатывают доступ к элементам многомерного массива.

Поскольку поддержка вектора обрабатывается специфическими инструкциями IL, предусмотренными специально для векторов, можно предположить, что использование вектором более эффективно, чем многомерных массивов, даже несмотря на то, что экземпляры обоих наследуются от `System.Array`.

Многомерные прямоугольные массивы

C# и CLR содержат прямую поддержку многомерных массивов, также известных под названием *прямоугольных массивов*, которые представляют собой нечто такое, что не поддерживается напрямую в C и C++. На C# вы можете легко объявить массив с несколькими измерениями. Просто вставляйте запятую между размерностями внутри квадратных скобок, как показано в следующем примере:

```
using System;
public class EntryPoint
{
    static void Main() {
        int[,] twoDim1 = new int[5,3];
        int[,] twoDim2 = { {1, 2, 3},
                           {4, 5, 6},
                           {7, 8, 9} };
```

```
        foreach( int i in twoDim2 ) {
            Console.WriteLine( i );
        }
    }
}
```

Здесь нужно отметить несколько моментов, касающихся использования прямоугольных массивов. Все обращения к этим массивам сводятся к вызову методов сгенерированного CLR ссылочного типа, и встроенный тип вектора здесь не участвует. Обратите внимание на два объявления. В каждом случае вам не нужно указывать размер каждого измерения при объявлении типа. Опять-таки это объясняется тем, что массивы типизированы на основе типа элементов и размерности (ранга). Однако, создав экземпляр типа массива, вы должны указать размер измерений. В данном примере я сделал это двумя разными способами. При создании twoDim1 я указал их явно, а при создании twoDim2 компилятор определяет размеры на основании выражения инициализации.

В данном примере я перечислил все элементы массива, используя цикл foreach. Этот цикл проходит по всем элементам массива, строка за строкой. Я мог бы достичь той же цели с помощью двух вложенных циклов for, и мне пришлось бы сделать это также в том случае, если бы я хотел пройти по элементам массива в любом другом порядке. Делая это, имейте в виду, что свойство Array.Length возвращает общее количество элементов в массиве. Чтобы получить размер каждого измерения, вы должны применить метод Array.GetLength, указав интересующее вас измерение. Например, я мог бы выполнить итерацию по элементам массива, используя следующий синтаксис, и результат был бы тем же самым:

```
using System;
public class EntryPoint
{
    static void Main() {
        int[,] twoDim = { {1, 2, 3},
                          {4, 5, 6},
                          {7, 8, 9} };
        for( int i = 0; i != twoDim.GetLength(0); ++i ) {
            for( int j = 0; j != twoDim.GetLength(1); ++j ) {
                Console.WriteLine( twoDim[i,j] );
            }
        }
        for( int i = twoDim.GetLowerBound(0);
                 i <= twoDim.GetUpperBound(0);
                 ++i ) {
            for( int j = twoDim.GetLowerBound(1);
                     j <= twoDim.GetUpperBound(1);
                     ++j ) {
                Console.WriteLine( twoDim[i,j] );
            }
        }
    }
}
```

Для наглядности я показал, как можно выполнять итерацию по размерностям массива двумя разными методами. Первый метод предполагает, что нижняя граница каждого измерения равна 0, а второй — нет. При всех вызовах GetLength(), GetUpperBound() и GetLowerBound() вы должны указывать номер интересующего измерения, начиная с нуля.

На заметку! Все массивы, создаваемые внутри C# с использованием стандартного синтаксиса объявления массивов, будут иметь нижнюю границу индекса, равную 0. Однако если вы имеете дело с массивами, используемыми в математических целях, а также с массивами, пришедшими из сборок, написанных на других языках, то нужно учитывать, что нижняя граница индекса может и не быть равной нулю.

Когда вы обращаетесь к элементам многомерного массива, компилятор генерирует вызовы методов Get и Set, которые подобны GetValue() и SetValue(). Эти методы перегружены для приема переменного списка целых чисел, специфицирующих порядок каждой размерности массива.

При отображении многомерных массивов на математические концепции, прямоугольный массив является наиболее естественным и предпочтительным для использования. Однако создание метода, чей аргумент может быть массивом переменной размерности, является непростой задачей, поскольку вы должны принимать аргумент типа System.Array и динамически обращаться с его размерностью. Размерность массива можно получить с помощью свойства Array.Rank. Поэтому создание обобщенного для размерности кода непросто, из-за синтаксической сложности обращения к элементам массива через вызовы методов System.Array, хотя это и вполне возможно. Более того, наиболее обобщенный код манипуляций с массивами должен учитывать возможность ненулевых границ индексов по отдельным размерностям.

Многомерные зубчатые массивы

Если вы пришли из мира C/C++ или Java, то вероятно, уже знакомы с зубчатыми (jagged) массивами, поскольку эти языки не поддерживают прямоугольных массивов, как это делает C#. Единственный способ реализации многомерных массивов в этих языках — это создавать массивы массивов, а именно это и представляют собой зубчатые массивы. Однако поскольку каждый элемент массива верхнего уровня является отдельным экземпляром массива, каждый экземпляр массива верхнего уровня может быть любого размера. Поэтому такой массив не обязательно прямоуголен — отсюда и термин *зубчатые массивы*.

Синтаксический шаблон объявления зубчатого массива в C# подобен тому, что принят в C++ и Java. Следующий пример показывает, как развернуть и использовать зубчатый массив:

```
using System;
using System.Text;
public class EntryPoint
{
    static void Main() {
        int[][] jagged = new int[3][];
        jagged[0] = new int[] { 1, 2};
```

```
        jagged[1] = new int[] {1, 2, 3, 4, 5};
        jagged[2] = new int[] {6, 5, 4};

        foreach( int[] ar in jagged ) {
            StringBuilder sb = new StringBuilder();
            foreach( int n in ar ) {
                sb.AppendFormat( "{0} ", n );
            }
            Console.WriteLine( sb.ToString() );
        }
        Console.WriteLine();
        for( int i = 0; i < jagged.Length; ++i ) {
            StringBuilder sb = new StringBuilder();
            for( int j = 0; j < jagged[i].Length; ++j ) {
                sb.AppendFormat( "{0} ", jagged[i][j] );
            }
            Console.WriteLine( sb.ToString() );
        }
    }
}
```

Как видите, выделение и создание зубчатого массива несколько сложнее, чем массива прямоугольного, потому что вы должны обрабатывать выделения всех подмассивов индивидуально, в то время как прямоугольный массив выделяется за один прием. Обратите внимание на "зубчатый" вывод, который объясняется разными размерами каждого подмассива:

```
1 2
1 2 3 4 5
6 5 4
```

В данном примере я демонстрирую два способа выполнения итерации по массиву — просто чтобы показать синтаксис доступа к индивидуальным элементам внутри зубчатого массива и его отличие от обращения к элементам прямоугольного массива. Синтаксис подобен принятому в С++ и Java. Метод `foreach` для итерации по массиву более элегантен, и как я расскажу далее, его применение позволяет вам использовать тот же код для итерации по коллекциям, которые могут и не быть массивами.

На заметку! Для итерации по массивам и коллекциям предпочтительно применять `foreach`. В результате вы можете изменять позднее тип контейнера, и до тех пор, пока он поддерживает интерфейс `IEnumerable`, блок `foreach` может не меняться. Если же вместо этого вы используете цикл `for`, вам, возможно, придется изменить метод доступа к каждому индивидуальному элементу. Вдобавок `foreach` работает и с массивами, у которых нижняя граница индекса не равна 0.

Часто имеет смысл использовать зубчатые массивы вместо прямоугольных. Например, вы можете читать информацию из базы данных, и каждый элемент массива верхнего уровня может представлять коллекцию, в которой каждая подколлекция может иметь широко варьируемое количество элементов. Если большинство подколлекций содержат лишь по несколько элементов, а одна из них — 100 элементов, то прямоугольный массив в этом случае тратит огромное количество мес-

та в памяти впустую, поскольку ему нужно выделять 100 вхождений в каждой из подколлекций, независимо от их реального количества. Зубчатые массивы обычно более эффективно используют память, но зато обращение к их элементам требует большей осторожности, поскольку вы не можете предположить, что каждый подмассив имеет в себе одно и то же количество элементов.

На заметку! Зубчатые массивы потенциально могут быть более эффективными, поскольку обычно состоят из одномерных массивом с нулевым минимальным индексом, которые CLR представляет в виде векторов, как было описано выше в настоящей главе.

Типы коллекций

С самого своего появления .NET Framework включал множество типов коллекций, предназначенных для управления всем — от расширяемых массивов `ArrayList`, очередей `Queue`, стеков `Stack` и даже словарей — через класс `HashTable`. С годами новые версии .NET Framework расширили и усовершенствовали эти типы. В общем случае коллекция — это любой тип, который может содержать в себе наборы объектов и реализует интерфейс `IEnumerable` или `IEnumerable<T>`. Объекты в таком наборе обычно имеют между собой отношения, определяемые проблемной областью.

Я предполагаю, что вы уже знакомы с не обобщенными типами коллекций и интерфейсами коллекций, доступными в .NET 1.1, а именно — определенными в пространствах имен `System.Collections` и `System.Collections.Specialized`. В MSDN вы найдете массу документации на эту тему. Поэтому дальше я буду называть старые типы коллекций не обобщенными типами коллекций, чтобы отличать их от новых типов коллекций и интерфейсов, определенных в пространствах имен `System.Collections.Generic` и `System.Collections.ObjectModel`.

Сравнение `ICollection<T>` с `ICollection`

Наиболее очевидные дополнения к типам коллекций, появившимся в .NET 2.0 Framework, являются типы, определенные внутри пространства имен `System.Collections.Generic`. Эти типы строго типизированы, что предоставляет компилятору больше возможностей для обеспечения безопасности типов за счет выявления ошибок несоответствия типов на этапе компиляции. Вдобавок, когда они используются для хранения типов значений, они намного более эффективны, поскольку исключают необходимость в упаковке. Возможно, корень всей системы обобщенных типов коллекций — это `ICollection<T>`. Приведу здесь его объявление:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
    bool IsReadOnly { get; }
    void Add( T item );
    void Clear();
    bool Contains( T item );
    void CopyTo( T[] array, int arrayIndex );
    bool Remove( T item );
}
```

Для сравнения приведу также определение необобщенного интерфейса ICollection:

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo( Array array, int index );
}
```

Рассмотрим отличия между ними и поговорим о том, что они означают для вашего кода. Одна вещь, которой недостает необобщенным коллекциям — это универсальный интерфейс управления содержимым коллекции. Например, оба не обобщенных типа — Stack и Queue — имеют метод Clear для очистки своего содержимого. Как можно ожидать, оба они реализуют интерфейс ICollection. Однако поскольку ICollection не содержит никаких модифицирующих методов, обычно вы не можете полиморфно трактовать в коде экземпляры этих двух типов. Поэтому вы всегда вынуждены выполнять приведение переменной экземпляра к типу Stack, чтобы вызвать Stack.Clear(), и приводить к типу Queue, чтобы вызвать Queue.Clear().

ICollection<T> помогает решить эту проблему, объявляя методы для модификации коллекции. Как в большинстве решений общего применения, это не обязательно применимо ко всем ситуациям. Например, ICollection<T> также объявляет свойство IsReadOnly, поскольку иногда вам нужно в своем дизайне представить неизменяемую коллекцию. Для таких экземпляров вызовы Add(), Clear() и Remove() приведут к генерации исключения InvalidOperationException.

На заметку! В интересах производительности я рекомендую, чтобы вызывающий код определял, разрешены ли такие операции, проверяя свойство IsReadOnly и таким образом вообще избегая исключений.

Поскольку главное назначение ICollection<T> — обеспечить более высокую безопасность типов, для ICollection<T> имеет смысл предоставление собственной строго типизированной версии CopyTo().

В то время как ICollection.CopyTo() знает, что его первый параметр — массив, ICollection<T>.CopyTo() также известна его размерность и тип содержащихся элементов. Ясно, что вы можете передавать методу ICollection<T>.CopyTo() только одномерные массивы.

Фактически не обобщенный метод ICollection.CopyTo() также принимает только одномерные массивы, но поскольку компилятор не может определить размерность типа System.Array во время компиляции, вы получите исключение времени выполнения ArgumentException, если передадите массив с более, чем одним измерением правильной реализации ICollection.CopyTo(). Обратите внимание, что я сказал "правильной реализации". Не только тот, кто вызывает ICollection.CopyTo() должен знать это правило, но также и тип, реализующий ICollection. Дополнительная информация о типе в ICollection<T>.CopyTo() не только предохранит от ошибки того, кто вызывает этот метод и того, кто его реализует, но и обеспечивает повышенную эффективность.

Вы заметите, что все обобщенные типы коллекций реализуют как ICollection<T>, так и ICollection. Оба интерфейса представляют собой удобный доступ к типу контейнера. Любые методы в ICollection, перекрывающие ICollection<T>, должны быть реализованы явно.

На заметку! При определении ваших собственных типов коллекций вы должны наследовать их от Collection<T> из пространства имен Collection.ObjectModel, если только у вас нет веской причины поступить иначе. Например, Collection<T> может предоставлять некоторую функциональность, которая вам не нужна, или же вы хотите реализовать собственный способ хранения элементов в коллекции. Если вы не наследуетесь от Collection<T>, ваша работа будет более трудной, поскольку вам придется реализовать большую часть того, что уже реализовано в Collection<T>.

Синхронизация коллекций

В ICollection присутствует одно средство, которого недостает его обобщенному аналогу, а именно — обеспечение обработки многопоточной синхронизации для всех коллекций. По умолчанию большинство типов коллекций не синхронизированы. Вы можете обратиться к свойству IsSynchronized для определения того, синхронизирована ли коллекция. В большинстве случаев, включая System.Array, ответ будет отрицательным (false). Однако иногда вам потребуется синхронизация при обращении к коллекции из нескольких потоков.

Есть два способа контролировать синхронизацию коллекций, возвращающих false из ICollection.IsSynchronized. Основной способ — использовать свойство ICollection.SyncRoot, возвращающее объект, который вы можете впоследствии применить с System.Monitor — обычно через оператор C# lock — чтобы защитить доступ к коллекции. Такой способ обеспечивает большую гибкость доступа к коллекции, поскольку вы тонко контролируете моменты захвата и освобождения блокировки. Однако на вас возлагается обязанность обеспечения правильной обработки блокирования, поскольку коллекция не пытается самостоятельно захватывать блокировку.

На заметку! Выбор способа реализации синхронизации — классический пример компромиссного технического решения, которое приходится принимать при проектировании новых коллекций, реализующих ICollection. Вы можете реализовать синхронизацию внутри коллекции, но тогда клиенты, которым она не нужна, будут нести излишние затраты. Вы можете также вынести синхронизацию наружу, реализуя ICollection.SyncRoot, но тогда на клиента возлагается обязанность правильно управлять синхронизацией. Вы должны учитывать особенности домена вашего приложения, делая тот или иной выбор.

В некоторых случаях типы коллекций просто возвращают для ICollection.SyncRoot ссылку this. Поэтому важно, чтобы вы никогда не синхронизировали доступ к коллекции, передавая ссылку на нее System.Monitor. Вместо этого всегда используйте объект, полученный через свойство SyncRoot, даже несмотря на то, что оно может в действительности вернуть this.

В качестве альтернативы ручному управлению SyncLock большинство не обобщенных типов коллекций из стандартной библиотеки реализуют метод Synchronized, который возвращает объект-оболочку коллекции, управляющий за вас блокировкой синхронизации. Вы можете попробовать применить такой же

шаблон при создании собственных типов коллекций. Используя оболочку, возвращенную методом Synchronized, клиентский код, работающий с этой коллекцией, не должен изменяться, чтобы позволить работу в многопоточной среде. При реализации ваших собственных коллекций всегда позволяйте клиентам выбирать возможность использования синхронизации, и никогда не навязывайте ее обязательное применение.

Списки

Одной вещью, которой недостает ICollection<T>, и на то есть веская причина, является операция индекса, которая позволяет обращаться к элементам внутри коллекции с использованием знакомого синтаксиса доступа к массиву. Дело в том, что не всем конкретным типам, реализующим ICollection<T>, нужна операция индекса, а в некоторых случаях для них и не имеет смысла в такой операции. Например, операция индекса для списка целых чисел могла бы, вероятно, иметь параметр типа int, в то время как тип словаря должен был бы принимать параметр типа, совпадающего с типом ключа такого словаря.

Если вы определяете коллекцию, для которой имеет смысл проиндексировать элементы, тогда вам нужно будет, чтобы эта коллекция реализовала IList<T>. Конкретные обобщенные типы коллекций-списков обычно реализуют интерфейсы IList<T> и IList. Интерфейс IList<T> реализует ICollection<T>, а интерфейс IList реализует ICollection, поэтому любой тип, являющийся списком, также является коллекцией. Интерфейс IList<T> выглядит следующим образом:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    T this[ int index ] { get; set; }
    int IndexOf( T item );
    void Insert( int index, T item );
    void RemoveAt( int index );
}
```

Интерфейс IList немного больше:

```
public interface IList : ICollection, IEnumerable
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[ int index ] { get; }
    int Add( object value );
    void Clear();
    bool Contains( object value );
    int IndexOf( object value );
    void Insert( int index, object value );
    void Remove( object value );
    void RemoveAt( int index );
}
```

Как видите, интерфейсы IList<T> и IList частично перекрываются, но в IList присутствует множество полезных свойств и методов, которых не может иметь такой обобщенный контейнер, как IList<T>, или любой другой обобщен-

ный контейнер, созданный вами. Как в случае с ICollection<T> и ICollection, типичный подход заключается в реализации обоих интерфейсов. Вы должны явно реализовать методы IList, которые перекрываются по функциональности соответствующие методы IList<T>, чтобы единственным способом обратиться к ним было явное преобразование экземпляра к типу IList.

На заметку! Обычно при реализации ваших собственных списочных типов вы должны наследовать реализацию от ICollection<T> из пространства имен System.Collections.ObjectModel.

Словари

В .NET 2.0 Framework появился тип IDictionary<TKey, TValue> — обобщенный, а потому строго типизированный аналог IDictionary. Как обычно, конкретные типы, реализующие IDictionary<TKey, TValue>, должны также реализовать IDictionary. Между ними есть много перекрытий, но обобщенный интерфейс объявляет более безопасные в отношении типов версии некоторых свойств и методов, объявленных в IDictionary. Однако в IDictionary<TKey, TValue> присутствует также и новый метод по имени TryGetValue, который вы можете использовать для попытки получения значения на основе заданного ключа. Этот метод возвращает значение в выходном параметре, а само возвращаемое значение говорит о том, присутствует ли элемент с таким ключом в словаре. Хотя вы можете добиться того же, используя операцию индекса и перехватывая исключение KeyNotFoundException, когда элемента в словаре нет, всегда более эффективным будет код, избегающий исключений, если вы знаете, что искомого элемента может и не оказаться в словаре. Во-первых, исключения не эффективны для управления потоком выполнения, поскольку дорого обходятся. Во-вторых, при этом игнорируется тот факт, что исключение должно быть действительно исключительным событием. При использовании исключений для управления потоком выполнения вы применяете исключения для обработки ожидаемых событий. Вы обнаружите больше случаев такого шаблона вызовов методов Try... по всему .NET Framework, поскольку команда разработчиков .NET предприняла значительные усилия для устранения узких мест эффективности вроде этого.

На заметку! При реализации обобщенных словарей у вас есть два варианта выбора — от чего наследовать реализацию. Для начала, вы можете использовать SortedDictionary<TKey, TValue>, который обеспечивает время доступа O(log n) и реализует IDictionary<TKey, TValue> в качестве интерфейса коллекции. Однако вы можете отдать предпочтение KeyedCollection<TKey, TValue> из пространства имен System.Collections.ObjectModel. Хотя этот класс на самом деле не реализует интерфейсов словарей, он обеспечивает время доступа O(1) в большинстве случаев. Подробности вы найдете в документации MSDN.

Наборы

.NET 3.5 Framework также вводит еще один полезный класс коллекций, известный как HashSet, определенный в пространстве имен System.Collections.Generic. Класс HashSet реализует типичные операции над множествами, которых

можно было ожидать. Например, вы можете вызывать метод IntersectWith для модификации текущего набора, чтобы он содержал пересечение текущего множества элементов с элементами, содержащимися в заданном экземпляре типа IEnumerable<T>. И наоборот, UnionWith модифицирует текущий набор таким образом, чтобы он включил объединение двух наборов. К другим полезным методам относятся IsSubsetOf, IsSupersetOf, ExceptWith, SymmetricExceptWith, Contains и т.д. Это лишь несколько из удобных методов, доступных в наборах.

На заметку! Обратите внимание, что различные методы операций над множествами, реализованные HashSet, принимают параметры типа IEnumerable<T>. Это очень удобно, поскольку позволяет вам использовать любой тип коллекций в качестве параметра этих методов, а не только экземпляры HashSet.

Как это принято в операциях над множествами, вы можете добавлять только уникальные значения в экземпляры HashSet. Например, если вы уже добавили значения 1, 2 и 3 к экземпляру HashSet<int>, то вы уже не сможете вставить другое целое число, равное одному из этих значений. По этой причине метод Add возвращает булевское значение, указывающее на то, удалась операция или нет. Было бы неэффективно генерировать исключения в таких случаях, поэтому результат обозначается возвращаемым значением метода Add.

System.Collections.ObjectModel

Для тех из вас, кому придется определять собственные типы коллекций, наиболее удобными типами окажутся те, что определены в пространстве имен System.Collections.ObjectModel. Фактически, если возможно, вы должны наследовать свои реализации от объектов из этого пространства имен. Оно содержит только три типа, и сам факт существования такого пространства имен является предметом дискуссии. Было три главных причины вынести эти три типа в отдельное пространство имен. Во-первых, в среде Visual Basic уже есть тип Collection, реализованный в пространстве имен, импортируемым по умолчанию, и команда Visual Basic предположила, что пользователи VB будут введены в заблуждение, увидев два типа с одинаковыми именами, но с совершенно отличающимся поведением, когда они появятся в IntelliSense. Во-вторых, команда, занимающаяся разработкой базовой библиотеки классов (BCL), решила, что пользователям редко понадобятся типы из этого пространства имен. Так ли это — время покажет. По моему мнению, эти типы исключительно полезны для написания библиотек кода, используемого другими. Одно из руководств Microsoft даже советует создавать подклассы этих типов при разработке коллекций, даже если только для того, чтобы представить более выразительное имя, описывающее коллекцию, и в качестве удобной исходной точки для расширений.

Эти типы весьма удобны, если вы определяете собственные типы коллекций. Вы можете легко наследовать свой тип от Collection<T>, чтобы получить поведение коллекции по умолчанию, включая реализации ICollection<T>, IList<T> и IEnumerable<T>. Collection<T> также реализует не обобщенные интерфейсы ICollection, IList и IEnumerable. Однако вам может понадобиться явно привести тип к одному из этих интерфейсов, чтобы получить доступ к его свойствам и методам, поскольку многие из них реализованы явно. Более того, тип Collection<T>

использует шаблон NVI[7] для обеспечения производных типов набором защищенных виртуальных методов, которые вы можете переопределить. Я не стану здесь приводить полный интерфейс Collection<T>, поскольку вы можете найти все подробности в документации MSDN. Однако защищенные виртуальные методы, которые вы можете переопределить, все-таки покажу:

```
public class Collection<T> : ICollection<T>, IList<T>, IEnumerable<T>,
                             ICollection, IList, IEnumerable
{
    ...
    protected virtual void ClearItems();
    protected virtual void InsertItem( int index, T item );
    protected virtual void RemoveItem( int index );
    protected virtual void SetItem( int index, T item );
    ...
}
```

Вы не можете модифицировать местоположение хранилища, переопределяя эти методы. Collection<T> управляет хранилищем элементов, и эти элементы содержатся в приватном поле типа IList<T>. Однако вы можете переопределить эти методы, чтобы управлять дополнительной информацией, порожденной этими операциями. Только не забывайте вызывать версии методов базового класса в своих переопределениях.

И, наконец, тип Collection<T> представляет два конструктора: один создает пустой экземпляр, а другой принимает IList<T>. Последний копирует переданное значение экземпляра IList<T> в новую коллекцию в порядке, определяемом перечислителем, полученным от IList<T>.GetEnumerator(). Важно упомянуть об этом порядке, поскольку вы увидите способ управления им в следующем разделе, где речь пойдет о блоках перечислителей и итераторов. Реализация перечислителя исходного списка умеет много вещей, таких как обращение порядка элементов на противоположный тому, в котором они помещались в коллекцию, просто предоставлением правильной реализации перечислителя. Лично я полагаю, что у Collection<T> должны были бы быть еще конструкторы, принимающие IEnumerator<T> и IEnumerable<T>, чтобы обеспечить больше гибких способов наполнения коллекции. Вы можете самостоятельно решить эту проблему, добавив дополнительные конструкторы к типам, производным от Collection<T>, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
public class MyCollection<T> : Collection<T>
{
    public MyCollection() : base() {
    }
    public MyCollection( IList<T> list )
        : base(list) { }
    public MyCollection( IEnumerable<T> enumerable )
        : base() {
```

[7] Шаблон NVI будет описан в главе 13.

```
        foreach( T item in enumerable ) {
            this.Add( item );
        }
    }
    public MyCollection( IEnumerator<T> enumerator )
        : base() {
        while( enumerator.MoveNext() ) {
            this.Add( enumerator.Current );
        }
    }
}
public class EntryPoint
{
    static void Main() {
        MyCollection<int> coll =
            new MyCollection<int>( GenerateNumbers() );

        foreach( int n in coll ) {
            Console.WriteLine( n );
        }
    }
    static IEnumerable<int> GenerateNumbers() {
        for( int i = 4; i >= 0; -i ) {
            yield return i;
        }
    }
}
```

В Main вы можете видеть экземпляр MyCollection<int>, созданный передачей ему типа IEnumerable<int>, возвращенного методом GenerateNumbers. Если ключевое слово yield в методе GenerateNumbers покажется вам незнакомым, это может объясняться тем, что данное средство было добавлено в C# 2.0. Чуть позже в этой главе я объясню смысл этого ключевого слова. По сути, оно определяет то, что называется *блоком итератора*, который создает из кода сгенерированный компилятором перечислитель. После создания конструирования экземпляра типа MyCollection<int> вы можете обращаться к нему исключительно через ссылку на Collection<T>. В конце концов, MyCollection<T> есть Collection<T>. Кстати, я не позаботился о создании конструкторов, принимающих не обобщенные IEnumerable и IEnumerator, просто потому, что предпочитаю обеспечить более строгую безопасность типов.

Возможно, вы отметили присутствие List<T> в пространстве имен System.Collections.Generic. Было бы соблазнительно использовать List<T> в ваших приложениях всякий раз, когда нужно предоставить потребителям обобщенный список. Однако вместо использования List<T> обратитесь к Collection<T>. Тип List<T> не реализует защищенных виртуальных методов, реализуемых Collection<T>. Поэтому, если вы наследуете ваш список от List<T>, ваш производный тип не имеет возможности определить, где именно были произведены модификации списка. С другой стороны, List<T> служит отличным инструментом, когда вам нужно встроить простую спискообразную реализацию внутрь типа, по-

скольку он свободен от таких вызовов виртуальных методов, как у Collection<T>, и, как следствие, более эффективен.

Другим полезным типом из пространства имен System.Collections.ObjectModel является тип-оболочка, который вы можете использовать для реализации коллекций, доступных только для чтения. Поскольку в языке C# отсутствует ключевое слово const, как в C++, важно иметь возможность создавать при необходимости неизменяемые типы и передавать их методам в виде константных параметров. Конструктор ReadOnlyCollection<T> принимает тип параметра IList<T>. Поэтому вы можете использовать ReadOnlyCollection<T> в качестве оболочки любого типа, реализующего IList<T>, включая Collection<T>. Естественно, если пользователь обратится к свойству ICollection<T>.IsReadOnly, он получит ответ true. При каждой попытке вызова модифицирующего метода, такого как ICollection<T>.Clear(), будет сгенерировано исключение NotSupportedException. Более того, для того, чтобы вызвать модифицирующий метод, ссылка ReadOnlyCollection<T> должна быть приведена к интерфейсу, объявляющему этот метод, поскольку ReadOnlyCollection<T> реализует все модифицирующие методы явно. Самый большой выигрыш от явной реализации этих методов заключается в том, что это помогает избежать их использования во время компиляции.

Эффективность

Имея выбор, вы всегда должны предпочитать обобщенные типы коллекций не обобщенным версиям — из-за дополнительной безопасности типов и повышенной эффективности. Рассмотрим аргумент эффективности более подробно. Обобщенные типы коллекций, хранящих типы значений, позволяют исключить излишнюю упаковку и распаковку. Упаковка — определенно более дорогая операция, чем распаковка, поскольку требует выделения памяти в куче, а распаковка — нет. Рико Мариани (Rico Mariani) указывает на многие другие узкие места эффективности в своем блоге *Rico Mariani's Performance Tidbits*[8]. Он подчеркивает, что команды разработчиков тратят массу времени на решение проблем производительности, и упрощение всегда помогает в этом. Он приводит один блестящий пример, доказывающий, что List<T> существенно быстрее, чем ArrayList, когда используется много операций foreach. Однако повышенная скорость объясняется не очевидными причинами, связанными с отсутствием упаковки/распаковки, а скорее тем, что ArrayList использует "бесплатный" набор виртуальных методов — особенно во время перечислений. ArrayList.GetEnumerator() — виртуальный метод, а вложенный тип перечисления ArrayListEnumeratorSimple также виртуально реализует метод MoveNext() и свойство Current. Это добавляет много дорогостоящих вызовов виртуальных методов при перечислении. Если только вы интенсивно не используете перечисления ArrayList, то, возможно, и не заметите ущерба для производительности, но это лишь демонстрирует, насколько много внимания команда разработчиков BCL уделила вопросам эффективности в последнее время.

Это — отличный пример того, почему вам стоит тщательно анализировать дизайн своего класса, чтобы оправдать его наследуемость. Не делайте метод виртуальным, если только точно не уверены в том, что кому-то понадобится переоп-

[8] Вы найдете блог Рико по адресу http://blogs.msdn.com/ricom/.

ределять его, и если уж объявили его виртуальным, убедитесь, что использовали шаблон NVI, описанный в главе 13. Я твердо уверен в том, что вам следует ориентироваться на создание герметизованных (sealed) классов, если только вы абсолютно не уверены в наличии причин, по которым кто-нибудь захочет наследоваться от вашего класса. Если вы не можете придумать убедительную причину для этого, не оставляйте класс открытым для наследования только потому, что считаете, что такая необходимость может у кого-нибудь возникнуть в будущем. Если убедительной причины для наследования нет, то маловероятно, чтобы вы создавали свой класс с прицелом на наследование, и может оказаться, что он не будет работать, как ожидается, в своих наследниках. Наследуемость должна быть осознанным решением, а не основанным лишь на интуиции.

На заметку! Даже если ваш класс наследуется от класса, использующего виртуальные методы, будет более эффективным, если вы объявите его как герметизованный, потому что компилятор в этом случае сможет вызывать их не виртуальным способом при вызове через ссылку производного типа.

Добавлю еще одно предупреждение к прозвучавшим ранее: бесплатное использование обобщений или любого средства подобного рода без знания их ограничений — всегда плохо. Всякий раз, когда конструируется конкретный тип на основе обобщенного, исполняющая система должна генерировать его код в памяти. К тому же, полностью сконструированные типы, созданные из обобщенных типов со статическими полями, получают каждый свою копию набора статических полей. Поэтому, если обобщение содержит поле вроде такого:

```
public class MyGeneric<T>
{
   public static int staticField;
}
```

то `MyGeneric<int>.staticField` и `MyGeneric<long>.staticField` будут ссылаться на разные места в памяти. Мораль этой истории в том, что вы должны учитывать инженерные компромиссы. Хотя обобщения помогают избежать упаковки и распаковки, и обычно порождают более эффективный код, они также увеличивают размер рабочего набора вашего приложения. Если есть сомнения, то применение инструментов анализа производительности помогут принять правильное решение.

IEnumerable<T>, IEnumerator<T>, IEnumerable и IEnumerator

Вы уже видели, как можно использовать оператор C# `foreach` для удобного выполнения итерации по коллекции объектов, включая `System.Array`, `ArrayList`, `List<T>` и т.п. Как он работает? Ответ состоит в том, что каждая коллекция, которая должна работать с `foreach`, должна реализовать интерфейс `IEnumerable<T>` или `IEnumerable`, который используется `foreach` для получения объекта, знающего, как перечислить, или выполнить итерацию, по элементам коллекции. Объект итератора, полученный от `IEnumerable<T>`, должен реализовать интерфейс `IEnumerator<T>` или `IEnumerator`. Обобщенные типы коллекций обычно

реализуют IEnumerator<T>, а объект перечислителя реализует IEnumerator<T>. IEnumerable<T> наследуется от IEnumerable, а IEnumerator<T> — от IEnumerator. Это позволяет вам применять обобщенные коллекции там же, где используются не обобщенные. Строго говоря, ваши типы коллекций не обязаны реализовать перечислители, и пользователь может выполнять итерации в цикле for, если вы предусмотрите операцию индекса, например, реализуя интерфейс IList<T>. Однако, поступая подобным образом, вы не заведете много друзей, и как только я продемонстрирую, насколько легко создавать перечислители с помощью блоков итераторов, вы убедитесь, что реализовать IEnumerable<T> и IEnumerator<T> — пара пустяков.

Многие из вас уже знакомы с не обобщенными интерфейсами перечислителей и с тем, как реализовать перечислители на ваших типах коллекций. В оставшейся части раздела я кратко пройдусь по обычно умалчиваемым моментам создания перечислителей "с нуля", и кратко опишу новый усовершенствованный способ создания перечислителей с использованием блоков итераторов. Если хотите, можете пропустить эту часть и перейти к следующему разделу об итераторах. Если же вы хотите освежить в памяти тему реализации перечислителей, читайте все.

Интерфейс IEnumerable<T> существует для того, чтобы клиенты имели четко определенный способ получения перечислителя для коллекции. В следующем коде определяются интерфейсы IEnumerable<T> и IEnumerable:

```
public interface IEnumerable<T> : IEnumerable
{
   IEnumerator<T> GetEnumerator();
}
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

Поскольку оба интерфейса реализуют GetEnumerator() с одной и той же сигнатурой перегрузки (напомню, что тип возвращаемого значения не учитывается при разрешении перегрузки), любая коллекция, реализующая IEnumerable<T>, должна явно реализовать метод GetEnumerator. Больше всего имеет смысл явно реализовать не обобщенный метод IEnumerable.GetEnumerator. Интерфейсы IEnumerator<T> и IEnumerator выглядят так:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
   T Current { get; }
}
public interface IEnumerator
{
   object Current { get; }
   bool MoveNext();
   void Reset();
}
```

Эти два интерфейса реализуют член с одинаковой сигнатурой — в данном случае свойство Current. При реализации IEnumerator<T> вы должны явно реали-

зовать IEnumerator.Current. К тому же заметьте, что IEnumerator<T> реализует интерфейс IDisposable. Позднее я объясню, чем это хорошо.

А теперь я продемонстрирую, как реализовать IEnumerable<T> и IEnumerator<T> для доморощенного типа коллекции. Хороший учитель всегда покажет вам, как что-то сделать "трудным способом", прежде чем представить "легкий способ". Я думаю, этот прием полезен, поскольку позволяет понять, что происходит "за кулисами". Когда вы понимаете, как работает внутренний механизм, вы лучше подготовлены к тому, чтобы иметь дело с техническими нюансами "легкого способа". Рассмотрим пример реализации IEnumerable<T> и IEnumerator<T> "трудным способом" для доморощенной коллекции целых чисел. Я покажу, как реализовать обобщенные версии, поскольку это предполагает также реализацию не обобщенных версий. В данном примере я не стану реализовывать ICollection<T>, чтобы не загромождать пример, а сосредоточиться только на интерфейсах перечисления.

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;
public class MyColl<T> : IEnumerable<T>
{
public MyColl( T[] items ) {
this.items = items;
}
public IEnumerator<T> GetEnumerator() {
return new NestedEnumerator( this );
}
IEnumerator IEnumerable.GetEnumerator() {
return GetEnumerator();
}
// Определение перечислителя.
class NestedEnumerator : IEnumerator<T>
{
    public NestedEnumerator( MyColl<T> coll ) {
        Monitor.Enter( coll.items.SyncRoot );
        this.index = -1;
        this.coll = coll;
    }
    public T Current {
        get { return current; }
    }
    object IEnumerator.Current {
        get { return Current; }
    }
    public bool MoveNext() {
        if( ++index >= coll.items.Length ) {
            return false;
        } else {
            current = coll.items[index];
            return true;
        }
    }
```

```
    public void Reset() {
        current = default(T);
        index = 0;
    }
    public void Dispose() {
        try {
            current = default(T);
            index = coll.items.Length;
        }
        finally {
            Monitor.Exit( coll.items.SyncRoot );
        }
    }
    private MyColl<T> coll;
    private T current;

    private int index;
    }
    private T[] items;
}
public class EntryPoint
{
    static void Main() {
        MyColl<int> integers =
            new MyColl<int>( new int[] {1, 2, 3, 4} );
        foreach( int n in integers ) {
            Console.WriteLine( n );
        }
    }
}
```

На заметку! В наиболее реальных случаях вы унаследуете ваш пользовательский класс коллекции от `Collection<T>` и получите реализацию `IEnumerable<T>` бесплатно.

Этот пример кода инициализирует внутренний массив внутри `MyColl<T>` случайным набором целых чисел, так что перечислитель получит некоторые данные, чтобы ему было с чем "поиграть". Конечно, реальный контейнер должен реализовать `ICollection<T>`, чтобы позволить динамически наполнять коллекцию элементами. Оператор `foreach` разворачивается в код, получающий перечислитель вызовом метода `GetEnumerator` на интерфейсе `IEnumerable<T>`. Компилятор достаточно интеллектуален, чтобы использовать в данном случае `IEnumerator<T>.GetEnumerator()` вместо `IEnumerator.GetEnumerator()`. Получив перечислитель, он запускает цикл, в котором сначала вызывает `MoveNext()`, а затем инициализирует переменную n значением, возвращенным от свойства `Current`. Если цикл не содержит других путей выхода, он продолжается до тех пор, пока `MoveNext()` не вернет `false`. В этот момент перечислитель завершает перечислять элементы коллекции, и вы должны вызвать `Reset()` на перечислителе, чтобы можно было использовать его снова.

Несмотря на то что вы можете создавать и использовать перечислитель явно, я рекомендую вместо этого применять конструкцию `foreach`. В этом случае при-

дется писать меньше кода, что означает меньшую вероятность непреднамеренного внесения ошибок. Конечно, вы можете иметь веские причины для непосредственной манипуляции перечислителем. Например, ваш перечислитель может реализовывать методы, специфичные для вашего конкретного типа перечислителя, которые вам нужно вызывать в процессе прохода по коллекции. Если вы должны манипулировать перечислением напрямую, всегда делайте это внутри блока `using`, поскольку `IEnumerator<T>` реализует `IDisposable`.

Обратите внимание, что по умолчанию в перечислитель не встроено никаких средств синхронизации. Поэтому один поток может выполнять перечисление элементов коллекции, в то время как другой модифицирует ее. Если коллекция модифицируется в процессе ее перечисления, то перечислитель становится семантически некорректным, и последующее его использование может привести к неопределенному поведению. Если вы должны предохранить целостность в таких ситуациях, то вам может понадобиться, чтобы перечислитель блокировал коллекцию через объект, представленный свойством `ICollection.SyncRoot`. Очевидное место для получения блокировки должно быть в конструкторе перечислителя. Однако вы должны где-то также снимать такую блокировку. Вы уже знаете, что для выполнения такой детерминированной очистки нужно реализовать интерфейс `IDisposable`. Именно в этом заключается причина того, почему `IEnumerator<T>` реализует интерфейс `IDisposable`. Более того, код, сгенерированный оператором `foreach`, создает "за кулисами" блок `try/finally`, который вызывает `Dispose()` на перечислителе внутри блока `finally`. Вы можете увидеть эту технику в действии в моем предыдущем примере.

Типы, производящие коллекции

Я уже коснулся того факта, что содержимое коллекции может измениться, пока перечислитель проходит по коллекции. Если коллекция изменяется, это может сделать перечислитель недействительным. В следующем разделе, посвященном итераторам, я покажу, как можно создать перечислитель, блокирующий доступ к контейнеру, пока идет перечисление. Хотя подобное возможно, это может оказаться не лучшим решением с точки зрения эффективности. Например, что если на проход по всем элементам коллекции потребуется много времени? Цикл `foreach` может выполнять какую-то длительную обработку каждого элемента, и при этом для кого-то другого коллекция будет недоступной для модификаций.

В подобных случаях может иметь смысл, чтобы каждый цикл `foreach` выполнялся над копией коллекции вместо самой исходной коллекции. Если вы решите сделать это, вам нужно убедиться в том, что вы понимаете, что означает копирование коллекции. Если коллекция содержит типы значений, то под копией понимается "глубокая" копия — до тех пор, пока типы значений внутри нее не содержат внутри себя ссылочных типов. Если же коллекция хранит ссылочные типы, вам нужно решить, должна ли копия коллекции клонировать каждый из содержащихся в ней элементов. В любом случае было бы неплохо иметь руководство по проектированию, чтобы следовать ему и знать, когда нужно возвратить копию.

Существующее эмпирическое правило гласит, что при возврате типов коллекций из ваших типов посредством методов всегда должны возвращаться копии, а при обращении к ним через свойства — ссылки на сами оригиналы коллекций. Хотя это правило и не отлито в бронзе, и вы не обязаны ему следовать, в нем есть

определенный семантический смысл. Применение методов обычно указывает на то, что вы выполняете некоторого рода операции над типом и можете ожидать результата такой операции. С другой стороны, доступ через свойства указывает на то, что вы хотите напрямую получить доступ к внутреннему состоянию самого объекта. Именно поэтому данное эмпирическое правило имеет хороший семантический смысл. Вообще имеет смысл применять такое же семантическое разделение ко всем свойствам и методам внутри ваших типов.

Итераторы

В предыдущем разделе я привел краткий и легковесный пример создания перечислителя для типа коллекции. После того, как вы сделаете это несколько раз, подобная задача покажется вам рутинной. А всякий раз, когда такое происходит, людям свойственно допускать досадные ошибки. C# предлагает новую конструкцию, называемую блоком итератора, чтобы еще более облегчить эту задачу. Прежде чем я погружусь в детальное описание итераторов, давайте быстро взглянем, как можно с помощью итератора решить ту же задачу, что и в примере из предыдущего раздела. Это будет именно тем "легким способом", о котором я говорил:

```
using System;
using System.Collections;
using System.Collections.Generic;
public class MyColl<T> : IEnumerable<T>
{
    public MyColl( T[] items ) {
        this.items = items;
    }
    public IEnumerator<T> GetEnumerator() {
        foreach( T item in items ) {
            yield return item;
        }
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
    private T[] items;
}
public class EntryPoint
{
    static void Main() {
        MyColl<int> integers =
            new MyColl<int>( new int[] {1, 2, 3, 4} );
        foreach( int n in integers ) {
            Console.WriteLine( n );
        }
    }
}
```

Ничего не может быть проще. Обратите внимание, что реализация перечислителя из примера предыдущего раздела теперь сведена к трем строкам внутри ме-

тода GetEnumerator. Секрет — в ключевом слове yield. Наличие ключевого слова yield определят блок кода как yield-блок. Когда вы видите его впервые, может быть нелегко догадаться о том, что здесь происходит. При вызове GetEnumerator() код в методе, содержащем оператор yield, на самом деле не выполняется в этот момент времени. Это — экземпляр класса, который возвращается из метода. Таким образом, когда оператор foreach в методе Main() вызывается через методы IEnumerator<T>, используется код блока yield.

Здесь пропущена одна вещь из предыдущего раздела — синхронизация. Давайте посмотрим, как добавить синхронизацию к перечислителю, возвращенному блоком yield. Вот замена предыдущего метода GetEnumerator:

```
public IEnumerator<T> GetEnumerator() {
    lock( items.SyncRoot ) {
        for( int i = 0; i < items.Length; ++i ) {
            yield return items[i];
        }
    }
}
```

Удивительно просто, не правда ли? Для примера я изменил способ итерации по коллекции, используя цикл for вместо foreach. Теперь позвольте мне объяснить, что за волшебство здесь делает компилятор. Как и раньше, блок кода yield не выполняется немедленно. Вместо этого возвращается объект перечислителя. Внутренне перечислитель может находиться в одном из нескольких состояний. При первом вызове MoveNext() на перечислителе блок кода выполняется до достижения первого оператора yield. Каждый последующий вызов MoveNext() продолжает выполнение цикла либо до достижения оператора yield break, либо когда цикл дойдет до конца метода. Как только это случится, перечислитель попадает в финальное состояние, и вы не можете использовать его для дальнейшего перечисления коллекции. Фактически метод Reset не доступен для использования на перечислителе, сгенерированного блоком yield, и если вы вызовете его, будет сгенерировано исключение NotSupportedException. В конце перечисления любой блок finally внутри блока yield выполняется, как ожидалось. В данном случае это означает снятие блокировки, поскольку оператор C# lock "за кулисами" сводится к конструкции try/finally. К тому же, если перечислитель будет освобожден (disposed) до конца цикла, то компилятор достаточно интеллектуален, чтобы поместить код внутрь блока finally в реализацию Dicpose() перечислителя, чтобы блокировка всегда снималась.

Как видите, компилятор делает за вас массу незаметной работы, когда вы используете итераторы. В качестве финального примера я покажу еще один способ итерации по элементам этой коллекции:

```
public IEnumerator<T> GetEnumerator( bool synchronized ) {
    if( synchronized ) {
        Monitor.Enter( items.SyncRoot );
    }
    try {
        int index = 0;
        while( true ) {
            if( index < items.Length ) {
                yield return items[index++];
```

```
            } else {
               yield break;
            }
         }
      }
      finally {
         if( synchronized ) {
            Monitor.Exit( items.SyncRoot );
         }
      }
   }
   public IEnumerator<T> GetEnumerator() {
      return GetEnumerator( false );
   }
```

Это не слишком красивый способ итерации по элементам, но я хотел продемонстрировать пример использования оператора `yield break`. К тому же заметьте, что я создал новый метод `GetEnumerator`, принимающий признак `bool`, указывающий на то, желает ли вызывающий код получить синхронизированный или не синхронизированный перечислитель. Здесь важно отметить, что объект перечислителя, созданный компилятором, теперь имеет общедоступное поле по имени `synchronized`. Любые параметры, переданные методу, содержащему блок `yield`, добавляются в виде общедоступных полей в сгенерированный класс перечислителя.

На заметку! Поскольку перечислитель, сгенерированный из блока `yield`, захватывает локальные переменные и параметры, не допускается объявлять параметры `ref` и `out` в методах, реализующих блок `yield`.

Вы можете спорить, что добавленные поля должны быть приватными, а не общедоступными, поскольку если обратиться к этим общедоступным полям и модифицировать их во время перечисления, то перечислитель можно разрушить. В данном случае, если вы модифицируете поле `synchronized` во время перечисления и присвоите ему `false`, то другие сущности никогда не смогут получить доступ к коллекции, поскольку блокировка никогда не будет снята. Даже несмотря на то, что вы должны использовать рефлексию, чтобы получить доступ к общедоступным полям перечислителя, сгенерированного из блока `yield`, очень легко и опасно сделать это неправильно. Я не хочу сказать, что такая техника не может быть полезной, что докажет пример из раздела "Прямые, обратные и двунаправленные итераторы", где я продемонстрирую создание двунаправленного итератора.

Вы можете избежать этого клубка червей, введя пресловутый дополнительный уровень посредничества. Вместо того чтобы сделать параметр `GetEnumerator` типа `bool`, объявите его с пользовательским неизменяемым типом, таким как `ImmutableBool`. Следующий пример демонстрирует, как можно сделать это, чтобы избежать доступа внешних сущностей к общедоступным полям сгенерированного компилятором перечислителя:

```
using System;
using System.Threading;
using System.Reflection;
using System.Collections;
using System.Collections.Generic;
```

```
public struct ImmutableBool
{
    public ImmutableBool( bool b ) {
        this.b = b;
    }
    public bool Value {
        get { return b; }
    }
    private bool b;
}
public class MyColl<T> : IEnumerable<T>
{
    public MyColl( T[] items ) {
        this.items = items;
    }
    public IEnumerator<T> GetEnumerator(
                        ImmutableBool synchronized ) {
        if( synchronized.Value ) {
            Monitor.Enter( items.SyncRoot );
        }
        try {
            foreach( T item in items ) {
                yield return item;
            }
        }
        finally {
            if( synchronized.Value ) {
                Monitor.Exit( items.SyncRoot );
            }
        }
    }
    public IEnumerator<T> GetEnumerator() {
        return GetEnumerator( new ImmutableBool(false) );
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
    private T[] items;
}
public class EntryPoint
{
    static void Main() {
        MyColl<int> integers =
            new MyColl<int>( new int[] {1, 2, 3, 4} );

        IEnumerator<int> enumerator =
            integers.GetEnumerator( new ImmutableBool(true) );

        // Получить ссылку на поле synchronized
        object field = enumerator.GetType().
            GetField("synchronized").GetValue( enumerator );
```

```
        while( enumerator.MoveNext() ) {
            Console.WriteLine( enumerator.Current );
            // Генерирует исключение
            field.GetType().GetProperty("Value").
                SetValue( field, false, null );
        }
    }
}
```

В `Main` вы можете видеть, что я использую перечислитель напрямую вместо того, чтобы сделать это через `foreach`. Несмотря на то что это не рекомендуется, я все равно так поступаю, потому что в данном случае играю роль зловредного разработчика, который хочет изменить состояние перечислителя во время перечисления. Вы можете видеть в цикле `while`, что я пытаюсь изменить значение свойства на `ImmutableBool` в перечислителе с помощью рефлексии. Это приведет к генерации исключения, поскольку данное свойство не имеет метода доступа `set`.

На заметку! Те из вас, кто знаком с хитросплетениями рефлексии, знают, что для кода технически возможно модифицировать приватное поле внутри экземпляра `ImmutableBool`. Это, однако, потребует, чтобы код отвечал требованиям безопасности `ReflectionPermission` доступа к коду (code access security — CAS). А для этого лицо, запустившее этот код, должно получить соответствующие полномочия явным образом, что маловероятно. Описание CAS не входит в перечень тем этой книги, но все тонкие детали CAS, включая способы расширения его в соответствие с вашими потребностями, вы найдете в книге Брайана А. Ла-Маччиа (Brian A. LaMacchia) *.NET Framework Security* (Upper Saddle River, NJ: Pearson Education, 2002 г.).

Используя неизменяемый тип, вы можете положиться на то, что никто не сможет вмешаться в состояние перечислителя и сделать что-то нехорошее.

До сих пор вы видели, как блоки итератора могут быть удобны для создания перечислителей. Однако также вы можете использовать их для генерации перечислимого типа. Например, предположим, что вы хотите выполнить итерацию по первым нескольким степеням двойки. Это можно сделать так:

```
using System;
using System.Collections.Generic;
public class EntryPoint
{
    static public IEnumerable<int> Powers( int from,
                                           int to ) {
        for( int i = from; i <= to; ++i ) {
            yield return (int) Math.Pow( 2, i );
        }
    }
    static void Main() {
        IEnumerable<int> powers = Powers( 0, 16 );
        foreach( int result in powers ) {
            Console.WriteLine( result );
        }
    }
}
```

В данном примере компилятор генерирует единственный тип, реализующий четыре интерфейса: IEnumerable<int>, IEnumerable, IEnumerator<int> и IEnumerator. Поэтому этот тип выступает и как перечислимый, и как перечислитель. В итоге любой метод, содержащий блок yield, должен возвращать тип IEnumerable<int>, IEnumerable, IEnumerator<int> или IEnumerator, где T — yield-тип метода. Компилятор сделает остальное. Я рекомендую придерживаться обобщенных версий, поскольку они позволяют избежать необходимости в упаковке типов значений и обеспечивают механизму контроля типов больше возможностей. В предыдущем примере значения from и to сохраняются в общедоступных полях перечислимого типа, как было показано ранее в этом разделе. Поэтому вы можете решить поместить их в оболочку неизменяемого типа, если нужно предотвратить возможность пользователям изменять их во время перечисления.

Совет. В книге Кшиштофа Квалины (Krzysztof Cwalina) и Брэда Абрамса (Brad Abrams) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Boston, MA: Addison-Wesley Professional, 2005 г.) предполагается, что тип никогда не должен реализовывать одновременно IEnumerable<T> и IEnumerator<T>, поскольку отдельный тип должен быть семантически либо коллекцией, либо перечислителем, но не тем и другим сразу. Однако объекты, сгенерированные блоками yield, нарушают это правило. Для закодированных вручную коллекций вы должны попытаться следовать данному правилу, даже несмотря на то, что C# его нарушает, чтобы сделать блоки yield более полезными.

Прямые, обратные и двунаправленные итераторы

Многие библиотеки, поддерживающие итераторы на своих контейнерных типах, поддерживают три основных разновидности итераторов — в форме прямых, обратных и двунаправленных итераторов. Прямые итераторы — аналог обычных перечислителей, реализующих IEnumerator<T>, которые можно получить от методов GetEnumerator контейнерных типов библиотеки .NET. Однако что, если вам понадобится обратный итератор или двунаправленный итератор? C# позволяет создавать такие вещи легко и просто.

Чтобы получить обратный итератор для вашего контейнера, все, что вам нужно — это создать блок yield, проходящий циклом по элементам коллекции в обратном порядке. Что еще удобнее — обычно вы можете объявлять блок yield внешним по отношению к вашей коллекции, как показано в следующем примере:

```
using System;
using System.Collections.Generic;
public class EntryPoint
{
    static void Main() {
        List<int> intList = new List<int>();
        intList.Add( 1 );
        intList.Add( 2 );
        intList.Add( 3 );
        intList.Add( 4 );
        foreach( int n in CreateReverseIterator(intList) ) {
            Console.WriteLine( n );
        }
    }
```

```
    static IEnumerable<T> CreateReverseIterator<T>( IList<T> list ) {
        int count = list.Count;
        for( int i = count-1; i >= 0; -i ) {
            yield return list[i];
        }
    }
}
```

Все "мясо" этого примера содержится в методе CreateReverseIterator<T>. Этот метод работает только с коллекциями типа IList<T>, но вы можете легко написать другую форму CreateReverseIterator<T>, принимающую какой-то другой тип коллекций. Когда вы создаете служебные методы такого рода, всегда лучше делать их насколько возможно обобщенными в отношении принимаемых типов. Например, можно было ли сделать CreateReverseIterator<T> методом еще более общего назначения, чтобы он принимал тип ICollection<T>? Нет, потому что ICollection<T> не объявляет операцию индекса. IList<T>, однако, делает это.

Теперь обратим наше внимание на двунаправленный итератор. Чтобы сделать из перечислителя двунаправленный итератор, нужно иметь возможность переключать его направление. Как я указывал ранее, перечислители, создаваемые из методов, принимающих параметры и содержащих блок yield, обладают общедоступными полями, которые вы можете модифицировать. Хотя для доступа к ним приходится использовать рефлексию, все же это можно делать. Для начала давайте взглянем на возможный сценарий использования для двунаправленного итератора:

```
static void Main() {
    LinkedList<int> intList = new LinkedList<int>();
    for( int i = 1; i < 6; ++i ) {
        intList.AddLast( i );
    }
    BidirectionalIterator<int> iter =
        new BidirectionalIterator<int>(intList,
                                       intList.First,
                                       TIteratorDirection.Forward);
    foreach( int n in iter ) {
        Console.WriteLine( n );
        if( n == 5 ) {
            iter.Direction = TIteratorDirection.Backward;
        }
    }
}
```

Вам нужен способ создания объекта итератора, поддерживающего IEnumerable<T>, чтобы затем использовать его внутри оператора foreach для запуска перечисления. В любой момент внутри блока foreach вам необходима возможность переключать направление итерации. В следующем примере демонстрируется класс BidirectionalIterator, который усовершенствует предыдущую модель использования.

```
public enum TIteratorDirection {
    Forward,
    Backward
};
```

```
public class BidirectionalIterator<T> : IEnumerable<T>
{
    public BidirectionalIterator( LinkedList<T> list,
                                  LinkedListNode<T> start,
                                  TIteratorDirection dir ) {
        enumerator = CreateEnumerator( list,
                                       start,
                                       dir ).GetEnumerator();
        enumType = enumerator.GetType();
    }

    public TIteratorDirection Direction {
        get {
            return (TIteratorDirection) enumType.GetField("dir")
                         .GetValue( enumerator );
        }
        set {
            enumType.GetField("dir").SetValue( enumerator,
                                               value );
        }
    }
    private IEnumerator<T> enumerator;
    private Type enumType;
    private IEnumerable<T> CreateEnumerator( LinkedList<T> list,
                                             LinkedListNode<T> start,
                                             TIteratorDirection dir ) {
        LinkedListNode<T> current = null;
        do {
            if( current == null ) {
                current = start;
            } else {
                if( dir == TIteratorDirection.Forward ) {
                    current = current.Next;
                } else {
                    current = current.Previous;
                }
            }
            if( current != null ) {
                yield return current.Value;
            }
        } while( current != null );
    }

    public IEnumerator<T> GetEnumerator() {
        return enumerator;
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

С технической точки зрения, мне не обязательно было помещать перечислитель в оболочку класса BidirectionalIterator. Я мог бы обращаться к переменной направления напрямую, через рефлексию, внутри блока foreach. Однако для того, чтобы сделать это, коду внутри блока foreach понадобилось бы имя параметра, переданного в методе BidirectionalIterator.CreateEnumerator с блоком yield. Чтобы избежать такого противоестественного применения, я связал его внутри класса-оболочки BidirectionalIterator и предоставил свойство Direction для доступа к общедоступному полю перечислителя.

И, наконец, в приведенном ниже примере демонстрируется применение той же техники для реализации циклического перечислителя. Вы можете использовать его для таких вещей, как игровые циклы, где нужно выполнять итерацию независимо по коллекциям сущностей, обновляя их состояние при каждом проходе — до тех пор, пока не поступит запрос на выход.

```
using System;
using System.Collections;
using System.Collections.Generic;
public class EntryPoint
{
    static void Main() {
        LinkedList<int> intList = new LinkedList<int>();
        for( int i = 1; i < 6; ++i ) {
            intList.AddLast( i );
        }
        CircularIterator<int> iter =
            new CircularIterator<int>(intList,
                                      intList.First);
        int counter = 0;
        foreach( int n in iter ) {
            Console.WriteLine( n );
            if( counter++ == 100 ) {
                iter.Stop();
            }
        }
    }
}
public class CircularIterator<T> : IEnumerable<T>
{
    public CircularIterator( LinkedList<T> list,
                             LinkedListNode<T> start ) {
        enumerator = CreateEnumerator( list,
                                       start,
                                       false ).GetEnumerator();
        enumType = enumerator.GetType();
    }
    public void Stop() {
        enumType.GetField("stop").SetValue( enumerator, true );
    }
    private IEnumerator<T> enumerator;
    private Type enumType;
```

```
    private IEnumerable<T> CreateEnumerator( LinkedList<T> list,
                                             LinkedListNode<T> start,
                                             bool stop ) {
        LinkedListNode<T> current = null;
        do {
            if( current == null ) {
                current = start;
            } else {
                current = current.Next;
                if( current == null ) {
                    current = start;
                }
            }
            yield return current.Value;
        } while( !stop );
    }
    public IEnumerator<T> GetEnumerator() {
        return enumerator;
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

Я включил метод Stop в CircularIterator<T>, чтобы можно было легко остановить итерацию. Конечно, как и в примере с двунаправленным итератором, метод Stop использует рефлексию для установки общедоступного поля stop сгенерированного компилятором перечислителя. Уверен, что вы согласитесь, что можно найти намного более творческие применения блоков yield, чтобы создавать сложные пути итераций.

Инициализаторы коллекций

C# 3.0 вводит новый сокращенный синтаксис инициализации коллекций, подобный синтаксису инициализаторов объектов. Если экземпляр типа коллекции, которую вы пытаетесь инициализировать, реализует ICollection<T> из пространства имен System.Collections.Generic, вы можете применить новый синтаксис, показанный в следующем коде:

```
using System;
using System.Collections.Generic;
public class Employee
{
    public string Name { get; set; }
    }

public class CollInitializerExample
{
    static void Main() {
        var developmentTeam = new List<Employee> {
            new Employee { Name = "Michael Bolton" },
```

```
            new Employee { Name = "Samir Nagheenanajar" },
            new Employee { Name = "Peter Gibbons" }
        };
        Console.WriteLine( "Development Team:" );
        foreach( var employee in developmentTeam ) {
            Console.WriteLine( "\t" + employee.Name );
        }
    }
}
```

Здесь "за кулисами" компилятор генерирует изрядный объем кода, чтобы помочь вам в этом. Для каждого элемента в списке инициализации генерируется вызов метода `Add()`. Обратите внимание, что я также использовал новый синтаксис для инициализации каждого экземпляра в списке инициализации.

Как я уже упоминал, тип коллекции должен реализовывать `ICollection<T>`. Если он этого не делает, вы получите ошибки времени компиляции. Вдобавок коллекция должна реализовывать только одну специализацию `ICollection<T>`; т.е. она может реализовывать `ICollection<T>` только для одного типа `T`. И, наконец, каждый элемент списка инициализации коллекции должен быть неявно преобразуемым к типу `T`.

Резюме

В этой главе я представил краткий обзор работы массивов в CLR и C#, в плане подготовки к дискуссии об обобщенных типах коллекций. После обзора обобщенных типов коллекций, определенных в `System.Collections.Generic`, я коснулся вопросов эффективности и применимости и представил вам полезные типы, определенные в пространстве имен `System.Collections.ObjectModel`. Затем я обратился к теме перечислителей и показал, как можно создавать эффективные перечислители посредством `yield`-блоков итераторов, появившихся в языке C# 2.0. И, наконец, я продемонстрировал новый синтаксис C# 3.0, упрощающий инициализацию коллекций во время создания экземпляров. Хотя в этой главе я не погружался в подробности каждого из типов коллекций, все же после ее прочтения вы должны быть вооружены информацией, необходимой для сознательного выбора — когда и какой тип обобщенных коллекций следует использовать. Я советую обратиться к документации MSDN, где вы найдете все тончайшие подробности, касающиеся API-интерфейсов типов коллекций.

В следующей главе я расскажу о делегатах, событиях и анонимных методах. Анонимные методы — замечательное дополнение к языку. Они удобны для создания встроенного вызываемого кода в точке, где вы регистрируете код обратного вызова.

ГЛАВА 10

Делегаты, анонимные функции и события

Делегаты обеспечивают встроенный, поддерживаемый языком механизм определения и выполнения обратных вызовов. Их гибкость позволяет определять точную сигнатуру обратного вызова, и эта информация становится частью самого типа делегата. Анонимные функции — форма делегатов, позволяющая сократить некоторую часть синтаксиса делегатов, который во многих случаях бывает громоздким и утомительным[1]. На делегатах построена поддержка механизма событий в C# и платформе .NET. События представляют собой унифицированный шаблон привязки реализаций обратных вызовов — даже по нескольку экземпляров сразу — к коду, инициирующему обратный вызов.

Обзор делегатов

CLR предоставляет исполняющую систему, которая явно поддерживает гибкий механизм обратных вызовов. С начала времен — по крайней мере, со времен появления Windows — всегда существовала необходимость в функциях обратного вызова, которые система либо какая-то другая сущность вызовет в определенный момент времени, чтобы известить вас о чем-нибудь интересном. В конце концов, обратные вызовы являются удобным механизмом, посредством которого пользователи могут расширить функциональность компонента. Даже наиболее базовые компоненты приложения Win32 GUI — оконные процедуры — представляют собой функции обратного вызова, регистрируемые в системе. Система вызывает такую функцию всякий раз, когда нужно известить вас о происхождении некоторого события в окне. Этот механизм работает почти хорошо в программной среде на базе языка C.

Все стало несколько сложнее с распространением объектно-ориентированных языков вроде C++. Разработчики немедленно захотели, чтобы система могла вызывать методы экземпляров на объектах вместо глобальных функций или статических методов. Существует много решений этой проблемы. Но независимо от того, какое из них вы используете, в итоге все сводится к тому, что где-то кто-то должен хранить указатель на экземпляр объекта и вызывать метод экземпляра через этот указатель на него. Реализации обычно состоят из *переходника* (thunk), который представляет собой не что иное, как промежуточный блок данных или кода,

[1] Средство, еще лучшее, чем анонимные функции — это лямбда-выражения, которым я посвятил целую главу 15.

вызывающий метод экземпляра через указатель на него. Переходник — в действительности функция, зарегистрированная в системе. За многие годы в C++ было разработано немало изобретательных реализаций переходников. Ваш доверчивый автор с сентиментальной нежностью может вспомнить множество примеров такого дизайна.

В настоящее время делегаты являются предпочтительным способом реализации обратных вызовов в CLR. Экземпляр делегата — это, по сути, то же самое, что переходник, за исключением того, что он является полноправным гражданином в стране CLR. Фактически, когда вы объявляете делегата в своем коде, компилятор C# генерирует класс-наследник `MulticastDelegate`, и CLR реализует все интересные методы делегата динамически, во время выполнения. Вот почему вы не увидите никакого IL-кода, стоящего за методами делегата, если заглянете в компилированный модуль с помощью ILDASM.

Делегат содержит два полезных поля. Первое хранит ссылку на объект, а второе — указатель на метод. Когда вы вызываете делегат, вызывается метод экземпляра на содержащейся в нем ссылке. Однако если ссылка на объект равна `null`, то исполняющая система понимает это так, что метод является статическим. Более того, вызов делегата — синтаксически то же самое, что вызов обычной функции. Поэтому делегаты — блестящее средство реализации обратных вызовов.

Как вы можете видеть, делегаты представляет собой отличный механизм отделения метода, вызываемого на экземпляре от вызывающего кода. Фактически вызывающий делегат код не имеет понятия, да и не нуждается в знании того, вызывается метод экземпляра либо статический метод, или же какой именно экземпляра вызывается. Для вызывающего кода это просто вызов произвольного кода. Он может получить экземпляр делегата любым подходящим способом, при этом полностью абстрагируясь от сущности, которую он на самом деле вызывает. Задумайтесь на минутку об элементах пользовательского интерфейса в диалоге, таких как кнопка `Commit`, и о том, насколько много внешних сторон могут быть заинтересованы в знании факта выбора этой кнопки. Если класс, представляющий кнопку, должен напрямую вызывать эти заинтересованные стороны, ему нужно обладать подробными знаниями о компоновке этих заинтересованных сторон, или объектов, и знать, какие именно их методы должны быть вызваны. Ясно, что такое требование приводит к чрезмерной зависимости между кнопкой и заинтересованными сторонами, причем зависимость эта чрезвычайно сложна. И здесь на помощь приходят делегаты и разрывают эту связь. Теперь заинтересованные стороны должны только зарегистрировать делегат с кнопкой, которая предварительно сконфигурирована так, что может вызывать любые методы, которые ей нужны. Этот механизм отделения описывает события, как поддерживаемые CLR. Далее, в разделе "События" мне еще будет, что сказать на эту тему. А пока давайте двинемся дальше и посмотрим, как создаются и используются делегаты в C#.

Создание и использование делегатов

Объявления делегатов выглядят почти так же, как объявления абстрактных методов, за исключением того, что снабжаются одним дополнительным ключевым словом — `delegate`. Вот как выглядит правильное объявление делегата:

```
public delegate double ProcessResults( double x, double y );
```

Когда компилятор C# встречает эту строку, он определяет тип-наследник MulticastDelegate, который также реализует метод по имени Invoke, имеющий в точности ту же сигнатуру, что и метод, описанный в объявлении делегата. Для практических нужд этот класс выглядит следующим образом:

```
public class ProcessResults : System.MulticastDelegate
{
   public double Invoke( double x, double y );
   // Остальное пропущено для ясности
}
```

Несмотря на то что компилятор создает тип, подобный приведенному, он также абстрагирует использование делегатов за синтаксическими сокращениями. Фактически компилятор не позволяет напрямую вызывать метод делегата Invoke. Вместо этого вы применяете синтаксис, подобный вызову функции, который я продемонстрирую ниже.

Создавая экземпляр делегата, вы должны связать его вызов с вызовом метода. Метод, который вы привязываете, может быть как статическим, так и методом экземпляра, обладающим сигнатурой, совместимой с делегатом. Таким образом, типы параметров и тип возврата должны либо совпадать с объявлением делегата, либо быть неявно преобразуемыми в типы, указанные в объявлении делегата.

На заметку! В .NET 1.x сигнатура методов, привязанных к делегатам, должна была точно совпадать с сигнатурой объявления делегата. В .NET 2.0 это требование было ослаблено и теперь допускает привязывать к делегату метод с совместимыми типами в объявлении.

Одиночный делегат

В следующем примере демонстрируется базовый синтаксис создания делегата.

```
using System;
public delegate double ProcessResults( double x,
                                       double y );
   public class Processor
   {
      public Processor( double factor ) {
         this.factor = factor;
      }
      public double Compute( double x, double y ) {
         double result = (x+y)*factor;
         Console.WriteLine( "InstanceResults: {0}", result );
         return result;
      }
      public static double StaticCompute( double x,
                                          double y ) {
         double result = (x+y)*0.5;
         Console.WriteLine( "StaticResult: {0}", result );
      return result;
   }
   private double factor;
}
```

```
public class EntryPoint
{
   static void Main() {
      Processor proc1 = new Processor( 0.75 );
      Processor proc2 = new Processor( 0.83 );
      ProcessResults delegate1 = new ProcessResults( proc1.Compute );
      ProcessResults delegate2 = new ProcessResults( proc2.Compute );
      ProcessResults delegate3 = new ProcessResults( Processor.StaticCompute );

      double combined = delegate1( 4, 5 ) +
                        delegate2( 6, 2 ) +
                        delegate3( 5, 2 );
      Console.WriteLine( "Вывод: {0}", combined );
   }
}
```

В данном примере я создал три делегата. Два из них указывают на методы экземпляра, а один — на статический метод. Обратите внимание, что делегаты создаются посредством создания экземпляров типа `ProcessResults`, который является типом, созданным объявлением делегата. Когда вы создаете экземпляр делегата, то передаете ему в конструкторе указатель на метод, который он должен вызвать. В примере в первых двух случаях переданы методы экземпляров `proc1` и `proc2`. Однако в третьем случае передан указатель метода типа, а не экземпляра. Таким образом, создается делегат, указывающий на статический метод вместо метода экземпляра. В точке вызова делегата синтаксис идентичен и не зависит от того, указывает ли делегат на метод экземпляра или статический метод. Конечно, этот пример довольно надуманный, но он дает представление об основах применения делегатов в C#.

Во всех случаях предыдущего кода при вызове делегата происходит единственное действие. Но можно связывать делегаты в цепочки, выполняющие по нескольку действий сразу.

Цепочки делегатов

Цепочка делегатов позволяет создавать связный список делегатов, так что когда вызывается делегат, находящийся в начале списка, вслед за ним выполняются все делегаты цепочки. Класс `System.Delegate` предоставляет несколько статических методов для управления списком делегатов. Чтобы создавать списки делегатов, вы полагаетесь на следующие методы, объявленные в типе `System.Delegate`:

```
public class Delegate : ICloneable, ISerializable
{
   public static Delegate Combine( Delegate[] );
   public static Delegate Combine( Delegate first, Delegate second );
}
```

Обратите внимание, что методы `Combine` принимают делегатов для связи в цепочку и возвращают другой `Delegate`. Возвращенный `Delegate` представляет собой новый экземпляр `MulticastDelegate`. Это потому, что экземпляры `Delegate` трактуются как неизменяемые. Например, код, вызывающий `Combine`, может создать список делегатов, оставив исходные делегаты в том состоянии, в котором они

были. Единственный способ сделать это — при создании цепочки трактовать экземпляры делегатов как неизменные.

Заметьте, что первая версия Combine, приведенная выше, принимает массив делегатов, чтобы сформировать непрерывную цепочку делегатов, а вторая форма принимает только пару делегатов. Однако в обоих случаях любой из экземпляров делегата, переданного в параметрах, может сам быть началом цепочки делегатов. Так что, как видите, подобным образом можно получать весьма сложные вложенные структуры.

Чтобы удалить делегат из списка, можно обратиться к следующим статическим методам Systev.Delegate:

```
public class Delegate : IClonable, ISerializable
{
    public static Delegate Remove( Delegate source, Delegate value );
    public static Delegate RemoveAll( Delegate source, Delegate value );
}
```

Как и в случае методов Combine, Remove и RemoveAll возвращают новый экземпляр Delegate, созданный из двух предшествующих. Метод Remove удаляет последнее вхождение значения из исходного списка делегатов, в то время как RemoveAll удаляет все вхождения значений списка делегатов из исходного списка делегатов. Обратите внимание, что параметр value может представлять список делегатов вместо одиночного делегата. Эти методы позволяют осуществлять очень сложные сценарии управления списками делегатов.

Взглянем на модифицированную форму примера кода из предыдущего раздела, демонстрирующую комбинацию делегатов:

```
using System;
public delegate double ProcessResults( double x,
                                       double y );
   public class Processor
   {
      public Processor( double factor ) {
      this.factor = factor;
   }
   public double Compute( double x, double y ) {
      double result = (x+y)*factor;
      Console.WriteLine( "InstanceResults: {0}", result );
      return result;
   }
   public static double StaticCompute( double x,
                                       double y ) {
      double result = (x+y)*0.5;
      Console.WriteLine( "StaticResult: {0}", result );
      return result;
   }
   private double factor;
}
public class EntryPoint
{
   static void Main() {
```

```
        Processor proc1 = new Processor( 0.75 );
        Processor proc2 = new Processor( 0.83 );
        ProcessResults[] delegates = new ProcessResults[] {
                            new ProcessResults( proc1.Compute ),
                            new ProcessResults( proc2.Compute ),
                            new ProcessResults( Processor.StaticCompute )
        };

        ProcessResults chained =
            (ProcessResults) Delegate.Combine( delegates );
        double combined = chained( 4, 5 );

        Console.WriteLine( "Вывод: {0}", combined );
    }
}
```

Обратите внимание, что вместо вызова всех делегатов по одному этот пример связывает их вместе и затем вызывает в один прием — через начало цепочки. Этот пример демонстрирует некоторые существенные отличия от предыдущего, которые перечислены ниже.

- Результирующее значение double, полученное от вызова цепочки, является результатом вызова последнего делегата, которым в данном случае является делегат, указывающий на статический метод StaticCompute. Значения, возвращенные другими делегатами в цепочке, просто теряются.
- Если любой из делегатов сгенерирует исключение, обработка цепочки немедленно прерывается и CLR начинает искать следующий фрейм обработчика исключения в стеке.
- Имейте в виду, что если вы объявляете делегаты, принимающие параметры по ссылке, то каждый делегат, использующий ссылочный параметр, увидит изменения, внесенные предыдущими делегатами в цепочке. Это может быть желательным эффектом или же окажется сюрпризом — в зависимости от ваших намерений.
- И, наконец, отметьте, что перед вызовом цепочки делегатов вы должны привести делегат к точному типу делегата. Это необходимо для того, чтобы компилятор знал, как вызывать делегат. Значение, возвращенное из методов Combine и Remove, имеет тип System.Delegate, в котором нет достаточной информации для того, чтобы компилятор знал, как его следует вызвать.

Итерация по цепочкам делегатов

Иногда вам нужно вызывать цепочку делегатов, но при этом получить возвращаемые значения каждого из них, или явно специфицировать последовательность вызовов делегатов в цепочке. Для этого тип System.Delegate, от которого наследуются все делегаты, предусматривает метод GetInvocationList, позволяющий захватить массив делегатов, в котором каждый элемент соответствует делегату в списке вызова. Получив этот массив, вы можете вызывать делегаты в любом порядке, какой вам нравится, и обрабатывать возвращенные значения каждого делегата соответствующим образом. Вы можете также построить фрейм исключения вокруг каждого элемента списка, чтобы исключение, сгенерированное из одного

вызова делегата, не отменяло остальных вызовов. Следующая модифицированная версия предыдущего примера демонстрирует, как явно вызвать каждый делегат цепочки по отдельности:

```
using System;
public delegate double ProcessResults( double x,
                                       double y );
public class Processor
{
    public Processor( double factor ) {
        this.factor = factor;
    }
    public double Compute( double x, double y ) {
        double result = (x+y)*factor;
        Console.WriteLine( "InstanceResults: {0}", result );
        return result;
    }
    public static double StaticCompute( double x,
                                        double y ) {
        double result = (x+y)*0.5;
        Console.WriteLine( "StaticResult: {0}", result );
        return result;
    }
    private double factor;
}
public class EntryPoint
{
    static void Main() {
        Processor proc1 = new Processor( 0.75 );
        Processor proc2 = new Processor( 0.83 );
        ProcessResults[] delegates = new ProcessResults[] {
            new ProcessResults( proc1.Compute ),
            new ProcessResults( proc2.Compute ),
            new ProcessResults( Processor.StaticCompute )
        };
        ProcessResults chained = (ProcessResults) Delegate.Combine( delegates );
        Delegate[] chain = chained.GetInvocationList();
        double accumulator = 0;
        for( int i = 0; i < chain.Length; ++i ) {
            ProcessResults current = (ProcessResults) chain[i];
            accumulator += current( 4, 5 );
        }
        Console.WriteLine( "Вывод: {0}", accumulator );
    }
}
```

Несвязанные делегаты (открытые экземпляры)

Все примеры делегатов, приведенные до сих пор, показывают, как привязать делегат к статическому методу определенного типа или методу определенного экземпляра. Эта абстракция представляет замечательное средство разделения (decoupling) кода, но делегат в действительности не имитирует и не представляет

указателя на метод, поскольку привязывается к методу на определенном экземпляре. Что, если вы хотите, чтобы делегат представлял метод экземпляра, но вызывать этот метод через делегат на целой коллекции экземпляров?

Для решения этой задачи вам понадобится *делегат открытого экземпляра*. Когда вы вызываете метод на экземпляре, в начале списка параметров ему передается скрытый параметр, известный как this, представляющий текущий экземпляр[2]. Когда вы привязываете делегат закрытого экземпляра к методу экземпляра на объекте экземпляра, то при вызове метода экземпляра делегат принимает ссылку на этот объект в скрытом параметре this. В случае делегатов открытого экземпляра это действие возлагается на то, что вызывает делегат. Таким образом, вы можете указывать экземпляр объекта для вызова в момент вызова делегата.

Рассмотрим пример, чтобы увидеть, на что это похоже. Представьте коллекцию типов Employee и компанию, руководство которой решило в конце года повысить на 10% зарплату всем сотрудникам. Все объекты Employee содержатся в коллекции, и вам нужно выполнить итерацию по всем сотрудникам, обеспечив повышение вызовом метода Employee.ApplyRaiseOf.

```
using System;
using System.Reflection;
using System.Collections.Generic;
delegate void ApplyRaiseDelegate( Employee emp,
                                  Decimal percent );
   public class Employee
   {
      private Decimal salary;
      public Employee( Decimal salary ) {
      this.salary = salary;
   }
   public Decimal Salary {
      get {
         return salary;
      }
   }
   public void ApplyRaiseOf( Decimal percent ) {
      salary *= (1 + percent);
   }
}
public class EntryPoint
{
   static void Main() {
      List<Employee> employees = new List<Employee>();
      employees.Add( new Employee(40000) );
      employees.Add( new Employee(65000) );
      employees.Add( new Employee(95000) );
      // Создать делегат открытого экземпляра.
      MethodInfo mi =
         typeof(Employee).GetMethod( "ApplyRaiseOf",
                                     BindingFlags.Public |
                                     BindingFlags.Instance );
```

[2] Подробнее о this в связи со ссылочными типами и типами значений читайте в главе 4.

```
        ApplyRaiseDelegate applyRaise = (ApplyRaiseDelegate )
           Delegate.CreateDelegate( typeof(ApplyRaiseDelegate), mi );
        // Выполнить повышение.
        foreach( Employee e in employees ) {
           applyRaise( e, (Decimal) 0.10 );
           // Вывести значения новой зарплаты на консоль.
           Console.WriteLine( e.Salary );
        }
     }
  }
```

Для начала обратите внимание, что объявление делегата имеет тип Employee в начале списка параметров. Так вы описываете скрытый указатель экземпляра, чтобы можно было привязать его позже. При объявлении делегата закрытого экземпляра этот параметр Employee пропускается. К сожалению, язык C# не предусматривает специального синтаксиса для создания делегатов открытого экземпляра. Поэтому вы должны использовать одну из обобщенных перегрузок Delegate.CreateDelegate для создания экземпляра делегата, как показано в примере, и перед тем, как сделать это, вы должны воспользоваться рефлексией для получения экземпляра MethodInfo, представляющего привязываемый метод.

Ключевой момент, который нужно здесь отметить — при создании экземпляра делегата нигде не указывается конкретный экземпляр объекта. Вы не указываете его до самого момента вызова делегата. Цикл foreach показывает, как вызывается делегат и в то же время указывается экземпляр для вызова. Несмотря на то что метод ApplyRaiseOf, к которому привязан делегат, принимает только один параметр, вызов делегата требует двух параметров, так что вы можете указать экземпляр, на котором нужно выполнить вызов метода.

Предыдущий пример демонстрирует, как создается и вызывается делегат открытого экземпляра; однако делегат может быть более обобщенным и в широком смысле более полезным. В данном примере вы объявили делегат так, что он знает, что он должен вызывать метод типа Employee. Поэтому в момент вызова вы можете использовать только экземпляр Employee или типа, производного от Employee. Можно использовать обобщенный делегат для такого объявления делегата, чтобы тип, на котором он вызывается, не был указан во время объявления[3]. Такой делегат потенциально более полезен. Он позволяет вам заявить следующее: "Я хочу представить метод, соответствующий данной сигнатуре, который поддерживается любым, пока еще не указанным типом". Только в точке создания экземпляра делегата вы должны будете указать конкретный тип, который будет вызван. Рассмотрим следующую модификацию предыдущего примера:

```
delegate void ApplyRaiseDelegate<T>( T instance,
                                     Decimal percent );
public class EntryPoint
{
   static void Main() {
      List<Employee> employees = new List<Employee>();
      employees.Add( new Employee(40000) );
      employees.Add( new Employee(65000) );
      employees.Add( new Employee(95000) );
```

[3] Обобщения рассматриваются в главе 11.

```
        // Создать делегат открытого экземпляра.
        MethodInfo mi =
           typeof(Employee).GetMethod( "ApplyRaiseOf",
                                       BindingFlags.Public |
                                       BindingFlags.Instance );
        ApplyRaiseDelegate<Employee> applyRaise =
           (ApplyRaiseDelegate<Employee> )
           Delegate.CreateDelegate(
                    typeof(ApplyRaiseDelegate<Employee>), mi );
        // Выполнить повышение.
        foreach( Employee e in employees ) {
           applyRaise( e, (Decimal) 0.10 );
           // Вывести значения новой зарплаты на консоль.
           Console.WriteLine( e.Salary );
        }
    }
}
```

Теперь делегат получился намного более общим. Можно представить, как он может оказаться удобным в разных ситуациях. Например, представьте графическую программу, которая поддерживает применение фильтров к различным объектам на холсте. Предположим, что вам нужен делегат для представления обобщенного типа фильтра, применение которого сопряжено с указанием некоторого процентного значения, задающего степень применения определенного эффекта к объекту. Используя обобщенные делегаты открытого экземпляра, вы можете реализовать такую общую концепцию.

События

Во многих случаях, когда вы используете делегаты в качестве механизма обратного вызова, вам может понадобиться просто известить кого-то о наступлении некоторого события вроде щелчка на кнопке в пользовательском интерфейсе. Предположим, что вы проектируете приложение медиа-проигрывателя. Где-то в пользовательском интерфейсе (UI) есть кнопка "Play" (Воспроизведение). В хорошо спроектированной системе UI отделен от логики управления посредством четкой абстракции — обычно реализуемой через шаблон Bridge (Мост). Абстракция облегчает последующее изменение UI или, что еще лучше, поскольку UI зависит от платформы — облегчает перенос приложения на другую платформу. Например, шаблон Bridge хорошо работает в ситуациях, когда вам нужно отделить логику управления от UI.

На заметку! Смысл шаблона Bridge, согласно книге Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) *Design Patterns: Elements of Reusable Object-Oriented Software* (Boston, MA: Addison-Professional,1995), заключается в отделении абстракции от реализации, чтобы то и другое можно было менять независимо.

Используя шаблон Bridge, вы можете облегчить сценарий, при котором изменения, происходящие в ядре системы, не влекут за собой изменений в UI, и что более важно — изменения в UI не требуют изменений в ядре системы. Один распространенный способ реализации этого шаблона заключается в создании четко

определенных интерфейсов в ядре системы, которые используются UI для взаимодействия с ней, и наоборот. Делегаты — блестящий механизм определения такого интерфейса. С делегатом вы можете сформулировать абстрактные вещи следующим образом: "Когда пользователь захочет включить воспроизведение, я хочу, чтобы были вызваны зарегистрированные методы, принимающие всю необходимую информацию для выполнения данного действия". Прелесть заключается в том, что ядро системы не волнует, как именно пользователь укажет UI, что он желает включить воспроизведение. Это может быть щелчок на кнопке или срабатывание какого-то устройства, улавливающего мозговые волны и угадывающего мысли пользователя. Для системы это не имеет значения, и вы в любой момент можете изменить любой компонент взаимодействия, не затрагивая другого. Обе стороны выполняют требования общего интерфейсного контракта, которым в данном случае выступает специально сформированный делегат и средства его регистрации в генерирующей события сущности[4].

Этот шаблон использования, также известный как "издатель/подписчик" (publish/subscribe), настолько распространен, даже за пределами мира разработки пользовательских интерфейсов, что проектировщики исполняющей системы .NET были настолько великодушны, что позаботились определить формализованный встроенный механизм событий. Когда вы объявляете событие внутри класса, компилятор реализует скрытые методы, позволяющие вам регистрировать и отменять регистрацию делегатов, которые вызываются при наступлении определенных событий. По сути, событие — это сокращение, позволяющее сэкономить ваше время, которое понадобилось бы для написания методов регистрации и отмены регистрации, управляющих цепочкой делегатов. Давайте взглянем на простой пример события, основанный на вышесказанном.

```
using System;
// Аргументы, переданные от UI при событии включения воспроизведения
public class PlayEventArgs : EventArgs
{
    public PlayEventArgs( string filename ) {
        this.filename = filename;
    }
    private string filename;
    public string Filename {
        get { return filename; }
        }
    }
    public class PlayerUI
    {
        // Определение события запуска воспроизведения.
        public event EventHandler<PlayEventArgs> PlayEvent;
        public void UserPressedPlay() {
        OnPlay();
    }
    protected virtual void OnPlay() {
        // Инициировать событие.
        EventHandler<PlayEventArgs> localHandler
            = PlayEvent;
```

[4] В главе 5 я раскрываю тему контрактов и интерфейсов во всех подробностях.

```
            if( localHandler != null ) {
               localHandler( this,
                             new PlayEventArgs("somefile.wav") );
            }
        }
    }
    public class CorePlayer
    {
        public CorePlayer() {
            ui = new PlayerUI();
            // Регистрация обработчика события.
            ui.PlayEvent += this.PlaySomething;
        }
        private void PlaySomething( object source,
                                    PlayEventArgs args ) {
            // Воспроизведение файла.
        }
        private PlayerUI ui;
    }
    public class EntryPoint
    {
        static void Main() {
            CorePlayer player = new CorePlayer();
        }
    }
```

Несмотря на то что синтаксис этого простого события может показаться усложненным, общая идея состоит в том, что вы создаете четко определенный интерфейс, через который извещаете все заинтересованные стороны о том, что пользователь желает воспроизвести файл. Этот интерфейс инкапсулирован внутри класса `PlayEventArgs`. События накладывают определенные правила на использование делегатов. Делегат должен что-нибудь возвращать и должен принимать два аргумента. Первый аргумент — ссылка на объект, представляющий сторону, которая генерирует сообщение. Второй аргумент — тип, унаследованный от `System.EventArgs`. Ваш производный класс `EventArgs` — это место, где вы определяете все специфичные для события аргументы.

На заметку! В .NET 1.1 приходилось явно определять тип делегата, стоящего за событием. Начиная с .NET 2.0, вы можете использовать новый обобщенный класс `EventHandler<T>`, изолирующий вас от этой рутины.

Обратите внимание, что я объявил событие, используя обобщенный класс `EventHandler<T>`. При регистрации обработчиков с использованием операции += в качестве сокращения вы можете представлять только ссылку на метод, который нужно вызвать, и компилятор создаст для вас экземпляр `EventHandler<T>`. Дополнительно вы можете применить после операции += новое выражение, создающее экземпляр `EventHandler<T>`, но если компилятор обеспечивает такое сокращение, зачем набирать сложный код, который может оказаться трудным для чтения?

Отметьте способ определения события внутри класса PlayerUI с применением ключевого слова event. За этим ключевым словом сначала следует определенный делегат события, а за ним — имя события, в данном случае PlayEvent. Идентификатор PlayEvent означает две совершенно разные вещи, в зависимости от того, на какой стороне "забора" вы находитесь. С точки зрения генератора события — в данном случае, PlayerUI — событие PlayEvent используется в точности как делегат. Вы можете видеть такое его применение внутри метода OnPlay. Обычно метод, названный OnPlay, вызывается в ответ на щелчок на кнопке пользовательского интерфейса. Он извещает всех зарегистрированных слушателей о вызове через событие (делегат) PlayEvent.

На заметку! Существует популярная идиома при генерации события — инициировать его внутри метода protected virtual по имени On<событие>, где <событие> заменяется именем события, в данном случае — OnPlay. Таким образом, производные классы могут легко модифицировать действия, предпринимаемые, когда должно быть инициировано событие. В C# вы должны проверить событие на равенство null, прежде чем вызывать его, иначе будет сгенерировано исключение NullReferenceException. Метод OnPlay создает локальную копию события перед проверкой на null. Это позволяет избежать состояния "гонок", когда событие устанавливается в null из другого потока после выполнения проверки на null и перед возбуждением события.

Со стороны потребителя события идентификатор PlayEvent используется совершенно иначе, как вы можете видеть в конструкторе CorePlayer. В C# перегружены операции += и -= для событий, чтобы предоставить компактную нотацию для регистрации и отмены регистрации слушателей событий. В конструкторе CorePlayer вы могли бы также зарегистрировать событие следующим образом:

```
ui.PlayEvent += this.PlaySomething;
```

Такова базовая структура событий. Как я упоминал ранее, события .NET — это сокращения для создания делегатов и интерфейсов, с которыми нужно регистрировать эти делегаты. В доказательство этого вы можете просмотреть код IL, полученный в результате компиляции предыдущего примера. "За кулисами" компилятор генерирует два метода — add_OnPlay и remove_OnPlay, которые вызываются, когда вы используете перегруженные операции += и -=. Эти методы управляют добавлением и удалением делегатов в цепочке делегатов событий. Фактически компилятор C# не позволяет вам вызывать эти методы явно, так что вы обязаны использовать перегруженные операции.

Механизм событий определяет две скрытые функции-члена, или средства доступа (accessors), подобно тому, как свойства определяют скрытые средства доступа. Вы можете спросить — есть ли какой-нибудь способ контролировать тело этих функций-членов, как это делается со свойствами. Ответ — да, и используемый для этого синтаксис подобен синтаксису свойств. Я модифицировал класс PlayerUI, чтобы продемонстрировать явный способ обработки добавления и удаления операций событий.

```
public class PlayerUI
{
    // Определить событие для уведомления о воспроизведении.
    private EventHandler<PlayEventArgs> playEvent;
```

```
    public event EventHandler<PlayEventArgs> PlayEvent {
    add {
       playEvent = (EventHandler<PlayEventArgs>)
                   Delegate.Combine( playEvent, value );
    }
    remove {
       playEvent = (EventHandler<PlayEventArgs>)
                   Delegate.Remove( playEvent, value );
    }
 }

 public void UserPressedPlay() {
    OnPlay();
 }

 protected virtual void OnPlay() {
    // Инициировать событие.
    EventHandler<PlayEventArgs> localHandler
       = playEvent;
    if( localHandler != null ) {
       localHandler( this,
              new PlayEventArgs("somefile.wav") );
    }
 }
}
```

Внутри разделов `add` и `remove` объявления события ссылкам на добавляемый или удаляемый делегат осуществляется по ключевому слову `value`, что идентично тому, как работает метод `set` свойства. Данный пример использует `Delegate.Combine` и `Delegate.Remove` для управления внутренней цепочкой делегатов по имени `playEvent`. Пример выглядит несколько надуманным, поскольку механизм событий по умолчанию делает, по сути, то же самое, но я показываю его для примера.

На заметку! Вам может понадобиться определять пользовательские средства доступа к событиям явно, чтобы построить некоторого рода специальный механизм хранения событий, или если вам понадобится выполнить любого рода специальную обработку при регистрации и отмене регистрации событий.

И теперь заключительный комментарий относительно шаблона проектирования. На основе сказанного вы можете видеть, что события идеальны для реализации шаблона проектирования "издатель/подписчик", когда множество слушателей регистрируются для получения уведомления (публикации) события. Аналогично вы можете использовать события .NET для реализации формы шаблона Observer (Обозреватель), когда различные сущности регистрируются для получения уведомлений об изменении некоторой другой сущности. Это лишь два шаблона проектирования, реализацию которых облегчают события.

Анонимные методы

Много раз вам случится создавать делегаты для обратного вызова, которые делают нечто очень простое. Представьте, что вы реализуете простой механизм обработки массива целых чисел. Предположим, что вы хотите спроектировать гибкую систему, чтобы когда процессор обрабатывал массив целых чисел, он использовал алгоритм, переданный ему в точке вызова. Такой шаблон использования называется шаблоном Strategy (Стратегия). Здесь вы можете выбирать различные стратегии вычислений для обеспечения механизма спецификации алгоритма для использования во время выполнения. Делегат — блестящий инструмент реализации такой системы. Посмотрим, как выглядит пример.

```
using System;
public delegate int ProcStrategy( int x );
public class Processor
{
    private ProcStrategy strategy;
    public ProcStrategy Strategy {
        set {
        strategy = value;
        }
    }
    public int[] Process( int[] array ) {
        int[] result = new int[ array.Length ];
        for( int i = 0; i < array.Length; ++i ) {
            result[i] = strategy( array[i] );
        }
        return result;
    }
}

public class EntryPoint
{
    private static int MultiplyBy2( int x ) {
        return x*2;
    }

    private static int MultiplyBy4( int x ) {
        return x*4;
    }

    private static void PrintArray( int[] array ) {
        for( int i = 0; i < array.Length; ++i ) {
            Console.Write( "{0}", array[i] );
            if( i != array.Length-1 ) {
                Console.Write( ", " );
            }
        }
        Console.Write( "\n" );
    }
```

```
    static void Main() {
        // Создать массив целых чисел.
        int[] integers = new int[] {
            1, 2, 3, 4
        };
        Processor proc = new Processor();
        proc.Strategy = new ProcStrategy( EntryPoint.MultiplyBy2 );
        PrintArray( proc.Process(integers) );
        proc.Strategy = new ProcStrategy( EntryPoint.MultiplyBy4 );
        PrintArray( proc.Process(integers) );
    }
}
```

Концептуально идея кажется простой. Однако на практике вам придется сделать несколько сложных вещей, чтобы заставить ее работать. Сначала нужно определить тип делегата для представления стратегического метода. В предыдущем примере этот тип делегата — `ProcStrategy`. Затем вы должны собственно написать методы различных стратегий. И, наконец, нужно создать делегаты, привязать их к этим методам и зарегистрировать в процессоре. По сути, эти действия кажутся не связанными в своем потоке. Казалось бы более естественным определять методы делегатов каким-то менее многословным способом. Часто случается так, что инфраструктура, необходимая для применения делегатов, затрудняет понимание кода, поскольку кусочки механизма разбросаны в разных местах кода.

Анонимные методы обеспечивают более легкий и компактный способ определения простых делегатов, подобных этим. Анонимные методы впервые появились в .NET 2.0 и, коротко говоря, они позволяют вам определять тело метода делегата в точке создания его экземпляра. Посмотрим, как можно модифицировать предыдущий пример для использования анонимных методов. Вот измененная часть примера:

```
public class EntryPoint
{
    private static void PrintArray( int[] array ) {
        for( int.i = 0; i < array.Length; ++i ) {
            Console.Write( "{0}", array[i] );
            if( i != array.Length-1 ) {
                Console.Write( ", " );
            }
        }
        Console.Write( "\n" );
    }
    static void Main() {
        // Создать массив целых чисел.
        int[] integers = new int[] {
            1, 2, 3, 4
        };

        Processor proc = new Processor();
        proc.Strategy = delegate(int x) {
            return x*2;
        };
        PrintArray( proc.Process(integers) );
```

```
        proc.Strategy = delegate(int x) {
            return x*4;
        };
        PrintArray( proc.Process(integers) );

        proc.Strategy = delegate {
            return 0;
        };
        PrintArray( proc.Process(integers) );
    }
}
```

Обратите внимание, что два метода — `MultiplyBy2` и `MultiplyBy4` — исчезли. Вместо этого делегат создается с применением специального синтаксиса анонимных методов в точке, где он присваивается свойству `Processor.Strategy`. Вы видите, что этот выглядит синтаксис почти так, как если бы вы взяли объявление делегата и метод, привязываемый к делегату, и смешали их в одно целое. Вообще в любом месте, где вы можете передать экземпляр делегата в качестве параметра, можно вместо него передать анонимный метод.

Когда вы передаете анонимный метод в списке параметров, принимающем делегат, или когда присваиваете типу делегата анонимный метод, вы должны учитывать преобразование типа анонимного метода. "За кулисами" ваш анонимный метод превращается в обычный делегат, который трактуется как любой другой экземпляр делегата.

При присваивании анонимного метода экземпляру делегата следует соблюдать несколько правил. Во-первых, типы параметров делегата должны быть совместимыми с типами параметров анонимного метода. В предыдущем примере в первых двух использованиях делегатов я показал длинный путь объявления анонимного метода. Некоторые из вас, возможно, заметили отличие синтаксиса третьего использования делегата в этом примере. Я пропустил список параметров, поскольку тело метода даже не использует их. Тем не менее, я все равно могу установить свойство `Strategy` по этому анонимному методу, потому ясно, что некоторое преобразование типов здесь происходит. Вообще говоря, если анонимный метод не имеет списка параметров, то он является преобразуемым к типу делегата со списком параметров до тех пор, пока этот список не включает никаких параметров `out` и `ref`. Если параметры `out` присутствуют, то анонимный метод обязан перечислить их в списке параметров в точке объявления.

Во-вторых, если анонимный метод перечисляет какие-то параметры в своем объявлении, их количество должно совпадать с параметрами типа делегата, и каждый из их типов должен быть неявно преобразуемым к этим типам в объявлении делегата. И, наконец, тип возврата анонимного метода должен быть неявно преобразуемым к объявленному типу возврата делегата. Поскольку синтаксис объявления анонимного метода явно не устанавливает типа возврата, компилятор должен проверить каждый оператор `return` и убедиться, что он возвращает тип, отвечающий правилам преобразования.

В приведенном примере анонимные методы позволили нам сэкономить некоторый объем клавиатурного набора и сделали код более читабельным. Но давайте рассмотрим правила областей видимости, связанные с анонимными методами. Вы уже знаете, что в C# фигурные скобки определяют единицы вложенных областей.

Скобки, отделяющие анонимные методы — не исключение. Взглянем на следующую модификацию предыдущего примера:

```
using System;
public delegate int ProcStrategy( int x );
public class Processor
{
   private ProcStrategy strategy;
   public ProcStrategy Strategy {
      set { strategy = value; }
   }
   public int[] Process( int[] array ) {
      int[] result = new int[ array.Length ];
      for( int i = 0; i < array.Length; ++i ) {
      result[i] = strategy( array[i] );
      }
      return result;
   }
}
public class Factor
{
   public Factor( int fact ) {
      this.fact = fact;
   }
   private int fact;
   public ProcStrategy Multiplier {
      get {
         // Это анонимный метод.
         return delegate(int x) {
            return x*fact;
         };
      }
   }
   public ProcStrategy Adder {
      get {
         // Это анонимный метод.
         return delegate(int x) {
            return x+fact;
         };
      }
   }
}
public class EntryPoint
{
   private static void PrintArray( int[] array ) {
      for( int i = 0; i < array.Length; ++i ) {
         Console.Write( "{0}", array[i] );
         if( i != array.Length-1 ) {
            Console.Write( ", " );
         }
      }
      Console.Write( "\n" );
   }
```

```
        static void Main() {
            // Создать массив целых чисел.
            int[] integers = new int[] {
                1, 2, 3, 4
            };

            Factor factor = new Factor( 2 );
            Processor proc = new Processor();
            proc.Strategy = factor.Multiplier;
            PrintArray( proc.Process(integers) );
            proc.Strategy = factor.Adder;
            factor = null;
            PrintArray( proc.Process(integers) );
        }
    }
```

Обратите особое внимание в этом примере на класс Factor. Я сделал Processor более гибким, чтобы можно было применять фактор по-разному, используя либо умножение, либо сложение.

Заметьте, что анонимные методы в классе Factor используют переменную, доступную в области, где она объявлена, а именно — поле экземпляра factor. Вы можете сделать это потому, что обычные правила областей видимости действуют и по отношению к блоку анонимного метода. Однако здесь есть некоторые тонкости. Видите, где я установил в null переменную экземпляра factor? Я сделал это перед использованием делегата, полученного от свойства Factor.Adder. Это нормально, потому что свойство Adder возвращает экземпляр делегата, даже несмотря на то, что я решил объявить делегат как анонимный метод, а не как это делается обычно. Но как насчет поля экземпляра Factor.fact? Если я устанавливаю значение переменной factor равным null в методе Main, то сборщик мусора (GC) может подобрать объект factor даже перед самим делегатом, использующим поле, и с ним будет покончено, правильно? Может ли это привести к запуску "гонок", если GC подберет экземпляр Factor.fact перед тем, как делегат закончит с ним? Ответ — нет, потому что делегат захватил переменную.

При объявлении анонимных методов любые переменные, объявленные вне контекста анонимного метода, но доступные внутри него, включая ссылку this, рассматриваются как внешние переменные. И если тело анонимного метода ссылается на эти переменные, то говорят, что анонимный метод "захватил" переменную. Поэтому поле Factor.fact в предыдущем примере продолжает существовать, поскольку на него ссылаются активные делегаты.

Способность тел анонимных методов обращаться к переменным, принадлежащим к контексту, в котором эти методы были определены, чрезвычайно удобна. Представьте, насколько сложнее было бы реализовать тот же механизм, что приведен в примере, с использованием обычных делегатов. Вам пришлось бы создавать механизм, внешний по отношению к делегату, который поддерживал бы переменную factor, планируемую для использования в делегате. Одним из возможных решений при использовании стандартных делегатов может быть ввод дополнительного промежуточного слоя в форме класса, как это часто делается для решения подобных проблем. Однако я уверен, вы согласитесь, что анонимные методы позволяют сэкономить массу работы, не говоря уже от том, что они делают ваш код короче и более читабельным.

Остерегайтесь сюрпризов захваченных переменных

Когда переменная захвачена экземпляром анонимного метода, вы должны соблюдать осторожность в отношении возможных последствий. Имейте в виду, что представление захваченной переменной существует где-то в куче, а переменная в экземпляре делегата — это просто ссылка на эти данные. Поэтому вполне возможно, что два экземпляра делегатов, созданных из анонимных методов, могут содержать ссылку на одну и ту же переменную. Ниже продемонстрирован пример сказанного.

```
using System;
public delegate void PrintAndIncrement();
public class EntryPoint
{
    public static PrintAndIncrement[] CreateDelegates() {
        PrintAndIncrement[] delegates = new PrintAndIncrement[3];
        int someVariable = 0;
        int anotherVariable = 1;
        for( int i = 0; i < 3; ++i ) {
            delegates[i] = delegate {
                Console.WriteLine( someVariable++ );
            };
        }
        return delegates;
    }
    static void Main() {
        PrintAndIncrement[] delegates = CreateDelegates();
        for( int i = 0; i < 3; ++i ) {
            delegates[i]();
        }
    }
}
```

Анонимные методы внутри метода `CreateDelegates` захватывают `someVariable` — локальную переменную в контексте метода `CreateDelegates`. Однако поскольку три экземпляра анонимных методов помещены в массив, получается, что эти три экземпляра захватывают один и тот же экземпляр переменной. Поэтому запуск приведенного выше кода выдает следующий результат:

```
0
1
2
```

При вызове каждого делегата он печатает и увеличивает одну и ту же переменную. Теперь рассмотрим, какой эффект даст небольшое изменение в методе `CreateDelegates`. Если вы переместите объявление `someVariable` в цикл, создающий массив делегатов, то свежий экземпляр локальной переменной будет создаваться на каждом шаге цикла. Обратите внимание на следующие изменения в методе `CreateDelegates`:

```
public static PrintAndIncrement[] CreateDelegates() {
    PrintAndIncrement[] delegates = new PrintAndIncrement[3];
```

```
    for( int i = 0; i < 3; ++i ) {
        int someVariable = 0;
        delegates[i] = delegate {
            Console.WriteLine( someVariable++ );
        };
    }
    return delegates;
}
```

На этот раз вывод будет таким:

```
0
0
0
```

Вот почему нужно соблюдать осторожность при использовании захвата переменных анонимными методами. В первом случае три делегата захватывают одну и ту же переменную. Во втором случае они захватывают отдельные экземпляры переменной, поскольку каждая итерация цикла `for` создает новую (отдельную) переменную в стеке. Хотя вам стоит включить это мощное средство в свой инструментарий, нужно четко представлять себе, что вы делаете, чтобы не споткнуться.

Сообразительные читатели могут недоумевать, как может успешно работать такой код, если захватываемые переменные относятся к типу значений, ведь типы значений по умолчанию находятся в стеке? Напомню, что типы значений создаются в стеке, только если они не являются полями ссылочного типа, экземпляры которого создаются в куче, что включает случай, когда они упаковываются. Однако `someVariable` — локальная переменная, поэтому при нормальных условиях она создается в стеке. Но здесь мы имеем дело с ненормальной ситуацией. Ясно, что для экземпляра анонимного метода невозможно захватить локальную переменную из стека и рассчитывать, что она останется там, когда методу понадобится к ней обратиться. Поэтому она должна располагаться в куче. Локальные переменные типов значений, которые захватываются анонимным методом, должны подчиняться другим правилам жизненного цикла, чем те же переменные, которые не захвачены. Поэтому компилятор "за кулисами" здесь допускает немножко "колдовства", когда обеспечивает возможность захватывать локальные переменные типов значений.

Когда компилятор встречает захваченную локальную переменную типа значений, он молча создает скрытый класс. Когда код инициализирует локальную переменную, компилятор генерирует код IL, создающий экземпляр этого прозрачного класса и инициализирует его поле, которое в данном случае представляет `someVariable`. Вы можете убедиться в этом, открыв скомпилированный код первого примера в ILDASM. Я включил дополнительную переменную `anotherVariable`, чтобы вы могли видеть отличие в том, как IL представляет их.

Поскольку `anotherVariable` не захватывается, она создается в стеке, как и можно было ожидать. Ниже показана часть кода IL для `CreateDelegates` после компиляции примера с включенной отладочной информацией.

```
// Code size 85 (0x55)
.maxstack 5
.locals init ([0] class PrintAndIncrement[] delegates,
        [1] int32 anotherVariable,
        [2] int32 i,
```

```
        [3] class PrintAndIncrement '<>9__CachedAnonymousMethodDelegate1',
        [4] class EntryPoint/'<>c__DisplayClass2' '<>8__locals3',
        [5] class PrintAndIncrement[] CS$1$0000,
        [6] bool CS$4$0001)
IL_0000: ldnull
IL_0001: stloc.3
IL_0002: newobj instance void EntryPoint/'<>c__DisplayClass2'::.ctor()
IL_0007: stloc.s '<>8__locals3'
IL_0009: nop
IL_000a: ldc.i4.3
IL_000b: newarr PrintAndIncrement
IL_0010: stloc.0
IL_0011: ldloc.s '<>8__locals3'
IL_0013: ldc.i4.0
IL_0014: stfld int32 EntryPoint/'<>c__DisplayClass2'::someVariable
IL_0019: ldc.i4.1
IL_001a: stloc.1
IL_001b: ldloc.1
IL_001c: call void [mscorlib]System.Console::WriteLine(int32)
```

Обратите внимание на использование двух переменных. В строке `IL_0002` создается новый экземпляр скрытого класса. В этом случае компилятор назвал этот класс `<>c__DisplayClass2`. Этот класс содержит общедоступное поле экземпляра по имени `someVariable`, значение которому присваивается в строке `IL_0014`. Компилятор прозрачно вставил тот самый пресловутый промежуточный слой в форме класса, чтобы решить проблему захвата локальной переменной анонимным методом. К тому же отметьте тот факт, что `anotherVariable` трактуется как нормальная локальная переменная, находящаяся в стеке, на что указывает ее объявление в разделе локальных переменных метода.

Анонимные методы как привязки параметров делегатов

Анонимные методы в сочетании с захватом переменных могут представлять удобное средство реализации *привязки параметров* делегатов. Привязка параметров — это техника, при которой вы хотите вызвать делегат, обычно с более чем одним параметром, таким образом, чтобы один или более параметров были фиксированы, а другие варьировались при вызовах делегата. Например, если у вас есть делегат, принимающий два параметра, и вы хотите преобразовать его в делегат, принимающий один параметр, а другой параметр зафиксировать, вы можете использовать привязку параметров (parameter binding) для выполнения этого трюка. Те из вас, кто программировал на C++ и знаком с STL или библиотекой Boost, должны быть знакомы с привязками параметров. Ниже приведен соответствующий пример.

```
using System;
public delegate int Operation( int x, int y );
public class Bind2nd
{
    public delegate int BoundDelegate( int x );
    public Bind2nd( Operation del, int arg2 ) {
        this.del = del;
        this.arg2 = arg2;
    }
```

```
      public BoundDelegate Binder {
         get {
            return delegate( int arg1 ) {
               return del( arg1, arg2 );
            };
         }
      }
      private Operation del;
      private int arg2;
   }
   public class EntryPoint
   {
      static int Add( int x, int y ) {
         return x + y;
      }
      static void Main() {
         Bind2nd binder = new Bind2nd(
               new Operation(EntryPoint.Add), 4 );
         Console.WriteLine( binder.Binder(2) );
      }
   }
```

В этом примере делегат типа `Operation` с двумя параметрами, который выполняет обратный вызов статического метода `EntryPoint.Add`, преобразуется в делегат, принимающий только один параметр. Второй параметр фиксируется с помощью класса `Bind2nd`. По сути, поле экземпляра `Bind2nd.arg2` устанавливается в значение, которое вы хотите зафиксировать за вторым параметром. Затем свойство `Bind2nd.Binder` возвращает новый делегат в форме экземпляра анонимного метода, который захватывает поле экземпляра и применяет его вместе с первым параметром, переданным в точке вызова.

Читатели, знакомые с C++ STL, возможно, воскликнут, что этот пример был бы намного более полезен, если бы `Bind2nd` был обобщенным типом, чтобы поддерживать обобщенный делегат с двумя параметрами — подобно тому, как это делает STL. В самом деле, это было бы хорошо; однако некоторые языковые барьеры это немного затрудняют. Начнем с попытки сделать обобщенным тип делегата в классе `Bind2nd`. Вы можете попробовать следующий код:

```
   // НЕ КОМПИЛИРУЕТСЯ !!!
   public class Bind2nd< DelegateType >
   {
      public delegate int BoundDelegate( int x );
      public Bind2nd( DelegateType del, int arg2 ) {
         this.del = del;
         this.arg2 = arg2;
      }
      public BoundDelegate Binder {
         get {
            return delegate( int arg1 ) {
               return del( arg1, arg2 ); // ОПА!
            };
         }
      }
```

```
    private DelegateType del;
    private int arg2;
}
```

Это выдающаяся попытка, но, к сожалению, неудачная, поскольку компилятор оказывается в недоумении внутри тела анонимного метода и жалуется на то, что поле экземпляра используется как метод. Компилятор прав. Это именно то, что вы хотите сделать, даже несмотря на то, что компилятор не может найти ни головы, ни хвоста. Что делать программисту?

Другая попытка включает ограничения обобщений. Используя ограничения, вы можете сказать, что невзирая на то, что тип является обобщенным, он должен наследоваться от определенного базового класса или реализовывать определенный интерфейс. Отлично! Давайте поможем компилятору и сообщим, что `DelegateType` наследуется от `System.Delegate`, как показано ниже:

```
// ВСЕ РАВНО НЕ КОМПИЛИРУЕТСЯ !!!
public class Bind2nd< DelegateType >
where DelegateType : Delegate
{
    public delegate int BoundDelegate( int x );
    public Bind2nd( DelegateType del, int arg2 ) {
        this.del = del;
        this.arg2 = arg2;
    }
    public BoundDelegate Binder {
        get {
            return delegate( int arg1 ) {
                return del( arg1, arg2 ); // ОПА!
            };
        }
    }
    private DelegateType del;
    private int arg2;
}
```

К сожалению, опять неудача! На этот раз компилятор говорит, что ограничение типа `Delegate` не разрешается. Это приводит нас к решению использовать для выполнения работы обобщенных делегатов в сочетании со статическим методом `Delegate.CreateDelegate`. И тогда решение проблемы выглядит так:

```
using System;
using System.Reflection;
public delegate int Operation( int x, int y );
public class Bind2nd<Arg1Type, Arg2Type, ReturnType>
{
    public delegate ReturnType UnboundDelegate<UBArg1Type, UBArg2Type>(
                UBArg1Type arg1,
                UBArg2Type arg2 );
    public delegate ReturnType BoundDelegate<BArg1Type>( BArg1Type x );
    public Bind2nd( Delegate del, Arg2Type arg2 ) {
        // Получить типы от делегата.
        object target = del.Target;
```

```
         MethodInfo targetMethod = del.Method;
         Type targetType = targetMethod.ReflectedType;
         if( target == null ) {
            // Статический метод.
            this.del = (UnboundDelegate<Arg1Type, Arg2Type>)
                            Delegate.CreateDelegate(
                            typeof(UnboundDelegate<Arg1Type, Arg2Type>),
                            targetType,
                            targetMethod.Name );
         } else {
            // Метод экземпляра.
            this.del = (UnboundDelegate<Arg1Type, Arg2Type>)
            Delegate.CreateDelegate(
                         typeof(UnboundDelegate<Arg1Type, Arg2Type>),
                         target,
                         targetMethod.Name );
         }
         this.arg2 = arg2;
      }
      public BoundDelegate<Arg1Type> Binder {
         get {
            return delegate( Arg1Type arg1 ) {
               return del( arg1, arg2 );
            };
         }
      }
      private UnboundDelegate<Arg1Type, Arg2Type> del;
      private Arg2Type arg2;
   }
   public class EntryPoint
   {
      static int Add( int x, int y ) {
         return x + y;
      }
      static void Main() {
         Bind2nd<int,int,int> binder = new Bind2nd<int,int,int>(
               new Operation(EntryPoint.Add),
               4 );
         Console.WriteLine( binder.Binder(2) );
      }
   }
```

В этом решении мы схитрили в двух местах. Во-первых, чтобы анонимный метод мог использовать поле del как метод, компилятор должен знать, что это делегат. Также он не может просто относиться к типу System.Delegate. Чтобы вызывать делегат, используя синтаксис вызова метода, он должен относиться к конкретному типу делегата. И здесь на помощь приходит делегат UnboundDelegate. Для пользы дела я также ввел тип BoundDelegate, который представляет собой обобщенный делегат, принимающий только один параметр — не связанный параметр. Вторая хитрость заключается в конструкторе. К сожалению, язык C# не предусматривает сокращения для преобразования делегата из одного типа в другой, даже если

оба типа делегатов поддерживают в точности одинаковые сигнатуры методов. Поэтому вы должны взломать целевой тип и информацию о методе оригинального делегата, чтобы построить экземпляр `UnboundDelegate` через вызов `Delegate.CreateDelegate`. Вооружившись этими двумя трюками, вы получаете работающую обобщенную привязку.

Шаблон Strategy

Делегаты представляют удобный механизм для реализации шаблона Strategy (Стратегия). В основе своей шаблон Strategy позволяет вам динамически заменять вычислительные алгоритмы, в зависимости от ситуации времени выполнения. Например, рассмотрим распространенный случай сортировки группы элементов. Предположим, что вы хотите, чтобы сортировка происходила насколько возможно быстро. Из-за особенностей системы, однако, потребовалось больше временной памяти, чтобы обеспечить нужную скорость. Это отлично работает для коллекций относительно управляемого размера, но если коллекция разрастается до громадных размеров, может случиться, что объем памяти, необходимый для выполнения быстрой сортировки, превысит емкость системных ресурсов. Для таких случаев вы можете предусмотреть алгоритм сортировки, который работает намного медленнее, но зато потребляет намного меньше системных ресурсов. Шаблон Strategy позволяет вам заменять эти алгоритмы во время выполнения, в зависимости от конкретных условий. Этот пример, несмотря на свою надуманность, отлично иллюстрирует предназначение шаблона Strategy.

Обычно вы реализуете этот шаблон с помощью интерфейсов. Вы объявляете интерфейс, который реализуют все стратегии. Затем потребитель алгоритма может не беспокоиться о том, какая именно реализация стратегии будет использована. На рис. 10.1 показана диаграмма, описывающая типичное применение этого шаблона.

Делегаты предоставляют более легковесную альтернативу использованию интерфейсов для реализации простой стратегии. Интерфейсы — это просто механизм для реализации программного контракта.

Рис. 10.1. Типичная реализация шаблона Strategy на основе интерфейсов

Вместо этого представьте, что ваше объявление делегата используется для реализации контракта, а любой метод, соответствующий сигнатуре делегата, представляет собой конкретную стратегию. Теперь вместо того, чтобы потребитель хранил ссылку на абстрактный интерфейс стратегии, он просто хранит экземпляр делегата. Следующий пример иллюстрирует применения этого сценария:

```
using System;
using System.Collections;
public delegate Array SortStrategy( ICollection theCollection );
public class Consumer
{
    public Consumer( SortStrategy defaultStrategy ) {
        this.strategy = defaultStrategy;
    }
    private SortStrategy strategy;
    public SortStrategy Strategy {
        get { return strategy; }
        set { strategy = value; }
    }
    public void DoSomeWork() {
        // Использовать стратегию.
        Array sorted = strategy( myCollection );
        // Сделать что-то с результатом.
    }
    private ArrayList myCollection;
}

public class SortAlgorithms
{
    static Array SortFast( ICollection theCollection ) {
        // Выполнять быструю сортировку.
    }
    static Array SortSlow( ICollection theCollection ) {
        // Выполнять медленную сортировку.
    }
}
```

Когда создается экземпляр объекта `Consumer`, он получает стратегию сортировки по умолчанию, которая представляет собой не что иное, как метод, реализующий сигнатуру делегата `SortStrategy`. Если во время выполнения складываются подходящие условия, то стратегия сортировки заменяется и метод `Consumer.DoSomeWork` автоматически обращается к заменяющей стратегии. Вы можете заметить, что реализация шаблона Strategy подобным образом еще более гибка, чем использование интерфейсов, поскольку делегаты могут привязываться как к статическим методам, так и к методам экземпляра. Поэтому вы можете создать конкретную реализацию стратегии, которая также содержит некоторые данные состояния, необходимые для выполнения операции, до тех пор, пока делегат указывает на метод экземпляра класса, содержащего эти данные о состоянии. Аналогично, делегат может быть анонимным методом, возвращенным свойством этого класса.

Резюме

Делегаты представляют собой первоклассный определенный и реализованный системой механизм унифицированного представления обратных вызовов. В данной главе вы увидели разные способы объявления и создания делегатов различного типа, включая одиночные делегаты, цепочки делегатов, делегаты открытого экземпляра и анонимные методы, которые сами по себе являются делегатами. Вдобавок я показал, как применять делегаты в качестве строительных блоков событий. Вы можете использовать делегаты для реализации широкого разнообразия шаблонов проектирования, поскольку делегаты — замечательное средство для определения программного контракта. А в основе почти любого из шаблонов проектирования лежит четко определенный контракт.

Следующая глава будет посвящена подробностям механизма обобщений, который определенно является одним из наиболее мощных средств CLR и языка C# для создания безопасного в отношении типов кода.

ГЛАВА 11

Обобщения

Поддержка обобщений (generics) — одно из самых замечательных средств C# и .NET. Обобщения позволяют вам создавать открытые (open-ended) типы, которые преобразуются в закрытые во время выполнения. Каждый уникальный закрытый тип сам по себе уникален. Могут существовать только экземпляры закрытых типов. Когда вы объявляете обобщенный тип, то указываете список параметров типа в его объявлении, и эти аргументы-типы задаются для создания конкретных закрытых типов, как в следующем примере:

```
public class MyCollection<T>
{
    public MyCollection() {
    }
    private T[] storage;
}
```

В данном случае я объявил обобщенный тип `MyCollection<T>`, который трактует тип внутри коллекции как неуказанный тип. В данном примере список параметров типа состоит только из одного типа, и он описывается в синтаксисе, который позволяет перечислять обобщенные типы, разделяя их запятыми, между угловыми скобками. Идентификатор `T` — это на самом деле только указатель места заполнения, куда подставляется любой тип. В некоторой точке потребитель `MyCollection<T>` объявляет то, что называется закрытым типом, подставляя конкретный тип, который должен представлять `T`. Например, предположим, что какая-то другая сборка желает создать контейнерный тип `MyCollection<T>`, содержащий члены типа `int`. Она может сделать это, как показано в следующем коде:

```
public void SomeMethod() {
    MyCollection<int> collectionOfNumbers = new MyCollection<int>();
}
```

`MyCollection<int>` — пример закрытого типа. `MyCollection<int>` может быть использован как любой другой объявленный тип, и он также следует всем тем же правилам, которым подчиняются другие не обобщенные типы. Единственное отличие в том, что он порожден от обобщенного типа. В точке создания экземпляра IL-код реализации `MyCollection<T>` подвергается JIT-компиляции таким образом, что все включения типа T в реализации `MyCollection<T>` заменяются типом `int`.

Обратите внимание, что все уникально сконструированные типы, созданные из одного и того же обобщенного типа, фактически являются совершенно разными типами, которые не разделяют никаких неявных возможностей пре-

образования. Например, MyCollection<long> — совершенно отличный тип от MyCollection<int>, и вы не можете сделать чего-либо вроде такого:

```
// ЭТО НЕ РАБОТАЕТ!!!
public void SomeMethod( MyCollection<int> intNumbers ) {
    MyCollection<long> longNumbers = intNumbers; // ОШИБКА!
}
```

Если вы знакомы с правилами ковариантности массивов, которые позволяют выполнить следующую операцию, то вас может удивить, почему невозможно сделать то же самое с обобщенными типами:

```
public void ProcessStrings( string[] myStrings ) {
    object[] objs = myStrings;
    foreach( object o in objs ) {
        Console.WriteLine( o );
    }
}
```

Дело в том, что в случае ковариантности массивов источник и цель операции присваивания относятся к одному и тому же типу — System.Array. Правила ковариантности массивов просто позволяют вам присваивать один массив другому — до тех пор, пока объявленный тип элементов массива неявно преобразуется во время компиляции. Однако в случае сконструированных обобщенных типов мы имеем дело с совершенно разными типами.

Разница между обобщениями и шаблонами C++

Не случайно синтаксис обобщений похож на синтаксис шаблонов C++, тем более что синтаксис всех прочих элементов C# основан на соответствующем синтаксисе C++. Этот подход позволяет вам опираться на имеющиеся знания. Это типично для C#, потому что проектировщики языка постарались упростить синтаксис и исключить излишнюю многословность. Однако здесь сходство и заканчивается, потому что обобщения C# ведут себя совершенно иначе, чем шаблоны C++, и если вы пришли из мира C++, то должны быть уверенными, что понимаете разницу. В противном случае может случиться так, что вы станете пытаться применить ваши знания шаблонов C++ таким способом, который просто не будет работать с обобщениями.

Главное отличие между этими двумя средствами в том, что расширение обобщений происходит динамически, в то время как расширение шаблонов C++ статично. Другими словами, шаблоны C++ всегда разворачиваются во время компиляции. И одной этой причины достаточно для того, чтобы шаблоны C++ невозможно было поместить в библиотеки. Я знаю, что многие разработчики бывают разочарованными этим фактом, только начиная изучение шаблонов C++. Я могу вспомнить массу случаев, когда было бы очень здорово иметь возможность упаковывать шаблоны C++ в статические библиотеки или DLL. К сожалению, это невозможно. Вот почему весь код шаблонов C++ обычно находится в заголовочных файлах. Это затрудняет защиту авторских прав на код шаблонов C++, поскольку, по сути, вы должны предоставлять такой код в открытом виде любому, кто в нем нуждается.

Блестящий пример — библиотека STL. Обратите внимание, что почти каждый кусочек вашей любимой реализации STL находится в заголовочных файлах.

Обобщения, с другой стороны, могут упаковываться в сборки и в таком виде потребляться позднее. Вместо формирования во время компиляции конструируемые типы формируются во время выполнения, или, точнее говоря — во время JIT-компиляции. Во многих отношениях это делает их более гибкими. Однако, как почти все в инженерном мире, преимущества сопровождаются недостатками. Вы должны трактовать обобщения во время проектирования существенно иначе, чем шаблоны C++, в чем вы убедитесь в конце настоящей главы.

На заметку! Всякий раз, когда JIT-компилятор формирует закрытый тип, новый тип инициализируется для домена приложения, использующего его. Естественно, это накладывает требования на потребление памяти приложением, также известной под именем рабочего набора (working set). Как только тип инициализирован и загружен в домен приложения, вы не можете отменить эту инициализацию и выгрузить его без одновременного разрушения домена приложения. В некоторых редких случаях вам может понадобиться учитывать эти ограничения, проектируя систему, использующую обобщения. Однако, вообще говоря, эти ограничения минимальны. Если ваш обобщенный тип объявляет массу статических полей, то создание множества закрытых типов на его основе может привести к существенной нагрузке на память, поскольку каждый закрытый тип получает свою собственную копию набора статических полей. Вдобавок, если эти закрытые типы используются во многих доменах приложений, то каждый из них также получает свою копию набора статических полей.

Эффективность и безопасность типов обобщений

Вероятно, дополнительная эффективность при использовании типов значений в коллекциях — одно из наибольших преимуществ, которые несут с собой обобщения в C#. Поскольку обычный массив, основанный на `System.Array`, может содержать гетерогенные коллекции экземпляров, созданных из множества типов, до тех пор, пока он хранит ссылки на некоторый общий для них всех базовый тип вроде `System.Object`, он имеет существенные недостатки. Взглянем на следующий пример применения:

```
public void SomeMethod( ArrayList col ) {
    foreach( object o in col ) {
        ISomeInterface iface = (ISomeInterface) o;
        o.DoSomething();
    }
}
```

Поскольку все в CLR наследуется от `System.Object`, экземпляр `ArrayList`, переданный через переменную `col`, может содержать просто невообразимую смесь разных сущностей. Некоторые из них могут и не реализовывать интерфейс `ISomeInterface`. И тогда, как можно было ожидать, такой код может сгенерировать исключение `InvalidCastException`. Однако разве плохо было бы заставить механизм контроля типов компилятора C# выявлять такие вещи еще на этапе компиляции? Именно это и делают обобщения. Используя обобщения, вы можете разработать нечто вроде следующего:

```
public void SomeMethod( IList<ISomeInterface> col ) {
    foreach( ISomeInterface iface in col ) {
        o.DoSomething();
    }
}
```

Здесь метод принимает интерфейс IList<T>. Поскольку параметром типа является тип ISomeInterface, то список может хранить только объекты, относящиеся к типу ISomeInterface. И тут же компилятор получает возможность контролировать безопасность типов.

На заметку! Повышение безопасности типов во время компиляции — всегда хорошая вещь, потому что намного лучше перехватывать ошибки несоответствия типов во время компиляции, чем позднее — во время выполнения.

Вы можете решить ту же проблему без использования обобщений, но это потребовало бы написания вручную класса, который служил бы той же цели, что и сконструированный тип List<ISomeInterface>. Поэтому другая прелесть обобщений подобна шаблонам C++: они представляют собой оболочку для легкой специализации новых типов, построенных на их основе.

Компилятор — ваш друг, и вы всегда должны предоставлять ему как можно больше информации о типах, чтобы помочь ему выполнять свою работу. Поскольку в C# и CLR все, так или иначе, наследуется от System.Object, вы всегда легко можете приведением удалить из объектов информацию о типе, тем самым обманывая компилятор. Если вы пришли из среды C++, просто представьте, что может случиться, если вы станете передавать все указатели как void*. И это я еще не упомянул о трудно поддающихся поиску ошибках, которые неизбежны при таком безумстве.

Пример, который вы только что видели, показывает, как использовать обобщения для обеспечения безопасности типов. Однако, применив его, вы не слишком много выиграете в смысле эффективности. Реальное повышение эффективности происходит тогда, когда аргумент-тип является типом значений. Напомню, что тип значения, вставляемый в коллекцию из пространства имен System.Collections, такую как ArrayList, сначала должен быть упакован, поскольку ArrayList поддерживает коллекцию типов System.Object. ArrayList, предназначенный для хранения только целых чисел, страдает от нескольких проблем, связанных с эффективностью, поскольку целочисленные значения должны упаковываться и распаковываться при каждой вставке или извлечении из коллекции, соответственно. К тому же, операция распаковки в C# обычно формируется в виде IL-операции unbox, сопровождаемой операцией копирования данных типа значения. Обобщения здесь приходят на помощь и прекращают это безумие. В качестве примера скомпилируйте следующий код и затем загрузите сборку в ILDASM, чтобы сравнить сгенерированный код IL для каждого из методов, принимающих экземпляр Stack:

```
using System;
using System.Collections;
using System.Collections.Generic;
public class EntryPoint
{
    static void Main() {
    }
```

```
    public void NonGeneric( Stack stack ) {
        foreach( object o in stack ) {
            int number = (int) o;
            Console.WriteLine( number );
        }
    }
    public void Generic( Stack<int> stack ) {
        foreach( int number in stack ) {
            Console.WriteLine( number );
        }
    }
}
```

Вы заметите, что код IL, сгенерированный методом `NonGeneric`, содержит, по меньшей мере, на 10 инструкций больше, чем обобщенная версия. Большая их часть предназначена для упаковки и распаковки элементов, которой приходится заниматься `NonGeneric`. Более того, метод `NonGeneric` может потенциально сгенерировать исключение `InvalidCastException`, если встретит объект, который не может быть явно приведен и распакован в целое число во время выполнения.

Ясно, что обобщения оказывают компилятору намного больше помощи в выполнении его работы, не исключая точную информацию о типе во время компиляции. Можно, конечно, спорить с тем утверждением, что выигрыш в эффективности был основной причиной ввода обобщений в CLR, чтобы избегнуть излишних операций упаковки. В любом случае оба преимущества (безопасность типов и эффективность) чрезвычайно существенны, и достойны того, чтобы пользоваться ими в полной мере.

Соглашения об именовании указателей мест заполнения в обобщенных типах

Хотя не существует жестких обязательных правил именования мест заполнения обобщенных параметров, рекомендуется, по крайней мере, давать им описательные имена, указывающие на то, как будет использоваться тип. Дополнительно идентификаторы мест заполнения принято именовать, начиная с заглавной буквы `T`, чтобы указать, что это тип. Это соглашение об именах подобно принятому для обозначения интерфейсов, где имена начинаются с заглавной `I`, что облегчает читабельность кода. Если определение обобщенного типа включает только один параметр типа, и понять его нетрудно, принято называть его `T`.

Определения обобщенных конструируемых типов

Как я уже говорил ранее, обобщенный тип — это компилированный тип, который не используется до тех пор, пока из него не будет создан закрытый тип. Не обобщенный тип также известен как закрытый тип, в то время как обобщенный тип называют открытым типом. Однако можно определить новый открытый тип через обобщенный, как показано в следующем примере:

```
public class MyClass<T>
{
    private T innerObject;
}
```

```
public class Consumer<T>
{
   private MyClass< Stack<T> > obj;
}
```

В этом случае определен обобщенный тип Consumer<T>, и он содержит поле, основанное на другом обобщенном типе. При объявлении типа поля Consumer<T>.obj тип MyClass< Stack<T> > остается открытым, пока кто-нибудь не объявит конструируемый тип на основе Consumer<T>, тем самым создав закрытый тип для содержащегося в нем поля.

Обобщенные классы и структуры

До сих пор все примеры, которые я приводил, были обобщенными классами, но все правила, касающиеся обобщений, в равной мере распространяются на структуры. Фактически, наиболее распространенные типы обобщенных объявлений, которые вы будете применять, будут обобщенными классами и структурами. Также до сих пор я был несколько небрежен в отношении терминологии, поэтому отныне постараюсь быть более аккуратным.

Вообще объявления всех обобщенных типов структур и классов следуют одним и тем же правилам, что и обычные структуры и классы. Всякий раз, когда объявление класса содержит список параметров-типов, такой класс можно считать обобщенным типом. Аналогично любая вложенное объявление класса — будь оно обобщенным или нет, — объявленное внутри контекста обобщенного типа, само является обобщенным. Это связано с тем, что полностью квалифицированное имя включенного типа требует аргумента типа, чтобы полностью специфицировать вложенный тип.

Обобщенные типы перегружаются на основе числа аргументов в их списке параметров. Следующий пример иллюстрирует то, что я имею в виду:

```
public class Container {}
public class Container<T> {}
public class Container<T, R> {}
```

Каждое из этих объявлений корректно в пределах одного и того же пространства имен. Вы можете объявить столько обобщенных типов на основе идентификатора Container, сколько хотите, пока они отличаются по количеству параметров типа. Вы не можете объявить другой тип по имени Container<X, Y>, даже несмотря на то, что в списке параметров указаны другие идентификаторы. Правила перегрузки имен для обобщенных объявлений основаны на количестве параметров типа, а не на именах, присвоенных их указателям мест заполнения.

Объявляя обобщенный тип, вы тем самым объявляете открытый тип. Он называется так потому, что полностью специфицированный тип еще не известен. Объявляя другой тип на основе определения обобщенного, вы объявляете то, что называется конструируемым типом, как показано ниже:

```
public class MyClass<T>
{
   private Container<int> field1;
   private Container<T> field2;
}
```

Оба поля в приведенном объявлении MyClass<T> относятся к конструируемому типу, поскольку они объявляют новый тип на базе обобщенного типа Container<T>. Однако не каждый конструируемый тип является закрытым. Только field1 является закрытым типом, в то время как field2 — открытый тип, поскольку его финальный тип должен быть определен во время выполнения на основе аргументов типа из MyClass<T>.

В C# все идентификаторы объявлены и корректны в пределах определенной области (контекста). В границах метода, например, любой идентификатор, объявленный внутри фигурных скобок тела метода, доступен в пределах этого метода. Аналогичные правила действуют по отношению к идентификаторам внутри обобщений. В предыдущем примере идентификатор T действителен в пределах области объявления класса. Рассмотрим следующий пример вложенного класса:

```
public class MyClass<T>
{
    public class MyNestedClass<R>
    {
    }
}
```

Идентификатор R действителен внутри контекста вложенного класса, и вы не можете использовать его в объемлющем контексте объявления MyClass<T>. Однако вы можете использовать T во вложенном классе, поскольку вложенный класс определен внутри контекста, в котором идентификатор T действителен. Обычно считается нежелательным скрывать идентификаторы внешних аргументов во вложенных контекстах, как и идентификаторы имен переменных внутри сложенных областей выполнения. Например, попробуйте разобраться в таком запутанном коде:

```
public class MyClass<T>
{
    public class MyNestedClass<T>
    {
    }
    private Containter<T> field1;
        static void Main() {
        // Что это значит для MyNestedClass?
        MyClass<int> closedTypeInstance = null;
    }
}
```

Когда закрытый тип MyClass<int> объявляется в Main, что это означает для вложенного типа? Ответ — ничего! Несмотря на то что объявление MyNestedClass<T> использует тот же аргумент типа, он не расширяется до такого:

```
// Этого НЕ происходит!
public class MyClass<int>
{
    public class MyNestedClass<int>
    {
    }
    private Containter<int> field1;
}
```

Только потому, что параметр типа для MyClass<T> был специфицирован, это не значит, что также был специфицирован MyNestedClass<T>. Фактически, было бы аккуратнее описать результирующий MyClass<int> следующим образом:

```
public class MyClass<int>
{
    public class MyNestedClass<T>
    {
    }
    private Containter<int> field1;
}
```

MyNestedClass<T> остается открытым, даже невзирая на то, что использует тот же идентификатор в своем списке параметров, что и содержащий его тип. На самом деле в фигурных скобках MyNestedClass<T> внешний аргумент из MyClass<T> скрыт от доступа идентификатором из внутреннего контекста. Лучше объявить его так:

```
public class MyClass<T>
{
    public class MyNestedClass<R>
    {
        private T innerfield1;
        private R innerfield2;
    }
    private Containter<T> field1;
    static void Main() {
        MyClass<int> closedTypeInstance = null;
    }
}
```

Теперь, как видим, в области определения MyNestedClass<R> доступны оба параметра — T и R.

Обобщенные структуры и классы, как и обычные структуры и классы, могут содержать статические типы. Однако каждый закрытый тип, базирующийся на обобщенном типе, содержит свой собственный экземпляр статического типа. Если вы рассматриваете каждый закрытый тип как отдельный конкретный тип, это имеет прямой смысл. Поэтому, если вам нужно разделить статические данные между различными закрытыми типами, основанными на общем обобщенным типе, вам нужно обеспечить это другими средствами. Один из возможных приемов предполагает наличие отдельного, не обобщенного типа, содержащего статические данные, на который ссылаются обобщенные типы. Такое устройство обычно реализуется шаблоном Singletone (Одиночный).

На заметку! Имейте в виду, что обобщенные типы со статическими инициализаторами требуют, чтобы код инициализации запускался всякий раз, когда CLR создает закрытый тип на основе обобщенного. Сложные инициализаторы типов или статические конструкторы, возможно, могут увеличить рабочий набор приложения, если слишком много закрытых типов создается на основе такого обобщенного типа. Например, если вы создаете значительного размера тип структуры данных в инициализаторе обобщенного типа, вы можете тем самым создать причину существенного расхода памяти, если из такого обобщенного типа будет формироваться много конкретных типов.

Обобщенные интерфейсы

Наряду с классами и структурами вы можете также создавать объявления обобщенных интерфейсов. Эта концепция — естественное развитие обобщений структур и классов. Конечно, многие интерфейсы, объявленные в базовой библиотеке классов .NET 1.1, являются отличными кандидатами на замену их обобщенными версиями. Блестящий пример — `IEnumerable<T>`. Обобщенные контейнеры создают гораздо более эффективный код, чем контейнеры не обобщенные, если они содержит элементы типов значений, поскольку позволяют избежать излишней упаковки. Конечно, это естественно, что любой обобщенный перечислимый интерфейс должен иметь средства перечисления обобщенных элементов внутри. Таким образом, `IEnumerable<T>` существует, и любые перечислимые контейнеры, которые вы реализуете самостоятельно, должны реализовывать этот интерфейс. Альтернативно, вы должны получить их бесплатно, наследуя ваш пользовательский контейнер от `Collection<T>`.

На заметку! При создании собственных типов коллекций вы должны наследовать их от `Collection<T>` из пространства имен `System.Collections.ObjectModel`. Другие типы, такие как `List<T>`, не предназначены для наследования, а должны использоваться в качестве низкоуровневого механизма хранения. `Collection<T>` реализует защищенные виртуальные методы, которые вы можете переопределить для настройки его поведения, в то время как `List<T>` такой возможности не дает.

Обобщенные методы

C# поддерживает обобщенные методы. Любое объявление метода внутри структуры, класса или интерфейса может быть сделано обобщенным. Сюда входят и статические, и виртуальные или абстрактные методы. К тому же вы можете объявлять обобщенные методы в не обобщенных типах. Чтобы объявить обобщенный метод, просто добавьте аргумент типа в список аргументов в конце имени метода, но перед списком его параметров. Вы можете объявить любые типы в списке параметров метода, включая и тип возврата метода, как обобщенные параметры. Как и в случае с вложенными классами, не стоит скрывать идентификаторы типов из внешнего контекста, повторно используя те же идентификаторы во вложенном контексте, которым в данном случае выступает контекст обобщенного метода. Рассмотрим пример, когда может быть полезен обобщенный метод. В следующем коде я создаю контейнер, в который хочу добавлять содержимое другого обобщенного контейнера:

```
using System;
using System.Collections.Generic;
public class MyContainer<T> : IEnumerable<T>
{
    public void Add( T item ) {
        impl.Add( item );
    }
    // Converter<TInput, TOutput> - новый тип делегата, введенный
    // в .NET Framework 2.0, который может быть привязан к методу,
    // знающему, как конвертировать тип TInput в тип TOutput.
```

```
    public void Add<R>( MyContainer<R> otherContainer,
            Converter<R, T> converter ) {
        foreach( R item in otherContainer ) {
            impl.Add( converter(item) );
        }
    }
    public IEnumerator<T> GetEnumerator() {
        foreach( T item in impl ) {
            yield return item;
        }
    }
    System.Collections.IEnumerator System.Collections.IEnumerable.
GetEnumerator() {
        return GetEnumerator();
    }
    private List<T> impl = new List<T>();
}
public class EntryPoint
{
    static void Main() {
        MyContainer<long> lContainer = new MyContainer<long>();
        MyContainer<int> iContainer = new MyContainer<int>();
        lContainer.Add( 1 );
        lContainer.Add( 2 );
        iContainer.Add( 3 );
        iContainer.Add( 4 );
        lContainer.Add( iContainer,
                    EntryPoint.IntToLongConverter );
        foreach( long l in lContainer ) {
            Console.WriteLine( l );
        }
    }
    static long IntToLongConverter( int i ) {
        return i;
    }
}
```

Прежде всего, обратите внимание на обобщенный метод Add<R>, и также отметьте, что в MyContainer<T> есть две перегрузки Add. Ясно, что нужен метод для добавления экземпляров типа T — отсюда необходимость в Add(T). Однако было бы действительно удобно иметь возможность добавлять целый диапазон объектов другого закрытого типа, сформированного из MyContainer<T>, до тех пор, пока включенный тип исходного контейнера может быть преобразован во включенный тип целевого. Если взглянуть на Main, намерения станут ясны. Я хотел поместить объекты, содержащиеся внутри экземпляра MyContainer<int>, в экземпляр MyContainer<long>. Поэтому я создал обобщенный метод Add<R> для того, чтобы позволить принимать любой контейнер, содержащий в себе произвольные типы.

Такой прием, однако, требует ухищрений. Логически то, что я пытаюсь сделать, имеет очевидный смысл. Я хочу добавить коллекцию элементов int в коллекцию элементов long, причем я знаю, что int легко неявно конвертируется в long, так

что это должно у меня получиться. Но при этом следует принять во внимание, что обобщения формируются динамически во время выполнения. И во время выполнения нет никаких гарантий, какой именно закрытый тип, сформированный из MyContainer<T>, увидит метод Add<R>. Это может быть MyContainer<Apples>, а Apples может и не быть неявно преобразуем в long, предполагая, что он передан MyContainer<long>.Add<Apples>. Те из вас, кто использует шаблоны C++, догадаются, что такой трюк работать не будет, поскольку компилятор уведомит, если вы попытаетесь выполнить некорректное преобразование во время компиляции. Однако обобщения лишены такой роскоши времени компиляции, поэтому предусмотрены дополнительные ограничения компиляции, чтобы исключить подобную вещь. Поэтому придется поискать другое решение и предусмотреть делегат преобразования для выполнения такой работы.

Библиотека базовых классов предлагает специально для этого случая System.Converter<T, R>. Синтаксис этого делегата может показаться несколько чужеродным, но это просто обобщенное объявление делегата, которое я подробно опишу в разделе "Обобщенные делегаты". Когда кто-то вызывает Add<R>, он может представить экземпляр обобщенного делегата Converter<T, R>, указывающий метод, который знает, как преобразовать исходный тип в целевой. Это объясняет потребность в методе IntToLongConverter в предыдущем примере. Метод Add<R> затем использует этот делегат для выполнения преобразования от одного типа к другому. В данном случае преобразование неявно, но оно должно было быть вынесено наружу, поскольку во время компиляции компилятор должен учесть тот факт, что метод Add<R> может принять любой тип.

Чтобы облегчить перечисление контейнера, я также объявил MyContainer<T> так, что он реализует IEnumerable<T>. Это позволит вам использовать синтаксически интуитивно понятную конструкцию foreach. Вы заметите здесь синтаксис, который может показаться вам незнакомым, если вы незнакомы с итераторами C#[1]. Однако обратите внимание, насколько легко создать перечислитель для этого класса с помощью ключевого слова yield. Это полезное дополнение языка, поскольку объявление и конструирование объектов, перечисляющих содержимое контейнеров — традиционно трудоемкая работа.

Обобщенные делегаты

Довольно часто обобщения используются в контексте контейнерных типов, где поле закрытого типа или внутренний массив основан на заданном аргументе типа. Обобщенные методы расширяют способность обобщенных типов, представляя более тонкую структурированность обобщенного контекста. Я еще расскажу о мощи обобщенных делегатов.

Вы уже знакомы с многоуважаемым делегатом. Если вам нужно объявить делегат, принимающий два параметра, первый из них типа long, а второй — object, то вы должны сделать это так:

```
public delegate void MyDelegate( long l, object o );
```

В предыдущем разделе вы уже видели пример обобщенного делегата, когда я показывал делегат для преобразования типов. Объявление такого обобщенного делегата для преобразования выглядит следующим образом:

---

[1] Тема итераторов полностью раскрывается в главе 9.

```
public delegate TOutput Converter<TInput, TOutput>(
   TInput input
);
```

Он выглядит как любой другой делегат, на с тем отличием, что сразу за его именем следует список параметров типа. Как и не обобщенные делегаты выглядят подобно объявлениям методов без тела, так и объявления обобщенных делегатов выглядят почти идентично обобщенным методам без тела. Список параметров-типов следует за именем делегата, но предшествует списку параметров самого делегата.

Обобщенный конвертер использует идентификаторы-указатели мест заполнения TInput и TOutput внутри списка параметров типов, и эти типы применяются повсюду в объявлении делегата. В объявлениях обобщенных делегатов типы из списка параметров типов находятся в контексте всего объявления делегата, включая тип возврата, как это показано в предыдущем объявлении делегата обобщенного конвертера.

Создание экземпляра делегата Converter<TInput, TOutput> — то же самое, что и создание экземпляра любого другого делегата. Когда вы создаете экземпляр обобщенного делегата, вы можете использовать операцию new и явно предоставить список типов во время компиляции. Или же можно просто использовать сокращенный синтаксис, который я применял в примере MyContainer<T> в предыдущем разделе — тогда компилятор сам выводит типы параметров. Для удобства я еще раз приведу текст метода Main этого примера:

```
static void Main() {
   MyContainer<long> lContainer = new MyContainer<long>();
   MyContainer<int> iContainer = new MyContainer<int>();
   lContainer.Add( 1 );
   lContainer.Add( 2 );
   iContainer.Add( 3 );
   iContainer.Add( 4 );
   lContainer.Add( iContainer,
                   EntryPoint.IntToLongConverter );
   foreach( long l in lContainer ) {
      Console.WriteLine( l );
   }
}
```

Обратите внимание, что второй параметр метода Add — просто ссылка на метод, а не явное создание самого делегата. Это работает благодаря групповым правилам преобразования, определенным в языке C#. Когда действительный делегат создается из метода, то закрытый тип выводится из обобщения с использованием сложного алгоритма сопоставления типов параметров самого метода IntToLongConverter. Фактически, вызов Add<T> лишен какого-либо явного списка параметров типа в точке этого вызова. Компилятор в состоянии выполнить ту же работу по сопоставлению типов, чтобы вывести закрытую форму вызванного метода Add<T>, которой в данном случае будет Add<int>. Вы можете также с успехом написать код, подобный следующему, в котором каждый тип указан явно:

```
static void Main() {
   MyContainer<long> lContainer = new MyContainer<long>();
   MyContainer<int> iContainer = new MyContainer<int>();
```

```
        lContainer.Add( 1 );
        lContainer.Add( 2 );
        iContainer.Add( 3 );
        iContainer.Add( 4 );
        lContainer.Add<int>( iContainer,
            new Converter<int, long>( EntryPoint.IntToLongConverter) );
        foreach( long l in lContainer ) {
            Console.WriteLine( l );
        }
    }
```

В этом примере все типы заданы явно, и компилятору не приходится выводить их во время компиляции. В любом случае, результирующий код IL будет одним и тем же. В большинстве случаев вы можете полагаться на механизм определения типа компилятором. Однако, в зависимости от сложности вашего кода, иногда может быть имеет смысл помочь компилятору, передав явный список параметров-типов.

Наряду с обеспечением возможности вынести преобразования типов из класса контейнера, как в предыдущем примере, обобщенные делегаты помогают решать специальные проблемы, что демонстрируется в следующем коде:

```
// ЭТО НЕ РАБОТАЕТ, КАК ОЖИДАЕТСЯ!!!
using System;
using System.Collections.Generic;
public delegate void MyDelegate( int i );
public class DelegateContainer<T>
{
    public void Add( T del ) {
        imp.Add( del );
    }
    public void CallDelegates( int k ) {
        foreach( T del in imp ) {
            // del( k );
        }
    }
    private List<T> imp = new List<T>();
}
public class EntryPoint
{
    static void Main() {
        DelegateContainer<MyDelegate> delegates =
            new DelegateContainer<MyDelegate>();
        delegates.Add( EntryPoint.PrintInt );
    }
    static void PrintInt( int i ) {
        Console.WriteLine( i );
    }
}
```

В таком виде, как он написан, этот код компилируется. Однако обратите внимание на закомментированную строку внутри метода `CallDelegates`. Если вы удалите комментарий с нее и попытаетесь перекомпилировать компилятором Microsoft, то получите следующую ошибку:

```
error CS0118: 'del' is a 'variable' but is used like a 'method'
ошибка CS0118: 'del' является 'переменной', но используется подобно 'методу'
```

Проблема в том, что компилятор никак не может знать, что тип, на который ссылается указатель места заполнения T, является делегатом. Те из вас, кто забежал вперед в этой главе, могут недоумевать — почему нет никакой формы ограничения (об ограничениях я расскажу дальше), чтобы подсказать компилятору, что это делегат. Но даже если бы такая форма ограничения была, компилятор может не знать, как вызвать этот делегат. Ограничение не может нести никакой информации о количестве параметров, принимаемых делегатом. Напомню, что в отличие от шаблонов C++, обобщения являются динамическими, и закрытые типы формируются во время выполнения, а не во время компиляции. Поэтому во время выполнения делегат, представленный del, может принять произвольное количество параметров. Я могу себе представить головную боль, вызванную попытками придумать способ занести динамическое количество параметров в стек перед вызовом делегата. По всем этим причинам редко имеет смысл создавать закрытый тип из обобщенного, когда один из аргументов-типов является типом делегата, так как, в конечном счете, вы все равно не сможете осуществить нормальный вызов через него.

Что можно сделать, чтобы помочь в этой ситуации — так это применить обобщенный делегат, чтобы предоставить компилятору немного больше информации о том, что вы хотите делать с этим делегатом. Например, используя обобщенный делегат, вы можете на самом деле сказать: "Я хотел бы, чтобы ты использовал делегаты, принимающие только два параметра и возвращать произвольный тип". Этой информации достаточно, чтобы позволить компилятору воспринять блок и сгенерировать код, имеющий смысл для обобщения. В конце концов, если вы предоставите компилятору этот объем информации, то, по крайней мере, он будет знать, сколько параметров нужно затолкнуть в стек перед вызовом через делегат. Следующий код демонстрирует, как можно исправить предыдущую ситуацию:

```
using System;
using System.Collections.Generic;
public delegate void MyDelegate<T>( T i );
public class DelegateContainer<T>
{
    public void Add( MyDelegate<T> del ) {
        imp.Add( del );
    }
    public void CallDelegates( T k ) {
        foreach( MyDelegate<T> del in imp ) {
            del( k );
        }
    }
    private List<MyDelegate<T> > imp = new List<MyDelegate<T> >();
}

public class EntryPoint
{
    static void Main() {
        DelegateContainer<int> delegates =
            new DelegateContainer<int>();
```

```
        delegates.Add( EntryPoint.PrintInt );
        delegates.CallDelegates( 42 );
    }
    static void PrintInt( int i ) {
        Console.WriteLine( i );
    }
}
```

Преобразование обобщенного типа

Как я уже упоминал ранее в этой главе, не существует неявных преобразований разных конструируемых типов, сформированных из одного обобщенного типа. Те же правила, которые действуют при определении, является ли объект типа X преобразуемым к объекту типа Y, в равной мере касаются определения конвертируемости объекта типа `List<int>` к объекту типа `List<object>`. Когда такое преобразование желательно, вы должны создать пользовательскую операцию преобразования, как в случае преобразования объектов типа X к объектам типа Y, когда они разделяют отношение наследования. В противном случае вам нужно создать метод преобразования, чтобы превратить один тип в другой. Например, следующий код неверен:

```
// НЕВЕРНЫЙ КОД!!!
public void SomeMethod( List<int> theList ) {
    List<object> theSameList = theList; // Так нельзя!!!
}
```

Если вы заглядывали в документацию по `List<T>`, то, вероятно, заметили там обобщенный метод по имени `ConvertAll<TOutput>`. Используя этот метод, вы можете конвертировать обобщенный список типа `List<int>` в `List<object>`. Однако вы должны передавать этому методу экземпляр обобщенного делегата преобразования, как было описано в предыдущем разделе. Это — единственный способ для метода узнать, как ему следует конвертировать каждый экземпляр содержимого из исходного типа в целевой. Несмотря на то что вы можете вызвать метод для преобразования `List<int>` в `List<object>`, все равно вам придется представить явные средства преобразования `int` в `object`.

Те, кто знаком с шаблоном Strategy (Стратегия), увидят здесь нечто знакомое. По сути, вы можете обеспечить метод `ConvertAll<TOutput>` во время выполнения средствами осуществления преобразования содержащихся в коллекции экземпляров, которые, в зависимости от сложности преобразования, могут быть настроены под платформу, на которой выполняются. Другими словами, если вы собираетесь конвертировать `List<Apples>` в `List<Oranges>`, то вам придется представить несколько методов различных преобразований на выбор, в зависимости обстоятельств. Например, может быть, один из них будет предусмотрен для среды с богатыми ресурсами, чтобы работать с максимальной скоростью. Другая версия может быть оптимизирована по минимальному потреблению ресурсов, но работать медленнее. Во время выполнения строится правильный преобразующий делегат для привязки к методу преобразования, максимально подходящий для выполнения работы в конкретном случае.

Выражение значения по умолчанию

Иногда при работе с определениями обобщенных типов и обобщенных методов вам нужно инициализировать объект или экземпляр значения параметризованного типа его значением по умолчанию. Вспомните, что значение по умолчанию для ссылки — это то же, что установка его в `null`, в то время как значение по умолчанию для типа значений эквивалентно установке всех его бит в `0`. Вам нужно выражение для обобщений, учитывающее это семантическое отличие, и для этой задачи вы можете использовать выражение значения по умолчанию (`default`), показанное в следующем примере кода:

```
using System;
public class MyContainer<T>
{
    public MyContainer() {
        // Начальное наполнение.
        imp = new T[ 4 ];
        for( int i = 0; i < imp.Length; ++i ) {
            imp[i] = default(T);
        }
    }

    public bool IsNull( int i ) {
        if( i < 0 || i >= imp.Length ) {
            throw new ArgumentOutOfRangeException();
        }
        if( imp[i] == null ) {
            return true;
        } else {
            return false;
        }
    }
    private T[] imp;
}
public class EntryPoint
{
    static void Main() {
        MyContainer<int> intColl =
            new MyContainer<int>();
        MyContainer<object> objColl =
            new MyContainer<object>();
        Console.WriteLine( intColl.IsNull(0) );
        Console.WriteLine( objColl.IsNull(0) );
    }
}
```

Обратите внимание на синтаксис внутри конструктора `MyContainer<T>`, где каждый элемент массива явно инициализируется его значением по умолчанию. Во время выполнения тип `T` может быть типом значений или ссылочным типом, поэтому вы не можете просто присвоить значение `null` и рассчитывать, что это будет работать с типами значений. Фактически, если вы попытаетесь присвоить

imp[i] значение null, то компилятор выдаст дружеское напоминание в виде следующей ошибки:

```
default_value_1.cs(8,13): error CS0403: Cannot convert null to
type parameter 'T' because it could be a value type. Consider using
'default(T)' instead.
default_value_1.cs(8,13): ошибка CS0403: Невозможно преобразовать null
к параметру типа 'T', поскольку он должен быть типом значения. Взамен
используйте 'default(T)'.
```

Также вы должны использовать выражение default при проверке переменной на null, поскольку, в конце концов, это может быть тип значения. Однако в этом случае компилятор не может помочь вам узнать, когда вы должны поступать так, как показано в примере. Если вы запустите предыдущий код, то получите следующий вывод:

```
False
True
```

Возможно, это не тот результат, которого вы ожидали. Если вы модифицируете код так, чтобы метод IsNull выглядел, как показано в следующем примере, то получите вывод, который более отвечает ожидаемому:

```
public class MyContainer<T>
{
    public MyContainer() {
        // Create initial capacity.
        imp = new T[ 4 ];
        for( int i = 0; i < imp.Length; ++i ) {
            imp[i] = default(T);
        }
    }
    public bool IsNull( int i ) {
        if( i < 0 || i >= imp.Length ) {
            throw new ArgumentOutOfRangeException();
        }
        if( Object.Equals(imp[i], default(T)) ) {
            return true;
        } else {
            return false;
        }
    }
    private T[] imp;
}
```

Типы, допускающие значения null

Предыдущей дискуссии касается концепция значений null и семантический смысл, который она может нести. Состояние null для ссылочных типов легко представить. Если значение ссылки установлено в null, обычно это значит, что переменная просто не имеет значения. Это семантически вовсе не то же самое, как если сказать, что значение равно 0. Семантически переменная, установленная

в null, не имеет значения — даже значения 0. В отношении же типов значений традиционно намного труднее представить семантическое значение null. Если вы устанавливаете значение 0, это может означать null. Но что делать, если вам действительно нужно представить значение 0, а не null? Многие приемы предполагают поддержку дополнительного булевского значения, которое сопровождает значение, наподобие isNull.

Чтобы избавить вас от необходимости использования такого нелепого, чреватого ошибками механизма, библиотека базовых классов .NET предлагает тип System.Nullable<T>, который демонстрируется в следующем коде:

```
using System;
public class Employee
{
    public Employee( string firstName,
                     string lastName ) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.terminationDate = null;
        this.ssn = default(Nullable<long>);
    }
    public string firstName;
    public string lastName;
    public Nullable<DateTime> terminationDate;
    public long? ssn; // Сокращенная нотация
}
public class EntryPoint
{
    static void Main() {
        Employee emp = new Employee( "Vasya",
                                     "Pupkin" );
        emp.ssn = 1234567890;
        Console.WriteLine( "{0} {1}",
                           emp.firstName,
                           emp.lastName );
        if( emp.terminationDate.HasValue ) {
            Console.WriteLine( "Start Date: {0}",
                               emp.terminationDate );
        }
        long tempSSN = emp.ssn ?? -1;
        Console.WriteLine( "SSN: {0}",
                           tempSSN );
    }
}
```

Этот код демонстрирует два способа объявления типа, допускающего null (nullable type). Первое поле, допускающее null, внутри типа Emloyee — terminationDate, объявленное с использованием типа System.Nullable<DateTime>. Одно из свойств Nullable<T> — это HasValue, которое возвращает true, если значение не равно null, и false в противном случае. Второе поле, допускающее null, внутри Employee — ssn; однако на этот раз я решил применить сокращенную нотацию C# для типов, допускающих значения null, когда объявление типа поля дополняется

вопросительным знаком. Внутренне компилятор обрабатывает его точно так же, как объявление поля terminationDate.

На заметку! Лично мне кажется, что даже несмотря на то, что явное применение Nullable<T> требует больше клавиатурного ввода, его гораздо сложнее не заметить при чтении кода, чем малюсенький вопросительный знак в конце типа поля. Всегда предпочитайте читабельный код чересчур изощренному.

И последнее, о чем надо сказать в связи с типами, допускающими null, — это выполнение присваиваний. В конструкторе Employee вы можете видеть, что я первым делом присвоил null полям типов, допускающих null. Компилятор использует неявное преобразование для значения null, чтобы все было правильно. Фактически, когда я инициирую поле ssn в конструкторе, то используется синтаксис выражения default, и то же самое делает компилятор, когда я присваиваю значение null полю terminationDate. И, наконец, в методе Main я хочу присвоить tempSSN значение emp.ssn. Однако поскольку emp.ssn может быть null, что будет присвоено tempSSN, если так случится, что в emp.ssn не окажется значения? Здесь вы должны применить операцию null-слияния ??. Эта операция позволяет назначить значение, отличное от null, которое должно быть установлено в том случае, когда переменная, от которой выполняется присваивание, не имеет значения. То есть в предыдущем примере я говорю: "Установить в tempSSN значение emp.ssn, а если emp.ssn не имеет значения, установить tempSSN равным -1". Вооружившись таким инструментом, очень легко представлять внутри системы значения, которые семантически могут быть null, что удобно для представления полей базы данных, которые допускают значения null.

Контроль доступа к конструируемым типам

Когда вы строите конструируемые типы из обобщенных, следует учитывать доступность как обобщенного типа, так и типов, указанных в аргументах типа, чтобы определить доступность всего конструируемого типа в целом. Например, следующий код неверен и компилироваться не будет:

```
public class Outer
{
   private class Nested
   {
   }
   public class GenericNested<T>
   {
   }
   private GenericNested<Nested> field1;
   public GenericNested<Nested> field2; // Ошибка!
}
```

Проблема связана с field2. Тип Nested объявлен приватным (private), поэтому как GenericNested<Nested> может быть public? Конечно, никак. Доступность конструируемых типов определяется как пересечение доступности обобщенного типа и типов, указанных в списке аргументов.

Обобщения и наследование

Обобщенные типы C# не могут напрямую наследоваться от параметра типа. Однако вы можете использовать следующие параметры типа для конструирования базовых типов, которые они могут наследовать:

```
// Это неправильно!!
public class MyClass<T> : T
{
}
// А это правильно.
public class MyClass<T> : Stack<T>
{
}
```

Совет. В шаблонах C++ прямое наследование от параметра типа обеспечивает особую гибкость. Если вы когда-либо имели дело с библиотекой ATL (Active Template Library — библиотека активных шаблонов) для разработки компонентов COM, вы знакомы с такой техникой, поскольку ATL интенсивно использует ее, чтобы избежать необходимости вызова виртуальных методов. Та же техника применяется в шаблонах C++ для генерации иерархий во время компиляции. За дополнительными примерами я рекомендую обратиться к книге Андрея Александреску (Andrei Alexandrescu) *Modern C++ Design: Generic Programming and Design Patterns Applied* (Boston, MA: Addison-Wesley Professional, 2001 г.). Это еще один пример, демонстрирующий статическую природу шаблонов C++, в то время как обобщения C# динамичны.

Давайте рассмотрим прием, который вы можете применять, чтобы до некоторой степени эмулировать такое же поведение. Как это часто бывает, для обеспечения чего-то похожего можно добавить дополнительный промежуточный слой. В C++, когда шаблонный тип наследуется напрямую от одного из типов-аргументов, часто предполагается, что тип, указанный в аргументе, представляет определенное желательное поведение. Например, применяя шаблоны C++, вы можете сделать следующее:

```
// ПРИМЕЧАНИЕ: Это код C++
class Employee
{
   public:
      long get_salary() {
         return salary;
      }
      void set_salary( long salary ) {
         this->salary = salary;
      }
   private:
      long salary;
};
template< class T >
class MyClass : public T
{
};
```

```
void main()
{
    MyClass<Employee> myInstance;
    myInstance.get_salary();
}
```

В функции main обратите внимание на вызов get_salary. Несмотря на то что на первый взгляд он выглядит странно, тем не менее, работает он хорошо, потому что MyClass<T> наследует реализацию типа, переданного в T в момент компиляции. В данном случае этот тип — Employee, он реализует get_salary, и MyClass<Employee> наследует эту реализацию. Ясно, что делается предположение, что тип, указанный в T, будет поддерживать метод get_salary. Если это не так, компилятор C++ сообщит об ошибке. Это — форма статического полиморфизма, или программирования на базе политик. В традиционных случаях полиморфизм, рассматриваемый в контексте виртуальных методов, является динамическим полиморфизмом. Вы не можете реализовать статический полиморфизм средствами обобщений C#. Однако вы можете потребовать, чтобы аргумент типа, переданный при формировании закрытого типа, поддерживал определенный контракт, используя для этого механизм *ограничений* (constraints), который я опишу в следующем разделе.

Ограничения

До сих пор большинство примеров обобщений, которые я приводил, включали некоторого рода классы в стиле коллекций, способные хранить множества объектов или значений определенного типа. Но часто у вас возникнет необходимость в создании обобщенных типов, которые не только содержат экземпляры различных типов, но также напрямую используют эти объекты, посредством вызова их методов или обращения к их свойствам. Например, предположим, что у вас есть обобщенный тип, хранящий экземпляры произвольных геометрических фигур, реализующих свойство по имени Area (площадь). Также вам нужно, чтобы обобщенный тип реализовал свойство, скажем, TotalArea, где все площади составных фигур складываются. Здесь нужна гарантия, чтобы все геометрические фигуры в обобщенном контейнере реализовывали свойство Area. Вы можете быть склонны написать следующий код:

```
using System;
using System.Collections.Generic;
public interface IShape
{
    double Area {
        get;
    }
}
public class Circle : IShape
{
    public Circle( double radius ) {
        this.radius = radius;
    }
```

```
    public double Area {
        get {
            return 3.1415*radius*radius;
        }
    }
    private double radius;
}
public class Rect : IShape
{
    public Rect( double width, double height ) {
        this.width = width;
        this.height = height;
    }
    public double Area {
        get {
            return width*height;
        }
    }
    private double width;
    private double height;
}
public class Shapes<T>
{
    public double TotalArea {
        get {
            double acc = 0;
            foreach( T shape in shapes ) {
                // Это не компилируется!!!
                acc += shape.Area;
            }
            return acc;
        }
    }
    public void Add( T shape ) {
        shapes.Add( shape );
    }
    private List<T> shapes = new List<T>();
}
public class EntryPoint
{
    static void Main() {
        Shapes<IShape> shapes = new Shapes<IShape>();
        shapes.Add( new Circle(2) );
        shapes.Add( new Rect(3, 5) );
        Console.WriteLine( "Общая площадь: {0}",
                           shapes.TotalArea );
    }
}
```

Есть одна главная проблема, из-за которой код не будет компилироваться. Причина — в строке, находящейся внутри свойства `TotalArea` класса `Shapes<T>`. Компилятор выдает следующую ошибку:

```
error CS0117: 'T' does not contain a definition for 'Area'
ошибка CS0117: 'T' не содержит определения 'Area'
```

Это требование, чтобы тип содержимого T поддерживал свойство Area, выглядит очень похожим на контракт. Так оно и есть! Обобщения C# являются динамическими, а не статическими по своей природе, поэтому вы не можете достичь желаемого эффекта, не предоставив некоторой дополнительной информации. Всякий раз, когда вы слышите слово *контракт* в мире C#, вы можете подумать об интерфейсах. Поэтому я решаю, чтобы обе мои фигуры реализовали интерфейс IShape. Таким образом, интерфейс IShape определит контракт, и фигуры реализуют его. Однако этого все еще недостаточно для компилятора C#, чтобы он мог скомпилировать приведенный код.

Обобщения C# должны иметь способ навязать правило, гласящее, что тип T обязан поддерживать определенный контракт во время выполнения, поскольку конструируемые типы формируются динамически во время выполнения. Наивная попытка решения этой проблемы могла бы выглядеть так:

```
public class Shapes<T>
{
    public double TotalArea {
        get {
            double acc = 0;
            foreach( T shape in shapes ) {
                // НЕ ДЕЛАЙТЕ ЭТОГО!!!
                IShape theShape = (IShape) shape;
                acc += theShape.Area;
            }
            return acc;
        }
    }
    public void Add( T shape ) {
        shapes.Add( shape );
    }
    private List<T> shapes = new List<T>();
}
```

Эта модификация Shapes<T> в самом деле будет компилироваться и работать в большинстве случаев. Однако такое обобщение теряет некоторую часть своей чистоты из-за приведения типа внутри цикла foreach. Только представьте, что будет, если во время ночных бдений, вызванных кофеином, вы попытаетесь создать сконструированный тип Shapes<int>. Компилятор с удовольствием его проглотит. Но что случится, когда вы попытаетесь получить свойство TotalArea от экземпляра Shapes<int>? Как и можно ожидать, вы получите исключение времени выполнения, когда метод доступа get свойства TotalArea попытается выполнить приведение int к IShape. Одним из главных преимуществ использования обобщений является повышенная безопасность типов, но в данном примере я попросту отбросил всю эту безопасность. Так что же делать? Ответ содержится в концепции, называемой *ограничениями обобщений* (generic constraints). Взгляните на правильную реализацию:

```
public class Shapes<T>
    where T: IShape
{
    public double TotalArea {
        get {
            double acc = 0;
            foreach( T shape in shapes ) {
                acc += shape.Area;
            }
            return acc;
        }
    }
    public void Add( T shape ) {
        shapes.Add( shape );
    }
    private List<T> shapes = new List<T>();
}
```

Обратите внимание на дополнительную строку, которая следует сразу за первой строкой объявления класса и использует ключевое слово `where`. Это говорит: "Определить класс `Shapes<T>`, где `T` должен реализовывать `IShape`". Теперь у компилятора есть все необходимое для обеспечения безопасности типов, и JIT-компилятор также имеет все необходимое для построения кода во время выполнения. Компилятор получает подсказку, помогающую известить вас об ошибке времени компиляции, когда вы попытаетесь создать конструируемый тип, в котором `T` не реализует `IShape`.

Синтаксис ограничений замечательно прост. Может существовать одна конструкция `where` для каждого параметра типа. Любое количество интерфейсов могут быть перечислены вслед за параметром типа в конструкции `where`, но класс — максимум один. Это ограничение интуитивно понятно, поскольку любой данный тип может наследоваться только от одного класса, но может реализовывать неограниченное количество интерфейсов. Дополнительно вы можете использовать специальные ключевые слова в конструкции ограничения для определенного аргумента типа. Только одно ограничение может указывать имя класса (поскольку CLR не поддерживает множественного наследования), поэтому такое ограничение называется *первичным ограничением*. Кроме того, вместо указания имени класса первичное ограничение может перечислять специальные слова `class` или `struct`, используемые для указания того, что параметр типа может быть классом или структурой. Конструкция ограничения может включать столько вторичных ограничений, сколько возможно, и обычно они представляют собой список интерфейсов. И, наконец, вы можете перечислить ограничения конструктора, имеющие форму `new()`. Они ограничивают параметризованный тип так, что он должен иметь конструктор по умолчанию, не имеющий параметров. Типы классов должны иметь явно определенный конструктор по умолчанию, в то время как ограничение `new()` для типов значений принято автоматически, поскольку они всегда имеют сгенерированный системой конструктор по умолчанию.

Принято указывать каждую конструкцию `where` в отдельной строке, в любом порядке под заголовком класса. Каждое ограничение, следующее за двоеточием после `where`, отделяется от соседних запятой. Взглянем на несколько примеров ограничений:

```
using System.Collections.Generic;
public class MyValueList<T>
    where T: struct
    // А так нельзя:
    // where T: struct, new()
{
    public void Add( T v ) {
        imp.Add( v );
    }
    private List<T> imp = new List<T>();
}
public class EntryPoint
{
    static void Main() {
        MyValueList<int> intList =
            new MyValueList<int>();
        intList.Add( 123 );
        // ТАК НЕЛЬЗЯ.
        // MyValueList<object> objList =
        // new MyValueList<object>();
    }
}
```

В этом коде вы можете видеть пример ограничения struct. По той или иной причине вы можете счесть необходимым создать контейнер, который может содержать только типы значений. Альтернативно может быть сформулировано ограничение, требующее разрешать только типы классов. Кстати, в версии компилятора C# из Visual Studio мне не удалось создать ограничение, включающее и класс и структуру. Конечно, это бессмысленно, поскольку тот же эффект дает отсутствие обоих ограничений. Тем не менее, компилятор выдаст ошибку, если вы попытаетесь сделать это, сообщив следующее:

```
error CS0449: The 'class' or 'struct' constraint must come before any
other constraints
```
ошибка CS0449: Ограничение 'class' или 'struct' должно находиться перед любыми другими ограничениями

Наверное, компилятору стоило бы сообщить, что допускается только одно первичное ограничение. Вы увидите далее, что я закомментировал строку альтернативного ограничения, где попытался включить ограничение new() для того, чтобы потребовать от типа T поддерживать конструктор по умолчанию. Ясно, что для типов значений это ограничение избыточно, но не должно было бы никак повредить. Тем не менее, компилятор не позволит вам применить ограничение new() вместе с ограничением struct. А теперь давайте рассмотрим немного более сложный пример, показывающий две конструкции ограничений:

```
using System;
using System.Collections.Generic;
public interface IValue
{
    // Методы IValue.
}
```

```
public class MyDictionary<TKey, TValue>
   where TKey: struct, IComparable<TKey>
   where TValue: IValue, new()
{
   public void Add( TKey key, TValue val ) {
      imp.Add( key, val );
   }
   private Dictionary<TKey, TValue> imp
      = new Dictionary<TKey, TValue>();
}
```

Я объявил `MyDictionary<TKey, TValue>` таким образом, что значение ключа ограничено типами значений. Также я хочу, чтобы значения ключей были сравнимы между собой, поэтому потребовал, чтобы тип `TKey` реализовывал `IComparable<TKey>`. Этот пример показывает две конструкции ограничений — по одному для каждого параметра типа. В этом случае я позволяю типу `TValue` быть структурой или классом, но при этом требую, чтобы он поддерживал интерфейс `IValue` вместе с конструктором по умолчанию.

Вообще механизм ограничений, встроенный в обобщения C#, прост и понятен. Сложностью ограничений легко управлять и расшифровывать ее с минимумом (или без) сюрпризов. По мере развития языка и CLR можно ожидать, что эта часть будет дополнена по мере нахождения областей применения обобщений. Например, возможность применения ограничений `class` и `struct` была добавлена к стандарту относительно недавно.

И, наконец, формат ограничений на обобщенные интерфейсы идентичен тому же формату для классов и структур.

Ограничения на неклассовых типах

До сих пор я говорил об ограничениях в контексте классов, структур и интерфейсов. В действительности любая сущность, которую вы можете объявить обобщенным образом, допускает применение ограничений. В объявлениях обобщенных методов и делегатов ограничение следует за списком формальных параметров метода или делегата. Применение операторов ограничения в объявлениях методов и делегатов порождает несколько странно выглядящий синтаксис, как показано в следующем примере:

```
using System;
public delegate R Operation<T1, T2, R>( T1 val1,
                                         T2 val2 )

   where T1: struct
   where T2: struct
   where R: struct;
public class EntryPoint
{
   public static double Add( int val1, float val2 ) {
      return val1 + val2;
   }
   static void Main() {
      Operation<int, float, double> op =
         new Operation<int, float, double>( EntryPoint.Add );
```

```
            Console.WriteLine( "{0} + {1} = {2}",
                1, 3.2, op(1, 3.2f) );
        }
    }
```

Я объявил обобщенный делегат для методы операции, принимающий два параметра и имеющий возвращаемое значение. Мое ограничение заключается в том, что параметры и возвращаемое значение должны быть типами значений. Для обобщенных методов конструкции ограничений следуют за объявлением метода, но предшествуют его телу. Обратите внимание, что в точке создания в методе Main я должен был сообщить компилятору точный сконструированный тип делегата `Operation<T1, T2, R>`, который мне нужен.

Обобщенные системные коллекции

Похоже, что наиболее естественное применение обобщений в C# и CLR относится к типам коллекций. Может быть, это потому, что вы можете получить огромный прирост эффективности, применяя обобщенные контейнеры для хранения типов значений, по сравнению с типами коллекций из пространства имен System.Collections. Конечно, вы не можете не заметить дополнительной безопасности типов, которую влечет за собой применение обобщенных коллекций. Всегда, когда возрастает безопасность типов, вы гарантированно получите сокращение количества исключений времени выполнения, поскольку компилятор может перехватить многие из них на этапе компиляции.

Я советую вам заглянуть в документацию .NET Framework по пространству имен System.Collections.Generic. Там вы найдете все классы обобщенных коллекций, которые стали доступными в .NET Framework. В это пространство имен входят `Dictionary<TKey, TValue>`, `LinkedList<T>`, `List<T>`, `Queue<T>`, `SortedDictionary<TKey, TValue>`, `SortedList<T>`, `HashSet<T>` и `Stack<T>`.

Если взглянуть на их имена, применение этих типов должно показаться вам знакомым, поскольку напоминает имена не обобщенных классов из System.Collections. Хотя набор контейнеров внутри пространства имен System.Collections.Generic может показаться недостаточным для ваших нужд, у вас всегда есть возможность создавать собственные коллекции, особенно на базе расширяемых типов из пространства System.Collections.ObjectModel.

При создании ваших собственных типов коллекций у вас часто будет возникать потребность в сравнении объектов, хранящихся в коллекции. При кодировании на C# кажется естественным использовать встроенные операции равенства и неравенства для выполнения сравнений. Однако я советую воздерживаться от этого, поскольку поддержка таких операций классами и структурами, хоть и возможна, но не является частью CLS. Некоторые языки имеют слишком медленную поддержку этих операций. Поэтому ваш контейнер должен быть специально подготовлен к тем случаям, когда содержащиеся в нем типы не поддерживают операций сравнения. Это одна из причин существования таких интерфейсов, как IComparer и IComparable.

Когда вы создаете экземпляр типа SortedList внутри System.Collections, у вас есть возможность представить экземпляр объекта, поддерживающего IComparer. SortedList затем использует этот объект, когда возникает необхо-

димость сравнения двух экземпляров ключей, содержащихся в нем. Если вы не представите объекта, поддерживающего IComparer, то SortedList ищет реализации интерфейса IComparable в содержащихся объектах ключей для выполнения сравнения. Естественно, вы должны будете представить явный компаратор, если содержащиеся в коллекции объекты ключей не поддерживают IComparable. Перегруженные версии конструктора, принимающие тип IComparer, предназначены специально для этого.

Обобщенная версия сортированного списка, SortedList<TKey, TValue>, следует тому же шаблону в отношении сортировки. Когда вы создаете экземпляр SortedList<TKey, TValue>, у вас есть возможность представить объект, реализующий интерфейс IComparer<T>, чтобы он сравнивал два ключа. Если вы этого не делаете, то SortedList<TKey, TValue> по умолчанию использует то, что называется *обобщенный компаратор* (generic comparer). Обобщенный компаратор — это просто объект, унаследованный от абстрактного класса Comparer<T>, который может быть получен от статического свойства Comparer<T>.Default. Исходя их не обобщенного варианта SortedList, вы можете подумать, что если создатель объекта SortedList<TKey, TValue> не предусмотрел компаратор, он может просто поискать реализацию IComparable<T> в типе содержащегося ключа. Такой подход привел бы к проблемам, поскольку тип содержащегося ключа может поддерживать как IComparable<T>, так и не обобщенный IComparable. Поэтому обобщенный компаратор действует как дополнительный промежуточный слой. Компаратор по умолчанию проверяет, реализует ли параметр типа IComparable<T>, и если нет, проверяет факт поддержки IComparable, используя затем первое, что найдет. Рассмотрим пример, иллюстрирующий вышесказанное:

```
using System;
using System.Collections.Generic;
public class EntryPoint
{
    static void Main() {
        SortedList<int, string> list1 =
            new SortedList<int, string>();
        SortedList<int, string> list2 =
            new SortedList<int, string>( Comparer<int>.Default );
        list1.Add( 1, "one" );
        list1.Add( 2, "two" );
        list2.Add( 3, "three" );
        list2.Add( 4, "four" );
    }
}
```

Я объявил два экземпляра SortedList<TKey, TValue>. В первом экземпляре я использовал конструктор без параметров, а во втором явно представил компаратор для целых чисел. В обоих случаях результат один и тот же, потому что я представил обобщенный компаратор по умолчанию в конструкторе list2. Я сделал это главным образом для того, чтобы вы могли увидеть синтаксис, применяемый для прохода обобщенного компаратора по умолчанию. Столь же легко вы можете представить любой другой тип в списке параметров типа для Comparer до тех пор, пока он поддерживает либо IComparable, либо IComparable<T>.

Обобщенные системные интерфейсы

Учитывая тот факт, что библиотека времени выполнения предоставляет обобщенные версии контейнерных типов, не должно быть сюрпризом, что она также представляет обобщенные версии часто используемых интерфейсов. Это замечательная вещь для тех, кто пытается достичь максимальной безопасности типов. Например, ваши классы и структуры могут реализовывать IComparable<T> и/или IComparable, равно как и IEquatable<T>. Естественно, IComparable<T> — более безопасная в отношении типов версия IComparable, и потому ей следует отдавать предпочтение, когда это возможно.

На заметку! Тип IEquatable<T> был добавлен в .NET 2.0, и представляет собой безопасный к типам интерфейс, через который можно обеспечивать проверку эквивалентности как типов значений, так и ссылочных типов.

Пространство имен System.Collections.Generic также определяет целый букет интерфейсов, представляющих обобщенные версии интерфейсов, имеющихся в пространстве имен System.Collections. Сюда входят ICollection<T>, IDictionary<TKey, TValue> и IList<T>. Два из этих интерфейсов имеют специальное назначение: IEnumerator<T> и IEnumerable<T>[2]. С опозданием команда разработчиков из Microsoft решила, что будет хорошей идеей наследовать IEnumerator<T> от IEnumerator, а IEnumerable<T> — от IEnumerable. Это решение вызвало немало споров. Андерс Хейлсберг (Anders Hejlsberg), один из разработчиков языка C#, указывает, что IEnumerable<T> наследует IEnumerable, потому что может это делать.

Его аргументы звучат примерно так: можете себе представить, как было бы хорошо, если бы контейнер, реализующий IList<T>, также реализовал IList. Если IList<T> наследуется от IList, это заставит автора контейнера реализовать две версии метода Add: Add<T> и Add. Но если конечный пользователь сможет вызывать не обобщенный Add, то весь выигрыш дополнительной безопасности типа IList<T> будет утерян, поскольку само наличие Add делает уязвимой реализацию контейнера для исключений приведения типов времени выполнения. Поэтому наследование IList<T> от IList — плохая идея. С другой стороны, IEnumerable<T> и IEnumerator<T> отличаются от других обобщенных интерфейсов в том, что тип T используется только в возвращаемых значениях. Поэтому никакой утери безопасности типов не происходит, когда реализованы оба.

Это оправдывает утверждение о том, что IEnumerable<T> может наследоваться от IEnumerable, а IEnumerator<T> — от IEnumerator только потому они могут это делать. Один из разработчиков Microsoft, работавших над библиотекой .NET Framework, говорил, что IEnumerable<T> и IEnumerator<T> реализованы таким образом для того, чтобы компенсировать недостаток ковариантности обобщений. Да уж...

Кодирование типа, реализующего IEnumerable<T>, требует некоторых ухищрений — в том смысле, что вы должны реализовать метод IEnumerable, используя явную реализацию интерфейса. Более того, чтобы не вводить в заблуждение ком-

---

[2] В главе 9 раскрываются возможности, предоставляемые IEnumerator<T> и IEnumerable<T>, а также то, как вы можете легко реализовать их, используя итераторы C#.

пилятор, вы можете иметь полностью квалифицированный `IEnumerable`, с указанием его пространства имен, как в следующем примере:

```
using System;
using System.Collections.Generic;
public class MyContainer<T> : IEnumerable<T>
{
    public void Add( T item ) {
        impl.Add( item );
    }
    public void Add<R>( MyContainer<R> otherContainer,
                        Converter<R, T> converter ) {
        foreach( R item in otherContainer ) {
            impl.Add( converter(item) );
        }
    }
    public IEnumerator<T> GetEnumerator() {
        foreach( T item in impl ) {
            yield return item;
        }
    }
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
    private List<T> impl = new List<T>();
}
```

Проблемы выбора и их решение

В этом разделе я хочу привести некоторые примеры создания обобщенных типов, которые продемонстрируют ряд полезных приемов создания обобщенного кода. Уверяю вас, что тропинка к овладению приемами эффективного использования обобщений время от времени содержит немало сюрпризов, поскольку иногда вам приходится вести разработку неестественным и запутанным образом, при этом делая нечто вполне концептуально естественное.

На заметку! Многие из вас, несомненно, почувствуют себя неуютно, если пришлось перейти к обобщениям от нотации шаблонов C++, и затем столкнуться с ограничениями, продиктованными динамической природой обобщений.

Преобразования и операции внутри обобщенных типов

Преобразование от одного типа к другому или применение операций к параметризованным типам внутри обобщений — неизбежно непростая задача. Чтобы проиллюстрировать это, давайте разработаем обобщенную структуру `Complex`, представляющую комплексное число. Предположим, что по какой-то причине вы хотите иметь возможность указывать тип значения, используемый внутри этого класса для представления реальной и мнимой частей комплексного числа. Этот пример довольно надуманный, поскольку обычно вполне достаточно представлять

компоненты комплексного числа чем-нибудь вроде System.Double. Однако для примера давайте предположим, что вы хотите иметь возможность представлять компоненты типом System.Int64. На протяжении этой дискуссии, чтобы не загромождать пример и сосредоточиться на проблемах, касающихся обобщений, я буду игнорировать все канонические конструкции, которые должна реализовать обобщенная структура Complex.

Начать можно со следующей реализации структуры Complex:

```
using System;
public struct Complex<T>
    where T: struct
{
    public Complex( T real, T imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public T Real {
        get { return real; }
        set { real = value; }
    }
    public T Img {
        get { return imaginary; }
        set { imaginary = value; }
    }
    private T real;
    private T imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex<Int64> c =
            new Complex<Int64>( 4, 5 );
    }
}
```

Это хорошее начало, но давайте сделаем этот тип значений немного более полезным. Вы можете выиграть от наличия свойства Magnitude, которое возвращает квадратный корень из результата умножения двух компонентов. Попробуем добавить это свойство:

```
using System;
public struct Complex<T>
    where T: struct
{
    public Complex( T real, T imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public T Real {
        get { return real; }
        set { real = value; }
    }
```

```
    public T Img {
        get { return imaginary; }
        set { imaginary = value; }
    }
    public T Magnitude {
        get {
            // НЕ КОМПИЛИРУЕТСЯ!!!
            return Math.Sqrt( real * real +
                              imaginary * imaginary );
        }
    }
    private T real;
    private T imaginary;
}
public class EntryPoint
{
    static void Main() {
        Complex<Int64> c =
            new Complex<Int64>( 3, 4 );
        Console.WriteLine( "Magnitude is {0}",
                           c.Magnitude );
    }
}
```

Если вы попытаетесь скомпилировать этот код, то вероятно, удивитесь, получив следующую ошибку компиляции:

```
error CS0019: Operator '*' cannot be applied to operands of type 'T' and 'T'
ошибка CS0019: Операция '*' не может быть применена к операндам типов 'T' и 'T'
```

Это блестящий пример проблемы использования операции в обобщенном коде. Проблема компиляции порождена тем фактом, что вы должны компилировать обобщенный код обобщенным способом, с учетом того, что конструируемые типы формируются во время выполнения и могут быть сформированы из типов значений, которые могут не поддерживать операцию. В этом случае компилятор никак не может знать, сможет ли тип, подставленный вместо T в конструируемом типе, когда-то в будущем поддержать операцию умножения. Что же делать? Распространенный прием заключается в том, чтобы вынести эту операцию за пределы определения Complex<T>. Подходящим инструментом для этого может быть делегат. Рассмотрим пример Complex<T>, который так и делает:

```
using System;
public struct Complex<T>
    where T: struct, IConvertible
{
    // Делегат для выполнения умножения.
    public delegate T BinaryOp( T val1, T val2 );
    public Complex( T real, T imaginary,
                    BinaryOp mult,
                    BinaryOp add,
                    Converter<double, T> convToT ) {
        this.real = real;
```

```
            this.imaginary = imaginary;
            this.mult = mult;
            this.add = add;
            this.convToT = convToT;
        }
        public T Real {
            get { return real; }
            set { real = value; }
        }
        public T Img {
            get { return imaginary; }
            set { imaginary = value; }
        }
        public T Magnitude {
            get {
                double magnitude =
                    Math.Sqrt( Convert.ToDouble(add(mult(real, real),
                    mult(imaginary, imaginary))) );
                return convToT( magnitude );
            }
        }
        private T real;
        private T imaginary;
        private BinaryOp mult;
        private BinaryOp add;
        private Converter<double, T> convToT;
    }
    public class EntryPoint
    {
        static void Main() {
            Complex<Int64> c =
                new Complex<Int64>(
                        3, 4,
                        EntryPoint.MultiplyInt64,
                        EntryPoint.AddInt64,
                        EntryPoint.DoubleToInt64 );
            Console.WriteLine( "Величина равна {0}",
                               c.Magnitude );
        }
        static Int64 MultiplyInt64( Int64 val1, Int64 val2 ) {
            return val1 * val2;
        }
        static Int64 AddInt64( Int64 val1, Int64 val2 ) {
            return val1 + val2;
        }
        static Int64 DoubleToInt64( double d ) {
            return Convert.ToInt64( d );
        }
    }
```

Возможно, вы смотрите на этот код и недоумеваете, что пошло не так, и почему настолько возросла сложность, когда вы попытались всего лишь найти величину (magnitude) комплексного числа. Как уже упоминалось, пришлось ввести делегат для внешней по отношению к обобщенному типу обработки операции умножения. Поэтому я определил делегат Complex<T>.Multiply. Конструктор Complex<T> должен получить третий параметр, ссылающийся на метод, который будет использовать делегат для выполнения умножения. В данном случае умножение выполняет EntryPoint.MultiplyInt64. Таким образом, когда свойству Magnitude нужно перемножить компоненты, оно должно использовать делегат вместо обычной операции умножения. Естественно, при вызове делегата в конечном итоге он сведет все к операции умножения. Однако применение операции теперь вынесено за пределы обобщенного типа Complecx<T>.

Несомненно, вы заметили усложнение методов доступа свойства. Во-первых, Math.Sqrt принимает тип System.Double. Это объясняет вызов метода Convert.ToDouble. И чтобы все проходило гладко, я добавил к T ограничение, чтобы этот тип обеспечивал поддержку IConvertible. Но это еще не все. Math.Sqrt возвращает System.Double, и вы должны конвертировать этот тип значения обратно в T. Чтобы сделать это, вы не можете полагаться на класс System.Convert, потому что на момент компиляции не знаете, какой тип придется преобразовывать. Опять-таки эту операцию преобразования нужно вынести наружу. Именно для таких случаев в .NET Framework предусмотрен делегат Converter<TInput, Touput>. В данном случае Complex<T> нуждается в делегате преобразования Converter<double, T>. Во время конструирования вы должны передать метод, который будет вызываться через этот делегат, и в данном случае таким методом является EntryPoint.DoubleToInt64. Теперь после всего этого свойство Complex<T>.Magnitude работает, как ожидалось, хотя и не без некоторых дополнительных усилий.

На заметку! Сложность применения Complex<T>, как показано в предыдущем примере, в значительной мере может быть сокращена с помощью лямбда-выражений, которые подробно описаны в главе 15. Используя лямбда-выражения, вы можете полностью исключить необходимость в определении методов операций, таких как MultiplyInt64, AddInt64 и DoubeToInt64, как показано в примере.

Теперь предположим, что вы хотите, чтобы экземпляры Complex<T> могли использоваться в качестве ключевых значений в обобщенном типе SortedList<TKey, TValue>. Чтобы это работало, Complex<T> должен реализовать IComparable<T>. Посмотрим, что надо сделать, чтобы это стало возможным.

```
using System;
public struct Complex<T> : IComparable<Complex<T> >
   where T: struct, IConvertible, IComparable
{
   // Делегат для выполнения умножения.
   public delegate T BinaryOp( T val1, T val2 );
   public Complex( T real, T imaginary,
                   BinaryOp mult,
                   BinaryOp add,
                   Converter<double, T> convToT ) {
      this.real = real;
      this.imaginary = imaginary;
```

```
            this.mult = mult;
            this.add = add;
            this.convToT = convToT;
        }

        public T Real {
            get { return real; }
            set { real = value; }
        }

        public T Img {
            get { return imaginary; }
            set { imaginary = value; }
        }

        public T Magnitude {
            get {
                double magnitude =
                    Math.Sqrt( Convert.ToDouble(add(mult(real, real),
                                    mult(imaginary, imaginary))) );
                return convToT( magnitude );
            }
        }
        public int CompareTo( Complex<T> other ) {
            return Magnitude.CompareTo( other.Magnitude );
        }
        private T real;
        private T imaginary;
        private BinaryOp mult;
        private BinaryOp add;
        private Converter<double, T> convToT;
    }
    public class EntryPoint
    {
        static void Main() {
            Complex<Int64> c =
                new Complex<Int64>(
                        3, 4,
                        EntryPoint.MultiplyInt64,
                        EntryPoint.AddInt64,
                        EntryPoint.DoubleToInt64 );
                Console.WriteLine( "Величина равна {0}",
                                   c.Magnitude );
        }
        static Int64 MultiplyInt64( Int64 val1, Int64 val2 ) {
            return val1 * val2;
        }
        static Int64 AddInt64( Int64 val1, Int64 val2 ) {
            return val1 + val2;
        }
        static Int64 DoubleToInt64( double d ) {
            return Convert.ToInt64( d );
        }
    }
```

Моя реализация интерфейса IComparable<Complex<T>> рассматривает два типа Complex<T> эквивалентными, если у них одинаковое значение величины (magnitude). Поэтому большая часть необходимой работы для сравнения уже выполнена. Однако вместо того, чтобы положиться на операцию неравенства языка C#, опять нужно применить механизм, не использующий операций. В данном случае я вызвал метод CompareTo. Конечно, это потребовало наложения еще одного ограничения на тип T: он должен поддерживать не обобщенный интерфейс IComparable.

Еще одна вещь, которую стоит упомянуть — предыдущее ограничение по не обобщенному интерфейсу IComparable несколько затрудняет Complex<T> хранение обобщенных структур, поскольку обобщенная структура может реализовывать вместо этого IComparable<T>. Фактически, при текущем определении невозможно указать тип Complex<Complex<int>>. Было бы неплохо, если бы Complex<T> мог конструироваться из типов, которые могут реализовывать IComparable<T>, или IComparable, или даже два сразу. Посмотрим, как это можно сделать.

```
using System;
using System.Collections.Generic;
public struct Complex<T> : IComparable<Complex<T> >
    where T: struct
{
    // Делегат для выполнения умножения.
    public delegate T BinaryOp( T val1, T val2 );
    public Complex( T real, T imaginary,
                    BinaryOp mult,
                    BinaryOp add,
                    Converter<double, T> convToT ) {
        this.real = real;
        this.imaginary = imaginary;
        this.mult = mult;
        this.add = add;
        this.convToT = convToT;
    }
    public T Real {
        get { return real; }
        set { real = value; }
    }
    public T Img {
        get { return imaginary; }
        set { imaginary = value; }
    }
    public T Magnitude {
        get {
            double magnitude =
                Math.Sqrt( Convert.ToDouble(add(mult(real, real),
                mult(imaginary, imaginary))) );
            return convToT( magnitude );
        }
    }
    public int CompareTo( Complex<T> other ) {
      return Comparer<T>.Default.Compare( this.Magnitude, other.Magnitude );
    }
```

```
        private T real;
        private T imaginary;
        private BinaryOp mult;
        private BinaryOp add;
        private Converter<double, T> convToT;
    }
    public class EntryPoint
    {
        static void Main() {
            Complex<Int64> c =
                new Complex<Int64>(
                3, 4,
                EntryPoint.MultiplyInt64,
                EntryPoint.AddInt64,
                EntryPoint.DoubleToInt64 );
            Console.WriteLine( "Величина равна {0}",
                               c.Magnitude );
        }
        static void DummyMethod( Complex<Complex<int> > c ) {
        }
        static Int64 AddInt64( Int64 val1, Int64 val2 ) {
            return val1 + val2;
        }
        static Int64 MultiplyInt64( Int64 val1, Int64 val2 ) {
            return val1 * val2;
        }
        static Int64 DoubleToInt64( double d ) {
            return Convert.ToInt64( d );
        }
    }
```

В этом примере я удалил ограничение на T, требующее реализации интерфейса IComparable. Вместо этого метод CompareTo полагается на обобщенный компаратор по умолчанию, определенный в пространстве имен System.Collections.Generic.

На заметку! Класс обобщенного компаратора Comparer<T> вводит еще один промежуточный уровень в форме класса для сравнения двух экземпляров. По сути, он выносит наружу возможность сравнения двух экземпляров. Если вам нужна пользовательская реализация IComparer, вы должны наследоваться от Comparer<T>.

Дополнительно мне пришлось удалить ограничение IConvertible на T, чтобы компилировался метод DummyMethod. Это объясняется тем, что Complex<T> не реализует IConvertible, и когда T заменяется Complex<T> (тем самым формируя Complex<Complex<T>>), то в результате T не реализует IConvertible.

На заметку! При создании обобщенных типов старайтесь не накладывать слишком много ограничений на содержащиеся в коллекции типы. Например, не требуйте от всех содержащихся типов реализации IConvertible. Часто можно вынести наружу такие ограничения, используя вспомогательный объект в сочетании с делегатом.

Задумаемся на минутку об удалении этого ограничения. В свойстве Magnitude вы полагаетесь на метод Convert.ToDouble. Однако, поскольку вы удалили ограничение, существует возможность получения исключения времени выполнения, например, когда тип, представленный T, не реализует IConvertible. Поскольку обобщения предназначены для обеспечения более высокой безопасности типов и помогают избежать исключений времени выполнения, должен существовать лучший способ. Фактически он есть, и вы поступите лучше, если предоставите Complex<T> еще один конвертер в форме делегата Convert<T, double> в конструкторе, как показано ниже:

```
using System;
using System.Collections.Generic;
public struct Complex<T> : IComparable<Complex<T> >
   where T: struct
{
   // Делегат для выполнения умножения.
   public delegate T BinaryOp( T val1, T val2 );
   public Complex( T real, T imaginary,
                   BinaryOp mult,
                   BinaryOp add,
                   Converter<T, double> convToDouble,
                   Converter<double, T> convToT ) {
      this.real = real;
      this.imaginary = imaginary;
      this.mult = mult;
      this.add = add;
      this.convToDouble = convToDouble;
      this.convToT = convToT;
   }
   public T Real {
      get { return real; }
      set { real = value; }
   }
   public T Img {
      get { return imaginary; }
      set { imaginary = value; }
   }
   public T Magnitude {
      get {
         double magnitude =
            Math.Sqrt( convToDouble(add(mult(real, real),
                                   mult(imaginary, imaginary))) );
         return convToT( magnitude );
      }
   }
   public int CompareTo( Complex<T> other ) {
     return Comparer<T>.Default.Compare( this.Magnitude, other.Magnitude );
   }
   private T real;
   private T imaginary;
   private BinaryOp mult;
```

```
        private BinaryOp add;
        private Converter<T, double> convToDouble;
        private Converter<double, T> convToT;
    }
    public class EntryPoint
    {
        static void Main() {
            Complex<Int64> c =
                new Complex<Int64>(
                        3, 4,
                        EntryPoint.MultiplyInt64,
                        EntryPoint.AddInt64,
                        EntryPoint.Int64ToDouble,
                        EntryPoint.DoubleToInt64 );
            Console.WriteLine( "Величина равна {0}",
                               c.Magnitude );
        }
        static void DummyMethod( Complex<Complex<int> > c ) {
        }
        static Int64 MultiplyInt64( Int64 val1, Int64 val2 ) {
            return val1 * val2;
        }
        static Int64 AddInt64( Int64 val1, Int64 val2 ) {
            return val1 + val2;
        }
        static Int64 DoubleToInt64( double d ) {
            return Convert.ToInt64( d );
        }
        static double Int64ToDouble( Int64 i ) {
            return Convert.ToDouble( i );
        }
    }
```

Теперь тип Complex<T> может содержать любые структуры, независимо от того, обобщенные они или нет. Однако вы должны оснастить его необходимыми средствами для преобразований в тип double и обратно, а также операцией умножения и компонентными типами. Эта структура Complex<T> не может служить руководством по созданию представлений комплексных чисел. На самом деле это просто довольно надуманный пример, предназначенный для того, чтобы проиллюстрировать многие нюансы, с которыми вам придется иметь дело при создании эффективных обобщенных типов.

Вы увидите примеры применения некоторых из описанных приемов, когда вам случится работать с обобщенными контейнерами, представленными в базовой библиотеке классов (BCL).

Динамическое создание конструируемых типов

Учитывая динамическую природу CLR и тот факт, что вы на самом деле генерируете классы и код во время выполнения, естественно предположить возможность конструирования закрытых типов из обобщений во время выполнения. До

настоящего момента все примеры этой книги имели дело с созданием типов во время компиляции.

Эта функциональность проистекает из естественного расширения спецификации метаданных для обслуживания обобщений. Тип System.Type — краеугольный камень функциональности, когда вам нужно работать с типами динамически внутри CLR, и естественно, он был расширен для того, чтобы иметь дело также и с обобщениями. Некоторые новые методы System.Type самоочевидны по их именам; к ним относятся GetGenericArguments, GetGenericParameterConstraints и GetGenericTypeDefinition. Эти методы полезны, когда у вас уже есть экземпляр System.Type, представляющий закрытый тип. Однако то, что делает все намного интереснее — это метод MakeGenericType, который позволяет передать массив объектов System.Type, представляющих типы, которые должны быть использованы в списке параметров для создания результирующего конструируемого типа.

Тех из вас, кто имеет опыт работы с шаблонами C++, обобщения могут время от времени разочаровывать, поскольку им недостает статических возможностей шаблонов. Однако я думаю, вы согласитесь с тем, что динамические возможности обобщений в конечном итоге перевешивают. Например, создание механизма анализа некоторого рода XML-ориентированного языка, определяющего новые типы на основе обобщений, становится очень простым. Давайте рассмотрим пример того, как можно использовать метод MakeGenericType:

```
using System;
using System.Collections.Generic;
public class EntryPoint
{
    static void Main() {
        IList<int> intList =
            (IList<int>) CreateClosedType<int>( typeof(List<>) );
        IList<double> doubleList =
            (IList<double>)
                CreateClosedType<double>( typeof(List<>) );

        Console.WriteLine( intList );
        Console.WriteLine( doubleList );
    }
    static object CreateClosedType<T>( Type genericType ) {
        Type[] typeArguments = {
            typeof( T )
        };
        Type closedType =
            genericType.MakeGenericType( typeArguments );
        return Activator.CreateInstance( closedType );
    }
}
```

"Мясо" этого кода содержится внутри обобщенного метода CreateClosedType<T>. Вся работа выполняется в общих терминах через ссылки на Type, созданные из доступных метаданных. Сначала вы должны получить ссылку на обобщенный, открытый тип List<T>, который передан в виде параметра. После этого вы просто создаете массив экземпляров Type для передачи методу MakeGenericType, чтобы

получить от него ссылку на закрытый тип. Как только эта стадия завершена, единственное, что остается — это вызов `CreateInstance` на классе `System.Activator`. Класс `System.Activator` — это средство, которое вы должны использовать для создания экземпляров типов, известных только во время выполнения. В данном случае я вызываю конструктор по умолчанию для закрытого типа. Однако `Activator` имеет перегрузки `CreateInstance`, позволяющие вызывать конструкторы, принимающие параметры.

На заметку! Я использовал операцию C# `typeof` вместо метода `Type.GetType`, чтобы получить экземпляр `Type` для типов. Если тип известен во время компиляции, то операция `typeof` сразу осуществляет поиск по метаданным вместо того, чтобы делать это во время выполнения, что более эффективно.

Когда вы запустите предыдущий пример, то увидите, что закрытые типы выводятся на консоль с указанием их полностью квалифицированных имен типов, тем самым доказывая, что закрытые типы были созданы правильно.

Возможность создания закрытых типов во время выполнения — еще один мощный инструмент в вашем распоряжении, предназначенный для создания высокодинамичных систем. Вы не только можете объявлять обобщенные типы внутри кода, чтобы получать гибкий код, но также можете создавать закрытые типы из этих обобщенных определений во время выполнения. Задумайтесь на минуту о круге проблем, которые можно решить с помощью такой техники, и вы легко увидите, что обобщения — исключительно мощное дополнение к C# и CLR.

Резюме

В этой главе я показал, как объявлять и использовать механизм обобщений в C#, включая обобщенные классы, структуры, интерфейсы, методы и делегаты. Также мы обсудили обобщенные ограничения, которые необходимы для того, чтобы компилятор создавал код, в котором некоторые функциональные предположения накладываются на аргументы типа, предоставляемые обобщенным типам во время выполнения. Типы коллекций получают ощутимое и реальное преимущество в отношении эффективности и безопасности благодаря обобщениям.

Поддержка обобщений в .NET и C# — весьма желанное дополнение. Обобщения не только позволяют генерировать более эффективный код при использовании типов значений в качестве параметров, но также придают компилятору больше мощности для обеспечения безопасности типов. Как правило, вы всегда должны предпочитать безопасность типов времени компиляции безопасности типов времени выполнения. Таким образом, вы можете исправить сбои времени компиляции до сдачи программного обеспечения в промышленную эксплуатацию. Сбой во время выполнения может стоить конечному пользователю огромных сумм денег, в зависимости от ситуации, и может скомпрометировать вас как разработчика. Поэтому всегда снабжайте компилятор максимальными возможностями контроля безопасности типов, чтобы он делал то, для чего предназначен — быть вашим другом.

В следующей главе я коснусь темы многопоточности в C# и исполняющей системе .NET. Рука об руку с многозадачностью идет не менее важная тема синхронизации.

ГЛАВА 12

Многопоточность в C#

Простое упоминание о многопоточности иногда вселяет страх в сердца некоторых программистов. Для остальных же — это хороший вызов. Независимо, как вы реагируете на это, скажу, что многопоточность — это область, усеянная минными полями. Если только вы не отнесетесь к ней с прилежанием, ошибка многопоточности однажды выскочит и ударит вас больно, причем там, где вы этого не ожидали. Ошибки многопоточности очень трудно найти, потому что они возникают асинхронно. Их трудно найти на однопроцессорной машине, но добавьте еще один процессор — и найти эти ошибки станет еще труднее. Фактически, некоторые ошибки многопоточности даже не поднимают свою уродливую голову до тех пор, пока вы не запустите ваше приложение на многопроцессорной машине, поскольку это единственный способ получить настоящую конкурентную многопоточность. По этой причине я всегда советую любому, кто разрабатывает многопоточное приложение, тестировать его, и тестировать часто на многопроцессорной машине. Иначе вы рискуете выпустить создаваемый продукт со скрытыми ошибками многопоточности.

Я помню, как будто вчера: у моего бывшего работодателя случилось так, что мы поспешили передать нашу золотую мастер-копию производителю компакт-дисков и получили сотни тысяч дисков, а потом напоследок решили протестировать приложение на многопроцессорной машине. Стоит ли говорить, что это послужило наглядным уроком всей команде разработчиков, когда ужасная ошибка была выявлена перед самым выпуском продукта в свет?

Многопоточность в C# и .NET

Даже несмотря на то, что многопоточные среды за годы принесли с собой множество сложностей и проблем, да и продолжают это делать, CLR и базовая библиотека классов .NET смягчают многие связанные с этим риски и предлагают ясную модель, на основе которой можно строить приложения. Все еще справедливо утверждение, что наибольшей сложностью при разработке высококачественного многопоточного кода остается синхронизация. .NET Framework позволяет легко, как никогда раньше, создавать новые потоки и использовать управляемые системой пулы потоков, и предоставляет интуитивные объекты, помогающие синхронизировать потоки между собой. Однако в ваши обязанности входит обеспечить правильное применение этих объектов.

Управляемые потоки — это виртуальные потоки, в том смысле, что они не отображаются однозначно на потоки операционной системы. Управляемые потоки

действительно работают параллельно, но было бы ошибкой предполагать, что поток операционной системы, исполняющий в данный момент управляемый код для одного из управляемых потоков, будет выполнять код только этого потока. Фактически, в текущей реализации CLR поток операционной системы выполняет управляемый код для нескольких управляемых потоков в нескольких доменах приложения. В итоге нужно подчеркнуть: не делайте никаких предположений о какой-либо корреляции между потоками операционной системы и управляемыми потоками. Если вы доберетесь до потока ОС, используя слой P/Invoke для выполнения прямых вызовов Win32, убедитесь, что вы применяете эту информацию только в отладочных целях, и не основываете на ней никакой программной логики. Иначе вы непременно столкнетесь с чем-то, что нарушит работу вашего приложения, как только перейдете на другую реализацию CLR.

Также ошибочно заключение, что многопоточное программирование сводится лишь к созданию дополнительных потоков, которые выполняют какую-то работу, требующую ощутимых затрат времени. Конечно да, но это лишь часть картины. Создавая настольное приложение, вы определенно захотите применить многопоточную технику, чтобы обеспечить отзывчивость пользовательского интерфейса во время длительных операций сетевого взаимодействия, поскольку все мы знаем, насколько нетерпеливыми становятся пользователи, когда настольные приложения вдруг начинают реагировать с задержками: они просто-таки убивают их! Однако важно понимать, что общая мозаика многопоточности не сводится просто к созданию дополнительного потока для выполнения некоторого случайного кода. Эта задача в среде C# действительно довольно проста, но давайте посмотрим, так ли здесь все просто, как кажется.

Запуск потоков

Как я сказал, создать поток очень просто. Взгляните на следующий код, чтобы понять, что я имею в виду:

```
using System;
using System.Threading;
public class EntryPoint
{
    private static void ThreadFunc() {
        Console.WriteLine( "Привет из потока {0}!",
        Thread.CurrentThread.GetHashCode() );
    }
    static void Main() {
        // Создание нового потока.
        Thread newThread =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        Console.WriteLine( "Главный поток {0}",
            Thread.CurrentThread.GetHashCode() );
        Console.WriteLine( "Запуск нового потока..." );

        // Запуск нового потока.
        newThread.Start();
```

```
        // Ожидание завершения работы нового потока.
        newThread.Join();
        Console.WriteLine( "Новый поток завершился" );
    }
}
```

Все, что вы должны сделать — это создать объект System.Thread и передать ему экземпляр делегата ThreadStart в качестве параметра конструктора. Делегат ThreadStart ссылается на метод, не принимающий параметров и не возвращающий параметров. В приведенном примере я решил использовать статический метод ThreadFunc в качестве стартовой точки выполнения нового потока. Точно также я мог бы выбрать любой другой метод, видимый коду, создающему поток — до тех пор, пока он не принимает и не возвращает параметров. Обратите внимание, что программа также выводит хеш-код потока, чтобы продемонстрировать, как вы можете идентифицировать потоки в мире управляемых приложений. В неуправляемом мире C++ вы должны использовать идентификатор (ID) потока, полученный через Win32 API. В управляемом мире вместо этого вы применяете значение, возвращенное методом GetHashCode. До тех пор, пока поток существует, гарантируется, что он никогда не вступит в коллизию с любым другим потоком в любом домене приложений данного процесса. Хеш-код потока не является глобально уникальным для всей системы. К тому же, вы можете видеть, как получить ссылку на текущий поток, обратившись к статическому свойству Thread.CurrentThread. И, наконец, обратите внимание на вызов метода Join объекта newThread. В "родном" коде Win32 вы обычно ждете завершения потока по его дескриптору. Когда поток завершает работу, операционная система извещает его дескриптор и ожидание прекращается. Метод Thread.Join инкапсулирует эту функциональность. В данном случае код ждет завершения потока вечно. Thread.Join также предусматривает несколько перегрузок, позволяющих вам указать время ожидания.

В управляемой среде класс System.Thread замечательно инкапсулирует все операции, которые вы можете выполнять на потоке. Если у вас есть некоторая информация о состоянии, которую вы должны передать новому потоку, чтобы она была доступна при его запуске, вы можете просто создать вспомогательный объект и инициализировать делегат ThreadStart, чтобы он ссылался на метод этого объекта. Опять же так вы решаете другую проблему, вводя новый промежуточный слой в форме класса. Предположим, что у вас есть система, в которой вы наполняете несколько очередей задач, и в некоторой точке хотите создать новый поток для обработки элементов определенной очереди, которую вы передадите ему. Следующий код демонстрирует один из возможных способов достижения цели:

```
using System;
using System.Threading;
using System.Collections;
public class QueueProcessor
{
    public QueueProcessor( Queue theQueue ) {
        this.theQueue = theQueue;
        theThread = new Thread( new ThreadStart(this.ThreadFunc) );
    }
    private Queue theQueue;
    private Thread theThread;
```

```
    public Thread TheThread {
        get {
            return theThread;
        }
    }

    public void BeginProcessData() {
        theThread.Start();
    }

    public void EndProcessData() {
        theThread.Join();
    }
    private void ThreadFunc() {
        // ... здесь извлекать элементы theQueue
    }
}
public class EntryPoint
{
    static void Main() {
        Queue queue1 = new Queue();
        Queue queue2 = new Queue();
        // ... операции наполнения очередей данными
        // Обработка каждой очереди в отдельном потоке.
        QueueProcessor proc1 = new QueueProcessor( queue1 );
        proc1.BeginProcessData();
        QueueProcessor proc2 = new QueueProcessor( queue2 );
        proc2.BeginProcessData();
        // ... между тем, выполнять какую-то другую работу
        // Ожидать окончания работы.
        proc1.EndProcessData();
        proc2.EndProcessData();
    }
}
```

Здесь таятся некоторые потенциальные проблемы, связанные с синхронизацией, если кто-то попытается обратиться к очередям после того, как новые потоки начнут работу. Но пока я отложу обсуждение вопросов синхронизации, об этом мы поговорим в настоящей главе, но позднее. Это решение ясно, и следует типичному шаблону асинхронной обработки, принятому в .NET Framework. Класс, добавляющий дополнительный промежуточный слой обработки — это QueueProcessor. Он четко инкапсулирует рабочий поток и предоставляет легковесный интерфейс для выполнения работы. В данном примере главный поток ожидает завершения вызовом EndProcessData. Этот метод просто вызывает Join на инкапсулированном потоке. Однако если вам понадобится некоторого рода данные о состоянии, связанные с завершенной работой, то метод EndProcessData может вернуть их вам.

Когда вы создаете отдельный поток, он, как и любой другой, подчиняется правилам планировщика потоков системы. Однако иногда вам нужно создать поток, который имеет немного больший или немного меньший приоритет, когда алгоритм планировщика потоков будет решать, какой именно поток выполнять следующим.

Вы можете управлять приоритетом потока через свойство. Thread.Priority. Его значение можно корректировать в процессе выполнения потока. Вообще такое случается довольно редко. Все потоки стартуют с приоритетом Normal из перечисления ThreadPriority.

Шаблон IOU и асинхронные вызовы методов

В разделе, озаглавленном "Асинхронные вызовы методов", где я обсуждаю асинхронный ввод-вывод и пулы потоков, вы увидите, что схема BeginProcessData/ EndProcessData является общепринятым шаблоном асинхронного программирования в .NET Framework. Шаблон асинхронного программирования BeginMethod/ EndMethod в .NET Framework подобен шаблону IOU, описанному Аланом Вермеуленом (Allan Vermeulen) в его статье *An Asynchronous Design Pattern* (Асинхронный шаблон проектирования) в журнале *Dr.Dobb's Journal* (июнь 1996 г.). В этом шаблоне функция вызывается для запуска асинхронной операции и в ее возврате вызывающий код получает объект IOU (I owe you — я владею тобой). Позднее вызывающий код может использовать этот объект для получения результата асинхронной операции. Прелесть этого шаблона в том, что он полностью отделяет вызывающий код, который желает получить готовую выполненную асинхронную работу, от механизма, применяемого для ее выполнения. Этот шаблон используется интенсивно в .NET Framework, и я советую применять его для асинхронных вызовов методов, поскольку он знаком вашим клиентам, и они будут себя чувствовать уверенно.

Состояния потока

Состояния управляемого потока хорошо определены исполняющей системой. Хотя переходы между состояниями могут иногда показаться запутанными, все же они не намного сложнее, чем переходы между состояниями потока операционной системы. Есть множество обстоятельств, которые нужно учитывать в управляемом мире, поэтому естественно, допустимые состояния и переходы между ними достаточно сложны. На рис. 12.1 показана диаграмма состояний управляемых потоков.

Состояния на этой диаграмме основаны на состояниях, которые определены CLR для управляемых потоков и представлены в перечислении ThreadState. Каждый управляемый поток начинает свое существование с состояния Unstarted. Как только вы вызываете Start на новом потоке, он входит в состояние Running. Потоки операционной системы, которые обслуживают управляемую исполняющую систему, немедленно стартуют в состоянии Running, минуя состояние Unstarted. Обратите внимание, что вернуться в состояние Unstarted невозможно. Доминирующее состояние на диаграмме — Running. Это то состояние, в котором находится поток, когда он нормально выполняет код, включая обработку любых исключений и выполнение блоков finally. Если главный метод потока, переданный в экземпляре делегата ThreadStart во время создания этого потока, завершается нормально, то поток входит в состояние Finished, как показано на рис. 12.1. Попав в это состояние, поток окончательно исчезает и уже никогда не возвращается к существованию. Если все фоновые потоки вашего процесса достигают состояния Finished, процесс завершается нормально.

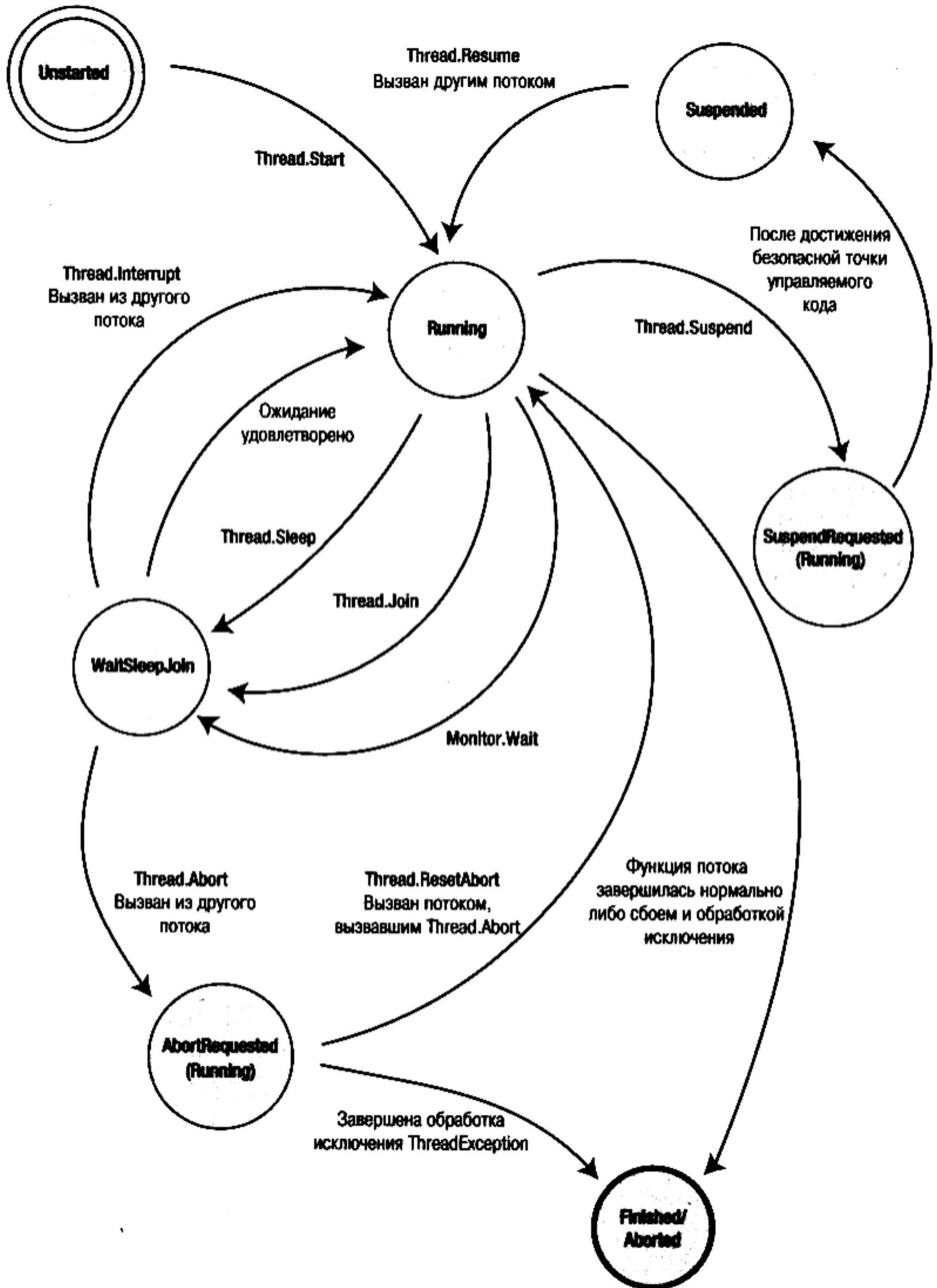

Рис. 12.1. Диаграмма состояний управляемых потоков

Названные три состояния покрывают все основные переходы состояний потока, если предположить, что ваш поток просто выполняет некоторый код и завершается. Как только вы станете добавлять в путь выполнения конструкции синхронизации или пожелаете управлять состоянием потока, будь то из другого потока или текущего, все существенно усложняется.

Например, предположим, что вы пишете код для нового потока, который вы хотите усыпить на время. Вам придется вызвать `Thread.Sleep` и указать таймаут — количество миллисекунд, на которые поток должен заснуть. Это подобно тому, как вы усыпляете поток операционной системы. Когда вызывается `Sleep`, поток входит в состояние `WaitSleepJoin`, при котором его выполнение приостанавливается на заданный период таймаута. Как только этот период истекает, поток возвращается в состояние выполнения.

Операции синхронизации также могут перевести потока в состояние `WaitSleepJoin`. Как должно быть очевидным из наименования состояния, вызов `Thread.Join` на другом потоке, предназначенный для того, чтобы заставить ждать его завершения, переводит вызывающий поток в состояние `WaitSleepJoin`. Вызов `Monitor.Wait` также переводит поток в состояние `WaitSleepJoin`. Теперь вам известны три фактора, прежде всего определивших наименование состояний. Вы можете использовать и другие методы синхронизации потоков, их я опишу ниже, в разделе "Синхронизация между потоками". Как и ранее, как только поток находится в состояния ожидания, он может быть принудительно возвращен обратно в состояние `Running`, когда другой поток вызовет `Thread.Interrupt` на ожидающем потоке. Программисты Win32 узнают в этом поведении нечто подобное изменяемым состояниям ожидания в операционной системе. Имейте в виду, что когда поток вызывает `Thread.Interrupt` на другом потоке, то прерываемый поток получает исключение `ThreadInterruptedException`. Поэтому, несмотря на то, что прерванный поток возвращается в состояние `Running`, он не может оставаться в нем долго, если только у него не окажется соответствующих средств обработки исключения. Иначе поток скоро войдет в состояние `Finished`, как только исключение завершит свой путь к вершине стека, будучи не обработанным.

Другой путь, которым поток может покинуть состояние `WaitSleepJoin` — это когда другой поток вызывает `Thread.Abort` на текущем потоке. Технически поток может вызвать `Abort` на самом себе. Однако я считаю, что это редкий вариант потока выполнения, и потому не показал его на рис. 12.1. Как только вызван `Thread.Abort`, поток входит в состояние `AbortRequested`. Это состояние в действительности формирует состояние выполнения, поскольку в потоке сгенерировано исключение `ThreadAbortException`, и он должен его обработать. Однако, как я объясню позже, управляемый поток трактует это исключение специальным образом — так, что следующим состоянием будет `Aborted`, если только поток, вызвавший `Thread.Abort`, не вызовет `Thread.ResetAbort` до того, как это случится. Кстати, ничего не может остановить поток, который прерывается по вызову `ResetAbort`. Однако вы должны воздерживаться от подобных вещей, поскольку они могут привести к некоторому болезненному поведению. Например, если поток переднего плана не сможет быть прекращен, потому что он находится в состоянии сброса (resetting), то процесс не завершится никогда.

На заметку! Начиная с .NET 2.0, хост имеет возможность принудительно уничтожать потоки во время останова домена приложения, используя т.н. *грубое прерывание потока* (rude thread abort). В такой ситуации поток не может продолжить свое существование с помощью `Thread.ResetAbort`.

И, наконец, выполняющийся поток входит в состояние `SuspendRequested` после вызова `Thread.Suspend` на нем самом, или же когда другой поток вызовет `Suspend` на нем. Очень скоро после этого поток автоматически входит в состояние `Suspended`. Далее, в разделе "Останавливающиеся и пробуждающиеся потоки" я расскажу, зачем нужны эти промежуточные состояния для приостановленного потока. А пока важно запомнить, что состояние `SuspendRequested` — это форма состояния выполнения в том смысле, что в этом состоянии продолжается выполнение управляемого кода.

Таков краткий обзор большой картины переходов в состояниях управляемых потоков. Не забывайте заглядывать на рис. 12.1, когда далее в этой главе речь пойдет о темах, связанных с состоянием потока.

Завершение потоков

Когда вы вызываете `Thread.Abort`, рассматриваемый поток, в конце концов, получает `ThreadAbortException`. Поэтому, естественно, для аккуратной обработки этой ситуации вы должны обрабатывать исключение `ThreadAbortException`, если ваш поток должен делать что-то особенное в случае его прерывания. Существует даже перегрузка `Abort`, которая принимает произвольную объектную ссылку, инкапсулируемую затем в `ThreadAbortException`. Это позволяет коду, прерывающему поток, передать некоторую контекстную информацию обработчику `ThreadAbortException`, например, причину вызова `Abort`.

CLR не доставляет `ThreadAbortException`, если только поток не выполняется внутри управляемого контекста. Если ваш поток вызвал "родную" функцию через слой P/Invoke, и эта функция требует длительного времени для завершения, то прерывание потока откладывается до тех пор, пока управление не вернется в управляемое пространство.

На заметку! В .NET 2.0 и позже при выполнении блока `finally` доставка `ThreadAbortException` откладывается до тех пор, пока выполнение не покинет блок `finally`. В .NET 1.x исключение прерывания доставляется независимо от этого.

Вызов `Abort` на потоке не прерывает его принудительно, поэтому если вам нужно ждать до тех пор, пока поток действительно не завершит выполнение, вы должны вызвать `Join` на этом потоке, чтобы подождать до тех пор, пока не завершится код обработчика исключения `ThreadAbortException`. Во время этого ожидания стоит установить таймаут, чтобы не ждать вечно, когда поток завершит уборку за собой. Даже несмотря на то, что код в обработчике исключений должен подчиняться другим правилам кодирования обработчиков, существует вероятность, что обработчику понадобится много времени, а то и бесконечно много, чтобы завершить работу. Давайте взглянем на обработчик `ThreadAbortException` и посмотрим, как он работает:

```
using System;
using System.Threading;
public class EntryPoint
{
    private static void ThreadFunc() {
        ulong counter = 0;
        while( true ) {
            try {
                Console.WriteLine( "{0}", counter++ );
            }
            catch( ThreadAbortException ) {
                // Попытка проглотить исключение и продолжиться.
                Console.WriteLine( "Abort!" );
            }
        }
    }
    static void Main() {
        Thread newThread =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        newThread.Start();
        Thread.Sleep( 2000 );
        // Прервать поток.
        newThread.Abort();
        // Ждать завершения потока.
        newThread.Join();
    }
}
```

После беглого взгляда на этот код может показаться, что вызов `Join` на экземпляре `newInstance` заблокирует выполнение навсегда. Однако этого не случается. Казалось бы, поскольку `ThreadAbortException` обрабатывается внутри цикла функции потока, исключение будет "проглочено" и цикл продолжится, независимо от того, сколько раз главный поток попытается прервать данный поток. Однако `ThreadAbortException`, сгенерированное через метод `Thread.Abort` ведет себя особым образом. Когда ваш поток завершает обработку исключения, исполняющая система заново неявно генерирует его в конце вашего обработчика. Это все равно, как если бы вы сами повторно сгенерировали исключение. Таким образом, любые внешние обработчики или блоки `finally` будут выполняться нормально. В примере вызов `Join` не будет ждать вечно, как можно было предполагать.

Можно предотвратить повторную генерацию `ThreadAbortException` системой вызовом статического метода `Thread.ResetAbort`. Однако общая рекомендация состоит в том, чтоб вы вызывали только `ResetAbort` из потока, вызвавшего `Abort`. Это потребовало бы некоторых ухищрений и сложной техники взаимодействия, если бы вы захотели сделать то же самое изнутри обработчика исключения `ThreadAbortException` прерываемого потока. Если вы пришли к заключению о необходимости реализации такой техники отмены прерывания потока, то это, скорее всего, свидетельствует о том, что вам, прежде всего, стоит пересмотреть дизайн своей системы. Другими словами, это признак плохого дизайна!

Хотя исполняющая система обеспечивает намного более ясный механизм для прерывающихся потоков, такой как информирование заинтересованных сторон о

факте прерывания потока, все равно вы должны правильно реализовать обработчик ThreadAbortException.

На заметку! Тот факт, что экземпляры ThreadAbortException могут генерироваться асинхронно в произвольном управляемом потоке, затрудняет создание устойчивого, безопасного к исключениям кода. Прочтите раздел "Ограниченные области выполнения" в главе 7.

Останавливающиеся и пробуждающиеся потоки

Подобно "родным" потокам операционной системы, существуют механизмы для погружения управляемых потоков в сон на определенный период времени или приостановки выполнения до тех пор, пока оно не будет явно возобновлено. Если поток просто желает приостановить себя на предопределенный период времени, он может вызвать статический метод Thread.Sleep. Единственный параметр, передаваемый методу Sleep — это количество миллисекунд, в течение которых поток должен спать. После его вызова этот метод заставляет поток отдать остаток кванта времени, выделенного ему процессором, и отправиться спать. По истечении указанного времени планировщик потоков возобновит его выполнение. Естественно, промежуток времени, который вы передаете Sleep, довольно точен, но не абсолютно. Это связано с тем, что в конце периода поток не получает немедленно времени процессора. В этот момент может выполняться другой, высокоприоритетный поток, и пробуждающемуся потоку придется подождать своей очереди. По этой причине применение Sleep для синхронизации выполнения между двумя потоками категорически не рекомендуется.

Внимание! Если вы обнаружите, что для решения проблемы синхронизации в вашем коде нужно применить Sleep, знайте — подобным образом вы проблему не решите вообще. Вы ее только закамуфлируете и тем самым усугубите.

Предусмотрено специальное значение, задающее период — Timeout.Infinite, которое вы можете передать Sleep, чтобы усыпить поток навсегда. Вы можете разбудить спящий поток, прервав его методом экземпляра Thread.Interrupt. Этот метод подобен Abort в том, что он пробуждает целевой поток и генерирует исключение ThreadInterruptedException. Поэтому, если ваша функция потока не оснащена обработчиком этого исключения, оно распространится по стеку до тех пор, пока не прекратит выполнение потока. Чтобы подстраховаться, вы должны поместить вызов Sleep внутри блока try и перехватывать ThreadInterruptException. В отличие от ThreadAbortException, ThreadInterruptException автоматически не повторяется исполняющей системой в конце обработчика.

На заметку! Другое специальное значение параметра для Thread.Sleep — 0. Если вы передаете 0, то метод Thread.Sleep заставит поток отдать остаток выделенного ему кванта времени. Возобновление выполнения такого потока произойдет тогда, когда планировщик потоков вернется к нему снова.

Другой способ отправить поток спать на неопределенное время предоставляет метод экземпляра Thread.Suspend. Вызов Suspend приостанавливает выполнение потока до тех пор, пока оно не будет явно возобновлено. Вы можете возобновить вы-

полнение потока вызовом метода экземпляра Resume или Interrupt. Однако в случае применения Interrupt целевой поток должен иметь правильный обработчик исключений вокруг вызова Suspend, иначе поток просто завершится. Технически вызов Abort также возобновляет выполнение потока, но лишь затем, чтобы послать ему ThreadAbortException и заставить завершиться. Имейте в виду, что любой поток с достаточными привилегиями может вызвать Suspend на потоке — даже текущий поток на самом себе. Если текущий поток вызывает Suspend, в этой точке он блокируется, ожидая, пока кто-нибудь не вызовет на нем Resume.

Важно отметить, что когда вы вызываете Suspend на потоке, он не приостанавливается немедленно. Вместо этого потоку разрешено выполняться до т.н. *безопасной точки*. Достигнув этой точки, поток приостанавливается. Безопасная точка помещается в управляемом коде, где можно провести сборку мусора. Например, если CLR определяет, что пришло время для сборки мусора, она должна временно приостановить все потоки, пока это не будет сделано (собран мусор). Однако, как вы можете представить, если поток находится на полпути в операции, состоящей из нескольких инструкций, обращающихся к объекту в куче, а в это время сработает сборщик мусора (GC) и переместит этот объект в другое место системной памяти, ничего хорошего не произойдет. По этой причине, когда GC приостанавливает потоки для сборки мусора, он должен подождать, пока все они не достигнут безопасной точки, когда можно будет перемещать объекты в памяти. По этой причине вызов Suspend позволяет потоку достичь безопасной точки, прежде чем в действительности приостановить его. Также я хотел подчеркнуть, что вы никогда не должны использовать Suspend и Resume для управления синхронизацией. Конечно, тот факт, что система позволяет потоку продолжать выполнение до достижения безопасной точки — достаточно веская причина полагаться на этот механизм, но, тем не менее, это — плохая практика дизайна.

Ожидание завершения потока

В предшествующих примерах настоящей главы я использовал метод Join для ожидания завершения определенного потока. Фактически именно для этого он и применяется. В неуправляемом приложении Win32, вы, возможно, привыкли ожидать, пока дескриптор потока не даст сигнал о завершении потока. Метод Join на самом деле реализует тот же механизм. Имя метода определено тем фактом, что вы присоединяете путь выполнения текущего потока к пути выполнения того потока, на котором вызван Join, и вы не можете продолжить работу, пока это соединение не произойдет.

Естественно, вы захотите избежать вызова Join на текущем потоке. Эффект от этого подобен вызову Suspend из текущего потока — поток блокируется до тех пор, пока не будет прерван. Даже когда поток блокируется вызовом Join, он может быть разбужен вызовом Interrupt или Abort, как описано в предыдущем разделе.

Иногда вы захотите вызвать Join для ожидания завершения другого потока, но наверняка не захотите ждать вечно. Join предусматривает перегрузки, позволяющие определять время ожидания. Эти перегрузки возвращают булевское значение, причем true означает, что поток был остановлен, а false свидетельствует об окончании таймаута.

Потоки переднего плана и фоновые потоки

Когда вы создаете поток в управляемой среде .NET, по умолчанию он существует как поток переднего плана. Это значит, что управляемая среда выполнения, а с ней и процесс, останутся живыми до тех пор, пока жив этот поток. Рассмотрим следующий код:

```
using System;
using System.Threading;
public class EntryPoint
{
    private static void ThreadFunc1() {
        Thread.Sleep( 5000 );
        Console.WriteLine( "Завершение дополнительного потока" );
    }
    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc1) );
        thread1.Start();
        Console.WriteLine( "Завершение главного потока" );
    }
}
```

Запустив этот код, вы увидите, что `Main` завершится до завершения дополнительного потока, как и следовало ожидать (разработчики C++ здесь обнаружат отличие от поведения, к которому они привыкли, где процесс обычно прерывается, как только завершается главная процедура приложения).

Временами вам может понадобиться обработать останов, когда главный поток завершается, даже несмотря на наличие дополнительных фоновых потоков. Вы можете достичь этого во время выполнения, превратив дополнительный поток в фоновый установкой свойства `Thread.IsBackground` в `true`. Это может понадобиться для потоков, выполняющих такую задачу, как прослушивание порта для сетевых соединений, или какую-то другую подобную фоновую задачу. Имейте в виду, однако, что всегда нужно быть уверенным, что ваши потоки получат шанс убрать за собой перед тем, как будут завершены. Когда фоновый поток завершается по завершении процесса, он не принимает исключений, как в случае, когда кто-то вызовет `Interrupt` или `Abort`. Поэтому, если поток работает с постоянно хранящимися данными, находящимися в каком-то промежуточном состоянии, то его останов определенно плохо на них отразится. Таким образом, создавая фоновые потоки, убедитесь, что они закодированы с учетом того, что могут быть грубо прерваны в любой момент, и это не вызовет никаких катастрофических последствий. Также вы можете реализовать некоторый механизм уведомления потока о том, что процесс собирается его скоро прервать. Создание такого механизма вносит дополнительную сложность, поскольку главный поток должен будет ждать в течение определенного времени после отправки уведомления дополнительному потоку, чтобы дать ему возможность выполнить работу по очистке. В этой точки почти имеет смысл превратить поток обратно в поток переднего плана.

Локальное хранилище потока

В управляемом окружении вы можете создать локальное хранилище потока (thread-local storage — TLS). В зависимости от вашего приложения вам может понадобиться иметь статическое поле класса, уникальное для каждого потока, в котором используется класс. В большинстве случаев на C# сделать это очень просто. Если у вас есть статическое поле, которое должно быть привязано к потоку, просто снабдите его атрибутом ThreadStaticAttribute. Если вы это сделаете, поле будет инициализировано отдельно для каждого потока, обращающегося к нему. "За кулисами" каждый поток получает собственное, связанное с ним место хранения значения или ссылки. Однако при использовании ссылок на объекты будьте осторожны, выдвигая предположения о создании объектов. Следующий код демонстрирует возможную ловушку, которой нужно избегать:

```
using System;
using System.Threading;
public class TLSClass
{
    public TLSClass() {
        Console.WriteLine( "Создание TLSClass" );
    }
}
public class TLSFieldClass
{
    [ThreadStatic]
    public static TLSClass tlsdata = new TLSClass();
}
public class EntryPoint
{
    private static void ThreadFunc() {
        Console.WriteLine( "Поток {0} запускается...",
                           Thread.CurrentThread.GetHashCode() );
        Console.WriteLine( "tlsdata для этого потока - \"{0}\"",
                           TLSFieldClass.tlsdata );
        Console.WriteLine( "Поток {0} завершается",
                           Thread.CurrentThread.GetHashCode() );
    }
    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        Thread thread2 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        thread1.Start();
        thread2.Start();
    }
}
```

Код создает два потока, которые обращаются к связанному с потоком статическому члену TLSFieldClass. Чтобы проиллюстрировать ловушку, содержащуюся в коде, я сделал специфичный для потока слот типа TLSClass, и код пытается инициализировать его посредством инициализатора в определении класса, который

просто вызывает new на конструкторе класса по умолчанию. Теперь смотрите, насколько неожиданный вывод получается:

```
Поток 3 запускается...
Поток 4 запускается...
Создание TLSClass
tlsdata для этого потока - "TLSClass"
Поток 3 завершается
tlsdata для этого потока - ""
Поток 4 завершается
```

Внимание! Всегда помните, что последовательность выполнения многопоточных программ никогда не гарантирована, если только вы не применяете специфических механизмов синхронизации. Этот вывод был сгенерирован на однопроцессорной системе. Если вы запустите то же приложение на многопроцессорной системе, вы, вероятно, увидите в выводе программы совершенно другую последовательность. Тем не менее, смысл этого примера не меняется.

Важный момент, который следует здесь отметить — конструктор TLSClass вызван лишь однажды. Конструктор был вызван для первого потока, но не для второго. Во втором потоке поле инициализируется значением null. Поскольку поле tlsdata статическое, его инициализация в действительности выполняется в момент вызова статического конструктора TLSFieldClass. Однако статические конструкторы могут быть вызваны только один раз для одного класса в одном домене приложений. По этой причине вы захотите избегать присваивания значений слотам, связанным с потоком, в точке объявления. Таким образом, они всегда будут получать значения по умолчанию. Для ссылочных типов это означает null, а для типов значений — эквивалент установки всех бит хранилища в 0. Затем, при первом доступе к специфичному для потока слоту, вы можете проверять его на равенство null и соответствующим образом создавать экземпляр. Конечно, самый чистый способ достижения этого — всегда обращаться к локальному по отношению к потоку слоту через статическое свойство.

В качестве дополнительного замечания: не думайте, что вы можете перехитрить компилятор, добавив промежуточный слой, такой как присваивание значения потоковому слоту на основе возвращаемого значения статического метода. Вы обнаружите, что ваш статический метод будет вызван только однажды. Если бы CLR попыталась "исправить" эту проблему для вас, она была бы без сомнения менее эффективна, поскольку пришлось бы проверять, осуществляется ли доступ к полю первый раз, и если это так — вызывать код инициализации. Если вы задумаетесь об этом, то поймете, что все намного сложнее, чем кажется, поскольку не удастся все делать правильно в 100% всего времени.

Имеется и другой путь использования локального для потока хранилища, который не предполагает декорацию статического метода атрибутом. Вы можете выделять локальное хранилище потока динамически, используя либо метод Thread.AllocateDataSlot, либо Thread.AllocateNamedDataSlot. Вы будете применять эти методы, если не знаете заранее, сколько специфичных для потока слотов вам понадобится до момента выполнения. В противном случае обычно намного проще использовать метод статического поля. Когда вы вызываете AllocateDataSlot, то выделяется новый слот во всех потоках, чтобы хранить ссылку на экземпляр типа System.Object. Метод возвращает дескриптор в форме экземпляра объекта

LocalDataStoreSlot. Вы можете обращаться к этому месту в памяти с помощью методов GetData и SetData потока. Взглянем на следующую модификацию предыдущего примера:

```
using System;
using System.Threading;
public class TLSClass
{
   static TLSClass() {
      tlsSlot = Thread.AllocateDataSlot();
   }
   public TLSClass() {
      Console.WriteLine( "Создание TLSClass" );
   }
   public static TLSClass TlsSlot {
      get {
         Object obj = Thread.GetData( tlsSlot );
         if( obj == null ) {
            obj = new TLSClass();
            Thread.SetData( tlsSlot, obj );
         }
         return (TLSClass) obj;
      }
   }
   private static LocalDataStoreSlot tlsSlot = null;
}
public class EntryPoint
{
   private static void ThreadFunc() {
      Console.WriteLine( "Поток {0} запускается...",
                         Thread.CurrentThread.GetHashCode() );
      Console.WriteLine( "tlsdata for this thread is \"{0}\"",
                         TLSClass.TlsSlot );
      Console.WriteLine( "Поток {0} завершается",
                         Thread.CurrentThread.GetHashCode() );
   }
   static void Main() {
      Thread thread1 =
         new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
      Thread thread2 =
         new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
      thread1.Start();
      thread2.Start();
   }
}
```

Как видите, использование динамических слотов несколько сложнее, чем метод статического поля. Однако этот вариант обеспечивает некоторую дополнительную гибкость. Обратите внимание, что слот выделяется в инициализаторе типа, которым является статический конструктор, присутствующий в примере. Таким образом, слот выделяется для всех потоков в точке, где исполняющая система ини-

циализирует тип для использования. Заметьте, что я проверяю слот на равенство null в средстве доступа свойства класса TLSClass. Когда вы выделяете слот с применением AllocateDataSlot, он инициализируется null для каждого потока.

Вы можете счесть удобным обращаться к специфичному для потока хранилищу по строковому имени, а не через ссылку на экземпляр LocalDataStoreSlot. Однако вы должны быть осторожны с применением разумно уникального имени, чтобы использование того же имени еще где-либо в коде не привело к нежелательным эффектам. Для именования вашего слота вы можете применить, к примеру, строковое представление уникального идентификатора GUID, чтобы быть уверенным, что никто не попытается создать еще один с точно таким именем. Когда вам нужен доступ к слоту, вы можете вызвать GetNameDataSlot, который просто транслирует вашу строку в экземпляр LocalDataStoreSlot. Я рекомендую прочесть документацию MSDN, касающуюся именования слотов локального хранилища потока, чтобы почерпнуть там необходимые детали.

Большая часть сказанного должна быть знакома тем разработчикам, кто использовал локальные для потока хранилища в Win32. Однако здесь есть одно усовершенствование: поскольку управляемые TLS-слоты реализованы иначе, ограничение на количество TLS-слотов Win32 здесь не действует.

Как неуправляемые потоки и апартаменты COM приспособлены друг к другу

Неуправляемые потоки могут входить в управляемую среду извне. Например, управляемые объекты могут быть представлены "родному" коду через промежуточный слой COM. Когда "родной" поток обращается к объекту, он входит в управляемую среду. Когда это случается, CLR фиксирует данный факт, и если это первый раз, когда неуправляемый поток осуществляет вызов CLR, он устанавливает необходимые учетные структуры, позволяющие ему выполняться как управляемый поток внутри управляемой исполняющей системы. Как я уже упоминал, потоки, входящие в управляемую среду подобным образом, начинают свое управляемое существование с состояния Running, как показано на рис. 12.1. Как только вся бухгалтерия запущена, то дальше всякий раз, когда тот же неуправляемый поток входит в исполняющую систему, он ассоциируется с тем же управляемым потоком.

Подобно тому, как управляемые объекты могут быть представлены внешнему миру в виде объектов COM, так и объекты COM могут быть представлены управляемому миру в виде управляемых объектов. Когда управляемый поток вызывает таким образом объект COM, исполняющая система ослабляет контроль над состоянием потока до тех пор, пока он не вернется в управляемую среду.

Предположим, что объект COM, написанный на "родном" C++, вызывает функцию Win32 API WaitForSingleObjectWin32, чтобы ожидать сигнала от определенного объекта синхронизации. Затем, если управляемый поток вызывает Thread.Abort или Thread.Interrupt, чтобы разбудить поток, то его пробуждение будет отложено до того момента, когда поток не войдет обратно в управляемое окружение. Другими словами, этот вызов не имеет эффекта до тех пор, пока поток выполняет неуправляемый код. Поэтому вам нужно быть в некоторой мере осведомленными о механизмах синхронизации, используемых родными объектами COM, к которым обращается ваш "родной" код.

И, наконец, если вам когда-нибудь в прошлом приходилось выполнять разработку COM, то вы наверняка знакомы с понятием апартамента COM (COM apartment), а также сопровождающими их прокси и заглушками[1]. Когда управляемый код осуществляет вызов объекта COM, то важно, чтобы управляемый код был настроен на вызов неуправляемого объекта COM либо через однопоточный апартамент (single-threaded apartment — STA), либо через многопоточный апартамент (multi-threaded apartment — MTA). Вы можете установить это свойство в новый управляемый поток, установив свойство Thread.ApartmentState. Как только поток выполняет вызов COM, это состояние блокируется. Другими словами, вы не сможете изменить его впоследствии. Вы можете как угодно устанавливать свойство после первого вызова COM, но это не будет иметь никакого эффекта. Когда вы обращаетесь к объекту COM из управляемого кода, то лучше знать тип апартамента, в котором будет выполняться этот COM-объект. Таким образом, вы сможете взвешенно выбрать тип апартамента COM, в котором вы хотите выполнять поток. Выбор неверного типа может плохо отразиться на эффективности, вынуждая все вызовы проходить через прокси и заглушки. А в еще худших ситуациях объекты COM могут быть вообще не вызываемыми из других типов апартаментов.

Используя Thread.ApartmentState, вы можете контролировать свойство COM-апартамента для новых управляемых потоков, которые вы создаете. Но как насчет главного потока приложения? Дело в том, что когда главный поток управляемого приложения запущен, уже слишком поздно устанавливать свойство ApartmentState. Это потому, что управляемая исполняющая система инициализирует главный поток в состояние MTA при инициализации управляемого приложения. Если вам нужно изменить ApartmentState главного потока на STA, то единственный способ сделать это заключается в декорировании метода Main атрибутом STAThreadAttribute.

Кстати, вы можете также декорировать его атрибутом MTAThreadAttribute, но это будет излишне, поскольку является выбором CLR по умолчанию. Следующий код показывает пример того, о чем я говорю:

```
public class EntryPoint
{
    [STAThread]
    static void Main() {
    }
}
```

Если вы когда-нибудь работали с приложениями Windows Forms, особенно с теми, что генерируются мастерами Visual Studio, то вы, вероятно, уже видели этот атрибут и недоумевали, зачем он нужен. Декорируя главный поток графического пользовательского интерфейса приложения этим атрибутом, вы можете легче интегрировать "родные" элементы управления ActiveX в графический интерфейс, поскольку они обычно выполняются в STA.

Обратите внимание, что свойство ApartmentState управляемого потока не оказывает никакого эффекта на выполнение управляемого кода. И что более важно, когда управляемые объекты потребляются "родным" приложением через промежуточный слой взаимодействия COM, то ApartmentState не управляет тем, в каком

[1] Детальное описание апартаментов COM и их работы можно найти в книге Дона Бокса (Don Box) *Essential COM* (Boston, MA: Addison-Wesley Professional, 1997 г.).

апартаменте находится объект с точки зрения "родного" приложения. С "родной" стороны забора все управляемые объекты выглядят объектами COM, находящимися в MTA и интегрирующими FTM (Free Threaded Marshaller — свободный потоковый маршализатор). К тому же все потоки, созданные в пуле потоков CLR, всегда располагаются в MTA для процесса.

Синхронизация между потоками

Синхронизация является, пожалуй, наиболее трудной частью создания многопоточных приложений. Вы можете создавать дополнительные потоки для выполнения какой-то работы хоть целый день, не заботясь о синхронизации, до тех пор, пока эти потоки потребляют при запуске некоторые данные, которые не использует ни один другой поток, и выполняют некоторую работу. Никому не интересно знать, когда они завершатся, или каким будет результат их действий. Очевидно, что это редкий случай, когда вам может понадобиться такой поток. В большинстве случаев вам нужно взаимодействовать с работающим потоком, ожидать, пока он достигнет определенного места в коде, или, возможно, работать с одними и теми же экземплярами объектов или значений, с которыми работают и другие потоки.

Во всех этих и многих других случаях вы можете положиться на технику синхронизации, чтобы синхронизировать потоки во избежание состояния "гонок" и взаимных блокировок. При состоянии гонок два потока могут нуждаться в доступе к одному фрагменту памяти, и только один из них может делать это безопасно в единицу времени. В таких случаях вы должны использовать механизм синхронизации, который позволяет только одному потоку в один момент времени обращаться к данным и блокирует другой поток, заставляя его ждать, пока первый не завершит. Многопоточная среда стохастична по своей природе, и вы никогда не можете знать, когда планировщик заберет управления у вашего потока. Классический пример — когда один поток находится на полпути в изменении блока памяти, теряет управление, затем другой поток получает управление и начинает читать память, предполагая, что она находится в корректном состоянии. Примером взаимной блокировки может служить ситуация, когда из двух потоков каждый ожидает освобождения ресурса другим потоком. Оба ждут друг друга, и поскольку ни один из них не может продолжить выполнение, пока не будет выполнено то, чего он ожидает, то оба остаются в состоянии вечного ожидания.

Во всех задачах синхронизации вы должны использовать наиболее легковесный механизм синхронизации, который есть в вашем распоряжении, и никак не тяжелее. Например, если вы пытаетесь разделить блок данных между двумя потоками в одном и том же процессе, и хотите разграничить доступ между ними двумя, используйте нечто вроде блокировки `Monitor`, а не `Mutex`. Почему? Потому, что `Mutex` предназначена для разграничения доступа к разделяемым ресурсам между процессами и представляет собой тяжеловесный объект операционной системы, который замедляет процесс, захватывая и освобождая блокировку. Если не требуется никакой межпроцессной блокировки, используйте вместо этого `Monitor`. Даже более легковесным, чем `Monitor`, является набор методов класса `Interlocked`. Он идеален, когда вы знаете, что вероятность ожидания при захвате блокировки является низкой.

На заметку! Любого рода ожидание на объекте ядра, такое как ожидание на Mutex, Semaphore, EventWaitHanldle или любом другом, которое в конечном итоге обеспечивается ожиданием на объекте ядра Win32, требует перехода в режим ядра. Такой переход обходится дорого, и вы всегда должны избегать его по мере возможности. Например, если потоки, которые вы синхронизируете, находятся в одном и том же процессе, объекты синхронизации ядра, вероятно, чересчур тяжелы. Наиболее легкая техника синхронизации предполагает изощренное использование класса Threading.Interlocked. Все его методы полностью реализованы в пользовательском режиме, что позволяет вам избежать переключений между пользовательским режимом и режимом ядра.

При использовании объектов синхронизации в многопоточной среде вы хотите удерживать блокировку в течение как можно более короткого промежутка времени. Например, если вы захватываете блокировку синхронизации для чтения разделенного экземпляра структуры, и код внутри метода, захватывающий блокировку, использует этот экземпляр структуры для каких-то своих целей, то лучше создать локальную копию структуры в стеке и затем немедленно освободить блокировку, если только это не является логически невозможным. Таким образом, вы не свяжете другие потоки в системе, которым нужен доступ к защищенной переменной.

Когда вам нужно синхронизировать выполнение потока, никогда не полагайтесь на такие методы, как Thread.Suspend или Thread.Resume, для управления синхронизацией потока. Если вы помните из предыдущего раздела настоящей главы, вызов Thread.Suspend на самом деле не приостанавливает поток немедленно. Вместо этого поток должен достичь безопасной точки управляемого кода, прежде чем он сможет прервать исполнение. Также никогда не применяйте Thread.Sleep для синхронизации потоков. Thread.Sleep подходит, когда выполняется некоторого рода цикл опроса на сущности вроде аппаратного устройства, которое только что было сброшено и не в состоянии никого известить о том, что оно вернулось на связь. В этом случае вы не хотите непрерывно в цикле проверять состояние устройства. Гораздо лучше ненадолго уснуть между опросом, чтобы позволить планировщику разрешить выполняться другим потокам. Я уже говорил об этом в предыдущем разделе, но повторю опять, потому что это важно: если вам когда-нибудь приходилось исправлять ошибки синхронизации, добавляя вызов Thread.Sleep в некоторую кажущуюся случайной точку кода, значит, вы вообще не решали проблему. Вы просто скрывали ее и усугубляли. Не поступайте так!

Легковесная синхронизация с помощью класса Interlocked

Те из вас, кто пришел из неуправляемого мира программирования на Win32 API, вероятно, знают о существовании семейства функций Interlocked... К счастью, эти функции предоставлены в распоряжение разработчиков C# через статические методы класса Interlocked из пространства имен System.Threading. Иногда при выполнении множества потоков возникает необходимость сопровождения одной простой переменной — обычно типа значения, но, может быть, и объекта — между несколькими потоками. Например, предположим, что вам по какой-то причине нужно отслеживать количество работающих потоков — где-то в статической целочисленной переменной. Когда поток стартует, он увеличивает значение этой переменной, а когда завершается — уменьшает это значение. Очевидно, что вы должны как-то синхронизировать доступ к этому значению, поскольку планировщик мо-

жет отобрать управление у одного потока и передать его другому, когда первый находится в процессе обновления значения счетчика. Еще хуже, когда тот же код должен выполняться параллельно на многопроцессорной машине. Для этой задачи вы можете использовать Interlocked.Increment и Interlocked.Decrement. Эти методы гарантированно модифицируют значение атомарно среди всех процессоров системы. Взгляните на следующий пример:

```
using System;
using System.Threading;
public class EntryPoint
{
    static private int numberThreads = 0;
    static private Random rnd = new Random();
    private static void RndThreadFunc() {
        // управляющий поток увеличивает счетчик и ожидает
        // произвольный период времени от 1 до 12 секунд.
        Interlocked.Increment( ref numberThreads );
        try {
            int time = rnd.Next( 1000, 12000 );
            Thread.Sleep( time );
        }
        finally {
            Interlocked.Decrement( ref numberThreads );
        }
    }
    private static void RptThreadFunc() {
        while( true ) {
            int threadCount = 0;
            threadCount =
                Interlocked.Exchange( ref numberThreads,
                                      numberThreads );
            Console.WriteLine( "{0} поток(ов) активно",
                               threadCount );
            Thread.Sleep( 1000 );
        }
    }
    static void Main() {
        // Старт отчетных потоков.
        Thread reporter =
            new Thread( new ThreadStart(
                          EntryPoint.RptThreadFunc) );
        reporter.IsBackground = true;
        reporter.Start();
        // Старт потоков, ожидающих случайный период времени.
        Thread[] rndthreads = new Thread[ 50 ];
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
            new Thread( new ThreadStart(
                          EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

Эта маленькая программа создает 50 потоков переднего плана, которые не делают ничего, кроме ожидания в течение произвольного периода времени между 1 и 12 секунд. Она также создает фоновый поток, который рапортует о том, сколько потоков активно в настоящий момент. Если вы взглянете на метод RndThreadFunc, являющийся функцией потока, которую используют 50 потоков программы, то увидите в ней увеличение и уменьшение целочисленного значения с использованием метода Interlocked. Обратите внимание, что я использую блок finally, чтобы гарантировать уменьшение счетчика, независимо от того, как завершается поток. Вы можете использовать подход "disposable/using" (освобождение/использование) с помощью ключевого слова using, упаковав инкремент и декремент счетчика в отдельный класс, реализующий IDisposable. Это позволит избавиться от громоздкого блока finally. Но в этом случае он вам не поможет, поскольку вы должны создать ссылочный тип, который будет хранить целочисленную переменную счетчика, поскольку невозможно применять ref к целому числу как к полю вспомогательного класса.

Вы уже видели Interlocked.Increment и Interlocked.Decrement в действии. Но как насчет Interlocked.Exchange, который применяет поток отчета? Напомню, что поскольку множество потоков пытаются записывать в переменную threadCount, отчетный поток должен читать ее значение также в синхронизированном режиме. И здесь на помощь приходит Interlocked.Exchange. Метод Interlocked.Exchange, как можно предположить из его имени, позволяет обменивать значение одной переменной с другой в атомарном режиме, и возвращает значение, которое было сохранено ранее в этом месте. Поскольку класс Interlocked не предусматривает метода для простого чтения значения Int32 в атомарной операции, я выполняю этот обмен значения переменной numberThreads с ее собственным значением, и в качестве побочного эффекта метод Interlocked.Exchange возвращает мне значение, которое было в слоте.

Методы Interlocked на SMP-системах

На симметричных многопроцессорных системах Intel (SMP) простое чтение и запись слотов памяти "родного" размера синхронизируются автоматически. В системе IA-32 чтение и запись свойств, выровненных по 32-битным значениям, синхронизированы. Поэтому в предыдущем примере, где я показал применение Interlocked.Exchange для простого чтения значения Int32, вызов этого метода был бы не обязателен, если переменная правильно выровнена в памяти.

По умолчанию CLR выполняет немало действий, чтобы правильно выровнять значения по естественным границам. Однако вы можете переопределить размещение значений в классе или структуре, применив к полям атрибут FieldOffsetAttribute, тем самым отменяя выравнивание полей данных. Если значение Int32 не выровнено, то гарантии, упомянутые в предыдущем абзаце, будут утеряны. В таком случае вы должны использовать Interlocked.Exchange для надежного чтения значения.

Все методы Interlocked... реализованы в системах IA-32 с использованием префикса lock. Этот префикс заставляет процессор применять сигнал LOCK#, что предотвращает параллельный доступ к значению других процессоров системы, что и требуется в сложных операциях, когда увеличиваются значения счетчиков и тому подобное. Одно удобное качество префикса lock заключается в том, что не выровненные поля данных не повлияют пагубным образом на целостность блокировки. Другими словами, он отлично работает с не выровненными данными. Вот почему Interlocked.Exchange — гарантия атомарного чтения не выровненных данных.

И, наконец, рассмотрим тот факт, что класс Interlocked реализует перегрузки некоторых методов таким образом, что они атомарно работают с 64-битными значениями на 32-разрядных системах. Естественно, не существует способа атомарно читать такие значения без обращения к классу Interlocked. Фактически по этой причине в версии .NET 2.0 класса Interlocked предусмотрен Interlocked.Read для значений типа Int64. Конечно, такой зверь не нужен на 64-разрядных системах, где он сводится к обычному чтению. Однако CLR предназначается для работы на многих платформах, поэтому вы всегда должны использовать Interlocked.Read при работе с 64-битными значениями.

По этим причинам лучше перестраховаться, чем потом сожалеть, и использовать Interlocked.Exchanged для чтения и записи значений в атомарном режиме, поскольку очень трудно проверить факт наличия или отсутствия выравнивания или превышения системного размера атомарных данных, прежде чем читать или писать их в "сыром" виде. Определение системного размера атомарной единицы данных для платформы и построение кода на основе этой информации полностью противоречит межплатформенному духу управляемого кода.

Последний метод класса Interlocked, о котором следует поговорить — CompareExchange. Этот маленький метод действительно удобен. Он похож на Interlocked.Exchange в том отношении, что позволяет передать значение из места положения или слота в атомарном режиме. Однако он выполняет такую передачу, только если оригинальное значение при сравнении оказывается эквивалентным представленному образцу. В любом случае метод всегда возвращает оригинальное значение. Одним из чрезвычайно удобных применений метода CompareExchange является создание легковесной спин-блокировки (spin lock). Название этой блокировки происходит из того факта, что если она не может захватить блокировку, то запускает маленький цикл, ожидая появления такой возможности. Обычно при реализации спин-блокировки вы погружаете поток в сон на очень краткий период времени при каждой неудачной попытке захватить блокировку. Таким образом, планировщик потоков получает возможность предоставить время для выполнения другому потоку, пока вы ждете. Если вы не хотите погружать поток в сон, а только освободить его текущий квант времени, то можете передать значение 0 методу Thread.Sleep. Рассмотрим пример:

```
using System;
using System.IO;
using System.Threading;
public class SpinLock
{
    public SpinLock( int spinWait ) {
        this.spinWait = spinWait;
    }
    public void Enter() {
        while( Interlocked.CompareExchange( ref theLock,
                                            1,
                                            0) == 1 ) {
            // Блокировка занята, ожидать.
            Thread.Sleep( spinWait );
        }
    }
```

```
    public void Exit() {
        // Сбросить блокировку.
        Interlocked.Exchange( ref theLock,
                              0 );

    }
    private int theLock = 0;
    private int spinWait;
}
public class SpinLockManager : IDisposable
{
    public SpinLockManager( SpinLock spinLock ) {
        this.spinLock = spinLock;
        spinLock.Enter();
    }
    public void Dispose() {
        spinLock.Exit();
    }
    private SpinLock spinLock;
}
public class EntryPoint
{
    static private Random rnd = new Random();
    private static SpinLock logLock = new SpinLock( 10 );
    private static StreamWriter fsLog =
        new StreamWriter( File.Open("log.txt",
                                    FileMode.Append,
                                    FileAccess.Write,
                                    FileShare.None) );
    private static void RndThreadFunc() {
        using( new SpinLockManager(logLock) ) {
            fsLog.WriteLine( "Поток запускается" );
            fsLog.Flush();
        }

        int time = rnd.Next( 10, 200 );
        Thread.Sleep( time );
        using( new SpinLockManager(logLock) ) {
            fsLog.WriteLine( "Поток завершается" );
            fsLog.Flush();
        }
    }
    static void Main() {
        // Запустить потоки, ожидающие в течение случайного периода времени.
        Thread[] rndthreads = new Thread[ 50 ];
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
                new Thread( new ThreadStart(
                                EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

Этот пример подобен предыдущему. Он создает 50 потоков, которые ожидают в течение случайного периода времени. Однако вместо управления счетчиком потоков он выводит строку в файл протокола. Поскольку эта запись происходит из множества потоков, и методы экземпляра `StreamWriter` не являются безопасными в отношении потоков, вы должны выполнять запись в безопасной манере внутри контекста блокировки. И здесь на помощь приходит класс `SpinLock`. Внутренне он управляется переменной блокировки в форме целочисленного значения и использует `Interlocked.CompareExchange` для регулировки доступа к блокировке. Вызов `Interlocked.CompareExchange` в `SpinLock.Enter` говорит следующее.

1. Если значение блокировки равно 0, заменить его на 1, чтобы обозначить установку блокировки; в противном случае не делать ничего.
2. Если значение слота уже содержит 1, значит, он занят, и вам нужно заснуть и ожидать.

Оба эти действия происходят в атомарном режиме через класс `Interlocked`, так что нет никакой возможности, чтобы более одного потока в единицу времени захватило блокировку. Когда вызывается метод `SpinLock.Exit`, то все, что ему нужно сделать — это сбросить блокировку. Однако это также должно быть сделано автоматически — отсюда и вызов `Interlocked.Exchange`.

В данном примере я решил проиллюстрировать применение идиомы "disposable/using" для реализации детерминированной деструкции, когда вводится другой класс, в данном случае — `SpinLockManager`, — чтобы реализовать идиому RAII. Этот избавляет от необходимости повсеместного написания блоков `finally`. Конечно, вам все равно приходится помнить о применении ключевого слова `using`, но если вы следуете идиоме более тщательно, чем в данном примере, то вы должны реализовать финализатор, который выдаст предупреждение в отладочной сборке, если объект не будет надлежащим образом освобожден[2].

Имейте в виду, что спин-блокировки, реализованные подобным образом, не реентерабельны. Любая функция, которая захватила блокировку, не может быть вызвана снова до тех пор, пока она не освободит блокировку. Это не значит, что вы не можете использовать спин-блокировки с техникой рекурсивного программирования. Это просто означает, что вы должны освободить блокировку перед рекурсивным вызовом, иначе спровоцируете состояние взаимной блокировки.

На заметку! Если вам нужен реентерабельный механизм, вы можете использовать более структурированные объекты ожидания, такие как класс `Monitor`, о котором я расскажу в следующем разделе, или же объекты, базирующиеся на ядре системы.

Кстати, если хотите увидеть некоторый, так сказать, фейерверк, попробуйте закомментировать использование спин-блокировки в методе `RndThreadFunc` и запустить результат несколько раз. Скорее всего, вы заметите, что вывод в файле протокола станет несколько некрасивым. И эта некрасивость вырастет при попытке выполнить тот же тест на многопроцессорной машине.

[2] Подробнее об этой технике читайте в главе 13.

Класс Monitor

В предыдущем разделе я показал, как реализовать спин-блокировку, используя методы класса Interlocked. Спин-блокировка не всегда является наиболее эффективным механизмом синхронизации, особенно, если вы используете его в среде, где синхронизация почти гарантирована. Планировщик потоков должен будить поток и позволять ему повторно проверять переменную блокировки. Как я уже упоминал ранее, спин-блокировка идеальна, когда вам нужен легковесный не реентерабельный механизм, и шансы того, что потоку придется ждать, невелики. Когда вам известно, что вероятность ожидания высока, вы должны использовать механизм синхронизации, позволяющий планировщику обойтись без пробуждения потока до тех пор, пока доступна блокировка. .NET предлагает класс System.Threading.Monitor для обеспечения синхронизации между потоками в пределах одного процесса. Вы можете использовать этот класс для защиты доступа к определенным переменным или для разграничения доступа к коду, который должен быть запущен только в одном потоке в единицу времени.

На заметку! Шаблон Monitor предоставляет способ обеспечения такой синхронизации, что только один метод или блок защищенного кода будет выполняться в единицу времени. Mutex обычно используется для той же цели. Однако Monitor намного легче и быстрее. Monitor подходит, когда вам нужно защитить доступ к коду внутри одного процесса. Mutex подходит в тех случаях, когда необходимо защитить доступ к ресурсу из множества процессов.

Одним потенциальным источником путаницы, связанной с классом Monitor, является то, что вы не можете создать экземпляр этого класса. Класс Monitor, во многом подобно классу Interlocked, представляет собой просто включающее пространство имен для коллекции статических методов, осуществляющих всю необходимую работу. Если вы привыкли к использованию критических секций в Win32, то вам известно, что в некоторой точке вы должны выделить и инициализировать структуру CRITICAL_SECTION. Затем, чтобы войти и выйти из блокировки, вы вызываете Win32-функции EnterCriticalSection и LeaveCriticalSection. Вы можете решить ту же задачу, используя класс Monitor в управляемой среде. Для входа и выхода из критической секции вы вызываете методы Monitor.Enter и Monitor.Exit. Там, где вы передаете объект CRITICAL_SECTION функциям критической секции Win32, передается ссылка на объект методам Monitor.

Внутренне CLR управляет блоком синхронизации для каждого экземпляра объекта в процессе. По сути, это флаг того же рода, что и целочисленное значение, использованное в примерах предыдущего раздела, описывающего класс Interlocked. Когда вы получаете блокировку на объекте, флаг устанавливается. Когда блокировка снимается, флаг сбрасывается. Класс Monitor — это ворота доступа к этому флагу. Непостоянство этой схемы проявляется в том, что каждый экземпляр объекта в CLR потенциально содержит одну из таких блокировок. Я говорю "потенциально", потому что CLR выделяет их в "ленивом" режиме, поскольку блокировка не каждого объекта будет использована. Чтобы реализовать критическую секцию, все, что вам нужно сделать — это создать экземпляр Object. Взглянем на пример использования класса Monitor, модифицировав пример из предыдущего раздела:

```
using System;
using System.Threading;
```

```
public class EntryPoint
{
    static private object theLock = new Object();
    static private int numberThreads = 0;
    static private Random rnd = new Random();
    private static void RndThreadFunc() {
        // Управляющий поток увеличивает счетчик и ожидает
        // произвольный период времени от 1 до 12 секунд.
        try {
            Monitor.Enter( theLock );
            ++numberThreads;
        }
        finally {
            Monitor.Exit( theLock );
        }

        int time = rnd.Next( 1000, 12000 );
        Thread.Sleep( time );
        try {
            Monitor.Enter( theLock );
            --numberThreads;
        }
        finally {
            Monitor.Exit( theLock );
        }
    }
    private static void RptThreadFunc() {
        while( true ) {
            int threadCount = 0;
            try {
                Monitor.Enter( theLock );
                threadCount = numberThreads;
            }
            finally {
                Monitor.Exit( theLock );
            }
            Console.WriteLine( "{0} поток(ов) активно",
                               threadCount );
            Thread.Sleep( 1000 );
        }
    }
    static void Main() {
        // Старт отчетного потока.
        Thread reporter =
            new Thread( new ThreadStart(
                             EntryPoint.RptThreadFunc) );
        reporter.IsBackground = true;
        reporter.Start();
        // Старт потоков, ожидающих случайный период времени.
        Thread[] rndthreads = new Thread[ 50 ];
```

```
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
                new Thread( new ThreadStart(
                    EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

Обратите внимание, что я осуществляю весь доступ к переменной numberThreads внутри критической секции в форме объекта блокировки. Перед каждым обращением средство доступа должно получить блокировку на экземпляре объекта theLock. Типом поля theLock является object — просто потому, что он не имеет значения. Единственное, что имеет значение — это то, что это ссылочный тип, а не тип значения. Поскольку экземпляр object вам нужен только для того, чтобы использовать его внутренний блок синхронизации, вы можете просто создать для этого экземпляр System.Object.

Вероятно, вы заметили еще одну вещь — код выглядит более громоздко, чем в версии, использующей методы Interlocked. Всякий раз, когда вы вызываете Monitor.Enter, вы хотите гарантировать, что неважно как, но обязательно будет запущен соответствующий ему Monitor.Exit. В примерах я смягчил эту проблему, применив класс Interlocked, упаковав использование его методов внутрь класса по имени SpinLockManager. Можете ли вы представить себе хаос, который может наступить, если вызов Monitor.Exit будет пропущен из-за исключения? Поэтому вы всегда должны использовать блок try/finally в таких ситуациях. Создатели языка C# осознавали, что разработчикам придется приложить много усилий, чтобы обеспечить наличие блоков finally во всех случаях, где нужен вызов Monitor.Exit. Поэтому они облегчили нам жизнь, введя ключевое слово lock. Рассмотрим тот же пример снова, на этот раз применив ключевое слово lock:

```
using System;
using System.Threading;
public class EntryPoint
{
    static private object theLock = new Object();
    static private int numberThreads = 0;
    static private Random rnd = new Random();
    private static void RndThreadFunc() {
        // Управляющий поток увеличивает счетчик и ожидает
        // произвольный период времени от 1 до 12 секунд.
        lock( theLock ) {
            ++numberThreads;
        }

        int time = rnd.Next( 1000, 12000 );
        Thread.Sleep( time );
        lock( theLock ) {
            --numberThreads;
        }
    }
```

```
    private static void RptThreadFunc() {
        while( true ) {
            int threadCount = 0;
            lock( theLock ) {
                threadCount = numberThreads;
            }
            Console.WriteLine( "{0} поток(ов) активно",
                               threadCount );
            Thread.Sleep( 1000 );
        }
    }
    static void Main() {
        // Старт отчетного потока.
        Thread reporter =
            new Thread( new ThreadStart(
                          EntryPoint.RptThreadFunc) );
        reporter.IsBackground = true;
        reporter.Start();
        // Старт потоков, ожидающих случайный период времени.
        Thread[] rndthreads = new Thread[ 50 ];
        for( uint i = 0; i < 50; ++i ) {
            rndthreads[i] =
                new Thread( new ThreadStart(
                    EntryPoint.RndThreadFunc) );
            rndthreads[i].Start();
        }
    }
}
```

Обратите внимание, что код стал намного яснее, и в нем теперь вообще нет явных вызовов методов Monitor. Однако "за кулисами" компилятор развертывает ключевое слово lock в знакомую конструкцию try/finally с вызовами Monitor.Enter и Monitor.Exit. Вы можете убедиться в этом, просмотрев сгенерированный код IL с помощью ILDASM.

Во многих случаях синхронизация, реализованная внутренне в пределах класса, также проста, как и реализация критических секций в такой манере. Но когда всем методам класса нужен только один объект блокировки, вы можете упростить эту модель еще больше, исключив дополнительный объект System.Object и используя ключевое слово this при захвате блокировки через класс Monitor. Вероятно, вы еще не раз встретите этот шаблон использования в коде C#. Хотя он избавляет вас от необходимости создания экземпляра объекта System.Object, который весьма легковесен, все же он несет с собой некоторые опасности. Например, внешний потребитель вашего объекта может в действительности попытаться использовать блок синхронизации внутри вашего объекта вызовом Minitor.Enter еще перед вызовом одного из ваших методов, которые пытаются захватить ту же блокировку. Технически это почти нормально, потому что один и тот же поток может вызывать Monitor.Enetr несколько раз. Другими словами, блокировки Monitor реентерабельны в отличие от спин-блокировок из предыдущего раздела. Однако когда блокировка освобождается, это должно быть сделано вызовом Monitor.Exit равное количество раз. Поэтому теперь вы должны полагаться на потребителя вашего объ-

екта — в том, что он использует либо ключевое слово lock, либо блок try/finally, чтобы гарантировать, что всем его вызовам Monitor.Exit будут соответствовать Monitor.Exit. Всякий раз, когда вы можете избежать неопределенности, делайте это. Поэтому я не советую использовать блокировку через ключевое слово this, а лучше вместо этого применять приватный экземпляр System.Object в качестве объекта блокировки. Вы могли бы достичь того же эффекта, если бы существовал способ объявить флаг блока синхронизации объекта как private, но, к сожалению, это невозможно.

Предупреждение об упаковке

Когда вы применяете методы Monitor для реализации блокировки, то внутренне класс Monitor использует блок синхронизации экземпляров объектов для управления блокировкой. Поскольку каждый экземпляр объекта потенциально может иметь блок синхронизации, вы можете использовать любую ссылку на объект, даже объектную ссылку на тип значения. Хотя вы можете это сделать, тем не менее, никогда не передавайте экземпляр типа значения Monitor.Enter, как показано в следующем примере кода:

```
using System;
using System.Threading;
public class EntryPoint
{
    static private int counter = 0;
    // НИКОГДА НЕ ДЕЛАЙТЕ ЭТОГО!!!
    static private int theLock = 0;
    static private void ThreadFunc() {
        for( int i = 0; i < 50; ++i ) {
            Monitor.Enter( theLock );
            try {
                Console.WriteLine( ++counter );
            }
            finally {
                Monitor.Exit( theLock );
            }
        }
    }
    static void Main() {
        Thread thread1 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        Thread thread2 =
            new Thread( new ThreadStart(EntryPoint.ThreadFunc) );
        thread1.Start();
        thread2.Start();
    }
}
```

Если вы попытаетесь выполнить этот код, то немедленно столкнетесь с исключением SynchronizationLockException, которое сообщит, что метод объекта синхронизации был вызван из не синхронизированного блока кода. Почему это случилось? Чтобы ответить на этот вопрос, вы должны вспомнить, что при передаче типа

значения методу, принимающему ссылочный тип, происходит неявная упаковка. К тому же многократная передача одного и того же экземпляра типа значения в один и тот же метод каждый раз даст в результате разные ссылки на упаковку. Поэтому ссылочный объект, используемый внутри тела Monitor.Exit, отличается от того, что используется внутри тела Monitor.Enter. Это еще один пример того, какими неприятностями для вас грозит неявная упаковка в языке C#. Возможно, вы заметили, что я применяю в данном примере подход на основе try/finally. Это потому, что проектировщики языка C# создали оператор lock таким образом, что он не принимает типов значений. Поэтому, если вы привыкли к использованию оператора lock для обработки критических секций, вам не нужно беспокоиться о нечаянной передаче упакованного типа значения методам Monitor.

Pulse и Wait

Я не могу преувеличить пользу от методов Monitor в реализации критических секций. Однако методы Monitor обладают и другими возможностями, помимо реализации простых критических секций. Вы также можете использовать их для реализации квитирования (handshaking) между потоками, а также для реализации поочередного доступа к разделенным ресурсам.

Когда поток успешно входит в блокированную область, он может встретить установленную блокировку и войти в очередь ожидания вызовом Monitor.Wait. Первый параметр Monitor.Wait — объектная ссылка, чей блок синхронизации представляет используемую блокировку. Второй параметр — значение таймаута.

Monitor.Wait возвращает булевское значение, указывающее на успех ожидания либо завершение его по истечении таймаута. Если ожидание завершилось успешно, возвращается true, в противном случае — false. Когда поток, вызвавший Monitor.Wait, завершает ожидание успешно, он покидает состояние ожидания как владелец блокировки.

Если потоки, столкнувшись с блокировкой, могут войти в состояние ожидания, должен существовать какой-нибудь механизм сообщения Monitor, что он может отдать блокировку обратно одному из ожидающих потоков как можно скорее. Таким механизмом является метод Monitor.Pulse. Только поток, который в данный момент удерживает блокировку, может вызвать Monitor.Pulse. Когда вызывается этот метод, то поток, стоящий первым в очереди ожидающих, перемещается в очередь готовых. Как только поток, владеющий блокировкой, снимает ее — вызовом Monitor.Exit или Monitor.Wait — то первый поток в очереди готовых получает разрешение работать. Потоки в очереди готовых включают помещенные с помощью Pulse и те, что были заблокированы после вызова Monitor.Enter. Вдобавок поток, владеющий блокировкой, может переместить все ожидающие потоки в очередь готовых вызовом Monitor.PulseAll.

Существует много причудливых задач синхронизации, которые вы можете выполнить, используя методы Monitor.Pulse и Monitor.Wait. Например, рассмотрим следующий пример, реализующий механизм квитирования между двумя потоками. Цель — заставить оба потока увеличить значение счетчика в альтернативной манере.

```
using System;
using System.Threading;
```

```
public class EntryPoint
{
   static private int counter = 0;
   static private object theLock = new Object();
   static private void ThreadFunc1() {
      lock( theLock ) {
         for( int i = 0; i < 50; ++i ) {
            Monitor.Wait( theLock, Timeout.Infinite );
            Console.WriteLine( "{0} из потока {1}",
                     ++counter,
                     Thread.CurrentThread.GetHashCode() );
            Monitor.Pulse( theLock );
         }
      }
   }
   static private void ThreadFunc2() {
      lock( theLock ) {
         for( int i = 0; i < 50; ++i ) {
            Monitor.Pulse( theLock );
            Monitor.Wait( theLock, Timeout.Infinite );
            Console.WriteLine( "{0} из потока {1}",
                                   ++counter,
            Thread.CurrentThread.GetHashCode() );
         }
      }
   }
   static void Main() {
      Thread thread1 =
         new Thread( new ThreadStart(EntryPoint.ThreadFunc1) );
      Thread thread2 =
         new Thread( new ThreadStart(EntryPoint.ThreadFunc2) );
      thread1.Start();
      thread2.Start();
   }
}
```

Вы заметите, что вывод этого примера демонстрирует, что потоки увеличивают `counter` в альтернативной манере.

В качестве другого примера вы можете реализовать черновой пул потоков, используя `Monitor.Wait` и `Monitor.Pulse`. Может быть, действительно излишне делать такие вещи, поскольку .NET Framework представляет объект `ThreadPool`, который достаточно устойчив и вероятно, использует оптимизированные порты ввода-вывода лежащей в основе операционной системы. Для примера, однако, я продемонстрирую, как реализовать пул рабочих потоков, которые ожидают постановки в очередь рабочих элементов:

```
using System;
using System.Threading;
using System.Collections;
public class CrudeThreadPool
{
```

```
    static readonly int MAX_WORK_THREADS = 4;
    static readonly int WAIT_TIMEOUT = 2000;
    public delegate void WorkDelegate();
    public CrudeThreadPool() {
        stop = 0;
        workLock = new Object();
        workQueue = new Queue();
        threads = new Thread[ MAX_WORK_THREADS ];
        for( int i = 0; i < MAX_WORK_THREADS; ++i ) {
            threads[i] =
                new Thread( new ThreadStart(this.ThreadFunc) );
            threads[i].Start();
        }
    }
    private void ThreadFunc() {
        lock( workLock ) {
            int shouldStop = 0;
            do {
                shouldStop = Interlocked.Exchange( ref stop,
                                                   stop );
                if( shouldStop == 0 ) {
                    WorkDelegate workItem = null;
                    if( Monitor.Wait(workLock, WAIT_TIMEOUT) ) {
                        // Обработать элемент из начала очереди.
                        lock( workQueue ) {
                            workItem =
                            (WorkDelegate) workQueue.Dequeue();
                        }
                        workItem();
                    }
                }
            } while( shouldStop == 0 );
        }
    }
    public void SubmitWorkItem( WorkDelegate item ) {
        lock( workLock ) {
            lock( workQueue ) {
                workQueue.Enqueue( item );
            }
            Monitor.Pulse( workLock );
        }
    }
    public void Shutdown() {
        Interlocked.Exchange( ref stop, 1 );
    }
    private Queue workQueue;
    private Object workLock;
    private Thread[] threads;
    private int stop;
}
```

```
public class EntryPoint
{
    static void WorkFunction() {
        Console.WriteLine( "WorkFunction() вызывается на потоке {0}",
                           Thread.CurrentThread.GetHashCode() );
    }
    static void Main() {
        CrudeThreadPool pool = new CrudeThreadPool();
        for( int i = 0; i < 10; ++i ) {
            pool.SubmitWorkItem(
                new CrudeThreadPool.WorkDelegate(
                                    EntryPoint.WorkFunction) );
        }
        pool.Shutdown();
    }
}
```

В этом случае рабочий элемент представлен делегатом, не принимающим параметров и не возвращающим значений. При создании объекта `CrudeThreadPool` он создает пул потоков и запускает их путем запуска главного метода обработки рабочего элемента. Этот метод просто вызывает `Monitor.Wait` для ожидания элемента для постановки в очередь. Когда вызывается `SubmitWorkItem`, элемент помещается в очередь и вызывается `Monitor.Pulse` для освобождения одного из рабочих потоков. Естественно, доступ к очереди должен быть синхронизирован. В этом случае для синхронизации доступа к очереди используется ссылочный тип. Вдобавок рабочие потоки не должны ждать вечно, поскольку они должны периодически просыпаться и проверять флаг, чтобы узнать, не пора ли им красиво завершиться. Опционально вы можете просто превратить рабочие потоки в фоновые установкой свойства `IsBackground` внутри метода `Shutdown`. Однако в этом случае рабочие потоки могут быть завершены перед тем, как они завершат выполнение своей работы. В зависимости от ситуации это может быть или не быть допустимым. Обратите внимание, что я решил использовать методы `Interlocked` для управления флагом останова для указания необходимости выхода рабочему потоку.

На заметку! Другой удобный прием сообщения потокам о том, что они должны завершиться, заключается в создании специального вида рабочего элемента, который скомандует потоку завершиться. Трюк состоит в том, что вы должны убедиться, что поместили столько специальных рабочих элементов в очередь, сколько потоков в пуле.

Блокирующие объекты

.NET Framework предоставляет несколько высокоуровневых блокирующих объектов, которые вы можете использовать для синхронизации доступа к данным из множества потоков. Я полностью посвятил предыдущий раздел одному типу блокировок — `Monitor`. Однако класс `Monitor` не реализует объекта блокировки ядра; вместо этого он предоставляет доступ к блокировке синхронизации каждого экземпляра объекта .NET. Ранее в этой главе я также описал методы примитивного класса `Interlocked`, которые можно применять для реализации спин-блокировок. Одной причиной того, что спин-блокировки можно считать примитивными, явля-

ется то, что они не реентерабельны, и потому не позволяют захватывать одну и ту же блокировку многократно. Другие высокоуровневые блокирующие объекты обычно позволяют это, до тех пор, пока вы обеспечиваете соответствие количества операций установки блокировки количеству операций по ее освобождению. В настоящем разделе я хочу рассказать о некоторых полезных блокирующих объектах, предлагаемых .NET Framework. Независимо от того, какой тип блокирующих объектов вы используете, всегда нужно стремиться писать код, который обеспечивает как можно более кратковременное удержание блокировки. Например, если вы захватили блокировку для доступа к некоторым данным внутри метода, который может потребовать ощутимого времени для обработки данных, удерживайте блокировку лишь столько времени, сколько необходимо для создания копии в локальном стеке, и затем как можно скорее снимайте блокировку. Применяя такую технику, вы гарантируете, что другие потоки в вашей системе не будут заблокированы в течение слишком длительного времени, ожидая доступа к тем же данным.

ReaderWriterLock

Синхронизируя доступ к разделенным данным между потоками, вы часто столкнетесь с ситуацией, когда несколько потоков читают, или потребляют, данные, в то время как только один поток пишет, или производит, эти данные. Очевидно, что все потоки должны захватывать блокировку прежде, чем они обратятся к данным, чтобы предотвратить состояние "гонок", когда один поток пишет данные, в то время как другой находится в процессе их чтения, потенциально порождая мусор для читателя. Однако взаимная блокировка определенно выглядит неэффективной для тех потоков, которые просто собираются читать данные, а не модифицировать их. Нет причины, по которой они не могли бы читать данные параллельно, не боясь наступать друг другу на пятки.

`ReaderWriterLock` позволяет элегантно избежать такой неэффективности. По сути, он позволяет множеству читателей обращаться к данным одновременно, но как только одному из потоков понадобится записать данные, все, кроме писателя, должны немедленно убрать свои руки. `ReaderWriterLock` управляет таким поведением, используя две внутренних очереди. Одна — для ожидающих читателей, а другая — для ожидающих писателей. На рис. 12.2 показана диаграмма высокоуровневой блокировки, как она выглядит внутри `ReaderWriterLock`. В этом сценарии в системе выполняются четыре потока, и в данный момент ни один из них не пытается обратиться к данным под блокировкой.

Чтобы обратиться к данным, читатель вызывает `AcquireReaderLock`. При состоянии блокировки, показанном на рис. 12.2, читатель будет немедленно помещен в категорию владельцев блокировки. Обратите внимание на использование множественного числа — может существовать несколько владельцев блокировки чтения. Все становится интереснее, как только потоки пытаются захватить блокировку записи вызовом `AcquireWriterLock`. В этом случае писатель помещается в очередь писателей, поскольку читатели в данный момент владеют блокировкой, как показано на рис. 12.3.

Как только все читатели освободят свои блокировки вызовом `ReleaseReaderLock`, то писатель — в данном случае поток B — сможет войти в область владельцев блокировки.

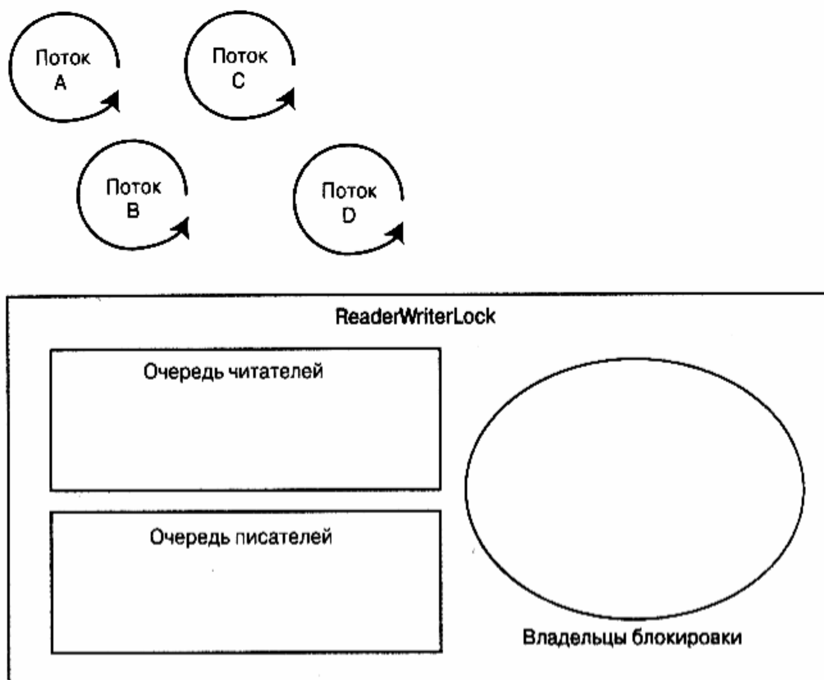

Рис. 12.2. Неинициализированный ReaderWriterLock

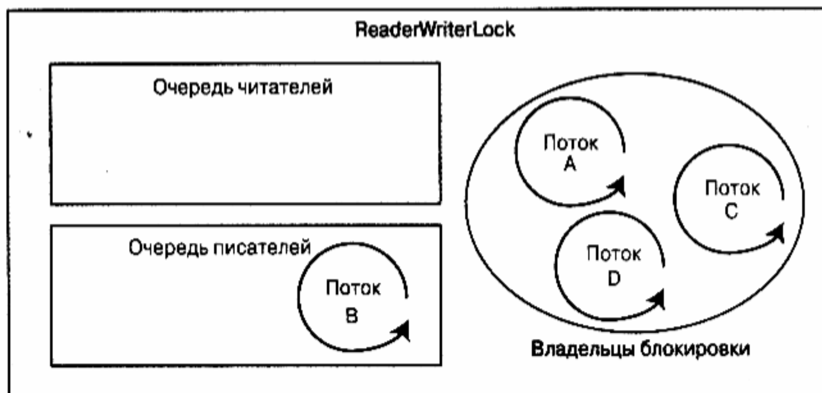

Рис. 12.3. Поток-писатель, ожидающий ReaderWriterLock

Но что случится, если поток А освободит свою блокировку чтения и затем попытается заново захватить блокировку читателя прежде, чем писатель получит шанс захватить блокировку? Если поток А сможет перезахватить блокировку, то любой поток, ожидающий в очереди, сможет потенциально пострадать от блокировки. Чтобы избежать этого, любой поток, который пытается затребовать блокировку чтения, в то время как писатель находится в очереди, помещается в очередь читателей, как показано на рис. 12.4.

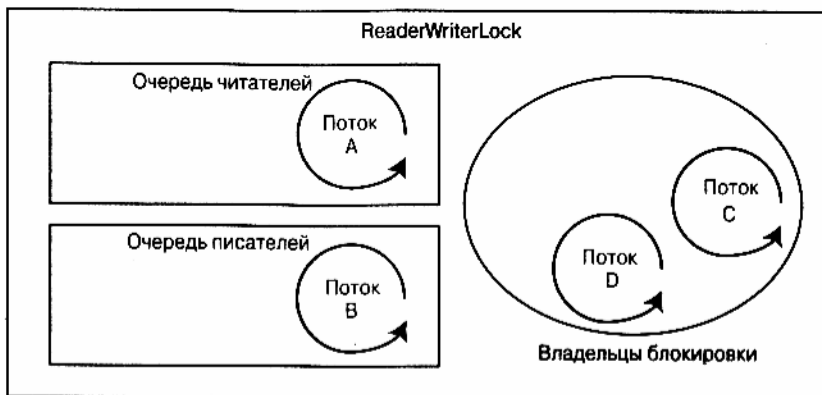

Рис. 12.4. Читатель пытается перезахватить блокировку

Естественно, эта схема предоставляет преимущества очереди писателей. Это имеет смысл, с учетом того факта, что любой читатель хотел бы получить наиболее свежую информацию. Конечно, если поток, нуждающийся в блокировке записи, вызывал бы `AcquireWriterLock`, пока `ReaderWriterLock` пребывает в состоянии, показанном на рис. 12.2, он бы немедленно был помещен в категорию владельцев блокировки, без необходимости проходить через очередь писателей.

`ReaderWriterLock` является реентерабельным. Поэтому поток может многократно вызывать любой из методов захвата блокировки, до тех пор, пока этому количеству вызовов будет соответствовать количество вызовов методов ее освобождения. Всякий раз, когда блокировка запрашивается заново, увеличивается значение ее внутреннего счетчика. Должно показаться очевидным, что один поток не может одновременно владеть как блокировкой чтения, так и блокировкой записи, как и не может ожидать сразу в двух очередях `ReaderWriterLock`. Однако поток может повышать или понижать тип блокировки, которой он владеет. Например, если поток в данный момент владеет блокировкой чтения и вызывает `UpgradeToWriterLock`, его блокировка читателя освобождается независимо от значения счетчика блокировок и затем помещается в очередь писателей. `UpgradeToWriterLock` возвращает объект типа `LockCookie`. Вы должны сохранить этот объект и передать его `DowngradeFromWriterLock`, когда завершите операцию записи. `ReaderWriterLock` использует cookie-набор для восстановления счетчика блокировки чтения на объекте. Даже несмотря на то, что вы можете увеличить счетчик блокировки записи, получив его через `UpgradeToWriterLock`, ваш вызов `DowngradeFromWriterLock` освободит блокировку записи, независимо от значения ее счетчика. Поэтому лучше избегать полагаться на счетчик внутри "повышенной" блокировки записи.

Как почти с любым другим объектом синхронизации в .NET Framework, вы можете указывать таймаут почти с любым методом захвата блокировки. Таймаут задается в миллисекундах. Однако вместо того, чтобы возвращать булевское значение, указывая на успех или неудачу захвата блокировки, эти методы генерируют исключение типа `ApplicationException` в случае истечения таймаута. Поэтому, если вы передаете одной из этих функций значение таймаута, отличное от `Timeout.Infinite`, не забудьте поместить вызов в блок `try`, чтобы перехватывать потенциальные исключения.

ReaderWriterLockSlim

.NET 3.5 представляет новый стиль блокировки читателей/писателей, называемый ReaderWriterLockSlim. Он несет в себе несколько усовершенствований, включая лучшую защиту от взаимных блокировок, повышенную эффективность и возможность освобождения (disposability). Также он по умолчанию не поддерживает рекурсии, что добавляет эффективности. Если все-таки вам нужна рекурсия, то на этот случай ReaderWriterLockSlim предусматривает перегруженный конструктор, принимающий значение из перечисления LockRecursionPolicy. Microsoft рекомендует применять во всех новых разработках ReaderWriterLockSlim вместо ReaderWriterLock.

В отношении ReaderWriterLockSlim поток может находиться в следующих четырех состояниях:

- свободный (unheld);
- режим чтения (read mode);
- обновляемый режим (upgradeable mode);
- режим записи (write mode).

Свободный режим означает, что поток вообще не пытается ни читать, ни записывать ресурс. Если поток находится в режиме чтения, он имеет доступ к ресурсу по чтению после успешного вызова метода EnterReadLock. Аналогично, если поток находится в режиме записи, он имеет доступ на записи после успешного вызова EnterWriteLock. Как и в случае ReaderWriterLock, только один поток в единицу времени может находиться в режиме записи, и пока поток находится в режиме записи, все потоки блокируются от входа в режим чтения. Естественно, поток, пытающийся войти в режим записи, блокируется до тех пор, пока любой другой поток находится в режиме чтения. Как только этот другой поток закончит чтение, поток, ожидающий права записи, освобождается. А что же такое обновляемый режим?

Обновляемый (upgradeable) режим удобен, если у вас есть поток, который нуждается в доступе по чтению к ресурсу, но при этом время от времени нуждается и в доступе по записи к тому же ресурсу. Без обновляемого режима потоку приходилось бы выходить из режима чтения и затем следом пытаться войти в режим записи. За время его пребывания в свободном режиме другой поток может войти в режим чтения и тем самым заблокировать последующую попытку войти в режим записи. Только один поток в единицу времени может находиться в обновляемом режиме, и входит он в этот режим посредством вызова EnterUpgradeableReadLock. Обновляемые потоки могут входить в режим чтения или записи рекурсивно, даже для экземпляров ReaderWriterLockSlim, которые были созданы с выключенной рекурсией. По сути, обновляемый режим является более мощной формой режима чтения, который обеспечивает повышенную эффективность при входе в режим записи. Если поток пытается войти в обновляемый режим, а другой поток в это время находится в режиме записи, либо есть потоки в очереди на вход в режим записи, то поток, вызывающий EnterUpgradeableReadLock, будет блокирован до тех пор, пока другой поток не выйдет из режима записи и все потоки, ожидающие в очереди, не войдут и выйдут из режима записи. Это поведение идентично и для потоков, пытающихся войти в режим чтения.

ReaderWriterLockSlim в некоторых ситуациях может сгенерировать исключение LockRecursionException. Поскольку экземпляры ReaderWriterLockSlim по умолчанию не поддерживают рекурсии, попытка многократно вызвать EnterReadLock, EnterWriteLock или EnterUpgradeableReadLock из одного и того же потока приведет к одному из этих исключений. Вдобавок независимо от того, поддерживает экземпляр рекурсию или нет, поток, уже находящийся в обновляемом режиме и пытающийся вызвать EnterReadLock, или поток, находящийся в режиме записи и пытающийся вызвать EnterReadLock, могут привести систему к состоянию взаимной блокировки, поэтому исключение LockRecursionException также генерируется и в этих случаях.

Если вы знакомы с классом Monitor, вы можете узнать здесь идиому, представленную в именах методов ReaderWriterLockSlim. Всякий раз, когда поток входит в состояние, он должен вызвать один из методов Enter..., и всякий раз, когда он покидает состояние, он должен вызвать один из соответствующих методов Exit... Вдобавок, подобно Monitor, ReaderWriterLockSlim предоставляет методы, позволяющие вам попытаться войти в блокировку, не рискуя заблокировать навсегда, с помощью таких методов, как TryEnterReadLock, TryEnterUpgradeableReadLock и TryEnterWriteLock. Каждый метод Try... позволяет передать ему значение таймаута, указывающее период времени, который вы готовы ждать.

Общее руководство, связанное с использованием Monitor, заключается в том, чтобы не использовать Monitor напрямую, а работать с ним опосредованно — через ключевое слово C# lock. Тогда вам не придется заботиться о том, чтобы не забыть вызвать Monitor.Exit, и не нужно набирать блок finally, чтобы гарантировать вызов Monitor.Exit при любых обстоятельствах. К сожалению, не существует доступного эквивалентного механизма облегчить установку и снятие блокировок при использовании ReaderWriterLockSlim. Всегда будьте осторожны с вызовом методов Exit... по завершении работы с блокировкой, и вызывайте их из блока finally, чтобы их вызов произошел даже в исключительных случаях.

Mutex

Объект Mutex — утяжеленный тип блокировки, которую вы можете использовать для реализации взаимно исключающего доступа к ресурсам. .NET Framework поддерживает два типа реализаций Mutex. Если он создается без имени, то вы получаете то, что называется локальным мьютексом (mutex). Но если вы создаете его с именем, то такой объект Mutex используется между несколькими процессами и реализуется с применением объекта ядра Win32 — одного из наиболее тяжеловесных объектов блокировки. Под этим я имею в виду, что это наиболее медленный и требующий максимальных накладных расходов объект, когда он используется для защиты ресурсов от управляемых потоков. Другие типы блокировок, такие как классы ReaderWriterLock и Monitor, предназначены для работы только в пределах одного процесса. Поэтому в интересах эффективности вы должны использовать объект Mutex только тогда, когда вы действительно нуждаетесь в синхронизации выполнения или доступа к некоторым ресурсам между несколькими процессами.

Как и другие высокоуровневые объекты синхронизации, Mutex является реентерабельным. Когда ваш поток должен установить эксклюзивную блокировку, вы вызываете метод WaitOne. Как обычно, вы можете передать значение таймаута, выраженное в миллисекундах, в ожидании объекта Mutex. Метод возвращает бу-

левское значение, которое будет равно `true`, если ожидание завершено успешно, и `false` — если истек таймаут. Поток может вызывать метод `WaitOne` столько раз, сколько хочет — до тех пор, пока количество вызовов совпадает с количеством вызовов метода `ReleaseMutex`.

Поскольку вы можете применять объекты `Mutex` между несколькими процессами, каждому из процессов нужно как-то идентифицировать `Mutex`. Поэтому вы можете указать не обязательное имя при создании экземпляра `Mutex`. Присваивание имени — простейший способ для другого процесса идентифицировать и открыть мьютекс. Поскольку все имена `Mutex` находятся в глобальном пространстве имен всей операционной системы, важно присваивать им достаточно уникальные имена, чтобы они не конфликтовали с именами `Mutex`, созданными другими приложениями. Я рекомендую использовать имя, базирующееся на строковой форме GUID, сгенерированной `GUIDGEN.exe`.

На заметку! Я уже упоминал, что имена объектов ядра являются глобальными для всей машины. Это утверждение не вполне соответствует действительности, если рассмотреть быстрое переключение пользователей Windows и Terminal Services. В этих случаях пространство имен, содержащее имя этих объектов ядра, создается для каждого зарегистрированного пользователя. Для ситуаций, когда вы действительно хотите, чтобы ваше имя существовало в глобальном пространстве имен, можете снабдить имя префиксом -- специальной строкой `\Global`. Дополнительную информацию на эту тему вы найдете в книге Марка Руссиновича (Mark E. Russinovich) и Дэвида Соломона (David A. Solomon) *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000* (Redmond,WA: Microsoft Press, 2004 г.).

Если все, что касается объектов `Mutex`, покажется совершенно знакомым тем из вас, кто имеет опыт “родной” разработки Win32, то это объясняется тем, что лежащий в их основе механизм фактически является объектом `Mutex` в Win32. То есть на самом деле вы получаете в свои руки действительный дескриптор операционной системы через свойство `SafeWaitHandle`, унаследованное от базового класса `WaitHandle`. Мне еще будет, что сказать о классе `WaitHandle`, в разделе “Объекты синхронизации Win32 и `WaitHandle`”, где речь пойдет о его достоинствах и недостатках. Важно отметить, что поскольку вы реализуете `Mutex` на основе семафора ядра, это влечет за собой переключение в режим ядра при каждой манипуляции или ожидании с `Mutex`. Такие переходы чрезвычайно медленны и должны быть сведены к минимуму, если вы выполняете критичный ко времени код.

Совет. Если возможно, избегайте использования объектов режима ядра для синхронизации между потоками одного и того же процесса. Отдавайте более легковесным механизмам, таким как класс `Monitor` или `Interlocked`. Но при необходимости синхронизации потоков между разными процессами у вас нет выбора, кроме как использовать объекты ядра. На моей текущей тестовой машине простой тест показывает, что применение `Mutex` требует в 44 раза больше времени, чем применение класса `InterLocked`, и в 34 раза больше — чем класса `Mutex`.

Семафоры

.NET Framework поддерживает семафоры через класс `System.Threading.Semaphore`. Они служат для того, чтобы обеспечить возможность определенному числу потоков использовать ресурсы совместно. Всякий раз, когда поток обращается к семафору через `WaitOne` или любой другой из методов `Wait...`, счетчик семафора уменьша-

ется. Когда поток-владелец вызывает Release, счетчик увеличивается. Если поток пытается обратиться к семафору, когда счетчик равен нулю, он блокируется до тех пор, пока другой поток не вызовет Release. При вызове Release счетчик увеличится.

Точно так же, как и с Mutex, когда вы создаете семафор, то можете указывать или не указывать имя, по которому другие процессы смогут идентифицировать его. Если вы создаете семафор без имени, то получаете локальный семафор, который доступен только в пределах одного процесса. В любом случае лежащая в основе реализация использует семафор — объект Win32 ядра. Поэтому он является очень тяжелым объектом синхронизации, медленным и неэффективным. Вы должны предпочитать локальные семафоры семафорам именованным, если только вам не нужно синхронизировать доступ между несколькими процессами по причинам безопасности.

Обратите внимание, что поток может входить в семафор рекурсивно. Однако он должен вызывать Release соответствующее количество раз, чтобы восстановить счетчик доступности семафора. Задача обеспечения соответствия вызовов методов Wait... последующим вызовам Release полностью ложится на вас. Ничто не удержит вас от излишних вызовов Release. Если вы это сделаете, то когда другой поток позднее вызовет Release, он может тем самым попытаться вытолкнуть счетчик за пределы допустимого лимита значений, что приведет к генерации исключения SemaphoreFullException. Такие ошибки очень трудно находить, потому что точка, в которой возникла причина ошибки, находится далеко от точки, где она проявляется.

События

В .NET Framework вы можете использовать два типа сигнальных событий: ManualResetEvent и AutoResetEvent. Как и с объектом Mutex, эти объекты событий отображаются непосредственно на объекты событий Win32. Если вы знакомы с применением событий Win32, то почувствуете себя как дома с объектами событий .NET. Подобно объектам Mutex, работа с объектами событий влечет за собой медленные переключения в режим ядра. Оба типа событий становятся сигнальными, когда кто-то вызывает метод Set на экземпляре события. В этой точке поток, ожидающий события, освобождается. Потоки ожидают события, вызывая унаследованный метод WaitHandle.WaitOne — тот же самый, который вы вызываете для ожидания, когда Mutex станет сигнальным.

Я с осторожностью утверждаю, что ожидающий поток освобождается, когда событие становится сигнальным. Возможно такое, что множество потоков могут быть освобождены, когда событие становится сигнальным. В этом фактически состоит разница между ManualResetEvent и AutoResetEvent. Когда ManualResetEvent становится сигнальным, все потоки, ожидающие на нем, освобождаются. Оно остается сигнальным до тех пор, пока кто-то не вызовет его метод Reset. Если любой поток вызывает WaitOne, когда ManualResetEvent уже является сигнальным, то ожидание немедленно завершается успешно. С другой стороны, объекты AutoResetEvent освобождают только один ожидающий поток и затем немедленно автоматически переходят в несигнальное состояние. Вы можете представить, что все потоки, ожидающие на AutoResetEvent, ожидают в очереди, где только первый поток из очереди освобождается, когда событие становится сигнальным.

Однако, несмотря на то, что удобно предположить, что ожидающие потоки стоят в очереди, вы не можете выдвигать никаких предположений о том, какой именно ожидающий поток будет освобожден первым. AutoResetEvent также известно, как *событие синхронизации* (sync events), исходя из такого поведения.

Используя тип AutoResetEvent, вы можете реализовать черновой пул потоков, где несколько потоков ожидают сигнала от AutoResetEvent, сообщающего, что некоторая часть работы доступна. Когда новая часть работы добавляется в рабочую очередь, событие сигнализируется, чтобы прекратить ожидание одного из потоков. Такая реализация пула потоков неэффективна и не лишена проблем. Например, все становится сложнее, когда все потоки заняты, а элементы работы продолжают поступать в очередь, особенно если только одному потоку разрешено завершать одну единицу работы, прежде чем возвращаться в очередь ожидания. Если все потоки заняты, а тем временем, скажем, пять единиц работы находятся в очереди, событие будет сигнализировано, но ни один поток не будет в состоянии ожидания. Первый поток в очереди ожидания будет освобожден, когда вызовет WaitOne, но другие — нет, даже несмотря на то, что в очереди находятся еще четыре элемента работы. Одним из решений этой проблемы может быть запрет рабочим элементам становиться в очередь, когда все потоки заняты. Вообще-то, это не выход, поскольку при этом некоторая логика синхронизации возлагается на поток, пытающийся поместить работу в очередь, заставляя его как-то реагировать на неудачные попытки поместить единицу работы в очередь. На самом деле создание эффективного пула потоков — задача, как минимум, непростая. Поэтому я рекомендую использовать класс ThreadPool перед тем, как решиться на такой подвиг. Детальное описание класса ThreadPool я приведу в разделе "Использование ThreadPool".

Поскольку объекты событий .NET основаны на объектах событий Win32, вы можете применять их для синхронизации выполнения между несколькими процессами. Наряду с Mutex, они также менее эффективны, чем альтернатива, такая как класс Monitor, поскольку также требуют переключения в режим ядра. Однако создатели ManualResetEvent и AutoResetEvent не предусмотрели возможности именования объектов событий в конструкторах, как это сделано для объекта Mutex. Поэтому если вам нужно создать именованное событие, вы должны вместо этого использовать класс EventWaitHandle.

Объекты синхронизации Win32 и WaitHandle

В предыдущих двух разделах я рассказал об объектах Mutex, ManualResetEvent и AutoResetEvent. Каждый из этих типов наследуется от WaitHandle — общего механизма, который вы можете использовать в .NET Framework для управления любого рода объектами синхронизации Win32, которые вам понадобятся. Сюда входят не только события и мьютексы. Независимо от того, как вы получаете дескриптор объекта Win32, для управления им вы можете применять объект WaitHandle. Я предпочитаю слово *управлять* (manage) слову *инкапсулировать*, потому что класс WaitHandle не решает, как следует, задачу инкапсуляции, да он для этого и не предназначен. Он предназначен просто служить оболочкой, которая помогает избежать массы прямых вызовов Win32 через слой P/Invoke, работая с этими дескрипторами операционной системы.

На заметку! Стоит потратить некоторое время, чтобы разобраться, когда и как нужно использовать WaitHanle, потому что многие API-интерфейсы все еще не получили своего отображения в .NET Framework, а многие из них, возможно, никогда и не получат.

Я уже обсуждал применение метода WaitOne для ожидания, когда объект станет сигнальным. Однако класс WaitHandle имеет два статических метода, который вы можете использовать для организации ожидания на нескольких объектах. Первый из них — WaitHandle.WaitAny. Вы передаете ему массив объектов WaitHandle, и когда любой из этих объектов становится сигнальным, метод WaitAny возвращает целочисленный индекс такого объекта в массиве.

Второй метод — WaitHandle.WaitAll, который, как вы можете представить, не возвращает управления до тех пор, пока все объекты не станут сигнальными. Оба эти метода определяют перегрузки, принимающие значение таймаута. В случае вызова WaitAny при завершении ожидания по таймауту возвращается значение, эквивалентное константе WaitHandle.WaitTimeout. В случае вызова WaitAll возвращается булевское значение — true в случае, если все объекты стали сигнальными, и false — в случае выхода по таймауту.

До появления класса EventWaitHandle, чтобы получить именованное событие, вы должны были создать лежащий в основе объект Win32, а затем упаковать его в WaitHandle, как я сделал в следующем примере:

```
using System;
using System.Threading;
using System.Runtime.InteropServices;
using System.ComponentModel;
using Microsoft.Win32.SafeHandles;
public class NamedEventCreator
{
    [DllImport( "KERNEL32.DLL", EntryPoint="CreateEventW",
            SetLastError=true )]
    private static extern SafeWaitHandle CreateEvent(
                            IntPtr lpEventAttributes,
                            bool bManualReset,
                            bool bInitialState,
                            string lpName );
    public const int INVALID_HANDLE_VALUE = -1;
    public static AutoResetEvent CreateAutoResetEvent(
                                bool initialState,
                                string name ) {
        // Создать именованное событие.
        SafeWaitHandle rawEvent = CreateEvent( IntPtr.Zero,
                                        false,
                                        false,
                                        name );
        if( rawEvent.IsInvalid ) {
            throw new Win32Exception(
                        Marshal.GetLastWin32Error() );
        }
        // Создать событие управляемого типа на основе дескриптора.
        AutoResetEvent autoEvent = new AutoResetEvent( false );
```

```
            // Очистить десриптор, находящийся в autoEvent
            // перед тем, как заменить его именованным.
            autoEvent.SafeWaitHandle = rawEvent;
            return autoEvent;
        }
    }
```

Здесь я использовал слой P/Invoke для вызова Win32-функции `CreateEventW` для создания именованного события. В этом примере следует отметить несколько моментов. Например, я полностью сделал ставку на безопасность дескриптора, как это принято и в остальных классах стандартной библиотеки классов .NET Framework. Поэтому первый параметр `CreateEvent` — `IntPtr.Zero`, что является лучшим способом передачи указателя `NULL` на ошибку Win32. Обратите внимание, что вы определяете успех или неудачу создания события, проверяя свойство `IsInvalid` на `SafeWaitHandle`. Когда вы обнаруживаете это значение, то генерируете исключение типа `Win32Exception`. Затем вы создаете новый объект `AutoResetEvent`, чтобы упаковать только что созданный "сырой" дескриптор. `WaitHandle` предоставляет свойство по имени `SafeWaitHandle`, посредством которого вы можете модифицировать лежащий в основе дескриптор Win32 в любом типе-наследнике `WaitHandle`.

На заметку! Возможно, вы отметили в документации унаследованное свойство `Handle`. Этого свойства лучше избегать, поскольку повторное присваивание ему нового дескриптора ядра не закрывает предыдущий дескриптор, в результате чего происходит утечка ресурсов, если только вы не закроете его самостоятельно. Вместо него вы должны использовать типы-наследники `SafeHandle`. Тип `SafeHandle` также использует ограниченные области выполнения, чтобы предотвратить утечки ресурсов в случае асинхронных исключений вроде `ThreadAbortException`. Подробнее об ограниченных областях выполнения читайте в главе 7.

В предыдущем примере вы можете видеть, что я объявил метод `CreateEvent` как возвращающий `SafeWaitHandle`. Хотя это не очевидно из документации `SafeHandle`, этот класс имеет приватный конструктор по умолчанию, который слой P/Invoke может использовать для создания и инициализации экземпляра этого класса.

Ознакомьтесь с остальными производными от `SafeHandle` типами из пространства имен `Microsoft.Win32.SafeHandles`. В частности, .NET 2.0 Framework предлагает `SafeHandleMinusOneIsInvalid` и `SafeHandleZeroOrMinusOneIsInvalid` для удобства определения ваших собственных основанных на Win32 наследников `SafeWaitHandle`.

Имейте в виду, что тип `WaitHandle` реализует интерфейс `IDisposable`. В связи с этим вы можете разумно использовать ключевое слово `using` в вашем коде всякий раз, когда применяете экземпляры `WaitHandle` или экземпляры любых его классов-наследников, таких как `Mutex`, `AutoResetEvent` и `ManualResetEvent`.

Еще один момент, о котором нужно знать, применяя объекты `WaitHandle` и объекты-наследники этого типа — вы не можете завершить или прервать управляемые потоки во временной манере, когда они блокированы методами `WaitHandle`. Поскольку действительный поток операционной системы, который стоит за управляемым потоком, блокируется внутри операционной системы — т.е. вне управляемой исполняющей среды — он может быть прерван только тогда, когда вернется в управляемую среду.

Поэтому, если вы вызовете `Abort` или `Interrupt` на одном из этих потоков, операция будет отложена до тех пор, пока не завершится ожидание потока на уровне

операционной системы. Вам следует помнить об этом, когда вы устанавливаете блокировку с использованием объекта WaitHandle в управляемых потоках.

Использование ThreadPool

Пул потоков — идеальное средство в системе, где небольшие единицы работы выполняются регулярно в асинхронном режиме. Хорошим примером может служить Web-сервер или сервер любого иного рода, который прослушивает запросы на сетевом порте. Когда поступает запрос, выделяется новый поток, которому он передается на обработку. Сервер достигает высокой степени параллельности и оптимальной утилизации, обслуживая эти запросы в нескольких потоках. Обычно наиболее медленная операция на компьютере — это операция ввода-вывода. Устройства хранения, такие как жесткие диски, работают очень медленно по сравнению с процессором и его возможностями обращения к памяти. Поэтому для оптимизации использования вашей системы вы захотите начать выполнение других элементов работы, пока идет ожидание завершения операции ввода-вывода в другом потоке. Создание пула потоков для управления такой системой — сложнейшая задача, изобилующая массой деталей и ловушек. Однако среда .NET предлагает готовый к использованию пул потоков в лице класса ThreadPool.

Класс ThreadPool подобен классам Monitor и Interlocked в том смысле, что вы не можете создавать его экземпляры. Вместо этого вы используете статические методы класса ThreadPool для управления пулом потоков, который каждый процесс получает по умолчанию от CLR. Фактически вам даже не нужно заботиться о создании пула потоков. Он создается при первом обращении к нему. Если вы использовали пулы потоков в мире Win32 — будь то через системный пул потоков, который появился в Windows 2000, или через порты завершения ввода-вывода (IO completion ports) — вы заметите, что пул потоков .NET — тот же “зверь”, но с управляемым интерфейсом поверх него.

Чтобы поставить в очередь элемент в пул потоков, вы просто вызываете ThreadPool.QueueUserWorkItem, передавая ему экземпляр делегата WaitCallback. Пул потоков создается тогда, когда вы первый раз вызываете код этой функции. Метод обратного вызова, который вызывается через делегат WaitCallback, принимает ссылку на System.Object и имеет тип возврата void. Объектная ссылка — необязательный объект контекста, который вызывающий код может применить к перегрузке QueueUserWorkItem. Если вы не указываете контекст, то ссылка на контекст должна быть null. Как только элемент работы помещается в очередь, поток из пула осуществит обратный вызов, как только станет доступным. Когда элемент работы помещается в очередь, он уже никак не может быть удален оттуда, кроме как посредством потока, который завершит выполнение этой единицы работы. Поэтому, если вы хотите отменить выполнение элемента работы, нужно предусмотреть способ дать знать вашему обратному вызову, что он не должен ничего делать, когда будет вызван.

Пул потоков настроен на максимально эффективную обработку рабочих элементов. Он использует алгоритм, основанный на том, сколько процессоров доступно в системе, чтобы определить количество потоков, которые следует создать в пуле. Однако даже после этого пул может содержать больше потоков, чем изначально вычислено. Например, представим, что алгоритм решит, что пул потоков должен

содержать в себе четыре потока. Затем предположим, что сервер принимает четыре запроса, требующих обращения к базе данных заднего плана, на что потребуется определенное время. Если в течение этого времени поступит пятый запрос, свободных потоков, чтобы его обслужить, не окажется. Что еще хуже, так это то, что четыре занятых потока в это время просто ждут завершения операций ввода-вывода. Чтобы заставить систему работать с максимальной производительностью, пул потоков в действительности создаст еще один поток, когда обнаружит, что все прочие блокированы. После завершения всех единиц работы, когда система вернется в устойчивое состояние, пул потоков избавится от дополнительных потоков, созданных подобным образом, уничтожив их. Даже несмотря на то, что вы не можете легко контролировать количество потоков в пуле, можно контролировать минимальное число простаивающих потоков, ожидающих работы, через вызовы `GetMinThreads` и `SetMinThreads`.

Я советую вам прочитать подробности о статических методах `System.Threading.ThreadPool` в документации MSDN, если вы планируете напрямую работать с пулом потоков. В действительности редко возникает необходимость в прямой вставке рабочих элементов в пул потоков. Есть другая, более элегантная точка входа в пул потоков через делегаты и вызовы асинхронных процедур, о которой я расскажу в следующем разделе.

Асинхронные вызовы методов

Хотя вы можете управлять помещением элементов работы в пул потоков непосредственно через класс `ThreadPool`, более популярный способ использования пула потоков заключается в вызовах асинхронных делегатов. Когда вы объявляете делегат, CLR создает класс-наследник `System.MulticastDelegate`. Один из определенных в нем методов — `Invoke` — принимает точно ту же сигнатуру функции, что и определение делегата. Язык C#, конечно, предусматривает синтаксическое сокращение для вызова метода `Invoke`. Фактически, вы не можете явно вызвать метод `Invoke` в C#. Но наряду с `Invoke` CLR также определяет два метода — `BeginInvoke` и `EndInvoke`, являющиеся сердцем шаблона асинхронной обработки, которая используется CLR. Этот шаблон подобен шаблону IOU, который был представлен ранее в этой главе.

Базовая идея, вероятно, очевидна из имен методов. Когда вы вызываете `BeginInvoke` на делегате, операция откладывается для выполнения в другом потоке. Когда вы вызываете метод `EndInvoke`, результаты операции возвращаются вам. Если операция не завершена на момент вызова `EndInvoke`, вызывающий поток блокируется до тех пор, пока она не будет завершена. Рассмотрим краткий пример, демонстрирующий общий шаблон в действии. Предположим, что у вас есть метод, вычисляющий ваши налоги за год, и вы хотите вызывать его асинхронно, потому что для своего выполнения он требует ощутимого времени:

```
using System;
using System.Threading;
public class EntryPoint
{
    // Объявление делегата для асинхронного вызова.
    private delegate Decimal ComputeTaxesDelegate( int year );
```

```
    // Метод для вычисления налогов.
    private static Decimal ComputeTaxes( int year ) {
        Console.WriteLine( "Вычисление налогов в потоке {0}",
                           Thread.CurrentThread.GetHashCode() );

        // Здесь происходит длительное вычисление.
        Thread.Sleep( 6000 );

        return 4356.98M;
    }
    static void Main() {
        // Выполним асинхронный вызов, создав
        // делегат и вызвав его.
        ComputeTaxesDelegate work =
            new ComputeTaxesDelegate( EntryPoint.ComputeTaxes );
        IAsyncResult pendingOp = work.BeginInvoke( 2004,
                                                   null,
                                                   null );
        // Выполнить другую полезную работу.
        Thread.Sleep( 3000 );

        // Завершить асинхронный вызов.
        Console.WriteLine( "Ожидание завершения операции." );
        Decimal result = work.EndInvoke( pendingOp );
        Console.WriteLine( "Сумма налогов: {0}", result );
    }
}
```

Первое, что вы заметите в этом шаблоне — это то, что сигнатура метода `BeginInvoke` не соответствует методу `Invoke`. Это объясняется тем, что вам нужен некоторый способ идентификации определенного элемента работы, который вы только что отложили вызовом `BeginInvoke`. Таким образом, `BeginInvoke` возвращает ссылку на объект, реализующий интерфейс `IAsyncResult`. Этот объект подобен cookie-набору, который вы сохраняете, чтобы идентифицировать выполняющийся элемент работы. Через методы интерфейса `IAsyncResult` вы можете проверять состояние операции, такое как ее готовность. Чуть дальше мы обсудим этот интерфейс подробно, наряду с двумя параметрами, добавленными в конец объявления метода `BeginInvoke`, и вместо которых я подставил `null`. Когда поток, запрошенный для выполнения операции, завершит свою работу, он вызывает `EndInvoke` на делегате. Однако, поскольку метод должен иметь способ идентификации асинхронной операции, результат которой нужно получить, вы должны передать ему объект, который получили от метода `BeginInvoke`. В данном примере вы заметите, что вызов `EndInvoke` будет блокирован на некоторое время, до окончания операции.

На заметку! Если в процессе асинхронного выполнения в пуле потоков целевого кода делегата будет сгенерировано исключение, оно будет сгенерировано повторно, когда инициирующий поток вызовет `EndInvoke`.

Отчасти красота асинхронного шаблона IOU, реализованного делегатом, заключается в том, что вызванный код даже может не знать о том, что он вызван асинхронно. Конечно, это не слишком практично — вызывать метод асинхронно, если он для этого не был предназначен, если он затрагивает данные системы, к которым обращаются другие методы, либо если не используются никакие механизмы синхронизации. Тем не менее, вся головная боль, связанная с созданием инфраструктуры асинхронного вызова метода, смягчается благодаря делегату, сгенерированному CLR, наряду с пулом потоков, связанным с процессом. Более того, инициатор асинхронного действия даже не обязан знать, как реализовано асинхронное поведение.

Теперь давайте присмотримся внимательней к интерфейсу IAsyncResult объекта, возвращенного методом BeginInvoke. Объявление интерфейса выглядит следующим образом:

```
public interface IAsyncResult
{
   Object AsyncState { get; }
   WaitHandle AsyncWaitHandle { get; }
   bool CompletedSynchronously { get; }
   bool IsCompleted { get; }
}
```

В предыдущем примере я решил ждать окончания вычислений, вызвав EndInvoke. Вместо этого я мог бы ожидать WaitHandle, возвращенного свойством IasyncResult.AsyncWaitHandle, перед тем, как вызвать EndInvoke. В любом случае конечный результат был бы тем же. Однако тот факт, что интерфейс IAsyncResult предоставляет WaitHandle, позволяет вам при необходимости иметь несколько потоков в системе, ожидающих завершения одного действия.

Два других свойства позволяют проверять, завершена ли операция. Свойство IsCompleted просто возвращает булевское значение, представляющее этот факт. Вы могли бы сконструировать цикл по пулу, периодически проверяющий значение этого флага. Однако это было бы намного менее эффективно, чем просто ожидание WaitHandle. Но, тем не менее, в вашем распоряжении есть и такой вариант. Еще одно булевское свойство — это CompletedSynchronously. Шаблон асинхронной обработки в .NET Framework предусматривает возможность вызова BeginInvoke для запуска работы в синхронном, а не асинхронном режиме. Свойство CompletedSynchronously позволяет определить, если это произойдет. В текущей реализации CLR никогда этого не делает, когда делегаты вызываются асинхронно, но это может измениться в любое время. Однако, поскольку рекомендуется применять тот же асинхронный шаблон всякий раз, когда вы проектируете тип, который может быть вызван асинхронно, такая возможность была встроена в шаблон. Например, предположим, что у вас есть класс, в котором поддерживается метод для синхронного выполнения обобщенных операций. Если одна из таких операций просто вернет номер версии класса, то вы точно знаете, что она выполняется быстро, и можете решить выполнить ее синхронно.

И, наконец, свойство AsyncState интерфейса IAsyncResult позволяет прикрепить любой тип специфичных контекстных данных к асинхронному вызову. Это — второй из двух дополнительных параметров, добавленных в конец сигнатуры BeginInvoke. В моем предыдущем примере я передал null, потому что не ну-

ждался в нем. Хотя я решил получить результат операции через вызов EndInvoke, с тем же успехом я мог бы предпочесть извещение о завершении операции через обратный вызов. Рассмотрим следующую модификацию предыдущего примера:

```
using System;
using System.Threading;
public class EntryPoint
{
    // Объявление делегата для асинхронного вызова.
    private delegate Decimal ComputeTaxesDelegate( int year );
    // Метод для вычисления налогов.
    private static Decimal ComputeTaxes( int year ) {
        Console.WriteLine( "Вычисление налогов в потоке {0}",
                           Thread.CurrentThread.GetHashCode() );

        // Здесь происходит длительное вычисление.
        Thread.Sleep( 6000 );

        return 4356.98M;
    }
    private static void TaxesComputed( IAsyncResult ar ) {
        // Теперь получим результат.
        ComputeTaxesDelegate work =
            (ComputeTaxesDelegate) ar.AsyncState;
        Decimal result = work.EndInvoke( ar );
        Console.WriteLine( "Сумма налогов: {0}", result );
    }
    static void Main() {
        // Выполним асинхронный вызов, создав
        // делегат и вызвав его.
        ComputeTaxesDelegate work =
            new ComputeTaxesDelegate( EntryPoint.ComputeTaxes );
        work.BeginInvoke( 2004,
                          new AsyncCallback(EntryPoint.TaxesComputed),
                          work );
        // Выполнить другую полезную работу.
        Thread.Sleep( 3000 );

        // Завершить асинхронный вызов.
        Console.WriteLine( "Ожидание завершения операции." );
        Thread.Sleep( 4000 );
    }
}
```

Теперь вместо вызова EndInvoke из потока, который вызвал BeginInvoke, я прошу, чтобы пул потоков вызвал метод TaxesComputed через экземпляр делегата AsyncCallback, который я передал в предпоследнем параметре BeginInvoke. Использование обратного вызова для обработки результата довершает шаблон асинхронной обработки, позволяя потоку, запустившему операцию, продолжать работу, без необходимости явного ожидания рабочего потока. Обратите внимание, что метод обратного вызова TaxesComputed все равно должен вызвать EndInvoke,

чтобы получить результат асинхронного вызова. Чтобы сделать это, однако, ему нужен экземпляр делегата. И здесь на помощь приходит объект контекста IasyncResult.AsyncState. В моем примере я инициализирую его так, чтобы он указывал на делегат, передав этот делегат в последнем параметре BeginInvoke. Главный поток, вызывающий BeginInvoke, не нуждается в объекте, возвращенном этим вызовом, поскольку он никогда в действительности не опрашивает состояния операции, как и не ожидает явно ее завершения. Метод sleep добавлен в конец метода Main просто для примера. Помните, что все потоки в пуле выполняются как фоновые. Поэтому если вы не станете ждать в этой точке, то процесс завершится задолго до завершения операций. Если вы хотите, чтобы асинхронная работа проходила в потоке переднего плана, лучше создать новый класс, реализующий асинхронный шаблон BeginInvoke/EndInvoke и использовать поток переднего плана для выполнения работы. Никогда не меняйте фоновый статус потока в пуле через свойство IsBackground в текущем потоке. Даже если вы попробуете это сделать, то обнаружите, что это не даст никакого эффекта.

На заметку! Важно понимать, что когда выполняется ваш асинхронный код и когда осуществляется обратный вызов, вы работаете в контексте произвольного потока. Вы не можете делать никаких предположений относительно того, какой поток выполнит ваш код. Во многих отношениях эта техника подобна разработке драйверов на платформе Windows.

Применение обратных вызовов для обработки завершения элемента работы очень удобно при создании серверного процесса, обрабатывающего входящие запросы. Например, предположим, что у вас есть процесс, прослушивающий определенный порт TCP/IP на предмет входящих запросов. Когда он получает такой запрос, то отвечает на него, пересылая запрошенную информацию. Чтобы достичь максимальной эффективности, вы определенно захотите выполнять эти операции асинхронно. Рассмотрим следующий пример, который слушает порт 1234 и, получив что-либо, просто отвечает строкой "Hello World!":

```
using System;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;
public class EntryPoint {
    private const int CONNECT_QUEUE_LENGTH = 4;
    private const int LISTEN_PORT = 1234;
    static void ListenForRequests() {
        Socket listenSock =
        new Socket( AddressFamily.InterNetwork,
                    SocketType.Stream,
                    ProtocolType.Tcp );
        listenSock.Bind( new IPEndPoint(IPAddress.Any,
                                        LISTEN_PORT) );
        listenSock.Listen( CONNECT_QUEUE_LENGTH );
        while( true ) {
            using( Socket newConnection = listenSock.Accept() ) {
                // Послать данные.
                byte[] msg =
```

```
                Encoding.UTF8.GetBytes( "Hello World!" );
            newConnection.Send( msg, SocketFlags.None );
        }
    }
}
static void Main() {
    // Запустить поток-слушатель.
    Thread listener = new Thread(
                        new ThreadStart(
                            EntryPoint.ListenForRequests) );
    listener.IsBackground = true;
    listener.Start();
    Console.WriteLine( "Нажмите <enter> для завершения" );
    Console.ReadLine();
}
}
```

Этот пример создает дополнительный поток, который просто в цикле слушает входящие соединения и обслуживает их, как только они поступают. У этого подхода много проблем. Во-первых, только один поток обрабатывает входящие соединения. Если соединения начнут поступать очень часто, он быстро будет перегружен. Подумайте о Web-сервере, который легко может получать тысячи запросов в секунду. Класс `Socket` реализует шаблон асинхронных вызовов .NET Framework. Используя этот шаблон, вы можете улучшить сервер, обслуживая входящие запросы с применением пула потоков, как показано ниже:

```
using System;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;
public class EntryPoint {
    private const int CONNECT_QUEUE_LENGTH = 4;
    private const int LISTEN_PORT = 1234;
    static void ListenForRequests() {
        Socket listenSock =
        new Socket( AddressFamily.InterNetwork,
                    SocketType.Stream,
                    ProtocolType.Tcp );
        listenSock.Bind( new IPEndPoint(IPAddress.Any,
                                        LISTEN_PORT) );
        listenSock.Listen( CONNECT_QUEUE_LENGTH );
        while( true ) {
            Socket newConnection = listenSock.Accept();
            byte[] msg = Encoding.UTF8.GetBytes( "Hello World!" );
            newConnection.BeginSend( msg,
                                     0, msg.Length,
                                     SocketFlags.None,
                                     null, null );

        }
    }
```

```
    static void Main() {
        // Запустить поток-слушатель.
        Thread listener = new Thread(
                            new ThreadStart(
                                EntryPoint.ListenForRequests) );
        listener.IsBackground = true;
        listener.Start();
        Console.WriteLine( "Нажмите <enter> для завершения" );
        Console.ReadLine();
    }
}
```

Сервер стал немного эффективнее, поскольку теперь отправка данных для входящих соединений происходит в потоке, взятом из пула. Этот код также демонстрирует стратегию "сделал и забыл" при использовании асинхронного шаблона. Вызывающий код не заинтересован в возврате объекта, реализующего `IAsyncResult`, как и не заинтересован в установке метода обратного вызова, который должен быть вызван при завершении работы. Этот вызов в стиле "сделал и забыл" — отважная попытка повысить эффективность сервера. Однако результат неудовлетворителен, поскольку здесь пропал оператор `using` из предыдущей версии сервера. `Socket` не закрывается вовремя, и удаленные соединения остаются открытыми до тех пор, пока GC не решит финализировать объекты `Socket`. Поэтому асинхронный вызов должен включать обратный вызов, чтобы закрыть соединение. Было бы бессмысленно для потока-слушателя ожидать метода `EndSend`, поскольку это возвращает нас в то же корыто неэффективности, где мы уже были ранее.

На заметку! Когда вы получаете объект, реализующий `IAsyncResult` от запуска асинхронной операции, этот объект должен реализовывать свойство `IAsyncResult.AsyncWaitHandle`, чтобы позволить пользователям получать дескриптор, на котором потом можно организовать ожидание. В случае `Socket` возвращается экземпляр `OverlappedAsyncResult`. Этот класс в конечном итоге наследуется от `System.Net.LazyAsyncResult`. Он в действительности не создает события для ожидания до тех пор, пока кто-нибудь не обратится к нему через свойство `IAsyncResult.AsyncWaitHandle`. Такое "ленивое" создание избавляет от ненужного создания объекта блокировки, который большую часть времени остается неиспользованным. К тому же на объект `OverlappedAsyncResult` возлагается обязанность закрывать дескриптор операционной системы по окончании работы с ним.

Однако перед тем как перейти к обратному вызову, присмотримся к потоку слушателя. Все, что он делает — это ожидание входящих запросов. А не будет ли более эффективно организовать прослушивание тоже через потоки из пула? Разумеется! Поэтому теперь позвольте представить новый и усовершенствованный сервер "Hello World!", который в полной мере использует пул потоков процесса:

```
using System;
using System.Text;
using System.Threading;
using System.Net;
using System.Net.Sockets;
public class EntryPoint {
    private const int CONNECT_QUEUE_LENGTH = 4;
    private const int LISTEN_PORT = 1234;
```

```
    private const int MAX_CONNECTION_HANDLERS = 4;
    private static void HandleConnection( IAsyncResult ar ) {
        Socket listener = (Socket) ar.AsyncState;
        Socket newConnection = listener.EndAccept( ar );
        byte[] msg = Encoding.UTF8.GetBytes( "Hello World!" );
        newConnection.BeginSend( msg,
                                 0, msg.Length,
                                 SocketFlags.None,
                                 new AsyncCallback(
                                   EntryPoint.CloseConnection),
                                 newConnection );
        // Поместить другой запрос в очередь.
        listener.BeginAccept(
            new AsyncCallback(EntryPoint.HandleConnection),
            listener );
    }

    static void CloseConnection( IAsyncResult ar ) {
        Socket theSocket = (Socket) ar.AsyncState;
       theSocket.Close();
    }

    static void Main() {
        Socket listenSock =
        new Socket( AddressFamily.InterNetwork,
                    SocketType.Stream,
                    ProtocolType.Tcp );
        listenSock.Bind( new IPEndPoint(IPAddress.Any,
                                        LISTEN_PORT) );
        listenSock.Listen( CONNECT_QUEUE_LENGTH );

        // Ожидать дескрипторов соединений.
        for( int i = 0; i < MAX_CONNECTION_HANDLERS; ++i ) {
            listenSock.BeginAccept(
                new AsyncCallback(EntryPoint.HandleConnection),
                listenSock );
        }

        Console.WriteLine( "Нажмите <enter> для завершения" );
        Console.ReadLine();
    }
}
```

Теперь сервер "Hello World!" полностью использует пул потоков процесса и может обрабатывать входящие клиентские запросы с максимальной степенью параллельности. Кстати, протестировать соединение весьма просто, если использовать встроенный в Windows клиент Telnet. Просто запустите Telnet из командной строки или из диалогового окна запуска программ, и введите команду для подключения к порту 1234 локальной машины при запущенном в другом командном окне процессе сервера:

```
Microsoft Telnet> open 127.0.0.1 1234
```

Таймеры

Еще одна точка входа в пул потоков находится в объектах класса Timer из пространства имен System.Threading. Как следует из наименования, вы можете настроить пул потоков на вызов делегата в определенное время или через регулярные интервалы. Рассмотрим пример использования объекта Timer:

```
using System;
using System.Threading;
public class EntryPoint
{
   private static void TimerProc( object state ) {
      Console.WriteLine( "Текущее время {0} на потоке {1}",
                         DateTime.Now,
                         Thread.CurrentThread.GetHashCode() );
      Thread.Sleep( 3000 );
   }
   static void Main() {
      Console.WriteLine( "Нажмите <enter> по завершении\n\n" );
      Timer myTimer =
         new Timer( new TimerCallback(EntryPoint.TimerProc),
                    null,
                    0,
                    2000 );

      Console.ReadLine();
      myTimer.Dispose();
   }
}
```

Когда создается таймер, вы должны предоставить ему делегат, который должен быть вызван в заданное время. Поэтому я создал делегат TimerCallback, указывающий на статический метод TimerProc. Второй параметр конструктора Timer — произвольный объект состояния, который вы можете передать ему. Когда происходит обратный вызов вашего таймера, этот объект состояния передается ему. В моем примере мне не нужен объект состояния, поэтому я передаю null. Последние два параметра конструктора определяют, когда должен произойти обратный вызов. Предпоследний параметр указывает, когда таймер должен сработать первый раз. В моем примере я передаю 0 — это говорит о том, что он должен быть запущен немедленно. Последний параметр — период, через который должен произойти следующий вызов. Я запросил двухсекундный период. Если вы не хотите, чтобы таймер запускался периодически, передайте в последнем параметре Timeout.Infinite. И, наконец, чтобы остановить таймер, просто вызовите его метод Dispose.

Вы можете удивиться, почему я делаю вызов Sleep внутри метода TimerProc. Это просто в целях иллюстрации, для того, чтобы показать, что TimerProc вызывает произвольный поток. Поэтому любой код, который выполняется в результате запуска вашего делегата TimerCallback, должен быть безопасным в отношении потоков. В моем примере первый поток из пула, вызывающий TimerProc, спит дольше, чем длительность следующего таймаута, поэтому пул потоков вызывает

TimerProc двумя секундами позже из другого потока, что видно из сгенерированного вывода. Вы можете действительно вызвать некоторое напряжение в пуле потоков, если увеличите период сна в TimerProc.

На заметку! Если вы когда-либо использовали класс Timer из пространства имен System.Windows.Forms, то должны понимать, что это — совсем другой "зверь", чем класс Timer из пространства System.Threading. Во-первых, Forms.Timer основан на системе сообщений Win32 Windows, а именно — на сообщении WM_TIMER. Одно из удобств Form.Timer заключается в том, что обратный вызов этого таймера всегда происходит в том же потоке. Однако обязательным условием работы такого таймера является то, чтобы поток графического интерфейса, частью которого является этот таймер, имел лежащее в его основе средство подкачки сообщений графического интерфейса. Если средство подкачки останавливается, то же происходит и с обратными вызовами Form.Timer. Поэтому, естественно, Threading.Timer — более мощный инструмент в том смысле, что он не страдает такой зависимостью. Однако его недостатком можно считать то, что вы должны кодировать обратные вызовы для Threading.Timer в безопасной в отношении потоков манере.

Резюме

В этой главе я раскрыл тонкости управляемых потоков в среде .NET. Я рассказал о различных механизмах обеспечения управляемой синхронизации между потоками, включая Interlocked, Monitor, AutoResetEvent, ManualResetEvent, объекты на базе WaitHandle и т.п. Затем я описал шаблон IOU и продемонстрировал, как .NET Framework интенсивно использует его для выполнения асинхронной работы. Эта дискуссия была сосредоточена вокруг использования CLR класса ThreadPool, основанного на реализации пула потоков Windows.

Многопоточность всегда усложняет приложения. Однако при правильном использовании она может сделать приложения более отзывчивыми на команды пользователя и более эффективными. Хотя многопоточная разработка не лишена некоторых ловушек, .NET Framework и CLR максимально смягчают все риски и предоставляют модель, которая в большинстве случаев защищает вас от сложностей операционной системы. Например, реализация пулов потоков всегда была трудной задачей, даже после того, как общая реализация была добавлена в операционную систему Windows. Среда .NET не только предлагает замечательный буфер между вашим кодом и сложностями пула потоков Windows, но также позволяет запускать ваш код на других платформах, реализующих .NET Framework, таких как исполняющая система Mono, работающая в среде Linux. Если вы понимаете детали средств многопоточности, предоставленных исполняющей системой .NET, и знакомы с приемами многопоточной синхронизации, описанными в этой главе, то вы на верном пути к производству эффективных многопоточных приложений.

В следующей главе я обращусь к исследованию канонической формы типов C#. Будет составлен перечень вопросов, которые вы, как разработчик, должны задать себе при проектировании типа с использованием C# для .NET Framework.

ГЛАВА 13

В поисках канонических форм C#

Многие объектно-ориентированные языки, включая C#, не предусматривают никакого принуждения в адрес разработчиков, чтобы те создавали хорошо спроектированное программное обеспечение. Лучший пример тому — применение C++ для реализации объектно-ориентированного дизайна. C# немного более структурирован, чем C++; например, вы не можете создавать свободных статических функций, находящихся вне контекста определенного типа. Тем не менее, C# не принуждает вас создавать программное обеспечение, следующее широко известным принципам хорошего дизайна программ.

Сообщество C++ быстро идентифицировало некоторые канонические формы, полезные для проектирования типов, предназначенных для определенной цели. В действительности эти канонические формы представляют собой просто списки или рецепты, которые вы можете использовать при проектировании новых классов. Прежде чем вывести самолет на взлетную полосу, пилот обязан пройти строгую процедуру, ответив на ряд вопросов. Цель настоящей главы — идентифицировать списки таких вопросов, позволяющие создавать устойчивые типы в мире C#.

Открывая вопросники подобного рода, вы должны учитывать, какого рода поведение требуется от объектов создаваемого вами нового типа. Например, должен ли ваш новый тип быть клонируемым? Если экземпляры нового типа помещаются в коллекцию, могут ли они упорядочиваться? Что означает сравнение на эквивалентность двух ссылок на объекты этого типа? Другими словами, хотите ли вы знать о том, что две ссылки указывают на один и тот же экземпляр? Или же вы хотите знать, что два экземпляра находятся точно в одном и том же состоянии? Такого рода вопросы вы должны задать себе при создании нового типа.

На заметку! Эта глава довольно длинная, но я решил, что важно собрать всю эту полезную и взаимосвязанную информацию в одном месте. Вообще говоря, глава разбита на два больших раздела. В первом описаны ссылочные типы, а во втором — типы значений. Я описал ссылочные типы сначала, выделив им больше места, поскольку некоторые материалы касаются как ссылочных, так и типов значений. И, наконец, глава завершается парой списков вопросов, на которые нужно ответить при проектировании новых типов.

Канонические формы ссылочных типов

Для начала давайте раскроем канонические формы ссылочных типов C#. В языке C# объекты находятся в управляемой куче, и доступ к ним осуществляется через типы значений, хранящие ссылки на них.

На заметку! В терминах C++ вы можете представить себе подобную систему, где все объекты создаются динамически с использованием `new`, и вы обращаетесь к ним только через указатели, возвращаемые `new`. Именно это происходит в CLR, но с тем отличием, что CLR отслеживает все эти "указатели", или ссылки, и знает, когда объекты в куче больше не имеют ссылок на них, а, следовательно — когда они могут быть уничтожены.

Чтобы быть немного точнее, рассмотрим вот что: с годами сообщество C++ накопило богатый массив идиом, полагающихся на стек и помогающих управлять ресурсами. Если вы создаете объект C++ в стеке, то компилятор гарантирует, что конструкторы и деструкторы вашего объекта будут вызваны в надлежащее время, тем самым предоставляя контролируемую точку, куда вы можете поместить ваш код очистки ресурсов. Доминирующая здесь идиома называется RAII (Acquisition Is Initialization — захват ресурсов является инициализацией), и она интенсивно используется в C++ и в любом другом объектно-ориентированном языке с детерминированной деструкцией объектов. По сути, идея, лежащая в основе этой идиомы, заключается в том, что любые ресурсы, которые требуют распределения, захватываются в теле конструктора, а освобождение этих ресурсов производится в соответствующем деструкторе. Эта идиома настолько укоренилась, что для того, чтобы писать устойчивый, безопасный и нейтральный к исключениям код, вы должны интенсивно пользоваться ею, включая все сопровождающие операции `new` и `delete` в тела конструкторов и деструкторов. В домене C# эта идиома не так легко доступна, потому что деструкторы C# не являются детерминированными. Поэтому вы должны подходить к проблемам, решаемым этой идиомой, как-то иначе, о чем и пойдет речь в настоящем разделе.

Классы должны помечаться как `sealed` по умолчанию

Я твердо уверен в том, что при создании новых классов вы должны автоматически помечать их как `sealed` (герметичный) и убирать это ключевое слово только в том случае, если у вас есть веская причина полагать, что кому-то понадобится наследоваться от вашего класса. Почему не поступить наоборот, оставляя классы не герметичными по умолчанию и запечатывать их только в том случае, когда нужно запретить наследование? Потому, что невозможно предсказать, каким образом кто-то попытается наследоваться от вашего класса, если вы не предприняли специальных усилий для поддержки наследования. За годы я встречал немало проектов, где кто-либо пытался наследоваться от класса, который никогда не предназначался для того, чтобы служить базовым. Например, при хорошем дизайне классы, лишенные виртуальных методов, обычно не предназначены для наследования. Отсутствие виртуальных методов, скорее всего, указывает на то, что автор класса не думал о том, что кто-то будет наследоваться от него, а потому должен был пометить класс словом `sealed`. Если ваш класс не является `sealed`, и вы собираетесь разрешить другим наследоваться от него, не забудьте снабдить его адекватной документацией, чтобы человек, наследующий свой класс от вашего, не ломал себе голову.

На заметку! Создать самодокументированный базовый класс вообще нелегко, а при наличии в классе хотя бы одного переопределяемого виртуального метода обойтись без документации почти невозможно. Почему — читайте далее.

Даже классы, имеющие виртуальные методы и предназначенные для того, чтобы служить базой для дальнейшего наследования, могут оказаться проблематичными. Например, если вы наследуетесь от класса, предоставляющего виртуальный метод `DoSomething`, и хотите расширить этот метод посредством его переопределения, должны ли вы вызывать версию базового класса в вашем переопределении? Если да, вызовите вы ее до или же после выполнения вашей производной работы? Имеет ли значение порядок? Может быть да, если в базовом классе объявлены защищенные поля[1]. Если у вас нет по-настоящему хорошей документации базового класса, то вы, возможно, никогда не узнаете ответов на эти вопросы. Фактически, это одна причина, почему расширение посредством включения обычно более гибко, а потому и более мощно по сравнению с расширением через наследование. Расширение включением является динамическим, и реализуется во время выполнения, в то время как расширение на основе наследования более ограничено, поскольку статично и осуществляется во время компиляции. И что еще важнее — вы можете выполнять расширение включением, даже если класс, который нужно расширить, помечен как `sealed`.

Если только у вас нет веских причин сделать ваш класс базовым, всегда помечайте его `sealed`. В противном случае будьте готовы предоставить очень подробную документацию о том, как лучше всего организовать наследование от вашего класса. Я гарантирую, что вы всегда можете разработать другой дизайн, применив наследование интерфейсов вместе с включением вместо наследования реализации (класса), который выполнит ту же работу. В силу вышесказанного, почти нет причин, почему бы все классы, которые вы проектируете, не объявлять как `sealed`. Не поймите меня превратно: я не говорю, что наследование — это всегда плохо. Наоборот, оно полезно, когда применяется правильно. К сожалению, часто оно применяется как раз неправильно. Глубокая иерархия, в противоположность мелкой, плоской — верный признак того, что ваш дизайн нужно пересмотреть.

На заметку! Когда листовые (терминальные) классы, унаследованные от других классов с виртуальными методами, помечены как `sealed`, или когда индивидуальные переопределенные методы помечены как `sealed`, то исполняющая система может превратить их вызовы в невиртуальные, поскольку никакие производные реализации этих методов не могут существовать. Естественно, это дает выигрыш в производительности.

Использование шаблона NVI

Очень часто, когда вы проектируете класс, специально предназначенный служить базовым в иерархии, вы объявляете виртуальные методы так, чтобы производные классы могли модифицировать поведение. Первое приближение такого базового класса может выглядеть примерно так:

[1] В главе 4 обсуждается инкапсуляция и ее важность в объектно-ориентированном дизайне. Важно отметить, что защищенные поля нарушают инкапсуляцию.

```
using System;
public class Base
{
   public virtual void DoWork() {
      Console.WriteLine( "Base.DoWork()" );
   }
}
public class Derived : Base
{
   public override void DoWork() {
      Console.WriteLine( "Derived.DoWork()" );
   }
}
public class EntryPoint
{
   static void Main() {
      Base b = new Derived();
      b.DoWork();
   }
}
```

Не удивительно, что вывод этого примера выглядит так:

```
Derived.DoWork()
```

Однако дизайн мог бы быть немного более устойчивым. Предположим, что вы — разработчик `Base`, и создали этот класс для миллионов пользователей. Многие люди по всему миру с удовольствие используют ваш класс `Base`, когда вы решаете по какой-то веской причине, что в `DoWork` нужно выполнять какую-то пред- и пост-обработку. Например, вы хотите предоставить отладочную версию `Base`, которая отслеживает количество вызовов метода `DoWork`. Если код останется таким, как показано выше, вы не сможете этого сделать, не внося разрушительных для миллионов пользователей вашего класса `Base` изменений. Например, вы можете добавить два новых метода — `PreDoWork` и `PostDoWork` — и вежливо попросить всех ваших пользователей заново реализовать их переопределения, чтобы они вызывали эти методы в надлежащее время. Ого! Теперь представим минимальную модификацию исходного дизайна, который даже не изменяет общедоступного интерфейса `Base`:

```
using System;
public class Base
{
   public void DoWork() {
      CoreDoWork();
   }
   protected virtual void CoreDoWork() {
      Console.WriteLine( "Base.DoWork()" );
   }
}
public class Derived : Base
{
   protected override void CoreDoWork() {
      Console.WriteLine( "Derived.DoWork()" );
   }
}
```

```
public class EntryPoint
{
    static void Main() {
        Base b = new Derived();
        b.DoWork();
    }
}
```

Этот маленький изящный шаблон называется NVI (Non-Virtual Interface — не виртуальный интерфейс), и он делает буквально следующее: объявляет общедоступный интерфейс базового класса не виртуальным, но переопределяемое поведение перемещает в другой, защищенный, метод под названием `CoreDoWork`. Шаблон NVI подобен шаблону Template Method (Шаблонный метод), описанному Эрихом Гаммой (Erich Gamma), Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидесом (John Vlissides) в книге *Design Patterns: Elements of Reusable Object-Oriented Software* (Boston, MA: Addison-Wesley Professional, 1995 г.). Библиотеки .NET Framework широко применяют шаблон NVI, и по совершенно очевидным причинам он кочует из одного в другое руководство по проектированию библиотек Microsoft. Чтобы добавить некоторого рода измеряющий код в метод `DoWork`, вам нужно всего лишь модифицировать `Base` и сборку, содержащую его. Любой из классов, наследующих сборку, не потребует изменений.

Другой прием, часто используемый с NVI в мире C++, заключается в том, что виртуальный метод объявляется приватным (`private`), как в следующем коде, который, к сожалению, не компилируется в C# по причинам, изложенным ниже:

```
// НЕ КОМПИЛИРУЕТСЯ!!!!!
using System;
public class Base
{
    public void DoWork() {
        CoreDoWork();
    }
    // НЕ КОМПИЛИРУЕТСЯ!!!!!
    private virtual void CoreDoWork() {
        Console.WriteLine( "Base.DoWork()" );
    }
}
public class Derived : Base
{
    // НЕ КОМПИЛИРУЕТСЯ!!!!!
    private override void CoreDoWork() {
        Console.WriteLine( "Derived.DoWork()" );
    }
}
public class EntryPoint
{
    static void Main() {
        Base b = new Derived();
        b.DoWork();
    }
}
```

Этот код компилировался бы в начальной версии C# из .NET 1.0, и такая техника вполне корректна для CLR, отражая факт, что в то время спецификации CLI, как и C#, стремились как можно ближе соответствовать семантике C++. Прежде чем я объясню, почему это не работает в C# теперь, давайте сначала рассмотрим, зачем это могло бы понадобиться.

Существует фундаментальное отличие между видимостью и доступностью методов. Если метод находится в объявлении класса или структуры, независимо от уровня защиты — такой метод видим. И традиционно для того, чтобы производный класс мог переопределять метод, он всего лишь должен был быть видимым, а не доступным.

На заметку! Единственный смысл слова `private` в методе `private virtual` C++ заключается в том, что производный класс не должен вызывать реализацию базового класса. Если вы не верите мне, попробуйте выполнить предложенный пример в "родном" C++. Вы обнаружите, что он работает, как ожидается.

Вся прелесть возможности объявления методов `private virtual` состоит в том, что вам не нужно беспокоиться о производных классах, которые неправильно используют ваш класс `Base`. Например, может быть, вам нужно, чтобы они не вызывали реализации виртуального метода вашего базового класса. Отлично — просто объявите его `private virtual`. Фактически, используя этот прием, вы должны сделать ваш метод `protected virtual`, если вы знаете, что есть хорошая причина, почему базовому классу может понадобиться вызвать его. И если вы сделаете это, то должны строго документировать, в какой точке переопределение должно вызывать базовую реализацию. Многие верят, что лишь потому, что метод объявлен `private`, он не может быть переопределен. Но, строго говоря, ограниченная доступность не делает его невидимым или переопределенным.

А теперь позвольте мне объяснить, почему это средство было отключено в версии C# из .NET 1.1. Это было загадкой для меня до тех пор, пока Брендон Брэй (Brandon Bray) из команды разработчиков Microsoft Visual C++ не объяснил все четко. Тот факт, что вы можете наследовать через границы доступности, превращал это средство в определенного рода брешь в безопасности. В "родном" C++ это никогда не было проблемой. Рассмотрим вот что: если метод `private virtual` мог бы быть переопределен, то же стало бы возможным и для метода `internal virtual`. И здесь возникает проблема. Это позволило бы вам переопределить поведение метода `internal virtual` в некотором произвольном классе в определенной сборке, что создает брешь в безопасности. Поэтому было принято компромиссное решение, и в каждой версии, начиная с 1.1, это средство было отключено. Кстати, то же исправление было внесено и в C++/CLI. Хотя "родные" классы C++ могут эффективно использовать средство методов `private virtual`, `ref`-классы C++ — нет. И это, конечно, потому, что `ref`-классы C++ представляют типы .NET `ref`, которые не могут быть унаследованы через границы сборок. Каково!

Является ли `Object` клонируемым?

Как вы знаете, объекты в C# и CLR находятся в куче и доступны через ссылки. Присваивая одну объектную переменную другой, вы на самом деле не создаете копии:

```
Object obj = new Object();
Object objCopy = obj;
```

После выполнения этого кода objCopy не ссылается на копию obj; вместо этого вы получаете две ссылки на один и тот же экземпляр Object.

Однако иногда имеет смысл создать копию объекта. Для этой цели в стандартной библиотеке предусмотрен интерфейс ICloneable. Если ваш объект реализует этот интерфейс, то можно говорить, что он поддерживает возможность создания копий самого себя. Другими словами, от него требуется, чтобы он мог служить прототипом для создания новых экземпляров объектов. Объекты подобного рода могут участвовать в шаблоне фабрики прототипов. Прежде чем двигаться дальше, взглянем, как выглядит интерфейс ICloneable:

```
public interface ICloneable
{
    object Clone();
}
```

Как видите, интерфейс объявляет только один метод — Clone, который возвращает объектную ссылку. Предполагается, что эта ссылка указывает на копию. Все, что вы должны сделать — это вернуть копию объекта, правильно? Не торопитесь.

Как видите, в определении интерфейса заложена не такая уж тонкая проблема. Документация по интерфейсу не указывает, должна возвращенная копия быть глубокой (deep) копией или же поверхностной (shallow). Фактически документация оставляет это на усмотрение проектировщика класса. Разница между поверхностной копией и глубокой существенна только в том случае, если объект содержит ссылки на другие объекты. Поверхностное копирование объекта создает копию, чьи содержащиеся объектные ссылки указывают на те же объекты, что и в прототипе. Глубокая копия, с другой стороны, создает копию прототипа, в которой все содержащиеся объекты также копируются. В глубокой копии дерево содержащихся объектов проходится сверху вниз, и по пути осуществляется копирование всех объектов. Поэтому результат глубокого копирования не разделяет никаких общих объектов с прототипом.

Этого достаточно, чтобы свести с ума хорошего разработчика программного обеспечения. Выглядит логичным, что если вы действительно хотите создать копию объекта, то глубокое копирования — единственно правильный путь. Отлично! С этого момента и далее, когда я говорю "клон", то имею в виду глубокую копию.

Для того чтобы объект эффективно реализовал клон, помните, что клон — это глубокая копия, поэтому все содержащиеся объекты должны предусматривать средства глубокого копирования самих себя. Вы можете легко увидеть проблему, возникающую из-за этого требования. Невозможно гарантировать глубокое копирование, если ваш объект содержит ссылки на объекты, которые сами не являются глубоко копируемыми. Вот почему мы страдаем от документации интерфейса ICloneable и недостатка в ней спецификации семантики копирования. Плюс, и это важно, такой недостаток спецификации заставляет вас четче документировать реализацию ICloneable в любом объекте, реализующем этот интерфейс, чтобы потребители знали, поддерживает объект поверхностную или глубокую копию.

Рассмотрим опции реализации интерфейса ICloneable в объектах. Если ваш объект содержит в себе только типы значений, такие как int, long или значения, основанные на определениях структур, не имеющих в себе ссылочных типов, то вы используете сокращение для реализации метода Clone, применяя Object.MemberwiseClone, как в следующем коде:

```
using System;
public sealed class Dimensions : ICloneable
{
    public Dimensions( long width, long height ) {
        this.width = width;
        this.height = height;
    }
    // Реализация ICloneable.
    public object Clone() {
        return this.MemberwiseClone();
    }
    private long width;
    private long height;
}
```

MemberwiseClone — защищенный метод, реализованный в System.Object, который может быть использован объектом для создания *поверхностной* копии самого себя. Однако важно отметить одну особенность, которая состоит в том, что MemberwiseClone создает копию объекта, не вызывая никаких конструкторов для нового объекта. Это сокращенное создание объекта. Если ваш объект полагается на обязательный вызов конструктора во время создания — например, если вы отправляете отладочную трассировку на консоль во время создания объекта — тогда MemberwiseClone не для вас. Если вы обязательно должны использовать MemberwiseClone, и ваш объект требует выполнения определенной работы при вызове конструктора, вы должны возложить эту работу на отдельный метод. Потом вы вызываете этот метод из конструктора и из вашего метода Clone после того, как будет вызван MemberwiseClone для создания нового экземпляра. Хотя такой подход возможен, все же он довольно утомителен. Альтернативный путь реализации клона состоит в применении приватного копирующего конструктора, как в следующем коде:

```
using System;
public sealed class Dimensions : ICloneable
{
    public Dimensions( long width, long height ) {
        Console.WriteLine( "Вызван Dimensions( long, long)" );
        this.width = width;
        this.height = height;
    }
    // Приватный копирующий конструктор, используемый при создании
    // копии этого объекта.
    private Dimensions( Dimensions other ) {
        Console.WriteLine( "Вызван Dimensions( Dimensions )" );
        this.width = other.width;
        this.height = other.height;
    }
    // Реализация ICloneable.
    public object Clone() {
        return new Dimensions(this);
    }
    private long width;
    private long height;
}
```

Этот метод клонирования объектов является наиболее безопасным в том смысле, что вы сохраняете полный контроль над тем, как выполняется копирование. Любые изменения, которые должны быть внесены в порядок копирования, могут быть внесены в копирующий конструктор. Вы должны учитывать то, что случается, когда вы объявляете конструктор класса. А случается то, что при этом компилятор не создает конструктор по умолчанию, что обычно делается, когда вы не предусматриваете никакого конструктора. Если приватный конструктор копирования окажется единственным определенным в классе, пользователи этого класса никогда не смогут создавать его экземпляры. Это потому, что конструктор по умолчанию пропадает, а никакого другого общедоступного конструктора нет. В данном случае нам можно не беспокоиться, поскольку определен общедоступный конструктор, принимающий два параметра. Тем не менее, это важный момент, который следует учитывать при проектировании класса.

Теперь давайте также рассмотрим объекты, которые содержат в себе ссылки на другие объекты. Предположим, что у вас есть база данных сотрудников, и вы представляете каждого сотрудника объектом типа `Employee`. Этот тип `Employee` содержит важнейшую информацию, такую как имя сотрудника, титул и идентификационный номер. Имя и, возможно, форматированный идентификационный номер представлены в виде строк, которые сами по себе являются объектами ссылочного типа. Для примера реализуем титул сотрудника в виде отдельного класса по имени `Title`. Если вы следуете руководству, изложенному мной ранее — всегда выполнять глубокое копирование для клона — то вы можете реализовать следующий метод клонирования:

```
using System;
// Класс Title
//
public sealed class Title : ICloneable
{
   public enum TitleNameEnum {
      GreenHorn,
      HotshotGuru
   }
   public Title( TitleNameEnum title ) {
      this.title = title;
      LookupPayScale();
   }
   private Title( Title other ) {
      this.title = other.title;
      LookupPayScale();
   }
   // Реализация ICloneable.
   public object Clone() {
      return new Title(this);
   }
   private void LookupPayScale() {
      // Находит тарифную сетку в базе данных.
      // Тарифная сетка определяется титулом.
   }
```

```
        private TitleNameEnum title;
        private double minPay;
        private double maxPay;
    }
    // Класс Employee
    //
    public sealed class Employee : ICloneable
    {
        public Employee( string name, Title title, string ssn ) {
            this.name = name;
            this.title = title;
            this.ssn = ssn;
        }
        private Employee( Employee other ) {
            this.name = String.Copy( other.name );
            this.title = (Title) other.title.Clone();
            this.ssn = String.Copy( other.ssn );
        }
        // Реализация ICloneable.
        public object Clone() {
            return new Employee(this);
        }
        private string name;
        private Title title;
        private string ssn;
    }
```

Обратите внимание, что вы не можете копировать объект `Title` методом `MemberwiseClone`, потому что побочный эффект от конструктора состоит в вызове `LookupPayScale` на новом объекте, чтобы получить из базы данных тарифную сетку по титулу. Предположим, что тарифная сетка должности может измениться между моментом создания прототипа и операцией клонирования, поэтому всегда нужно обращаться к базе данных. К тому же заметьте, что копии содержащихся объектов выполняются с использованием соответствующих методов `ICloneable`. Для объекта `Title` вы просто вызываете его реализацию `Clone`. Оказывается, `System.String` реализует `ICloneable`. Однако вы не можете использовать метод `Clone` для создания глубокой копии `Employee`. Если вы внимательно прочитаете описание реализации `String.Clone`, то увидите, что она просто возвращает ссылку на самого себя. Это — блестящий пример проблем, о которых я говорил относительно несогласованности реализаций `Clone`. Вместо этого вы должны применять статический метод `String.Copy`, чтобы получить настоящую копию исходной строки.

Тот факт, что `System.String` возвращает ссылку на себя при вызове метода `ICloneable.Clone`, объясняется оптимизацией, предусмотренной разработчиками. Даже несмотря на то, что они удерживают вас от создания глубокого клона любого объекта, содержащего ссылки на строковые объекты, эта оптимизация корректна по двум причинам. Во-первых, документация не специфицирует, должны вы реализовать глубокую или поверхностную копию. Я уже говорил об аргументах за и против такого упущения в спецификации контракта. Во-вторых, `System.String` — не изменяемый объект. Неизменяемость объекта — мощная концепция, о которой

я расскажу ниже, а разделе "Всегда отдавайте предпочтение безопасности типов". Общая идея состоит в том, что, однажды создав строковый объект, вы не можете изменить его на протяжении его жизни. Поэтому реализация String.Clone, выполняющая глубокое копирование, нанесла бы ущерб эффективности. Клиенты System.String работают одинаково, независимо от того, выполняет String.Clone глубокое или поверхностное копирование, именно из-за неизменности объектов String.

Стараясь сделать так, чтобы реализация ICloneable документировала сама себя, вы можете использовать специальный атрибут, которым пометить метод Clone. Таким образом, потребители вашего объекта смогут определять во время проектирования или во время выполнения, поддерживает ваш объект глубокое или поверхностное копирование. Рассмотрим следующий пользовательский атрибут[2]:

```
using System;
namespace CloneHelpers
{
    public enum CloneStyle {
        Deep,
        Shallow
    }

    [AttributeUsageAttribute(AttributeTargets.Method)]
    public sealed class CloneStyleAttribute : Attribute
    {
        public CloneStyleAttribute( CloneStyle clonestyle ) {
            this.clonestyle = clonestyle;
        }
        public CloneStyle Style {
            get {
                return clonestyle;
            }
        }
        private CloneStyle clonestyle;
    }
}
```

Используя этот атрибут, вы можете пометить реализации клонов таким образом, чтобы они сами несли в себе информацию о том, какой тип операции клонирования они поддерживают. Но имейте в виду, что этот атрибут — всего лишь маркер, который ни к чему не обязывает во время выполнения. Я не говорю о том, что вы не можете создать некоторый другой тип, который навязывает политику во время выполнения, основываясь на прикрепленных пользовательских атрибутах. Давайте вернемся к классу Dimensions и соответственно применим этот атрибут:

```
using System;
using CloneHelpers;
```

[2] Полное раскрытие темы пользовательских атрибутов в .NET Framework выходит за рамки настоящей книги. Дополнительную информацию вы можете найти в документации MSDN или любой из книг, посвященных CLR, например, в книге Эндрю Троелсена (Andrew Troelsen) *Pro C# with .NET 3.0* (Berkeley, CA: Apress, 2007г.).

```
public sealed class Dimensions : ICloneable
{
   public Dimensions( long width, long height ) {
      this.width = width;
      this.height = height;
   }
   // Реализация ICloneable.
   [CloneStyleAttribute(CloneStyle.Deep)]
   public object Clone() {
      return this.MemberwiseClone();
   }
   private long width;
   private long height;
}
```

Теперь нет вопросов о том, как реализован метод Clone, и потребители этого объекта будут информированы.

После этой дискуссии я уверен, что вы согласитесь, что реализация такой на первый взгляд безобидной вещи, как ICloneable, вовсе не так проста.

Внимание! Избегайте реализации ICloneable. Как бы тревожно это не звучало, Microsoft действительно выдвигает такую рекомендацию. Проблема происходит из того факта, что контракт не специфицирует, должна ли копия быть глубокой или же поверхностной. Фактически, как описано в книге Кшиштофа Квалины (Krzysztof Cwalina) и Брэда Абрамса (Brad Abrams) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Boston, MA: Addison-Wesley Professional, 2005 г.), авторы пересмотрели всю кодовую базу .NET Framework и не нашли нигде кода, использующего ICloneable. Если бы проектировщики и разработчики .NET Framework использовали этот интерфейс, они бы наверно обнаружили недостатки в спецификации ICloneable и устранили их. Однако эта рекомендация не направлена на то, чтобы помешать вам реализовать метод Clone, если он действительно необходим. Если вашему классу нужен метод клонирования, вы можете реализовать его в общедоступном контракте класса, не реализуя интерфейса ICloneable.

Является ли Object одноразовым?

Я уже описывал достоинства и недостатки одноразовых, или освобождаемых (disposable), объектов, но давайте поговорим об эффекте, который они могут оказать на ваш дизайн. Во-первых, вам нужно определить, должен ли ваш объект быть одноразовым. Обычно если объект управляет определенного рода неуправляемым ресурсом, таким как порция виртуальной памяти (или любым другим "родным" ресурсом), то объект должен быть одноразовым. Если ваш объект содержит другие объекты, которые сами являются одноразовыми, то ваш объект также должен быть одноразовым. Например, объект, хранящий ссылку на открытый файл с исключительными привилегиями на чтение/запись должен быть одноразовым, чтобы клиент этого объекта мог контролировать закрытие и очистку лежащего в основе ресурса.

Объект объявляется одноразовым, если он реализует интерфейс IDisposable. Интерфейс IDisposable — еще один из упрощенно выглядящих интерфейсов, подобный ICloneable, который таит в себе массу нюансов и проблем. Посмотрим на объявление интерфейса:

```
public interface IDisposable
{
   void Dispose();
}
```

Выглядит достаточно просто. Необходимо лишь реализовать метод Dispose, чтобы он очищал ресурс, и готово, верно? Ну, может быть...

Если вы создаете одноразовый объект, содержащий в себе другие одноразовые объекты, то в вашей реализации Dispose вы должны вызвать методы Dispose содержащихся объектов. К тому же, вполне допускается многократный вызов Dispose клиентами. Поэтому вместо генерации исключения при последующих вызовах, что неправильно, согласно документации по IDisposable, вы должны просто ничего не делать. Потому вам нужно поддерживать какой-то внутренний флаг, чтобы ваш код не дал сбой, если Dispose будет вызван много раз. Этот внутренний флаг может быть использован и для других целей. Обычно недопустимо вызывать этот метод на освобожденном (disposed) объекте, поэтому в таких случаях вы можете проверить флаг, и если он говорит о том, что объект уже освобожден ранее, вы можете сгенерировать исключение ObjectDisposedException. Вы уже могли видеть, что требования реализации IDisposable являются возрастающими, и то, что казалось простым интерфейсом, становится в реализации все более и более сложным. Взглянем на пример реализации IDisposable. Следующий код состоит из пользовательского объекта кучи, использующего функции Win32 для управления локальной кучей:

```
using System;
using System.Runtime.InteropServices;
public sealed class Win32Heap : IDisposable
{
   [DllImport("kernel32.dll")]
   static extern IntPtr HeapCreate(uint flOptions, UIntPtr dwInitialSize,
                                   UIntPtr dwMaximumSize);
   [DllImport("kernel32.dll")]
   static extern bool HeapDestroy(IntPtr hHeap);
   public Win32Heap() {
      theHeap = HeapCreate( 0, (UIntPtr) 4096, UIntPtr.Zero );
   }
   // Реализация IDisposable.
   public void Dispose() {
      if( !disposed ) {
         HeapDestroy( theHeap );
         theHeap = IntPtr.Zero;
         disposed = true;
      }
   }
   private IntPtr theHeap;
   private bool disposed = false;
}
```

Этот объект не содержит в себе никаких объектов, реализующих IDisposable, поэтому вам не нужно проходить по дереву включения, вызывая Dispose.

Важно отметить, что в шаблоне Disposable реализация содержащихся объектов заставляет объект-контейнер реализовать `IDisposable`, если содержащиеся в нем объекты реализуют `IDisposable`. Это отношение "изнутри-наружу".

Поскольку шаблон Disposable требует от пользователя явного вызова метода `Dispose`, на пользователя возлагается ответственность за его вызов, даже в случае исключений. Это делает реализацию клиентского кода утомительной. Например, рассмотрим следующий код, который открывает файл для записи:

```
using System;
using System.IO;
public sealed class WriteStuff
{
    static void Main(){
        StreamWriter sw = new StreamWriter("Output.txt");
        try {
            sw.WriteLine( "Проверка безопасности механизма освобождения"
);
        }
        finally {
            if( sw != null ) {
                ((IDisposable)sw).Dispose();
            }
        }
    }
}
```

Проектировщики C# поняли, насколько болезненным может оказаться написание такого кода, поэтому в .NET Framework появился интерфейс `IDisposable` и разработчики перегрузили ключевое слово `using` для создания оператора `using`, который должен был помочь в этой ситуации. В операторе `using` вы объявляете одноразовые переменные внутри пары скобок и затем, когда управление покидает следующий блок кода, все объекты освобождаются. Внутренне оператор `using` делает, по сути, то же самое, что и конструкция `try/finally`. Чтобы убедиться в этом, достаточно взглянуть на сгенерированный код IL. Оператор `using` определенно помогает: однако клиент объекта все равно должен, прежде всего, не забыть использовать его. Давайте модифицируем предыдущий пример, добавив оператор `using`:

```
using System;
using System.IO;
public sealed class WriteStuff
{
    static void Main(){
        using( StreamWriter sw = new StreamWriter("Output.txt") ) {
          sw.WriteLine( "Проверка безопасности механизма освобождения" );
        }
    }
}
```

Теперь можете ли вы представить, что случится, если клиент вашего объекта забудет вызвать `Dispose` или не воспользуется оператором `using`? Ясно, что в этом случае появляется шанс утечки ресурса. И потому вы должны реализовать финализатор, как я объясню в следующем разделе.

На заметку! В предыдущих примерах я не рассматривал, что произойдет, если несколько потоков параллельно вызовут `Dispose`. Хотя ситуация кажется маловероятной, вы должны предусмотреть ее, если разрабатываете код библиотеки, которую будут использовать неизвестные вам клиенты.

Нужен ли финализатор `Object`?

Финализатор — это метод, который вы можете реализовать в вашем классе, и который вызывается перед тем, как сборщик мусора (GC) вычистит ваш неиспользуемый объект из кучи. Давайте сначала проясним одну важную концепцию: финализаторы — это не деструкторы, и вы не должны видеть в них деструкторы.

Деструкторы ассоциируются с детерминированной деструкцией объектов. Финализаторы относятся к недетерминированной деструкции объектов. К сожалению, значительная путаница между финализаторами и деструкторами обусловлена тем фактом, что проектировщики языка C# решили отобразить финализаторы на синтаксис деструкторов C#, который идентичен синтаксису деструкторов C++. Фактически вы обнаружите, что в C# невозможно явно перегрузить `Object.Finalize`. Вы перегружаете его неявно, используя синтаксис деструктора, который применяете в том случае, если пришли из мира C++. Единственная хорошая вещь, присущая реализации финализаторов в C#, — вам никогда не нужно заботиться о явном вызове финализатора класса. За вас это делает компилятор.

Большую часть времени, когда ваш код нуждается в определенного рода коде очистки — например, объект, абстрагирующий файл из файловой системы — этот код должен быть вызван детерминировано, например, при манипуляциях с неуправляемыми ресурсами. Другими словами, это должно произойти явно, когда пользователь завершил работу с объектом, а не тогда, когда GC наконец-то соберется уничтожить объект. В этих случаях вам нужно реализовать эту функциональность с использованием шаблона Disposable, реализуя интерфейс `IDisposable`. Не позволяйте себя обмануть тому факту, что деструктор, который вы пишете в классе с применением знакомого синтаксиса деструктора, будет вызван, когда объект выйдет из своей области определения, как это происходит в C++. Фактически, если вы задумаетесь об этом, то увидите, что очень редко возникает необходимость в реализации финализатора. Трудно представить себе задачу очистки, которую невозможно выполнить с применением `IDisposable`.

На заметку! В действительности вообще редко возникает необходимость в написании деструктора. Большую часть времени вы должны реализовывать шаблон Disposable, чтобы выполнять необходимую очистку ресурсов в вашем объекте. Однако финализаторы могут быть полезны для гарантированной очистки неуправляемых ресурсов, т.е. когда пользователь забыл вызвать `IDisposable.Dispose`.

В идеальном мире вы могли бы просто реализовать весь типичный код деструктора в методе `IDisposable.Dispose`. Однако существует один серьезный побочный эффект от того, что в языке C# не поддерживается детерминированная деструкция. Компилятор C# не вызывает автоматически `IDisposable.Dispose` на вашем объекте, когда он выходит из своей области определения. C#, как я уже говорил ранее, возлагает на пользователя обязанность вызова `IDisposable.Dispose`. Язык C# упрощает задачу гарантировать это поведение, перегружая ключевое слово `using`, но

все же требует от клиента вашего объекта не забыть применить это ключевое слово. Это важно иметь в виду, и именно это разрушает вашу мечту об "идеальном мире".

Мы живем не в идеальном мире, поэтому для того, чтобы напрямую надежно очистить используемые ресурсы, для любого объекта, реализующего интерфейс `IDisposable`, необходимо также реализовать финализатор, который просто вызовет метод `Dispose`[3]. Таким образом, вы можете перехватить эти блуждающие ошибки, связанные с тем, что пользователи забывают использовать шаблон Disposable и не освобождают (dispose) правильно ваш объект. Конечно, очистка неосвобожденных объектов теперь произойдет по инициативе GC, но по крайней мере, она все-таки произойдет. Имейте в виду: финализатор GC вызывают финализатор объекта, подлежащего очистке, из отдельного потока. Теперь вас могут обеспокоить проблемы многозадачности в ваших одноразовых объектах. Маловероятно, что они заденут вас при финализации, поскольку теоретически финализируемый объект уже не имеет ссылок ниоткуда. Однако это может стать фактором, зависящим от того, что вы делаете в вашем методе `Dispose`. Например, если ваш метод `Dispose` использует внешний, возможно, неуправляемый объект для выполнения работы, на который может иметь ссылку другая сущность, то такой объект должен надежно работать в многопоточных средах. Лучше перестраховаться, чем потом сожалеть и бороться с проблемами многозадачности, реализуя финализатор.

Есть еще одна важная вещь, которую нужно рассмотреть, и которой я коснулся в предыдущей главе. Когда вы вызываете метод `Dispose` через финализатор, вы не должны использовать ссылочных объектов, содержащихся в полях внутри данного объекта. На первый взгляд, это звучит не слишком понятно, но вы должны понимать, что не существует никаких гарантий последовательности финализации объектов. Объекты в полях вашего объекта могут быть финализированы перед запуском вашего финализатора. Потому это может вызвать ветвящееся неопределенное поведение, если вы их используете и окажется, что они уже уничтожены. Думаю, вы согласитесь, что выявить причину такой ошибки будет весьма нелегко. Теперь вам должно быть ясно, что финализаторы ведут вас в страну, полную ловушек.

Внимание! Будьте осторожны с любым объектом, используемым во время финализации, даже если речь не идет о поле вашего финализируемого объекта, потому что, опять-таки, он уже может быть помечен к финализации, и может уже быть или не быть финализированным. Использование объектных ссылок внутри финализатора действительно ведет вас на скользкую дорожку. Фактически многие школы рекомендуют воздерживаться от применения внешних объектов внутри финализатора. Объективным фактом является то, что в любой момент, когда объект, поддерживающий финализатор, перемещается в очередь финализации GC, все элементы графа объектов перемещаются, независимо от того, финализируемы они или нет. Поэтому если ваш финализируемый объект содержит в себе приватный, нефинализируемый объект, то вы можете касаться этого объекта в финализаторе, потому что знаете, что он еще существует, поскольку он помещается в очередь финализации вместе с вашим объектом, и никак не может быть финализирован перед вашим объектом, поскольку не имеет финализатора. Обязательно прочтите следующую вкладку "На заметку!" далее в тексте!

---

[3] Важно отметить, что объекты, реализующие `IDisposable` только по причине того, что содержащиеся в них типы реализуют `IDisposable`, не должны иметь финализатора. Они не управляют ресурсами напрямую, и такой финализатор только излишне увеличит нагрузку на поток финализатора GC.

Давайте вернемся к примеру Win32Heap из предыдущего раздела и модифицируем его финализатором. Последуем рекомендованному шаблону Disposable и посмотрим, как изменится код:

```
using System;
using System.Runtime.InteropServices;
public class Win32Heap : IDisposable
{
    [DllImport("kernel32.dll")]
    static extern IntPtr HeapCreate( uint flOptions,
                                     UIntPtr dwInitialSize,
                                     UIntPtr dwMaximumSize);
    [DllImport("kernel32.dll")]
    static extern bool HeapDestroy(IntPtr hHeap);
    public Win32Heap() {
        theHeap = HeapCreate( 0, (UIntPtr) 4096, UIntPtr.Zero );
    }
    // Реализация IDisposable.
    protected virtual void Dispose( bool disposing ) {
        if( !disposed ) {
            if( disposing ) {
                // Здесь допускается использовать любые внутренние объекты.
                // Этот класс, однако, их не имеет.
            }
            // Если используются объекты, о которых известно,
            // что они еще существуют, - такие, как объекты,
            // реализующие шаблон Singleton, важно убедиться,
            // что они являются безопасными в отношении потоков.
            HeapDestroy( theHeap );
            theHeap = IntPtr.Zero;
            disposed = true;
        }
    }
    public void Dispose() {
        Dispose( true );
            GC.SuppressFinalize( this );
        }
        ~Win32Heap() {
        Dispose( false );
    }
    private IntPtr theHeap;
    private bool disposed = false;
}
```

Давайте проанализируем изменения, внесенные для поддержки финализатора. Для начала обратите внимание, что я добавил финализатор, применив знакомый синтаксис деструктора[4]. Также отметьте, что я добавил второй уровень посредничества в реализацию Dispose. Это для того, чтобы можно было знать, вызван ваш приватный метод Dispose из вызова Dispose или через финализатор. Также в этом примере Dispose(bool) реализован виртуально, так что любой производный

[4] Не переставайте напоминать себе, что это — не деструктор!

тип может просто переопределить этот метод для модификации поведения освобождения. Если класс Win32Heap был бы помечен как sealed, вы могли бы изменить модификатор доступа метода с protected на private и убрать ключевое слово virtual. Как я упоминал ранее, вы не можете надежно использовать подобъекты, если ваш метод Dispose был вызван из финализатора.

На заметку! Некоторые предпочитают подход, основанный на том, что все объектные ссылки лишены каких-либо ограничений внутри метода Dispose, если он был вызван финализатором. Нет причин, по которым нельзя было бы использовать эти объекты, если вам известно, что они живы-здоровы. Однако имейте в виду, что если финализатор был вызван в результате останова всего домена приложения, то объекты, которые вы считаете живыми, уже таковыми могут не оказаться. В действительности почти невозможно определить со 100% уверенностью, действительна ли объектная ссылка в таких случаях. Поэтому лучше просто не обращаться ни к каким ссылочным типам на стадии финализации, если этого можно избежать.

Метод Dispose наносит удар по производительности — обратите внимание на вызов GC.SupressFinalize. Финализатор этого объекта просто вызывает приватный метод Dispose, а вы знаете, что если вызван ваш общедоступный метод Dispose, то вашему финализатору больше ничего делать не нужно. Поэтому вы можете приказать GC, чтобы он не помещал экземпляр объекта в очередь финализации, когда вызван метод IDisposable.Dispose. Такая оптимизация более чем тривиальна, если учесть, что объекты, реализующие финализатор, существуют дольше, чем те, которые этого не делают. Когда GC просматривает кучу в поисках мертвых объектов, которые нужно убрать, обычно он сжимает кучу и освобождает память. Однако если объект оснащен финализатором, то вместо немедленного освобождения памяти, GC перемещает объект в список финализации, который обрабатывается отдельным потоком финализации. Как только поток финализации завершает свою работу над объектом, объект помечается к удалению, и GC освобождает место во время следующего прохода. Таким образом, объект, реализующий финализатор, существует дольше, чем объект без финализатора. Если ваш объект съедает много памяти кучи, либо ваша система создает много таких объектов, финализация становится значительным фактором. Она не только снижает эффективность GC, но также поглощает время процессора в потоке финализации. Вот почему по возможности следует подавлять финализацию внутри Dispose.

На заметку! Когда у объекта есть финализатор, он помещается во внутреннюю очередь CLR для отслеживания этого факта, и ясно, что этот статус оказывает влияние на GC.SuppressFinalize. При нормальном выполнении, как упоминалось ранее, вы не можете гарантировать доступность других объектных ссылок. Однако во время останова приложения поток финализации действительно финализирует объекты из этой внутренней очереди финализации, и потому эти объекты доступны, и на них можно ссылаться в финализаторе. Вы можете определить, так ли это, используя Environment.HasShutdownStarted или AppDomain.IsFinalizingForUnload. Однако из того, что вы можете это сделать, не следует, что вы должны это делать без тщательного обдумывания. Не удивляйтесь, если это поведение изменится в будущих версиях CLR.

Теперь давайте внимательней рассмотрим влияние финализаторов на производительность GC. Сборщик мусора CLR реализован как учитывающий поколения (generational). Это значит, что выделенные объекты, которые относятся к старшим

поколениям, существуют дольше, чем те, что относятся к младшим поколениям, и собираются менее часто, чем те, что относятся к поколению над ними. Тонкие детали алгоритма сборки мусора GC выходят за рамки настоящей книги. Однако полезно коснуться их на самом высоком уровне. Например, GC обычно пытается разместить новые объекты в поколении 0. Более того, GC предполагает, что объекты из поколения 0 будут существовать в течение относительно короткого времени. Поэтому, когда GC пытается выделить место для объекта и видит, что кучу пора сжать, он освобождает место, удерживаемое умершими объектами поколения 0, а те объекты, которые еще живы, перемещаются в поколение 1 во время этого сжатия. На этой стадии, если GC в состоянии найти достаточно места для выделения, он прекращает процесс сжатия кучи. Он не будет пытаться сжать поколение 1, если только ему не понадобится еще больше места, или же он обнаружит, что куча поколения 1 полна и, вероятно, нуждается в сжатии. Затем он проходит по всем поколениям, если это необходимо. Однако во время этого прохода сборщика мусора объект может быть перемещен только на один уровень. Поэтому, если объект перемещается из поколения 0 в поколение 1 во время сборки, и GC должен последовательно сжимать поколение 1 в том же проходе, то только что перемещенный объект остается в поколении 1. В настоящее время куча CLR состоит только из трех поколений. Потому, естественно, если объект живет в поколении 2, он не может быть перемещен в старшее поколение. CLR также содержит специальную кучу для выделения крупных объектов, которая в настоящей версии содержит объекты, чей размер превышает 80 Кбайт. Эта цифра может измениться в будущих версиях, так что не полагайтесь на ее неизменность.

Теперь рассмотрим, что происходит, когда объект из поколения 0 перемещается в поколение 1 во время сжатия кучи. Даже если все корневые ссылки на объект в поколении 1 находятся вне своих областей видимости, пространство может и не возвращаться в течение длительного времени, поскольку GC не слишком часто сжимает область кучи поколения 1.

Объекты, реализующие финализаторы, попадают в очередь финализации. Эта ссылка в очереди считается корневой ссылкой. Поэтому объект будет перемещен в поколение 1, если он в данный момент находится в поколении 0. Однако вы уже знаете, что объект умирает. Фактически, как только очередь финализации пустеет, это означает, что объект, скорее всего, будет мертв, если только он не воскреснет во время процесса финализации. Этот объект с финализатором умирает, но поскольку он был помещен в очередь финализации, и потому перемещен в более старшее поколение, весьма вероятно, что он останется где-то у истоков GC до тех пор, пока не случится сжатие старшего поколения.

По этой причине важно, чтобы вы не реализовывали финализаторы, если только это не абсолютно необходимо. И, как упоминалось ранее, это должно быть необходимо только когда ваш объект напрямую поддерживает ресурсы, которые должны выделяться и освобождаться детерминировано.

На заметку! Ресурсы, которые должны очищаться детерминировано, могут быть как управляемыми, так и неуправляемыми. В качестве примера неуправляемого ресурса можно представить нечто вроде экземпляра `System.IO.FileStream`, где `IDisposable.Dispose` вызывается через `FileStream.Close`, который и освобождает лежащие в основе неуправляемые ресурсы. Скорее всего, это нежелательно для вас — ожидать, пока GC почувствует вызов финализатора на `FileStream`, прежде чем файл будет разблокирован.

Я хочу немного сконцентрировать внимание на том факте, что Dispose никогда не вызывается автоматически, и как ваш финализатор может помочь в решении потенциальных проблем эффективности вашим клиентам. Предположим, что вы создаете объект, который выделяет нетривиальную порцию неуправляемых системных ресурсов. И предположим, что клиент создает новый экземпляр вашего объекта. Производительность клиентской системы существенно деградирует, если клиент забудет своевременно вызвать Dispose этих объектов перед тем, как ссылки на эти объекты исчезнут. Конечно, если вы реализуете финализатор так, как было показано ранее, то объект в конечном итоге будет освобожден. Однако это произойдет, только когда GC сочтет это необходимым, поэтому вероятно истощение ресурсов системы. Более того, если забыть вызвать Dispose, то это, скорее всего, задержит финализацию, что еще больше нагрузит GC. Клиентский код может принудительно инициировать работу GC через вызов метода GC.Collect. Однако этого делать настоятельно не рекомендуется, поскольку вызовет вмешательство в алгоритм работы GC. В 99,9% случаев GC лучше вас знает, как управлять памятью.

Было бы неплохо, если бы вы могли информировать клиентов вашего объекта, когда они забывают вызвать Dispose в своих отладочных сборках. Фактически вы можете протоколировать ошибку всякий раз, когда запускается финализатор вашего объекта, извещая о том, что объект не был отброшен правильно. Вы можете даже указать клиентам точное место создания объекта, выводя трассировку стека в точке создания. Таким образом, они будут знать, какая строка кода создает объект-нарушитель. Давайте модифицируем пример Win32Heap на основе этого подхода:

```
using System;
using System.Runtime.InteropServices;
using System.Diagnostics;
public sealed class Win32Heap : IDisposable
{
    [DllImport("kernel32.dll")]
    static extern IntPtr HeapCreate(uint flOptions,
    UIntPtr dwInitialSize,
    UIntPtr dwMaximumSize);
    [DllImport("kernel32.dll")]
    static extern bool HeapDestroy(IntPtr hHeap);
    public Win32Heap() {
        creationStackTrace = new StackTrace(1, true);
        theHeap = HeapCreate( 0, (UIntPtr) 4096, UIntPtr.Zero );
    }
    // Реализация IDisposable
    private void Dispose( bool disposing ) {
        if( !disposed ) {
            if( disposing ) {
                // Здесь допускается использовать любые внутренние объекты.
                // Этот класс, однако, их не имеет.
            } else {
                // Опа! Мы финализировали этот объект, а он не был освобожден.
                // Уведомим пользователя, если только домен приложения
                // не завершается здесь.
                AppDomain currentDomain = AppDomain.CurrentDomain;
```

```
                if( !currentDomain.IsFinalizingForUnload() &&
                    !Environment.HasShutdownStarted )
                {
                    Console.WriteLine(
                    "Сбой в освобождении объекта!!!" );
                    Console.WriteLine( "Объект размещен в:" );
                    for( int i = 0; i < creationStackTrace.FrameCount;
                         ++i ) {
                        StackFrame frame =
                        creationStackTrace.GetFrame(i);
                        Console.WriteLine( " {0}",
                        frame.ToString() );
                    }
                }
            }
            // Если используются объекты, о которых известно,
            // что они еще существуют, такие как объекты,
            // реализующие шаблон Singleton, важно убедиться,
            // что они являются безопасными в отношении потоков.
            HeapDestroy( theHeap );
            theHeap = IntPtr.Zero;
            disposed = true;
        }
    }
    public void Dispose() {
        Dispose( true );
        GC.SuppressFinalize( this );
    }
    ~Win32Heap() {
        Dispose( false );
    }
    private IntPtr theHeap;
    private bool disposed = false;
    private StackTrace creationStackTrace;
}
public sealed class EntryPoint
{
    static void Main()
    {
        Win32Heap heap = new Win32Heap();
        heap = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

Обратите внимание, что в методе `Main` я распределяю новый объект `Win32Heap`, после чего немедленно вызываю его финализацию. Поскольку объект не был освобожден, это вызывает код построения дампа стека внутри метода `Dispose`. Поскольку вы, вероятно, не заботитесь о финализации объектов в результате выгрузки домена приложения, я поместил код построения дампа стека в условный блок, за-

висящий от результата AppDomain.IsFinalizingForUnload && Environment.HasShutdownStarted. Если бы в Main я вызвал Dispose перед установкой ссылки в null, то трассировка стека не была бы отправлена на консоль. Клиенты вашей библиотеки могут поблагодарить вас за указание на не освобожденные объекты. И это правильно.

На заметку! Когда вы скомпилируете предыдущий пример, вы получите более осмысленный и читабельный вывод, если укажете ключ компилятора /debug+. Вы можете даже рассмотреть включение генерации отчета только в отладочных и тестовых сборках.

Надеюсь, эта дискуссия наглядно показала вам опасности реализации финализаторов. Они представляют собой потенциальную причину огромного расхода ресурсов, поскольку заставляют объекты существовать дольше, при этом скрываясь за безобидным синтаксисом деструкторов. Единственной компенсацией этого со стороны финализаторов является способность указать, где объекты не были освобождены правильно, но я советую применять эту технику только в отладочных целях.

В методе Main я создал новый экземпляр System.Object и затем немедленно сделал копию этой ссылки. В результате я получил нечто подобное тому, что демонстрирует диаграмма на рис. 13.1.

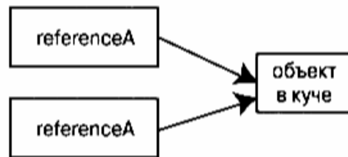

Рис. 13.1. Ссылочные переменные

В CLR переменные, представляющие ссылки, в действительности являются типами значений, которые хранят местоположение в памяти (т.е. указатели на объекты, которые они представляют) вместе с ассоциированным типом. Однако обратите внимание, что когда ссылка копируется, объект, на которую она указывает, не копируется. Вместо этого вы получаете две ссылки на один и тот же объект. Операции над объектом, выполненные через одну ссылку, будут видимы клиенту, использующему другую ссылку.

Теперь рассмотрим, что означает сравнение двух ссылок. Что на самом деле означает эквивалентность двух ссылочных переменных? Ответ состоит в том, что это зависит от того, что вам нужно, и как вы определяете эквивалентность. По умолчанию эквивалентность ссылочных переменных означает сравнение на идентичность. Это значит, что две ссылочные переменные считаются эквивалентными, если они ссылаются на один и тот же объект, как на рис. 13.1. Опять же эта ссылочная эквивалентность является поведением по умолчанию для определения эквивалентности двух ссылок на объекты, находящиеся в куче.

С точки зрения клиентского кода вы должны быть осторожны при сравнении двух объектных ссылок на эквивалентность. Рассмотрим следующий код:

```
public class EntryPoint
{
   static bool TestForEquality( object obj1, object obj2 )
   {
      return obj1.Equals( obj2 );
   }
   static void Main()
   {
      object obj1 = new System.Object();
      object obj2 = null;
      System.Console.WriteLine( "obj1 == obj2 дает {0}",
                                TestForEquality(obj1, obj2) );
   }
}
```

Здесь я создаю экземпляр System.Object, и хочу узнать, эквивалентны ли переменные obj1 и obj2. Поскольку я сравниваю ссылки, тест эквивалентности определяет, указывают ли они на один и тот же экземпляр объекта. Взглянув на этот код, вы можете видеть, что очевидный результат — obj1!=obj2, поскольку obj2 равен null. Это то, что ожидается. Однако рассмотрим, что произойдет, если поменять порядок параметров в вызове TestForEquality. Вы быстро обнаружите, что ваша программа завершается крахом с необработанным исключением, где TestForEquality пытается вызвать Equals на null-ссылке. Поэтому вы должны модифицировать код следующим образом:

```
public class EntryPoint
{
   static bool TestForEquality( object obj1, object obj2 )
   {
      if( obj1 == null && obj2 == null ) {
         return true;
      }
      if( obj1 == null )
      {
         return false;
      }
      return obj1.Equals( obj2 );
   }
   static void Main()
   {
      object obj1 = new System.Object();
      object obj2 = null;
      System.Console.WriteLine( "obj1 == obj2 дает {0}",
                                TestForEquality(obj2, obj1) );
      System.Console.WriteLine( "null == null дает {0}",
                                TestForEquality(null, null) );
   }
}
```

Теперь код может поменять порядок аргументов в вызове TestForEquality, и вы получите ожидаемый результат. Обратите внимание, что для получения правильного результата я также предусмотрел проверку случая, когда оба аргу-

мента равны null. Теперь метод TestForEquality завершен. Конечно, кажется, что здесь слишком много работы всего-навсего для того, чтобы проверить две ссылки на эквивалентность. Да, и проектировщики стандартной библиотеки .NET Framework осознали эту проблему и предусмотрели статическую версию Object.Equals, выполняющую такое сравнение. К счастью, благодаря существованию статической версии Object.Equals, вам не нужно беспокоиться о создании кода TestForEquality в этом примере.

Итак, вы увидели, как проверка эквивалентности ссылок на объекты проверяет по умолчанию их идентичность. Однако могут быть случаи, когда такая проверка эквивалентности не имеет смысла. Рассмотрим неизменяемый объект, представляющий комплексное число:

```
public class ComplexNumber
{
    public ComplexNumber( int real, int imaginary )
    {
        this.real = real;
        this.imaginary = imaginary;
    }
    private int real;
    private int imaginary;
}
public class EntryPoint
{
    static void Main()
    {
        ComplexNumber referenceA = new ComplexNumber( 1, 2 );
        ComplexNumber referenceB = new ComplexNumber( 1, 2 );
        System.Console.WriteLine( "Результат проверки на эквивалентность: {0}",
                                  referenceA == referenceB );
    }
}
```

Вывод этого кода выглядит так:

```
Результат проверки на эквивалентность: False
```

На рис. 13.2 показана диаграмма, представляющая компоновку памяти для этих двух ссылок.

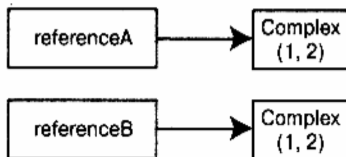

Рис. 13.2. Ссылки на ComplexNumber

Такого результата можно было ожидать, если полагаться на обычный смысл эквивалентности двух ссылок, принятый по умолчанию. Однако это совсем не очевидно пользователю этих объектов ComplexNumber. Имело бы больше смысла, чтобы сравнение двух объектов, представленных на диаграмме, вернуло true, по-

скольку значения двух объектов одинаковы. Чтобы получить такой результат, вы должны представить собственную реализацию эквивалентности таких объектов. Чуть ниже я покажу, как это делается, но сначала давайте кратко обсудим смысл понятия эквивалентности.

Эквивалентность значений

На основе предыдущего раздела смысл эквивалентности значений должен быть очевиден. Проверка эквивалентности двух значений возвращает `true`, когда действительные значения полей, представляющих состояние объекта или его значение, эквивалентны. В примере `ComplexNumber` из предыдущего раздела проверка эквивалентности значений даст `true`, когда значения полей `real` и `imaginary` эквивалентны между двумя экземплярами этого класса.

В CLR, а, следовательно, и в C#, именно так и трактуется эквивалентность типов значений, определенных как структуры. Типы значений наследуются от `System.ValueType`, и `System.ValueType` переопределяет метод `Object.Equals`. `ValueType.Equals` использует рефлексию для перебора полей типа значений при сравнении этих полей. Такая обобщенная реализация будет работать со всеми типами значений. Однако намного эффективнее будет переопределить метод `Equals` в ваших типах структур, сравнивая поля напрямую. Хотя применение рефлексии для выполнения этой задачи — обычно подходящий подход, он очень неэффективен.

На заметку! Прежде чем реализация `ValueType.Equals` обратится к рефлексии, она выполняет пару быстрых проверок. Если два сравниваемых типа разные, эквивалентности нет. Если они одного типа, сначала проверяются типы содержащихся в них полей — относятся ли они к простым типам данных, которые можно сравнить бит за битом. Если так, то весь тип можно сравнивать побитно. Невыполнение обоих условий затем ведет к применению рефлексии.

Так как реализация `ValueType.Equals` по умолчанию выполняет итерацию по значениям содержащихся полей, используя рефлексию, она определяет эквивалентность этих индивидуальных полей, полагаясь на реализации `Object.Equals` этих объектов. Таким образом, если ваш тип значения содержит поле ссылочного типа, вы можете получить неожиданный результат, в зависимости от семантики методов `Equals`, реализованных этим ссылочным типом. В общем случае включение ссылочных типов в тип значения не рекомендуется.

Переопределение `Object.Equals` для ссылочных типов

Часто вам может понадобиться переопределить семантическое значение эквивалентности объекта. Вы можете пожелать, чтобы эквивалентность для вашего ссылочного типа трактовалась как эквивалентность значений в противоположность ссылочной эквивалентности, или идентичности. Или же, как вы увидите в одном из следующих разделов, у вас может быть специальный пользовательский тип, в котором вы хотите переопределить метод `Equals` по умолчанию, предоставленный `System.ValueType`, чтобы повысить эффективность операции сравнения. Независимо от причин переопределения `Equals`, вы должны следовать перечисленным ниже правилам.

- `x.Equals(x) == true`. Это — рефлексивное свойство эквивалентности.
- `x.Equals(y) == y.Equals(x)`. Это — симметричное свойство эквивалентности.

- `x.Equals(y) && y.Equals(z)` подразумевает `x.Equals(z) == true`. Это — транзитивное свойство эквивалентности.
- `x.Equals(y)` должно возвращать один и тот же результат до тех пор, пока внутреннее состояние `x` и `y` не меняется.
- `x.Equals(null) == false` для всех `x`, отличных от `null`.
- Метод `Equals` не должен генерировать исключений.

Реализация `Equals` должна подчиняться этим непреложным правилам. Кроме того, вы должны следовать и некоторым другим рекомендациям, чтобы сделать реализации `Equals` ваших классов более устойчивыми.

Как уже говорилось, версия по умолчанию `Object.Equals`, унаследованная классами, проверяет ссылочную эквивалентность, иначе называемую идентичностью. Однако в случаях, подобных примеру с `ComplexNumber`, такая проверка не является интуитивно понятной. Будет более естественно и ожидаемо, если экземпляры такого типа будут сравниваться на основе сравнения их полей — одного за другим. Это главная причина, по которой вы должны переопределять `Object.Equals` для таких классов, которые имеют семантику значений.

Давайте еще раз вернемся к примеру `ComplexNumber`, чтобы посмотреть, как это можно сделать:

```
public class ComplexNumber
{
    public ComplexNumber( int real, int imaginary )
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public override bool Equals( object obj )
    {
        ComplexNumber other = obj as ComplexNumber;
        if( other == null )
        {
            return false;
        }
        return (this.real == other.real) &&
               (this.imaginary == other.imaginary);
    }

    public override int GetHashCode()
    {
        return (int) real ^ (int) imaginary;
    }

    public static bool operator==( ComplexNumber me, ComplexNumber other )
    {
        return Equals( me, other );
    }
```

```
    public static bool operator!=( ComplexNumber me, ComplexNumber other )
    {
        return Equals( me, other );
    }
    private double real;
    private double imaginary;
}
public class EntryPoint
{
    static void Main()
    {
        ComplexNumber referenceA = new ComplexNumber( 1, 2 );
        ComplexNumber referenceB = new ComplexNumber( 1, 2 );
        System.Console.WriteLine( "Результат проверки на эквивалентность: {0}",
                                  referenceA == referenceB );
        // Если нам действительно нужна ссылочная эквивалентность.
        System.Console.WriteLine( "Идентичность ссылок: {0}",
                              (object) referenceA == (object) referenceB );
        System.Console.WriteLine( "Идентичность ссылок: {0}",
                              ReferenceEquals(referenceA, referenceB) );
    }
}
```

Как видно из этого примера, реализация Equals вполне прямолинейна, за исключением проверки некоторых условий. Я должен убедиться, что обе сравниваемые объектные ссылки не равны null, и что они действительно ссылаются на экземпляры CompexNumber. Проверив это, я могу просто проверить поля обеих ссылок, чтобы убедиться в их эквивалентности. Вы можете провести оптимизацию и сравнивать в Equals this с other. Если они ссылаются на один и тот же объект, можно вернуть true, не сравнивая поля. Однако сравнение двух полей — совершенно тривиальная задача в данном случае, поэтому я пропустил проверку на идентичность.

В большинстве случаев вам не понадобится переопределять Object.Equals для ваших объектов ссылочных типов. Рекомендуется, чтобы ваши объекты трактовали эквивалентность, используя сравнение на идентичность, что вы и так получаете бесплатно от Object.Equals. Однако бывают случаи, когда имеет смысл переопределить Equals для объекта. Например, если ваш объект представляет нечто такое, что ведет себя как значение и является неизменным, наподобие комплексного числа или класса System.String, тогда может быть, имеет смысл переопределить Equals для того, чтобы придать реализации этого метода в данном объекте семантику эквивалентности значений.

Во многих случаях при переопределении виртуальных методов в производных классах, таких как Object.Equals, имеет смысл в некоторой точке вызвать реализацию того же метода из базового класса. Однако если ваш объект напрямую наследует System.Object, в этом смысла нет. Дело в том, что Object.Equals, скорее всего, несет в себе семантику, отличающуюся от той, что вы переопределяете. Помните, что единственная причина для переопределения Equals для объектов — изменение семантического смысла с идентичности на эквивалентность значений. К тому же, вы не захотите смешивать вместе эти две семантики. Но эта история имеет нелепый поворот. Вы *должны* вызывать версию Equals базового класса,

если ваш класс наследуется от класса, отличного от System.Object, и этот другой класс переопределяет Equals. Дело в том, что наиболее вероятная причина того, что базовый класс переопределяет Object.Equals — переключение к семантике значения. Это означает, что вы должны быть близко знакомы с вашим базовым классом, если планируете переопределить Object.Equals, чтобы знать, нужно ли вызывать версию этого метода базового класса. В этом состоит горькая правда о переопределении Object.Equals для ссылочных типов.

Иногда, даже когда вы имеете дело со ссылочными типами, вам действительно нужно проверить ссылочную эквивалентность — не важно, чего. Вы не можете всегда полагаться на метод Equals объекта для оценки ссылочной эквивалентности, поэтому придется прибегнуть к другим средствам, поскольку метод уже может быть переопределен, как в примере ComplexNumber.

К счастью, существуют два способа выполнить эту работу, и вы можете видеть их в конце метода Main в предыдущем примере кода. Компилятор C# гарантирует, что если вы применяете операцию == к двум ссылками типа Object, то вы всегда получите ссылочную эквивалентность. К тому же System.Object предлагает статический метод по имени ReferenceEquals, принимающий два параметра-ссылки и возвращающий true, если проверка идентичности дает положительный результат. Какой бы способ вы ни выбрали, результат будет одинаковым.

Если вы измените семантическое значение Equals для объекта, стоит четко документировать этот факт для клиентов вашего объекта. Если вы переопределяете Equals для класса, я настоятельно рекомендую указать семантическое значение в пользовательском атрибуте — подобный прием был представлен ранее в реализации ICloneable. Таким образом, люди, выполняющие наследование от вашего класса и желающие изменить семантический смысл Equals, смогут быстро определить, должны ли они первым делом вызывать вашу реализацию. Для максимальной эффективности пользовательский атрибут должен служить целям документирования. Хотя этот атрибут и можно проверить во время выполнения, это будет очень неэффективно.

На заметку! Никогда не генерируйте исключений внутри реализации Object.Equals. Вместо этого лучше возвращайте результат false.

На протяжении всей дискуссии я намеренно избегал упоминания операций эквивалентности, поскольку их стоит рассматривать как дополнительный слой к Object.Equals. Поддержка перегрузки операций не является обязательным требованием для CLS-совместимых языков. Поэтому не все языки, ориентированные на CLR, поддерживают их. Visual Basic — один из языков, которые долго не поддерживали перегрузку операций, и в нем такая поддержка в полной мере появилась лишь в Visual Basic 2005. Visual Basic .NET 2003 поддерживает вызов перегруженных операций только на объектах, определенных на языках, поддерживающих эту перегрузку, при этом они должны вызываться по имени специальной функции, сгенерированной для операции. Например, operator== реализована с именем op_Equality в сгенерированном коде IL. Наилучший подход — реализовать Object.Equals как основу для любых реализаций operator== и operator!=, представляя последние только в качестве дополнительного удобства для языков, поддерживающих их.

На заметку! Рассмотрите реализацию IEquatable<T> в вашем типе, чтобы получить безопасную в отношении типов версию Equals. Это особенно важно для типов значений, поскольку специфичные для типов версии методов избегают излишней упаковки.

Если переопределили Equals, переопределите и GetHashCode

GetHashCode вызывается, когда объекты используются в качестве ключей хеш-таблицы. Когда хеш-таблица ищет элемент по заданному ключу, она опрашивает ключ на предмет его хеш-кода и затем использует его для идентификации сегмента (bucket), в котором находится объект. Как только сегмент найден, можно проверить, есть ли там искомый ключ. Теоретически поиск в сегменте должен быть быстрым, потому что он содержит очень небольшое количество ключей. Это случается, если ваш метод GetHashCode возвращает достаточно уникальное значение для экземпляров вашего объекта, поддерживающего семантику эквивалентности значений.

С учетом сказанного, вы можете видеть, что будет очень плохо, если ваш алгоритм вычисления хеш-кода будет возвращать разные значения для двух экземпляров, содержащих эквивалентные значения. В таком случае хеш-таблица не сможет найти сегмент, в котором находится искомый ключ. По этой причине нужно обязательно переопределять GetHashCode, если вы переопределили Equals для объекта. Фактически, если вы переопределите только Equals, но не GetHashCode, то компилятор C# сообщит вам об этом в дружественном предупреждении. А поскольку мы стараемся строить код нашей версии с нулевым количеством предупреждений, придется отнестись к предупреждению компилятора серьезно.

На заметку! Предыдущая дискуссия со всей очевидностью должна показать, что любой тип, используемый в качестве ключа хеш-таблицы, должен быть неизменным. В конце концов, значение GetHashCode обычно вычисляется на основе состояния самого объекта. Если это состояние меняется, то результат GetHashCode, вероятно, изменится вместе с ним.

Реализации GetHashCode должны подчиняться перечисленным ниже правилам.

- Если для двух экземпляров x.Equals(y) равно true, то x.GetHashCode() == y.GetHashCode().
- Хеш-коды, сгенерированные GetHashCode, не обязаны быть уникальными.
- В GetHashCode не разрешено генерировать исключения.

Если два экземпляра возвращают одинаковые значения хеш-кода, они должны быть далее сравнены методом Equals, чтобы определить, эквивалентны ли они. Кстати, если ваш метод GetHashCode очень эффективен, вы можете построить на нем реализации operator!= и operator==, поскольку разные хеш-коды объектов одного типа предполагают отсутствие эквивалентности. Такая реализация операций в некоторых случаях может оказаться более эффективной, но все зависит от эффективности вашей реализации GetHashCode и сложности метода Equals. В некоторых случаях при использовании этого приема вызовы операций могут оказаться менее эффективными, чем простой вызов Equals, а в других случаях они могут быть заметно более эффективными. Например, рассмотрим объект, который моделирует многомерную точку в пространстве. Предположим, что количество из-

мерений (ранг) этой точки может приближаться к сотням. Внутренне вы можете представить измерения точки, используя массив целых чисел. Скажем, вы хотите реализовать метод GetHashCode, вычисляя код CRC32 точек измерений в массиве. Это также подразумевает, что тип Point является неизменным. Вызов GetHashCode потенциально может оказаться дорогостоящим, если вычислять CRC32 при каждом вызове. Поэтому может быть стоит заранее вычислить хеш-код и сохранить его в объекте. В таком случае вы можете написать операцию эквивалентности, как показано ниже.

```
sealed public class Point
{
    // Прочие методы опущены для ясности
    public override bool Equals( object other ) {
        bool result = false;
        Point that = other as Point;
        if( that != null ) {
            result = (this.coordinates == that.coordinates);
        }
        return result;
    }
    public override int GetHashCode() {
        return precomputedHash;
    }
    public static bool operator ==( Point pt1, Point pt2 ) {
        if( pt1.GetHashCode() != pt2.GetHashCode() ) {
            return false;
        } else {
            return Object.Equals( pt1, pt2 );
        }
    }
    public static bool operator !=( Point pt1, Point pt2 ) {
        if( pt1.GetHashCode() != pt2.GetHashCode() ) {
            return true;
        } else {
            return !Object.Equals( pt1, pt2 );
        }
    }
    private float[] coordinates;
    private int precomputedHash;
}
```

В этом примере до тех пор, пока предварительно вычисленный хеш достаточно уникален, перегруженные операции в некоторых случаях будут выполняться быстро. В худших случаях вместе с вызовами функции для их получения потребуется одно дополнительное сравнение двух целых чисел — хеш-значений. Если вызов Equals дорог, то такая оптимизация даст определенный выигрыш для большого количества сравнений. Если один вызов Equals не дорог, то такая техника добавляет накладных расходов и снижает эффективность кода. Лучше вспомнить старую истину, что преждевременная оптимизация — это плохая оптимизация. Такую оптимизацию вы должны применять только после того, как профилировщик направит вас в этом направлении, и вы уверены, что она должна помочь.

Метод Object.GetHashCode существует потому, что разработчики стандартной библиотеки решили, что будет удобно иметь возможность использовать любой объект в качестве ключа хеш-таблицы. На самом деле не все объекты — хорошие кандидаты в ключи хеша. Обычно лучше использовать в качестве таких ключей неизменяемые типы. Хорошим примером неизменяемого типа из стандартной библиотеки может служить System.String. Однажды созданный объект этого типа изменить невозможно. Поэтому вызов GetHashCode на экземпляре строки гарантированно всегда возвращает одно и то же значение для одного и того же экземпляра строки. Намного сложнее генерировать хеш-коды для изменяемых объектов. В таких случаях лучше основывать вашу реализацию GetHashCode на вычислениях, выполняемых по неизменяемым полям внутри изменяемого объекта.

Детальное описание алгоритмов генерирования хеш-кодов выходит за рамки этой книги. Я рекомендую обратиться к фундаментальному труду Дональда Кнута *Искусство программирования, том 3. Сортировка и поиск* (ИД "Вильямс", 2007 г.). Для примера предположим, что вы хотите реализовать GetHashCode для типа ComplexName. Одно из возможных решений заключается в вычислении хеша на основе величины (magnitude) комплексного числа, как показано в следующем примере:

```
using System;
public sealed class ComplexNumber
{
    public ComplexNumber( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public override bool Equals( object other ) {
        bool result = false;
        ComplexNumber that = other as ComplexNumber;
        if( that != null ) {
            result = (this.real == that.real) &&
            (this.imaginary == that.imaginary);
        }
        return result;
    }
    public override int GetHashCode() {
        return (int) Math.Sqrt( Math.Pow(this.real, 2) *
                                Math.Pow(this.imaginary, 2) );
    }
    public static bool operator ==( ComplexNumber num1, ComplexNumber num2 ) {
        return Object.Equals(num1, num2);
    }
    public static bool operator !=( ComplexNumber num1, ComplexNumber num2 ) {
        return !Object.Equals(num1, num2);
    }
    // Прочие методы опущены для ясности
    private readonly double real;
    private readonly double imaginary;
}
```

Предложенный алгоритм GetHashCode не является образцом высокой эффективности. На самом деле он вообще не эффективен, поскольку основан на нетри-

виальных математических процедурах с плавающей точкой. К тому же округление потенциально может привести к тому, что много комплексных чисел попадут в один сегмент хеша. В этом случае эффективность хеш-таблицы деградирует. Я предлагаю читателям в качестве упражнения реализовать более эффективный алгоритм самостоятельно. Обратите внимание, что я не использую метод `GetHashCode` для реализации `operator!=` из соображений эффективности. Но что более важно, я полагаюсь на статический метод `Object.Equals` для сравнения их на предмет эквивалентности. Этот удобный метод проверяет ссылки на равенство `null` перед вызовом экземпляра `Equals`, избавляя вас от необходимости делать это. Если бы я использовал `GetHashCode` для реализации `operator!=`, мне пришлось бы проверять ссылки на `null` перед вызовом `GetHashCode` на них. Также заметьте, что оба поля, используемые для вычисления хеш-кода, являются неизменными. Поэтому экземпляр этого объекта всегда возвращает одно и то же значение хеш-кода на протяжении всего его существования. Фактически вы можете применить кэширование хеш-кода, однажды вычислив его, чтобы добиться повышенной эффективности.

Поддерживает ли объект упорядочивание?

Иногда вам придется проектировать класс объектов, предназначенных для хранения в коллекции. Когда объекты в коллекции должны быть отсортированы, например, вызовом `Sort` на `ArrayList`, вам понадобится четко определенный механизм для сравнения двух объектов. Шаблон, предложенный разработчиками базовой библиотеки классов, предусматривает реализацию такими объектами следующего интерфейса `IComparable`[5]:

```
public interface IComparable
{
    int CompareTo( object obj );
}
```

Мы опять видим интерфейс с единственным методом. К счастью, `IComparable` не таит в себе такой глубины и ловушек, как `ICloneable` и `IDisposable`. Метод `CompareTo` достаточно прямолинеен. Он может возвратить положительное, отрицательное или нулевое значение. В табл. 13.1 описан смысл возвращаемых значений.

Таблица 13.1. Смысл возвращаемых значений метода IComparable.CompareTo

Возвращаемое значение CompareTo	Смысл
Положительное	`this > obj`
Нулевое	`this == obj`
Отрицательное	`this < obj`

При реализации `IComparable.CompareTo` нужно помнить о нескольких моментах. Во-первых, обратите внимание, что спецификация возвращаемого значения ничего не сообщает о действительном значении возвращенного целого числа. Определен только знак возвращаемых значений. Поэтому для того, чтобы обозна-

[5] Для типов значений рассмотрите возможность применения обобщенного интерфейса `IComparable<T>`, как показано в главе 10.

чить ситуацию, когда `this` меньше `obj`, вы можете просто вернуть `-1`. Когда ваш объект представляет значение, которое имеет целочисленный смысл, то эффективный способ вычисления возвращаемого значения состоит в вычитании одного из другого. Может возникнуть соблазн трактовать возвращаемое значение как степень неравенства. Хотя это возможно, я не рекомендую подобное, поскольку это выходит за рамки спецификации `IComparable`, и не от всех объектов можно такого ожидать.

Во-вторых, имейте в виду, что `CompareTo` не предусматривает никакого определенного возвращаемого значения, когда два объекта не могут быть сравнены. Поскольку типом параметра `CompareTo` является `System.Object`, вы легко можете попытаться сравнить экземпляр `Apple` с `Orange`. В таких случаях сравнение невозможно, и вы должны указать на это, сгенерировав исключение `ArgumentException`.

И, наконец, семантически интерфейс `IComparable` является надмножеством `Object.Equals`. Если вы осуществляете наследование от класса, который переопределяет `Equals` и реализует `IComparable`, то разумно будет и переопределить `Equals`, и заново реализовать `IComparable` в вашем производном классе, или же не делать ни того, ни другого. Следует убедиться в том, что ваши реализации `Equals` и `CompareTo` согласованы друг с другом.

На основе вышесказанного можно сделать вывод, что совместимый с `IComparable` интерфейс должен подчиняться перечисленным ниже правилам.

- `x.CompareTo(x)` должно возвращать `0`. Это рефлексивное свойство.
- Если `x.CompareTo(y) == 0`, то и `y.CompareTo(x)` должно быть равно `0`. Это — свойство симметричности.
- Если `x.CompareTo(y) == 0` и `y.CompareTo(z) == 0`, то и `x.CompareTo(z)` должно быть равно `0`. Это — свойство транзитивности.
- Если `x.CompareTo(y)` возвращает значение, отличное от `0`, то `y.CompareTo(x)` также должно возвращать ненулевое значение с противоположным знаком. Другими словами, это положение говорит: если `x < y`, то `y > x`, или если `x > y`, то `y < x`.
- Если `x.CompareTo(y)` возвращает значение, отличное от `0`, и `y.CompareTo(z)` возвращает значение, отличное от `0`, с тем же знаком, что и первое, то `x.CompareTo(y)` должно вернуть ненулевое значение с тем же знаком, что и предыдущие два. Другими словами, это означает, что если `x < y` и `y < z`, то `x < z`, или если `x > y` и `y > z`, то и `x > z`.

В следующем коде показана модифицированная форма класса `ComplexNumber`, реализующего `IComparable`, и консолидирующая тот же код в приватных вспомогательных методах:

```
using System;
public sealed class ComplexNumber : IComparable
{
    public ComplexNumber( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
```

```
    public override bool Equals( object other ) {
        bool result = false;
        ComplexNumber that = other as ComplexNumber;
        if( that != null ) {
            result = InternalEquals( that );
        }
        return result;
    }

    public override int GetHashCode() {
        return (int) this.Magnitude;
    }

    public static bool operator ==( ComplexNumber num1, ComplexNumber num2 ) {
        return Object.Equals(num1, num2);
    }

    public static bool operator !=( ComplexNumber num1, ComplexNumber num2 ) {
        return !Object.Equals(num1, num2);
    }

    public int CompareTo( object other ) {
        ComplexNumber that = other as ComplexNumber;
        if( that == null ) {
            throw new ArgumentException( "Неверное сравнение!" );
        }
        int result;
        if( InternalEquals(that) ) {
            result = 0;
        } else if( this.Magnitude > that.Magnitude ) {
            result = 1;
        } else {
            result = -1;
        }
        return result;
    }

    private bool InternalEquals( ComplexNumber that ) {
        return (this.real == that.real) &&
        (this.imaginary == that.imaginary);
    }
    public double Magnitude {
        get {
            return Math.Sqrt( Math.Pow(this.real, 2) +
                              Math.Pow(this.imaginary, 2) );
        }
    }
    // Прочие методы опущены для ясности.
    private readonly double real;
    private readonly double imaginary;
}
```

Является ли Object форматируемым?

Когда вы создаете новый объект, или экземпляр типа значения, он наследует от System.Object метод по имени ToString. Этот метод не принимает параметров и просто возвращает строковое представление объекта. Во всех случаях, если вызов ToString на вашем объекте имеет смысл, следует переопределить этот метод. Реализация по умолчанию, предоставленная System.Object, просто возвращает строковое представление имени типа объекта, что, конечно же, не слишком полезно для объекта, от которого требуется строковое представление, основанное на его внутреннем состоянии. Вы всегда должны переопределять Object.ToString для всех ваших типов, даже только для удобства вывода состояния объекта в отладочный выходной протокол.

Object.ToString удобен для получения быстрого строкового представления объекта; однако иногда этого не достаточно. Например, рассмотрим предыдущий пример ComplexNumber. Предположим, что вы хотите переопределить ToString для этого класса. Очевидная реализация должна выводить комплексное число как упорядоченную пару внутри пары скобок, такую, например, как "(1, 2)". Однако реальная и воображаемая составляющие ComplexType имеют тип double. К тому же числа с плавающей точкой не всегда представляются одинаково в разных культурах. Американцы используют точку для отделения дробной части числа с плавающей точкой, в то время как европейцы применяют запятую. Эта проблема легко решается, если вы используете информацию о культуре по умолчанию, ассоциированной с потоком. Обращаясь к свойству System.Threading.Thread.CurrentThread.CurrentCilture, вы можете получить ссылки на информацию о культуре по умолчанию, детализирующую представление числовых величин, включая денежные суммы, а также информацию о том, как представляются значения даты и времени.

На заметку! Тема глобализации и информации о культурах раскрывается в главе 8.

По умолчанию свойство CurrentCulture предоставляет вам доступ к System.Globalization.DateTimeFormatInfo и System.Globalization.NumberFormatInfo. Используя информацию, представленную в этих объектах, вы можете вывести ComplexNumber в форме, отвечающей культуре по умолчанию, установленной на машине, на которой работает ваше приложение. Загляните в главу 8, чтобы увидеть пример того, как это работает.

Это решение кажется довольно легким. Однако вы должны понимать, что бывают случаи, когда использовать установки текущей культуры недостаточно, и пользователь вашего объекта может пожелать самостоятельно указывать желаемую культуру. Мало того, пользователь может пожелать специфицировать точный формат вывода. Например, он может предпочесть, чтобы реальная и мнимая части ComplexNumber отображались только с пятью значимыми цифрами, используя культурную информацию Германии. Если вы разрабатываете программное обеспечение для сервера, вы знаете, насколько необходима такая возможность. Компании, которые эксплуатируют сервер финансовых служб в Соединенных Штатах, и обслуживают запросы из Японии, хотят отображать символ японской валюты в формате, принятом в японской культуре. Вам нужно как-то специфи-

цировать, как форматировать объект, когда он преобразуется в строку методом ToString, не изменяя предварительно CurrentCulture текущего потока.

Фактически стандартная библиотека предоставляет интерфейс для решения этой задачи. Когда структура или класс должен отвечать таким требованиям, он реализует интерфейс IFormattable. Следующий код демонстрирует довольно просто выглядящий интерфейс IFormattable. Однако не позволяйте себя обмануть его кажущейся простотой, потому что в зависимости от сложности вашего объекта, реализовать его может быть непросто:

```
public interface IFormattable
{
    string ToString( string format, IFormatProvider formatProvider );
}
```

Давайте сначала рассмотрим второй параметр. Если клиент передает null для formatProvider, вы должны по умолчанию использовать культурную информацию, ассоциированную с текущим потоком, как это было показано ранее. Однако если параметр formatProvider не равен null, вы должны получить информацию о форматировании от поставщика через метод IFormatProvider.GetFormat. Интерфейс IFormatProvider выглядит так:

```
public interface IFormatProvider
{
    object GetFormat( Type formatType );
}
```

Чтобы эти усилия были как можно более обобщены, 'дизайнеры стандартной библиотеки спроектировали GetFormat так, чтобы он принимал объект типа System.Type. Таким образом, он является расширяемым в отношении типов поддерживаемых объектов, реализующих IFormatProvider. Эта гибкость удобна, если вы намерены разрабатывать собственные поставщики формата, которые должны возвращать какую-то, пока еще не определенную информацию формата.

Стандартная библиотека представляет тип System.Globalization.CultureInfo, который, скорее всего, удовлетворит все ваши нужды. Объект CultureInfo реализует интерфейс IFormatProvider, и вы можете передавать его экземпляры в качестве второго параметра IFormattable.ToString. Скоро я покажу пример его использования, внеся модификации в пример ComplexNumber, но сначала взглянем на первый параметр ToString.

Параметр формата ToString позволяет вам специфицировать способ форматирования определенного числа. Поставщик формата может описать, как отображать дату, или как отображать валюту на основе культурных предпочтений, но первым делом вы должны знать, как форматировать объект. Все типы из стандартной библиотеки, такие как Int32, поддерживают стандартные спецификаторы формата, описанные в разделе "Standard Numeric Format Strings" библиотеки MSDN. В основе своей строка формата состоит из единственной буквы, специфицирующей формат, сопровождаемой необязательным числом от 0 до 99, которое указывает точность. Например, вы можете специфицировать, что тип double должен отображаться как пятизначное число с плавающей точкой, указав F5. Не все типы обязаны поддерживать все форматы, за исключением формата G, который означает "general" (общий). Фактически формат G — это то, что вы получаете, вызывая вер-

сию Object.ToString без параметров на большинстве объектов из стандартной библиотеки. Некоторые типы игнорируют спецификаторы формата в специальных случаях. Например, System.Double может содержать специальные значения, представляющие NaN (Not a Number — не число), PositiveInfinity (положительная бесконечность) или NegativeInfinity (отрицательная бесконечность). В таких случаях System.Double игнорирует спецификацию формата и отображает символ, соответствующий культуре, как указывает NumberFormatInfo.

Спецификатор формата может также состоять из пользовательских строк формата. Пользовательские форматные строки позволяют пользователю специфицировать точную компоновку чисел, смешивая со строчными литералами и т.п., используя синтаксис, описанный в библиотеке MSDN в разделе "Custom Numeric Format String". Клиент может специфицировать один формат для отрицательных чисел, другой — для положительных, а третий — для нулевых значений. Я не хочу тратить много времени на подробное описание этих разнообразных возможностей форматирования. Вместо этого советую обратиться к материалам MSDN, чтобы получить детальную информацию на эту тему.

Как вы можете видеть, реализация IFormattable.ToString может быть довольно утомительной, особенно в связи с тем, что ваша форматная строка может быть в высшей степени настраиваемой. Однако во многих случаях — и пример ComplexNumber один из них — вы можете положиться на реализации IFormattable стандартных типов. Поскольку ComplexNumber использует System.Double для представления реальной и мнимой частей, вы можете возложить большую часть вашей работы по реализации IFormattable на System.Double. Давайте рассмотрим модификации примера ComplexNumber, необходимые для поддержки IFormattable. Предположим, что тип ComplexNumber будет принимать форматную строку в точности так же, как это делает System.Double, и что каждый компонент комплексного числа будет выведен с использованием этого формата. Конечно, лучшая реализация может предоставлять больше возможностей вроде спецификации того, должен ли вывод быть представлен в декартовом или полярном формате, но я оставлю это вам в качестве упражнения.

```
using System;
using System.Globalization;
public sealed class ComplexNumber : IFormattable
{
    public ComplexNumber( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public override string ToString() {
        return ToString( "G", null );
    }
    // Реализация IFormattable.
    public string ToString( string format,
        IFormatProvider formatProvider ) {
            string result = "(" +
                            real.ToString(format, formatProvider) +
                            " " +
                            real.ToString(format, formatProvider) +
                            ")";
```

```
        return result;
    }
    // Прочие методы опущены для ясности.
    private readonly double real;
    private readonly double imaginary;
}
public sealed class EntryPoint
{
    static void Main() {
        ComplexNumber num1 = new ComplexNumber( 1.12345678,
                                                2.12345678 );
        Console.WriteLine( "Формат US: {0}",
                           num1.ToString( "F5",
                                          new CultureInfo("en-US") ) );
        Console.WriteLine( "Формат DE: {0}",
                           num1.ToString( "F5",
                                          new CultureInfo("de-DE") ) );
        Console.WriteLine( "Object.ToString(): {0}",
                           num1.ToString() );
    }
}
```

Так выглядит вывод этого примера:

```
Формат US: (1.12346 2.12346)
Формат DE: (1,12346 2,12346)
Object.ToString(): (1.12345678 2.12345678)
```

В `Main` обратите внимание на создание и использование разных экземпляров `CultureInfo`. Первый `ComplexNumber` выводится с использованием форматирования культуры American, а второй — German. В обоих случаях я специфицирую вывод только пяти значимых разрядов.

Вы увидите, что реализация `IFormattable.ToString` типа `System.Double` даже округляет значение, как ожидается. И, наконец, вы можете видеть, что переопределение `Object.ToString` реализовано так, что поручает форматирование методу `IFormattable.ToString` с общим форматом G.

`IFormattable` предоставляет клиентам ваших объектов мощные возможности, когда у них возникают специфические потребности в форматировании ваших объектов. Однако эта мощь достигается ценой реализации. Реализация `IFormattable.ToString` может быть очень детально-ориентированной задачей, которая потребует много времени и внимания.

Является ли `Object` преобразуемым?

Компилятор C# предоставляет поддержку преобразования экземпляров простых встроенных типов значений, таких как `int` и `long`, из одного типа в другой через приведение, генерируя код IL, который использует IL-инструкцию `conv`. Инструкция `conv` хорошо работает с простыми встроенными типами, но что делать, если вы хотите конвертировать строку в целое число и наоборот? Компилятор не может сделать этого автоматически, поскольку такое преобразование является потенциально сложным и даже требует параметров наподобие информации о культуре.

.NET Framework предлагает несколько способов выполнения такой работы. Для нетривиальных преобразований, которые невозможно выполнить простым приведением, вы должны полагаться на класс System.Convert. Я не стану перечислять здесь все функции, реализованные классом Convert, поскольку этот список очень большой. Как всегда, советую вам заглянуть в библиотеку MSDN. Класс Convert содержит методы для преобразования почти любого встроенного типа в любой другой, если только такое преобразование имеет смысл. Поэтому, если вы хотите преобразовать double в String, вам достаточно просто вызвать статический метод ToString, передав ему double, как показано ниже:

```
static void Main()
{
    double d = 12.1;
    string str = Convert.ToString( d );
}
```

Подобно IFormattable.ToString, Convert.ToString имеет различные перегрузки, которые также позволяют вам передавать объект CultureInfo или любой другой объект, поддерживающий IFormatProvider, чтобы специфицировать информацию о культуре при выполнении преобразования. Вы можете использовать также такие методы, как ToBoolean и ToUint32. Общий шаблон имен методов — очевидно, ToXXX, где XXX — тип, к которому выполняется преобразование. System.Convert даже имеет методы для преобразования байтовых массивов к закодированным base64 строкам и обратно. Если вы храните двоичные данные в тексте XML или любом другом текстовом носителе, то сочтете эти методы очень удобными.

Convert удовлетворит большинство ваших потребностей в преобразованиях между встроенными типами. Это — универсальный инструмент для преобразования объекта одного типа в другой. На это указывает богатство методов, которые он поддерживает. Однако что случится, если вам понадобится преобразовать тип, о котором Convert ничего не знает? Ответ кроется в методе Convert.ChangeType.

ChangeType — это механизм расширения System.Convert. Он имеет несколько перегрузок, включая некоторые, принимающие поставщика формата для культурной информации. Однако общая идея состоит в том, что он принимает объектную ссылку и преобразует ее в тип, представленный переданным объектом System.Type. Рассмотрим следующий код, использующий ComplexNumber из предыдущих примеров и пытающийся конвертировать строку с помощью System.Convert.ChangeType:

```
using System;
public sealed class ComplexNumber
{
    public ComplexNumber( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Прочие методы опущены для ясности.
    private readonly double real;
    private readonly double imaginary;
}
```

```
public sealed class EntryPoint
{
    static void Main() {
       ComplexNumber num1 = new ComplexNumber( 1.12345678, 2.12345678 );

       string str =
          (string) Convert.ChangeType( num1, typeof(string) );
    }
}
```

Вы увидите, что код нормально компилируется. Однако столкнетесь с сюрпризом во время выполнения, когда обнаружите, что он генерирует исключение с сообщением "Object must implement `IConvertible`" (Объект должен реализовать `IConvertible`). Несмотря на то что `ChangeType` — это механизм расширения `System.Convert`, такая расширяемость не дается даром. Вам придется предпринять определенные усилия, чтобы заставить `ChangeType` работать с `ComplexNumber`. И, как вы, возможно, догадываетесь, эти усилия заключаются в реализации интерфейса `IConvertable`.

Интерфейс `IConvertable` — последнее средство, когда речь идет о преобразовании объектов. Если вы хотите, чтобы ваши пользовательские объекты хорошо работали с `System.Convert` и другими типами, преобразование которых может понадобиться, то вам лучше реализовать `IConvertable`. Как и в отношении `System.Convert`, я не стану перечислять здесь методы `IConvertable`, хотя их и не так много. Советую вам заглянуть в документацию MSDN. Вы увидите один метод для преобразования каждого из встроенных типов. Вдобавок `Convert` использует универсальный метод `IConvertable.ToType` для преобразования одного пользовательского типа в другой пользовательский тип. К тому же методы `IConvertable` принимают поставщика формата, чтобы можно было указать культурную информацию методу преобразования.

Напомню, что реализуя интерфейс, вы должны представить реализации всех его методов. Однако если конкретное преобразование не имеет смысла для вашего объекта, то вы можете сгенерировать исключение `InvalidCastException`. Естественно, ваша реализация, скорее всего, будет генерировать исключение внутри `IConvertible.ToType` для любого обобщенного типа, преобразование к которому не поддерживается.

Чтобы подытожить, скажу, что может показаться, что существует несколько способов преобразования одного типа в другой в C#, и так оно и есть. Однако главное эмпирическое правило заключается в том, что нужно полагаться на `System.Convert`, когда простое приведение не работает. Более того, ваши пользовательские объекты вроде класса `ComplecNumber` должны реализовать `IConvertable`, чтобы иметь возможность работать в сочетании с классом `System.Convert`.

На заметку! C# представляет операции преобразования, позволяющие вам делать по сути то же самое, что позволяет реализация `IConvertable`. Однако явные и неявные операции преобразования C# не являются совместимыми с CLS. Поэтому не каждый язык, потребляющий ваш код C#, сможет вызывать их для выполнения преобразований. Не рекомендуется полагаться исключительно на них в выполнении преобразований. Конечно, если ваш проект разработан только на языках .NET, поддерживающих операции преобразования, то вы можете свободно их использовать, но при этом также рекомендуется поддерживать `IConvertable`.

.NET Framework предлагает еще один тип механизма преобразования, который работает через System.ComponentModel.TypeConverter. Это еще один конвертер, внешний по отношению к классу преобразуемого экземпляра, как и System.Convert. Преимущество использования TypeConverter заключается в том, что вы можете применять его как во время проектирования, внутри IDE, так и во время выполнения. Вы создаете собственный специальный тип конвертера для вашего класса, который наследуется от TypeConverter, и затем ассоциируете этот ваш новый тип конвертера с вашим классом через атрибут TypeConverterAttribute. Во время проектирования IDE может просмотреть метаданные вашего типа, и на основании информации, почерпнутой оттуда, создать экземпляр вашего типа конвертера. Таким образом, IDE может конвертировать ваш тип в нужное представление и обратно, которое подходит для использования. Я не стану углубляться в детали создания наследников TypeConverter, но если вам нужна дополнительная информация, обратитесь к разделу "Generalized Type Conversion" документации MSDN.

Всегда отдавайте предпочтение безопасности типов

Вы уже знаете, что C# — строго типизированный язык. Строго типизированный язык и его компилятор формируют динамический дуэт, способный выявлять ошибки до их проявления. Даже несмотря на то, что каждый объект в управляемом мире унаследован от System.Object, будет очень плохой идеей трактовать каждый объект обобщенно, через ссылку на System.Object. Одна причина — эффективность; например, если вам нужно сопровождать коллекцию объектов Employee через ссылки на System.Object, вам всегда придется осуществлять приведение их экземпляров к типу Employee, прежде чем вы сможете вызвать метод класса Employee. Хотя эта проблема неэффективности не так серьезна, когда используются ссылочные типы и приведение выполняется успешно, она значительно усугубляется, когда дело касается типов значений, поскольку в этом случае требует излишних операции упаковки, генерируемых в коде IL. Я еще вернусь к теме неэффективности упаковки в следующих разделах, посвященных типам значений. Самая большая проблема со всеми этими приведениями при работе со ссылочными типами проявляется тогда, когда приведение завершается неудачей, и происходит генерация исключения. Используя строгие типы, вы можете перехватывать эти проблемы и справляться с ними еще на этапе компиляции.

Другая наглядная причина предпочесть использование строгих типов связана с перехватом ошибок. Рассмотрим случай реализации интерфейсов, таких как ICloneable. Обратите внимание, что метод Clone возвращает экземпляр типа Object. Ясно, что это делается для того, чтобы интерфейс мог работать со всеми типами. Однако за это приходится платить определенную цену.

И C++, и C# — строго типизированные языки. Каждая переменная объявляется со своим типом. Это обеспечивает безопасность типов, на страже которой стоит компилятор, помогая вам избегать ошибок. Например, он предотвращает присваивание экземпляру класса Apple значение экземпляра класса MonkeyWrench. Однако C# (и C++) позволяет вам работать в менее безопасной к типам манере. Вы можете ссылаться на каждый объект через тип Object. Однако, поступая так, вы полностью отбрасываете всю безопасность типов, и тогда компилятор позволяет вам выполнять присваивание экземпляру типа Apple значение экземпляра класса MonkeyWrench — до тех пор, пока обе ссылки имеют тип Object. К сожалению, не-

смотря на то, что такой код компилируется, вы рискуете получить ошибку времени выполнения, когда CLR станет выполнять этот код и обнаружит глупость, которую вы пытаетесь провернуть. Поэтому чем больше вы используете средства контроля безопасности типов компилятора, тем больше ошибок можно обнаружить во время компиляции, а перехват ошибок на этапе компиляции *всегда* более желателен, чем их обнаружение во время выполнения.

Давайте присмотримся к аспекту эффективности этой проблемы. Обобщенная трактовка объектов обычно наносит ущерб эффективности во время выполнения, когда вам приходится осуществлять приведение к действительному типу. В действительности этот ущерб эффективности очень невелик, когда речь идет об управляемых ссылочных типах C#, если только вы не делаете этого много раз в цикле.

В некоторых ситуациях компилятор C# генерирует намного более эффективный код, если вы предложите ему безопасную в отношении типов реализацию хорошо определенного метода. Рассмотрим типичный оператор foreach на C#:

```
foreach( Employee emp in collection ) {
    // Делать что-то
}
```

Все достаточно просто: код проходит циклом по всем элементам collection. Внутри тела оператора foreach переменная emp типа Employee ссылается на текущий элемент коллекции во время итерации. Одно из правил, устанавливаемым компилятором C# для коллекций, гласит о том, что они должны реализовать общедоступный метод GetEnumerator, который возвращает тип, используемый для перечисления элементов коллекции. Этот метод реализован как результат того, что тип коллекции реализует интерфейс IEnumerable, и обычно возвращает прямой итератор на коллекции содержащихся объектов[6]. Одно из правил для типа перечислителя гласит, что он должен реализовать общедоступное свойство по имени Current, которое позволяет получить доступ к текущему элементу. Это свойство — часть интерфейса IEnumerator; однако заметьте, что тип IEnumerator.Current — это System.Object. Отсюда вытекает другое правило, касающееся оператора foreach. Оно устанавливает, что тип объекта IEnumerator.Current — реальный тип объекта — должен быть неявно преобразуем к типу итератора в операторе foreach, которым в данном случае является Employee.

Итак, что же вы можете сделать, чтобы исправить эту ситуацию в мире C#? По сути, всякий раз, когда вы реализуете интерфейс, содержащий методы с безтиповыми возвращаемыми значениями, рассмотрите возможность явной реализации интерфейса, чтобы скрыть метод от общедоступного интерфейса класса, в то же время реализуя более безопасную к типам версию, как часть общедоступного интерфейса класса. Давайте взглянем на пример использования интерфейса IEnumerator:

```
using System;
using System.Collections;
public class Employee
{
```

[6] Я использовал слово *обычно*, поскольку итераторы могут быть обратными. В главе 9 я показал, как вы можете легко создавать обратные и двунаправленные итераторы, реализующие IEnumerator.

```
    public void Evaluate() {
        Console.WriteLine( "Проверка Employee..." );
    }
}
public class WorkForceEnumerator : IEnumerator
{
    public WorkForceEnumerator( ArrayList employees ) {
        this.enumerator = employees.GetEnumerator();
    }
    public Employee Current {
        get {
            return (Employee) enumerator.Current;
        }
    }
    object IEnumerator.Current {
        get {
            return enumerator.Current;
        }
    }
    public bool MoveNext() {
        return enumerator.MoveNext();
    }
    public void Reset() {
        enumerator.Reset();
    }
    private IEnumerator enumerator;
}
public class WorkForce : IEnumerable
{
    public WorkForce() {
        employees = new ArrayList();
        // Вставим сюда сотрудника в целях демонстрации
        employees.Add( new Employee() );
    }
    public WorkForceEnumerator GetEnumerator() {
        return new WorkForceEnumerator( employees );
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return new WorkForceEnumerator( employees );
    }
    private ArrayList employees;
}
public class EntryPoint
{
    static void Main() {
        WorkForce staff = new WorkForce();
        foreach( Employee emp in staff ) {
            emp.Evaluate();
        }
    }
}
```

Посмотрите внимательно на этот пример и отметьте, как безтиповые версии методов интерфейса реализуются явно. Напомню, что для того, чтобы обратиться к этим методам, сначала вы должны выполнить приведение экземпляра к типу интерфейса. Однако компилятор не делает этого, когда генерирует цикл `foreach`. Вместо этого он просто ищет методы, которые подчиняются упомянутым выше правилам[7]. Так он находит строго типизированные версии и использует их. Я настоятельно рекомендую пройтись отладчиком по этому коду, чтобы увидеть его в действии. Фактически, эти типы даже не обязаны реализовывать интерфейсы, которые они реализуют, а именно — `IEnumerable` и `IEnumerator`. Вы можете закомментировать имена интерфейсов и просто реализовать их методы, чтобы соответствовали сигнатурам этих интерфейсов. К тому же вы можете заметно повысить эффективность этого кода, используя обобщения, о которых говорилось в главе 11.

Давайте внимательней присмотримся к циклу `foreach`, сгенерированному компилятором, чтобы лучше понять, о какого рода эффективности идет речь. В следующем коде я удалил строго типизированные версии методов интерфейса и, как ожидалось, с внешней точки зрения пример работает так же, как и раньше:

```
using System;
using System.Collections;
public class Employee
{
   public void Evaluate() {
      Console.WriteLine( "Проверка Employee..." );
   }
}

public class WorkForceEnumerator : IEnumerator
{
   public WorkForceEnumerator( ArrayList employees ) {
      this.enumerator = employees.GetEnumerator();
   }
   public object Current {
      get {
         return enumerator.Current;
      }
   }
   public bool MoveNext() {
      return enumerator.MoveNext();
   }
   public void Reset() {
      enumerator.Reset();
   }
   private IEnumerator enumerator;
}
```

---

[7] Этот прием обычно называют *утиным следом* (duck typing). "Утиный след" — это стиль программирования, при котором тип реализует контракт или интерфейс, просто реализуя методы, определенные в контракте, вместо наследования от определенного типа или интерфейса. Дополнительную информацию ищите по адресу `http://en.wikipedia.org/wiki/Duck_typing`.

```
public class WorkForce : IEnumerable
{
   public WorkForce() {
      employees = new ArrayList();
      // Вставим сюда сотрудника в целях демонстрации.
      employees.Add( new Employee() );
   }
   public IEnumerator GetEnumerator() {
      return new WorkForceEnumerator( employees );
   }
   private ArrayList employees;
}
public class EntryPoint
{
   static void Main() {
      WorkForce staff = new WorkForce();
      foreach( Employee emp in staff ) {
         emp.Evaluate();
      }
   }
}
```

Конечно, сгенерированный код IL не настолько эффективен. Чтобы увидеть выигрыш эффективности в цикле `foreach`, вы должны загрузить скомпилированные версии каждого примера в ILDASM и открыть IL-код метода `Main`. Вы увидите, что слабо типизированный пример содержит дополнительные инструкции `castclass`, которых нет в строго типизированном примере. На моей машине для разработки я запустил цикл `foreach` 20 000 000 раз, чтобы протестировать производительность. Типизированная версия перечислителя оказалась на 15% быстрее нетипизированной. Это существенный выигрыш, если вы работаете над главным циклом очередной игры-бестселлера "Укрощение DirectX".

Использование неизменных ссылочных типов

При создании тщательно спроектированного контракта или интерфейса вы всегда должны учитывать изменчивость или неизменность типов, объявленных в этом контексте. Например, если у вас есть метод, который принимает параметр, вы должны задуматься — разрешено ли методу модифицировать этот параметр. Предположим, вы хотите гарантировать, что тело метода не сможет модифицировать параметр. Если параметр относится к типу значений и передан без ключевого слова `ref`, то метод принимает копию параметра, и вы гарантированы, что исходное значение не модифицируется. Однако для ссылочных типов все намного сложнее, поскольку копируются только ссылки, а не объекты, на которые они ссылаются.

На заметку! Если вы пришли из мира C++, то знаете, что неизменность реализуется ключевым словом `const`. Следование этой технике означает соблюдение "константности". Даже если превосходство C++ кажется неоспоримым тем, кто жалуется на отсутствие `const` в C#, следует вспомнить, что в C++ вы всегда можете избавиться от требования константности, используя `const_cast`. Таким образом, реализация неизменности на самом деле превосходит ключевое слово C++ `const`, поскольку вы не можете просто отбросить ее.

Замечательным примером неизменного класса в стандартной библиотеке является System.String. Как только вы создали объект String, изменить его уже невозможно. Обойти это ограничение тоже никак нельзя: так уж спроектирован этот класс. Вы можете создавать копии, и эти копии могут быть модифицированной формой оригинала, но вы просто не можете изменить исходный экземпляр на протяжении его жизни, не прибегая е небезопасному (unsafe) коду. Если вы понимаете это, значит, вы ухватили суть того, что я пытаюсь вам донести: для того, чтобы гарантировать, что ссылочные объекты, переданные методу, не изменятся в вызове этого метода, они сами должны быть неизменными.

В таком мире, как CLR, где объекты по умолчанию доступны через ссылки, такое понятие о неизменности становится чрезвычайно важным. Давайте предположим на минуту, что System.String был бы изменяемым, и что вы могли бы написать нечто вроде следующего фиктивного метода:

```
public void PrintString( string theString )
{
    // Предположим, что следующая строка не создает новый
    // экземпляр String, а модифицирует theString.
    theString += ": эта строка печатается!";
    Console.WriteLine( theString );
}
```

Представьте себе испуг того, кто вызвал этот метод, когда после вызова он обнаружит, что к его строке добавлен этот новый кусок. Вот что случилось бы, будь System.String изменяемым. Как видите, неизменность System.String имеет свои причины, и возможно, вы решите использовать это и в своем дизайне.

Есть много способов решения проблемы const-параметров C# для объектов, которые должны быть изменяемыми. Одно общее решение состоит в создании двух классов для каждого изменяемого класса, созданного вами, если вы хотите, чтобы ваши клиенты могли передавать константную версию объекта в качестве параметра. Для примера давайте вернемся к нашему прежнему классу ComplexNumber. Будучи реализованным как объект, а не как тип значения, этот класс является блестящим кандидатом стать неизменным подобно String. В этом случае такие операции, как ComplexNumber.Add, должны были бы создавать новый экземпляр ComplexNumber вместо того, чтобы модифицировать объект, на который указывает ссылка this. Но давайте посмотрим, что вы можете сделать, позволив ComplexNumber быть изменяемым. Вы можете открыть доступ к действительной и мнимой части через свойства, допускающие чтение-запись. Но как можно передать объект методу и не допустить, чтобы метод изменил его через метод set одного из свойств? Один ответ, как это принято в объектно-ориентированном дизайне, заключается в приеме введения нового класса. Рассмотрим следующий код:

```
using System;
public sealed class ComplexNumber
{
    public ComplexNumber( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public double Real {
        get {
```

```
            return real;
        }
        set {
            real = value;
        }
    }
    public double Imaginary {
        get {
            return imaginary;
        }
        set {
            imaginary = value;
        }
    }
    // Прочие методы опущены для ясности.
    private double real;
    private double imaginary;
}
public sealed class ConstComplexNumber
{
    public ConstComplexNumber( ComplexNumber pimpl ) {
        this.pimpl = pimpl;
    }
    public double Real {
        get {
            return pimpl.Real;
        }
    }
    public double Imaginary {
        get {
            return pimpl.Imaginary;
        }
    }
    private readonly ComplexNumber pimpl[8];
}
public sealed class EntryPoint
{
    static void Main() {
        ComplexNumber someNumber = new ComplexNumber( 1, 2 );
        SomeMethod( new ConstComplexNumber(someNumber) );
        // Контракт ConstComplexNumber гарантирует
        // в этой точке неизменность someNumber.
    }
    static void SomeMethod( ConstComplexNumber number ) {
        Console.WriteLine( "( {0}, {1} )",
        number.Real,
        number.Imaginary );
    }
}
```

[8] Для любопытных, которым интересно, почему так названо поле: читайте об идиоме Pimpl в книге Херба Саттера (Herb Sutter) *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* (Boston, MA: Addison-Wesley Professional, 2000 г.).

Обратите внимание, что я ввел класс-прокладку по имени ConstComplexNumber. Когда метод желает принять объект ComplexNumber, но при этом гарантировать неизменность параметра, то он принимает ConstComplexNumber вместо ComplexNumber. Конечно, в случае ComplexNumber лучшим решением было бы сделать его сразу неизменным[9]. Но можно представить себе класс, намного более сложный, чем ComplexNumber (никаких каламбуров... правда!) который может потребовать применения подобной техники, чтобы гарантировать невозможность изменения параметра методом.

Как это часто бывает с проблемами программного дизайна, одной и той же цели можно достичь разными способами. Схема, которую я продемонстрировал — не единственный путь решения проблемы. Того же можно добиться с помощью интерфейсов.

Вы могли бы определить интерфейс, объявляющий все методы, которые модифицируют объект, скажем, IModifiableComplexNumber, и другой интерфейс, объявляющий только не модифицирующие методы, например, IConstantComplexNumber. Затем вы могли создать третий интерфейс IComplexNumber, наследующий оба предыдущих, и, наконец, ComplexNumber мог бы реализовать интерфейс IComplexNumber. Методам, которые должны воспринимать параметр как неизменяемый, вы бы просто передавали экземпляр как тип IConstantComplexNumber.

Прежде чем применить этот прием для написания академического упражнения, найдите время, чтобы изучить и понять мощь концепции неизменности для устойчивого программного дизайна. Неспроста так много статей написано о корректности const в среде сообщества разработчиков C++. И ничто не мешает вам применять те же приемы в вашем дизайне C#.

Канонические формы типов значений

Исследуя канонические формы типов значений, вы обнаружите, что некоторые концепции, применимые к ссылочным типам, могут быть применены и здесь. Однако есть немало заметных отличий. Например, не имеет смысла реализация ICloneable для типа значений. Технически вы можете это сделать, но поскольку ICloneable возвращает экземпляр Object, ваша реализация ICloneable.Clone для типа значений вернет просто упакованную копию самого себя. Вы можете получить в точности то же поведение, выполнив простое приведение экземпляра типа значения к ссылке на System.Object, до тех пор, пока ваш тип значение не содержит в себе никаких ссылочных типов. Фактически, вы могли бы возразить, что типы значений, содержащие в себе изменяемые ссылочные типы — свидетельство плохого дизайна. Типы значений наиболее подходят для неизменяемых легковесных порций данных. Поэтому до тех пор, пока ссылочные типы, содержащиеся в ваших типах значений, являются неизменяемыми — например, как тип System.String — вам не нужно беспокоиться о реализации ICloneable для вашего типа значений. Если вы будете вынуждены реализовать ICloneable в вашем типе значений, присмотритесь к его дизайну. Возможно, ваш тип значения должен быть заменен ссылочным типом.

---

[9] Чтобы избежать такого сложного клубка, многие типы значений, определенные в .NET Framework, фактически являются неизменными.

Типы значений не нуждаются в финализаторе, и фактически C# не позволит вам создать финализатор для структуры в синтаксисе деструктора. Аналогично, типы значений не нуждаются в реализации интерфейса IDisposable, если только не содержат объектов по ссылке, реализующих IDisposable, или же не потребляют значительных системных ресурсов. В таких случаях нужно, чтобы типы значений реализовали IDisposable. Фактически вы можете применять оператор using с типами значений, реализующими IDisposable.

Совет. Поскольку типы значений не могут реализовывать финализаторы, они не могут гарантировать выполнение кода очистки в Dispose, если пользователь забудет вызвать его явно. Таким образом, объявления полей ссылочного типа внутри типа значений не рекомендуется. Если есть поле типа значений, которое требует вызова Dispose, ничто не гарантирует, что этот вызов будет выполнен.

Типы значений и ссылочные типы разделяют между собой много идиом реализации. Например, для обоих имеет смысл рассмотреть реализацию IComparable, IFormattable и, возможно, IConvertible.

В оставшейся части раздела я раскрою различные канонические концепции, которые вы должны применять при проектировании типов значений. В частности, если вы хотите переопределить Equals для повышения эффективности во время выполнения, то желаете знать, что это означает для типа значений — реализовать интерфейс. Давайте приступим к этому.

Переопределение Equals для повышения производительности

Вы уже видели основные отличия между двумя видами эквивалентности в CLR и C#. Так, например, вы знаете, что ссылочные типы (экземпляры классов) по умолчанию определяют эквивалентность как идентичность, а типы значений (экземпляры структур) используют эквивалентность значений. Ссылочные типы получают свою реализацию по умолчанию от Object.Equals, в то время как типы значений — от переопределения Equals из System.ValueType. Все типы структур неявно унаследованы от System.ValueType. Вы должны реализовать ваше собственное переопределение Equals для каждой структуры, которую определяете. Можно сравнивать поля вашего объекта более эффективно, поскольку вы знаете их типы, и что они представляют собой во время компиляции. Давайте обновим пример ComplexNumber из предыдущих разделов, преобразуя его в структуру и реализуя собственное переопределение Equals:

```
using System;
public struct ComplexNumber : IComparable
{
    public ComplexNumber( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public override bool Equals( object other ) {
        bool result = false;
        if( other is ComplexNumber ) {
            ComplexNumber that = (ComplexNumber) other ;
```

```
            result = InternalEquals( that );
        }
        return result;
    }

    public override int GetHashCode() {
        return (int) this.Magnitude;
    }

    public static bool operator ==( ComplexNumber num1,
                                    ComplexNumber num2 ) {
        return num1.Equals(num2);
    }
    public static bool operator !=( ComplexNumber num1,
                                    ComplexNumber num2 ) {
        return !num1.Equals(num2);
    }
    public int CompareTo( object other ) {
        if( !(other is ComplexNumber) ) {
            throw new ArgumentException( "Неверное сравнение!" );
        }
        ComplexNumber that = (ComplexNumber) other;
        int result;
        if( InternalEquals(that) ) {
            result = 0;
        } else if( this.Magnitude > that.Magnitude ) {
            result = 1;
        } else {
            result = -1;
        }
        return result;
    }
    private bool InternalEquals( ComplexNumber that ) {
        return (this.real == that.real) &&
               (this.imaginary == that.imaginary);
    }
    public double Magnitude {
        get {
            return Math.Sqrt( Math.Pow(this.real, 2) +
                              Math.Pow(this.imaginary, 2) );
        }
    }
    // Прочие методы опущены для ясности.
    private readonly double real;
    private readonly double imaginary;
}
public sealed class EntryPoint
{
    static void Main()
    {
        ComplexNumber num1 = new ComplexNumber( 1, 2 );
        ComplexNumber num2 = new ComplexNumber( 1, 2 );
        bool result = num1.Equals( num2 );
    }
}
```

Взглянув на код примера, вы увидите, что понадобились минимальные изменения по сравнению с версией ссылочного типа. Теперь тип объявлен как структура, а не класс, и также обратите внимание, что он поддерживает IComparable. У меня еще будет, что сказать о реализации интерфейсов структурами в разделе, озаглавленном "Поддерживают ли значения этого типа какие-либо интерфейсы?". Дотошный читатель может заметить, что эффективность этого варианта далека от совершенства. Дело, конечно, в упаковке и распаковке. Напомню, что каждый раз, когда экземпляр типа значения передается в виде объекта в списке параметров метода, он должен быть неявно упакован, если не был упакован ранее. Это значит, что когда метод Main вызывает метод Equals, он должен сначала упаковать значение num2. Что еще хуже — методу затем приходится распаковывать значение, чтобы использовать его. Таким образом, в процессе сравнения двух значений на эквивалентность, приходится дважды копировать одно из них.

Чтобы решить эту проблему, вы можете определить две перегрузки Equals. Вам нужна безопасная в отношении типов версия, принимающая ComplexNumber в качестве типа параметра, и вам также нужно переопределить метод Object.Equals, как и раньше.

На заметку! .NET Framework формализовал эту концепцию обобщенным интерфейсом IEquatable<T>, который объявляет один метод, являющийся безопасной в отношении типов версией Equals.

Посмотрим, как изменился код:

```
using System;
public struct ComplexNumber : IComparable,
                              IComparable<ComplexNumber>,
                              IEquatable<ComplexNumber>
{
    public ComplexNumber( double real, double imaginary ) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public bool Equals( ComplexNumber other ) {
        return (this.real == other.real) &&
               (this.imaginary == other.imaginary);
    }
    public override bool Equals( object other ) {
        bool result = false;
        if( other is ComplexNumber ) {
            ComplexNumber that = (ComplexNumber) other ;
            result = Equals( that );
        }
        return result;
    }
    public override int GetHashCode() {
        return (int) this.Magnitude;
    }
    public static bool operator ==( ComplexNumber num1,
                                     ComplexNumber num2 ) {
```

```
        return num1.Equals(num2);
    }
    public static bool operator !=( ComplexNumber num1,
                                    ComplexNumber num2 ) {
        return !num1.Equals(num2);
    }
    public int CompareTo( object other ) {
        if( !(other is ComplexNumber) ) {
            throw new ArgumentException( "Неверное сравнение!" );
        }
        return CompareTo( (ComplexNumber) other );
    }
    public int CompareTo( ComplexNumber that ) {
        int result;
        if( Equals(that) ) {
            result = 0;
        } else if( this.Magnitude > that.Magnitude ) {
            result = 1;
        } else {
            result = -1;
        }
        return result;
    }
    public double Magnitude {
        get {
            return Math.Sqrt( Math.Pow(this.real, 2) +
            Math.Pow(this.imaginary, 2) );
        }
    }
    // Прочие методы опущены для ясности.
    private readonly double real;
    private readonly double imaginary;
}
public sealed class EntryPoint
{
    static void Main()
    {
        ComplexNumber num1 = new ComplexNumber( 1, 2 );
        ComplexNumber num2 = new ComplexNumber( 1, 2 );
        bool result = num1.Equals( num2 );
    }
}
```

Теперь сравнение внутри Main стало более эффективным, поскольку отпала необходимость в упаковке значений. Компилятор выбирает наиболее подходящую из двух перегрузок, которой, конечно, является строго типизированная перегрузка Equals, принимающая ComplexNumber вместо обобщенного типа объекта. Внутренне Object.Equals переопределяет делегаты в безопасную к типам версию Equals после того, как проверит тип объекта и распакует его. Важно отметить, что Object.Equals переопределяет первые проверки типа, чтобы убедиться, что это ComplexNumber или, точнее, упакованный ComplexNumber, прежде чем распаковать его, чтобы избежать

генерации исключения. Документация по стандартной библиотеке ясно указывает, что перегрузки `Object.Equals` не должны генерировать исключений. И, наконец, заметьте, что для структур существует то же эмпирическое правило относительно `GetHashCode`, что и для классов. Если вы переопределите `Object.Equals`, то также должны переопределить и `Object.GetHashCode`, и наоборот.

Обратите внимание, что я также реализовал `IComparable<ComplexNumber>`, который использует ту же технику, что и `IEquatable<ComplexNumber>` для предоставления безопасной к типам версии `IComparable`. Вы всегда должны рассматривать реализацию этих обобщенных интерфейсов, чтобы у компилятора было больше свободы в обеспечении безопасности типов.

Поддерживают ли значения этого типа какие-либо интерфейсы?

Разница в поведении между типами значений и ссылочными типами внутри CLR иногда может служить причиной путаницы и головной боли, особенно для новичков в CLR и C#. Это обычно связано со сложностями взаимодействия этих двух миров — мира ссылочных типов и мира типов значений. Учтите тот факт, что все типы значений (структуры) неявно наследуются от `System.ValueType`. Также учтите, что `System.ValueType` наследуется от `System.Object`. Вы можете быть склонны думать, что можете просто выполнить приведение типа значений, такого как `ComplexNumber`, к `Object`, и тем самым преодолеть пропасть между миром типов значений и миром ссылочных типов. Это случится, но, наверно, совсем не так, как вы могли ожидать.

В действительности при этом CLR создаст новый объект, и этот новый объект содержит копию вашего типа значений. Вы уже видели эту концепцию, определенную в виду упаковки. "За кулисами", когда CLR встречает определение структуры или типа значений, она также внутренне определяет ссылочный тип, представляющий собой упаковку, о которой я уже говорил, когда шла речь об упаковочной операции. Вы не можете явно создать экземпляр этого типа, но делаете это при инициации операции упаковки на экземпляре значения.

Когда CLR создает этот внутренний тип упаковки во время выполнения, она использует рефлексию для реализации всех методов, реализованных вашим типом значений, а реализация метода просто перенаправляет вызовы, содержащейся в нем копии вашего типа значения. К тому же динамически сгенерированный упаковочный тип также реализует любые интерфейсы, реализуемые этим типом значений. Таким образом, ссылки на экземпляры этого динамического упаковочного типа, который является типом ссылочным, могут быть приведены к ссылкам на реализуемые интерфейсные типы, как это принять для ссылочных типов. Но как вы думаете, что случится, когда вы выполняете приведение экземпляр типа значения к интерфейсному типу? Ответ состоит в том, что сначала значений должно быть упаковано. Это имеет смысл, если учесть то, что ссылки на интерфейс — это всегда ссылки на ссылочный тип.

Вы уже знаете, какие неудобства несет с собой упаковка в C#. Это объясняется тем, что упаковка осуществляется автоматически, чтобы помочь вам выйти из затруднения. Но если вы не знаете, что происходит "за кулисами" этого процесса, это может привести к еще большей путанице, потому что вы можете неосторожно модифицировать значение внутри упаковки и затем отбросить его прочь, не рас-

пространив эти изменения обратно из упакованного значения на исходное значение. Ужас, не правда ли?

Можете ли вы представить себе способ, которым можно модифицировать значение, находящееся в упаковке? Если вы приведете экземпляр упаковки обратно к его типу значений, то получите новую копию значения в упаковке. Поэтому не пытайтесь выполнить этот трюк. Что вам нужно — так это добраться до внутреннего значения упаковки. Интерфейсы — не ответ. Как я уже сказал, внутренне созданный ссылочный тип упаковки, который вы никогда не видите, реализует все интерфейсы, которые реализует структура. Поскольку интерфейсные ссылки указывают на объекты, они могут модифицировать состояние значения внутри упаковки, если выполнять вызовы через интерфейс. Вы не можете модифицировать содержимое значения внутри упаковки иначе, кроме как через интерфейс.

В заключение важно отметить, что типы значений, реализующие интерфейсы, провоцируют неявную упаковку, если вы приводите один из этих типов к типу интерфейса, который он реализует. В то же время интерфейсы — единственный механизм, через который вы можете изменить значение внутри упаковки. Пример того, как это делается, ищите в разделе "Упаковка и распаковка" главы 4.

Реализация безопасных к типам форм членов интерфейса и унаследованных методов

Я уже раскрывал эту тему в контексте ссылочных типов в разделе "Всегда отдавайте предпочтение безопасности типов". Большинство ее положений применимы и к типам значений, но с учетом некоторых соображений эффективности. Проблемы эффективности происходят из явных операций преобразования от типов значений к ссылочным типам и наоборот. Как вы знаете, эти преобразования приводят к скрытым операциям упаковки и распаковки в генерированном коде IL. Операции упаковки могут легко уничтожить эффективность во многих ситуациях. Замечания, сделанные ранее относительно того, как безопасные к типам версии методов перечисления помогают компилятору C# создавать более эффективный код в цикле `foreach`, в десятикратной мере касаются типов значений. Это объясняется тем, что операции упаковки, вызванные преобразованиями к типам значений и обратно, требуют намного больше времени процессора по сравнению с приведением ссылочных типов, которое относительно быстрое.

Вы уже видели, как тип значения `ComplexNumber` реализует интерфейс — в данном случае `IComparable`. Это нужно для того, чтобы обеспечить возможность сортировки элементов этого типа при помещении их в контейнер. Вы также заметили, что основные типы CLR вроде `System.Int32` также поддерживают такие интерфейсы, как `IComparable`. Однако с точки зрения эффективности вам не нужно упаковывать тип значения каждый раз, когда вы хотите сравнить один его экземпляр с другим. Фактически, в том виде, как он написан, следующий код упаковывает оба значения:

```
public void Main()
{
    ComplexNumber num1 = new ComplexNumber( 1, 3 );
    ComplexNumber num2 = new ComplexNumber( 1, 2 );
    int result = ((IComparable)num1).CompareTo( num2 );
}
```

Можете ли вы увидеть здесь обе операции упаковки? Как было показано в предыдущем разделе, экземпляр num1 должен быть упакован для того, чтобы получить ссылку на интерфейс IComparable. Кроме того, поскольку CompareTo принимает ссылку типа System.Object, экземпляр num2 также должен быть упакован. Это отражается губительно на эффективность. Технически мне не обязательно упаковывать num1, чтобы осуществить вызов через IComparable. Однако в предыдущем примере ComplexNumber интерфейс IComparable был реализован явно. У меня не было выбора.

Чтобы решить эту проблему, вы можете реализовать безопасную к типам версию метода CompareTo, в то же время реализуя метод IComparable.CompareTo. Если применить такой прием, вызов сравнения из предыдущего кода совершенно не инициирует операции упаковки. Давайте посмотрим, как следует модифицировать структуру ComplexNumber, чтобы сделать это:

```
using System;
public struct ComplexNumber : IComparable,
                              IComparable<ComplexNumber>,
                              IEquatable<ComplexNumber>
{
   public ComplexNumber( double real, double imaginary ) {
      this.real = real;
      this.imaginary = imaginary;
   }

   public bool Equals( ComplexNumber other ) {
      return (this.real == other.real) &&
      (this.imaginary == other.imaginary);
   }

   public override bool Equals( object other ) {
      bool result = false;
      if( other is ComplexNumber ) {
         ComplexNumber that = (ComplexNumber) other ;
         result = Equals( that );
      }
      return result;
   }

   public override int GetHashCode() {
      return (int) this.Magnitude;
   }

   public static bool operator ==( ComplexNumber num1,
                                   ComplexNumber num2 ) {
      return num1.Equals(num2);
   }

   public static bool operator !=( ComplexNumber num1,
      ComplexNumber num2 ) {
      return !num1.Equals(num2);
   }
```

```
    public int CompareTo( ComplexNumber that ) {
        int result;
        if( Equals(that) ) {
            result = 0;
        } else if( this.Magnitude > that.Magnitude ) {
            result = 1;
        } else {
            result = -1;
        }
            return result;
        }
        int IComparable.CompareTo( object other ) {
            if( !(other is ComplexNumber) ) {
                throw new ArgumentException( "Неверное сравнение!" );
        }
        return CompareTo( (ComplexNumber) other );
    }
    public double Magnitude {
        get {
            return Math.Sqrt( Math.Pow(this.real, 2) +
                              Math.Pow(this.imaginary, 2) );
        }
    }
    // Прочие методы опущены для ясности
    private readonly double real;
    private readonly double imaginary;
}
public sealed class EntryPoint
{
    static void Main()
    {
        ComplexNumber num1 = new ComplexNumber( 1, 3 );
        ComplexNumber num2 = new ComplexNumber( 1, 2 );
        int result = num1.CompareTo( num2 );

        // Теперь попробуем безопасную к типам версию
        result = ((IComparable)num1).CompareTo( num2 );
    }
}
```

После модификаций первый вызов `CompareTo` в методе `Main` не инициирует операции упаковки. Также вы должны заметить, что я пошел на шаг дальше и явно реализовал метод `IComparable.CompareTo`. Это затрудняет нечаянный вызов безтиповой версии `IComparable` без явного приведения экземпляра значения в ссылке на тип `IComparable`. Для полноты картины метод `Main` демонстрирует, как вызывать безтиповую версию `CompareTo`. Теперь клиенты, использующие значение `ComplexNumber`, могут писать код естественным образом, получая преимущества повышенной производительности. Клиенты, которым нужно работать через интерфейс, такие как определенные типы контейнеров, могут использовать интерфейс `IComparable`, хотя и с упаковкой. Если вы любопытны, откройте исполняемый файл кода предыдущего примера внутри ILDASM и исследуйте метод `Main`.

Вы увидите, что первый вызов `CompareTo` не приводит к излишней упаковке, в то время как второй вызов `CompareTo` сопряжен с двумя операциями упаковки, чего и следовало ожидать.

В качестве главного эмпирического правила вы можете применять эту идиому почти ко всем методам типов значений, которые принимают или возвращают упакованный экземпляр типа значения. До сих пор вы видели два случая применения этой идиомы: первый — при реализации `Equals` для типа `ComplexNumber`, а второй — при реализации `IComparable.CompareTo`.

Резюме

Эту главу можно подытожить двумя удобными списками вопросов, которые вы можете применять всякий раз, когда проектируете новый тип в C#. Когда вы проектируете новый класс или структуру, полезно сверится с каждым из этих списков, подобно тому, как это делает пилот перед выводом самолета на взлетную полосу. Если вы примете на вооружение этот подход, то всегда будете уверены в своем дизайне.

Предлагаемые списки вопросов формировались в течение некоторого времени. Разумеется, их нельзя считать исчерпывающими. Вы можете обнаружить необходимость их дополнения или создания новых экземпляров для новых сценариев использования классов или структур. Эти списки предназначены для обслуживания наиболее распространенных сценариев, с которыми вы столкнетесь в процессе дизайна программ C#.

Список вопросов для ссылочных типов

- *Должен ли данный класс допускать наследование?* Классы должны объявляться `sealed` по умолчанию, если только они четко не предназначены для использования в качестве базовых. И даже тогда вы должны хорошо документировать, как использовать их в таком качестве. Отдавайте предпочтение герметичным (sealed) классам перед не герметичными.
- *Является ли объект клонируемым?*
  - *Реализуйте `ICloneable`, если по умолчанию нужны глубокие копии.* Если объект изменяемый, он должен копироваться глубоко. Если же он не изменяемый, рассмотрите возможность поверхностного копирования для оптимизации.
  - *Избегайте использования `MemberwiseClone`.* Вызов `MemberwiseClone` создает новый объект без вызова каких-либо конструкторов. Такая практика может быть опасна.
- Является ли объект одноразовым?
  - *Реализуйте `IDisposable`.* Если вы обнаружите необходимость в реализации деструктора, используйте вместо этого интерфейс `IDisposable`.
  - *Реализуйте финализатор.* Одноразовые объекты должны реализовывать финализатор, чтобы перехватывать объекты, для которых клиент забыл вызвать `Dispose`, либо для предупреждения клиентов об этом. Не возлагайте обязанности детерминированной деструкции на деструктор C#, которым является финализатор. Выполняйте такую работу только в методе `Dispose`.

- *Подавляйте финализацию при вызове* `Dispose`. Это позволит сборщику мусора более эффективно выполнять свою работу. В противном случае объекты задерживаются в куче дольше, чем это необходимо.

- Должна ли эквивалентность объектов иметь семантику значений?
  - *Переопределяйте* `Object.Equals`. Перед изменением семантического значения `Equals` убедитесь, что у вас есть серьезные основания для этого; в противном случае оставьте эквивалентность идентичности по умолчанию. Нельзя генерировать исключения изнутри ваших переопределений `Equals`.
  - *Знайте, где вызывать реализацию* `Equals` *базового класса*. Если ваш объект наследуется от типа, версия `Equals` которого имеет семантический смысл, отличающийся от вашей реализации, не вызывайте в переопределении версию базового класса. В противном случае включите ее результат в ваш.
  - *Переопределяйте* `GetHashCode` *тоже*. Это — обязательный шаг, чтобы гарантировать возможность использования вашего типа в качестве ключа хеш-кода. Если вы переопределили `Equals`, также переопределите и `GetHashCode`.

- Являются ли объекты этого типа сравнимыми?
  - *Реализуйте* `IComparable` *и переопределите* `Equals` и `GetHashCode`. Вам следует переопределить все их в группе, поскольку их реализации взаимосвязаны.

- Является ли объект преобразуемым в `System.String` или обратно?
  - *Переопределяйте* `Object.ToString`. Реализация, унаследованная от `Object.ToString`, просто возвращает строковое имя типа объекта.
  - *Реализуйте* `IFormattable`*, если вам нужен более тонкий контроль форматирования строки*. Реализуйте переопределение `Object.ToString` вызовом `IFormattable.ToString` с форматной строкой `G` и поставщиком формата `null`.

- Является ли объект преобразуемым?
  - *Переопределяйте* `IConvertible`*, чтобы класс работал с* `System.Convert`. В C# вы должны реализовать все методы интерфейса. Однако из методов преобразования, которые не имеют смысла для вашего класса, просто генерируйте исключение `InvalidCastException`.

- Должен ли объект быть неизменяемым?
  - *Рассмотрите возможность сделать поля доступными только для чтения и предоставить доступные только для чтения свойства*. Объекты, фундаментально представляющие простое значение — такие как строка или комплексное число — блестящие кандидаты стать неизменяемыми объектами.

- Нужно ли передавать этот объект в качестве константного неизменяемого параметра метода?
  - *Рассмотрите возможность реализации неизменяемого прокладочного класса, содержащего ссылку на изменяемый объект, который может передаваться в качестве параметра метода*. Первым делом, подумайте, имеет ли смысл вашему классу быть неизменяемым. Если да, то в этом действии нет необходимости. Если же вам нужно передавать ваши изменяемые объекты в методы как неизменные, то вы можете достичь того же эффекта, используя интерфейсы.

Список вопросов для типов значений

- Нужна ли вам повышенная эффективность для ваших типов значений?
  - *Переопределите* `Equals` *и* `GetHashCode`. Общая версия `ValueType.Equals` неэффективна, поскольку для выполнения работы полагается на рефлексию. Вообще лучше предоставить безопасную к типам версию `Equals`, реализуя `IEquatable<T>`, и затем заставив безтиповую версию вызывать ее. Не забудьте также переопределить `GetHashCode`.
- Нужно ли модифицировать упакованные экземпляры значения?
  - *Реализуйте интерфейс, чтобы делать это*: Вызов через член интерфейса, реализованного типом значения — единственный способ изменить экземпляр типа значения внутри упакованного экземпляра.
- Являются ли значения данного типа сравнимыми?
  - *Реализуйте* `IComparable` *и переопределите* `Equals` *и* `GetHashCode`. Вам следует переопределить все их в группе, поскольку их реализации взаимосвязаны. Если вы переопределили `Equals`, последуйте предыдущему совету и предусмотрите также безопасную к типам версию.
- Является ли значение конвертируемым в `System.String` или обратно?
  - *Переопределяйте* `ValueType.ToString`. Реализация, унаследованная от `ValueType`, просто возвращает строковое имя типа значения.
  - *Реализуйте* `IFormattable`, *если пользователю нужен более тонкий контроль форматирования строки*. Реализуйте переопределение `ValueType.ToString` вызовом `IFormattable.ToString` с форматной строкой `G` и поставщиком формата `null`.
- Является ли значение конвертируемым?
  - *Переопределите* `IConvertible`, *чтобы структуры работали с* `System.Convert`. В C# все методы интерфейса должны быть реализованы. Однако методы преобразования, которые не имеют смысла для вашей структуры, должны просто генерировать исключение `InvalidCastException`.
- Должна ли данная структура быть неизменяемой?
  - *Рассмотрите возможность сделать поля доступными только для чтения и предоставить доступные только для чтения свойства*. Значения — блестящие кандидаты на то, чтобы быть неизменяемыми типами.

ГЛАВА 14

Расширяющие методы

Применяя расширяющие методы (extension methods), вы можете объявлять методы, расширяющие общедоступный интерфейс или контракт типа. На первый взгляд это может показаться способом расширения классов, которые не предназначены для расширения. Однако очень важно отметить, что расширяющие методы имеют доступ только к общедоступным членам расширяемого ими типа. Это объясняется тем, что они на самом деле вовсе не являются методами экземпляра, а потому не могут нарушить оболочку инкапсуляции расширяемого типа.

Введение в расширяющие методы

Как упоминалось ранее, расширяющие методы создают впечатление, что вы можете модифицировать общедоступный интерфейс любого типа. Рассмотрим краткий пример расширяющих методов в действии:

```
using System;
namespace ExtensionMethodDemo
{
    public static class ExtensionMethods
    {
        public static void SendToLog( this String str ) {
            Console.WriteLine( str );
        }
    }
    public class ExtensionMethodIntro
    {
        static void Main() {
            String str = "Полезная для протоколирования информация";
            // Вызов расширяющего метода.
            str.SendToLog();
            // Вызов того же метода старым способом.
            ExtensionMethods.SendToLog( str );
        }
    }
}
```

Сначала взгляните на метод Main. Обратите внимание, что я сначала объявил System.String, а затем вызвал метод SendToLog на экземпляре str. Минуточку! В определении типа System.String нет метода по имени SendToLog. Дело в том, что метод SendToLog — это расширяющий метод, объявленный в предыдущем классе по имени ExtensionMethods.

На первый взгляд ExtensionMethods.SendToLog выглядит так же, как любой обычный статический метод. Но обратите внимание на две вещи. Он объявлен внутри статического класса по имени ExtensionMethods, и тип первого параметра статического метода SendToLog предварен ключевым словом this. Такое применение ключевого слова this — способ сообщить компилятору, что данный метод является расширяющим методом.

Обратите внимание на конец метода Main, где я показываю, что вы можете метод SendToLog точно так же, как любой другой статический метод. Фактически расширяющие методы не предоставляют в ваше распоряжение никакой дополнительной функциональности сверх обычных статических методов. Расширяющие методы добавляют некоторого рода "синтаксическую приправу" к языку, позволяя вам вызывать их так, как если бы они были методами экземпляра, с которым вы их применяете. Но, как и любым другим средством языка, ими не стоит злоупотреблять. Поэтому далее, в разделе "Рекомендации по использованию" я представлю некоторые полезные советы и рекомендации по использованию расширяющих методов.

Как компилятор находит расширяющие методы?

Когда вы вызываете метод экземпляра на конкретном экземпляре типа, компилятор должен определить, какой именно метод следует вызвать, учитывая такие вещи, как тип экземпляра, его базовый тип (если есть) и все интерфейсы, которые он и его базовый тип могут реализовывать. Как показано в главе 5, шаги, выполняемые компилятором для определения того, какой метод нужно вызвать, достаточно сложны. Каким же образом компилятор справляется с дополнительной сложностью, связанной с нахождением расширяющего метода для вызова?

Расширяющие методы обычно импортируются в текущую единицу компиляции из пространств имен посредством ключевого слова using. Когда вы применяете ключевое слово using для импорта типов из определенного пространства имен в текущий контекст, вы также обеспечиваете доступ ко всем расширяющим методам, реализованным в статических классах этого пространства имен, посредством применения нового синтаксиса. Если вы не импортируете пространство имен со статическими классами, реализующими нужные вам расширяющие методы, то вы можете вызывать их только как обычные статические методы, используя их полные квалифицированные имена. Напомню, что вовсе не обязательно импортировать пространство имен, чтобы использовать типы, определенные внутри него. Например, вы можете применять тип System.Console в вашем приложении, не импортируя пространство имен System. Но обычно для удобства вы все-таки делаете это. Аналогично, для того, чтобы можно было вызывать расширяющие методы с использованием синтаксиса вызова методов экземпляра, вы должны импортировать пространство имен.

На заметку! В C# 3.0 импортирование пространства имен, как более удобный способ обращения к типам внутри него, имеет побочный эффект, если это пространство имен содержит расширяющие методы. Мы поговорим об этом подробнее ниже, в разделе "Рекомендации по использованию" настоящей главы.

Когда вы вызываете метод экземпляра, компилятор ищет тип для сопоставления метода экземпляра. Если поиск не обнаруживает никаких подходящих методов, компилятор начинает искать подходящие методы расширения. Он начинает с просмотра всех расширяющих методов, объявленных во включающем пространстве имен, и если не находит подходящего, рекурсивно продолжает поиск в следующем включающем пространстве имен. Если ему не удается найти подходящий метод, он останавливается с выдачей ошибки компиляции. И наоборот, если текущее пространство имен имеет более одного подходящего импортированного метода расширения, компилятор выдает ошибку, сообщающую о неоднозначности. В таких случаях вы должны вернуться к вызову этого метода как обычного статического, указав его полное квалифицированное имя.

Рассмотрим следующий пример, иллюстрирующий все эти моменты:

```
using System;
public static class ExtensionMethods
{
   static public void WriteLine( this String str ) {
      Console.WriteLine( "Пространство имен по умолчанию: " + str );
   }
}
namespace A
{
   public static class ExtensionMethods
   {
      static public void WriteLine( this String str ) {
         Console.WriteLine( "Пространство имен A: " + str );
      }
   }
}
namespace B
{
   public static class ExtensionMethods
   {
      static public void WriteLine( this String str ) {
         Console.WriteLine( "Пространство имен B: " + str );
      }
   }
}
namespace C
{
   using A;
   public class Employee
   {
      public Employee( String name ) {
         this.name = name;
      }
```

```
        public void PrintName() {
            name.WriteLine();
        }
        private String name;
    }
}
namespace D
{
    using B;
    public class Dog
    {
        public Dog( String name ) {
            this.name = name;
        }
        public void PrintName() {
            name.WriteLine();
        }
        private String name;
    }
}
namespace E
{
    public class Cat
    {
        public Cat( String name ) {
            this.name = name;
        }
        public void PrintName() {
            name.WriteLine();
        }
        private String name;
    }
}
namespace Demo
{
    using A;
    using B;
    public class EntryPoint
    {
        static void Main() {
            C.Employee fred = new C.Employee( "Fred" );
            D.Dog thor = new D.Dog( "Thor" );
            E.Cat sylvester = new E.Cat( "Sylvester" );
            fred.PrintName();
            thor.PrintName();
            sylvester.PrintName();
            // String str = "Etouffe";
            // str.WriteLine();
        }
    }
}
```

В данном примере один и тот же расширяющий метод объявлен в трех разных пространствах имен, а именно — в A, B и пространстве имен по умолчанию. Вдобавок определены три типа — каждый в своем собственном пространстве имен. Это — `Employee`, `Dog` и `Cat`. В методе `Main` создается экземпляр каждого из них, и затем на каждом экземпляре вызывается метод `PrintName`. Тело метода `PrintName` каждого типа затем вызывает расширяющий метод `WriteLIne` на поле типа `String`, и здесь все становится намного интереснее. Если вы скомпилируете и выполните этот код, то увидите на консоли следующий вывод:

```
Пространство имен A: Fred
Пространство имен B: Thor
Пространство имен по умолчанию: Sylvester
```

Отметьте, что в этом случае конкретная реализация вызываемого расширяющего метода определяется пространством имен, в котором определен данный тип (`Employee`, `Dog` или `Cat`). Поскольку пространство имен C импортирует пространство A, следовательно `Employee` вызовет метод `WriteLine`, определенный в пространстве имен A. Аналогично, поскольку пространство имен D импортирует пространство B, то вызывается метод расширения `WriteLine` из пространства B. Поскольку пространство E не импортирует ни A, ни B, компилятор ищет в пространстве имен по умолчанию, которое также включает собственную реализацию метода расширения `WriteLine`. И, наконец, обратите внимание на закомментированный код в конце метода `Main`. Если вы уберете комментарии с этих строк и попытаетесь скомпилировать, то увидите, как компилятор пожалуется, что не может определить, какой метод `WriteLine` ему нужно вызвать, поскольку пространство имен `Demo` импортирует оба пространства — и A, и B. Этот пример подчеркивает опасность неправильного использования этого нового синтаксиса.

За кулисами

Как же компилятор реализует расширяющие методы? Поскольку они представляют собой просто синтаксические сокращения, никаких модификаций в исполняющей системе для поддержки методов расширения не потребовалось. Вместо этого компилятор целиком реализует методы расширения через метаданные. Ниже приведен код IL метода `ExtensionMethods.WriteLine` из пространства имен по умолчанию, сгенерированного при компиляции предыдущего примера:

```
.method public hidebysig static void WriteLine(string str) cil managed
{
    .custom instance void [System.Core]System.Runtime.CompilerServices.
ExtensionAttribute::.ctor() = ( 01 00 00 00 )
    // Code size 19 (0x13)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr    "Default Namespace: "
    IL_0006: ldarg.0
    IL_0007: call string [mscorlib]System.String::Concat(string,string)
    IL_000c: call     void [mscorlib]System.Console::WriteLine(string)
    IL_0011: nop
    IL_0012: ret
} // end of method ExtensionMethods::WriteLine
```

Выделенный текст перенесен, чтобы уместиться на странице, но из него видно, что метод снабжен новым атрибутом. Этот атрибут — ExtensionAttribute, и он определен в пространстве имен System.Runtime.CompilerServices. Компилятор также применяет атрибут к содержащему классу, в данном случае — ExtensionMethods. Во время компиляции эта информация используется при поиске расширяющего метода — потенциального кандидата на вызов. Это — наглядный пример мощи метаданных и того, чего можно достичь с помощью пользовательских атрибутов в метаданных.

Если вы найдете в сгенерированном коде IL место вызова расширяющего метода, то увидите, что он вызывается точно так же, как нормальный статический метод.

Читабельность или понятность

Признаем один факт. Существует масса компаний, которые предоставляют очень мало документации к коду их приложений. Многие компании могут иметь документацию по дизайну высшего уровня, которая, по сути, демонстрирует крупные компоненты приложения, связанные неясными линиями, выражающими отношения между ними. Но слишком часто случается так, что в спешке обозначается лишь начало проекта, несколько поворотов туда-сюда, а затем уже никого не заботит обновление исходной документации до актуального состояния. Через несколько версий кода и документации продукта они приходят в такое состояние, что приходится просто отправлять документацию в мусорную корзину. Другая крайность — это когда вы встречаете организацию, которая документирует абсолютно каждый кусочек приложения средствами моделирования UML, еще до начала кодирования.

Наиболее успешные проекты относятся к промежуточной категории. Обычно вы имеете достаточно документации, так что ведущий разработчик, вынужденный покинуть проект, оставляет своему преемнику достаточно информации, содержащейся в документации и коде, чтобы тот мог подхватить флаг и двигаться дальше. Ключом к успеху в этом случае является читабельный код, но также и код, который легко понять.

В чем разница? Хорошо читабельный код четко обозначает намерения программиста, вы можете легко прочесть его и постичь. Например, представьте, что вы читаете чей-то код, автор которого должен был вывести некоторую строку в какой-нибудь протокольный файл. Вы можете встретить там нечто вроде:

```
String s = "интересная информация";
Logger.LogOutput( s );
```

Вроде все понятно, что здесь происходит. Но так ли это? Что делает LogOutput? Пишет ли он строку в файл? Отправляет ее на консоль? Выводит в поток отладки? Глядя на этот код, ответить невозможно. Вместо этого нужно обратиться к коду Logger.LogOutput, чтобы понять, что происходит на самом деле. Таким образом, кто-то может сказать, что ключ к понятному коду — легкость его отслеживания, т.е. навигации по нему.

Посмотрим, как мог бы выглядеть этот код при использовании расширяющих методов:

```
String info = "интересная информация";
info.WriteTo( logFile );
```

Сразу можно сказать, что этот код гораздо легче читать. Он читается почти как предложение на естественном языке. Эту технику вы встретите повсюду а .NET Framework. Во второй строке все начинается с субъекта, за ним следует глагол, а заканчивается все целевым объектом действия. Это можно воспринимать как императивную команду, отданную объекту `info`.

Но что, если `WriteTo` реализован как расширяющий метод? Так ли легко понять код, как его легко читать? В действительности это может зависеть от применяемых вами инструментов. Средство Intellisense в Visual Studio определенно может помочь, потому что подскажет вам, какой статический класс реализует расширяющий метод `WriteTo`. Но если ваш любимый редактор — Emacs или Vim, вам придется определить, какие пространства имен были импортированы в текущий контекст, и потом вы должны начать в этих пространствах поиск класса, реализующего расширяющий метод. В зависимости от сложности кода приложения и от того, насколько много пространств имен импортирует текущая единица компиляции, это может превратиться в настоящий кошмар!

Мораль этой истории такова. Как и любое другое средство языка, используйте средства расширения корректно, сдержанно, и только когда это действительно необходимо. Только то, что вы можете реализовать нечто с расширяющими методами, еще не значит, что вы всегда должны это делать. Вокруг инженерных решений построена целая инженерная дисциплина, которая говорит о том, как собирать информацию и делать правильный выбор в ситуациях, когда ни один из вариантов не является идеальным[1].

Рекомендации по использованию

В следующих разделах детализируются некоторые передовые приемы и содержатся рекомендации по использованию расширяющих методов. Конечно, по мере развития методов расширения в языке, этот набор будет расти.

Использование расширяющих методов вместо наследования

Когда впервые сталкиваешься с расширяющими методами, естественно воспринять их как средство расширения общедоступного контракта отдельного типа или группы типов. Это очевидное заключение исходит из того факта, что расширяющие методы могут быть вызваны посредством синтаксиса вызова методов экземпляра. Однако я думаю, что намного эффективнее воспринимать методы расширения как способ описания операций, которые вы можете применить к данному типу или множеству типов, в более синтаксически привлекательном виде, потому что, опять же, на самом деле они вовсе не расширяют контракт типа.

Ранее я продемонстрировал тривиальные примеры использования расширяющих методов, позволяющие вызывать `WriteLine` на экземплярах типа `String`. Альтернативно можно было бы осуществить наследование от `String`, чтобы обеспечить то же поведение. Но в этом случае вы обнаружите, что это невозможно, потому что класс `String` является `sealed` и не допускает наследования. Я главе 13 я объяснил, почему считаю, что лучше по умолчанию сразу герметизировать соз-

---
[1] Это можно назвать выбором из двух зол.

даваемые вами типы. Допускать наследование от ваших классов вы должны лишь после тщательного обдумывания дизайна (и документирования), позволяющего им служить в качестве базовых классов. Проектировщики System.String имели веские причины удержать нас от использования этого типа в качестве базового.

Вдобавок неосмотрительное использование наследования для этой цели излишне усложняет ваш дизайн. В главе 4 объясняется, как можно сдержанно применять наследование, и как использовать отношение включения — обычно более гибкое, чем наследование. Наследование — одна из наиболее строгих форм статической связи двух типов. Злоупотребление этим клеем создает монолит, с которым чрезвычайно трудно работать.

Например, у вас может возникнуть необходимость в применении вашей операции WriteLine или какого-нибудь удобного расширяющего метода на экземпляре типа, который создавали не вы. Рассмотрим экземпляр типа, возвращенного некоторым фабричным методом, как в следующем коде:

```
public class MyFactory
{
    public Widget CreateWidget() {
        ...
    }
}
```

Здесь CreateWidget — метод, создающий экземпляр возвращаемого Widget. Это то, что называется фабричным методом (factory method). Обычно у вас будет иерархия специализаций типа, унаследованная от Widget, и CreateWidget может принимать некоторые параметры, сообщающие ему, какого именно типа нужно создать Widget. Независимо от этого, мы не управляем тем, как и когда создается экземпляр. Поэтому невозможно использовать наследование для расширения контракта типа Widget, если только также не контролировать метод CreateWidget. И даже если у нас будет такая возможность, может оказаться, что этот метод уже опубликован в какой-то другой сборке, которая была сертифицирована и не может быть изменена. Прямое наследование в данной ситуации представляет собой неверный подход для добавления функциональности WriteLine к типам Widget.

Расширяющие методы также позволяют вам представлять операции, которые можно применять ко всей иерархии типов. Например, рассмотрим расширяющий метод, чей первый параметр будет относиться к типу System.Object. Этот расширяющий метод может быть вызван на любом типе. Вы не можете достичь того наследованием. Чтобы сделать это, потребовалась бы возможность создания производного типа от System.Object, скажем, MyObject, а затем каким-то образом заставить каждый тип в CLR наследоваться от MyObject вместо System.Object. Стоит ли говорить о том, что это невозможно. Но только то, что вы можете создать расширяющий метод, который можно будет вызывать со всеми объектами, еще не значит, что вы должны это делать. Если расширяющий метод не оперирует только общедоступным интерфейсом System.Object, скорее всего, в расширяющем методе у вас будет код, определяющий тип во время выполнения, чтобы можно было выполнять какие-то специфичные для типа операции. Такой стиль кодирования сводит на нет строго типизированную природу языка C# и его компилятора.

На заметку! Вы также можете использовать обобщения для применения расширяющих методов к нескольким типам. Вы можете объявлять обобщенные расширяющие методы почти также легко, как и обобщенные методы экземпляра или обобщенные делегаты. В разделе "Шаблон Visitor" я покажу пример использования обобщенного расширяющего метода.

Изоляция расширяющих методов в отдельном пространстве имен

Одним из фундаментальных принципов при написании методов является избежание побочных эффектов. Это значит, что если вы создаете такой метод, как `LogToFile`, и внутри его реализации решаете модифицировать некоторое глобальное состояние, используемое другими компонентами приложения, вы тем самым создаете потенциально опасный побочный эффект. Подобные побочные эффекты обычно являются причиной многих трудно обнаруживаемых ошибок. В данном случае модификация глобального состояния может быть не интуитивно понятной, исходя из имени метода.

По той же причине старайтесь избегать создания побочных эффектов для ваших клиентов, когда они импортируют пространства имен. А именно: лучше будет, если вы объявляете любые расширяющие методы во вложенном пространстве имен. Если не предоставлять вашим клиентам такой степени детализации, это может привести к путанице, когда компилятор попытается найти метод для вызова метода экземпляра.

Представьте на мгновение, какую путаницу может внести определение ваших расширяющих методов в пространстве имен `System`. Не существует механизма, который бы удержал вас от этого. Почти все импортируют пространство `System` в свой код. Поэтому, если вы определите ваши расширения подобным образом, большинство разработчиков импортирует их, и единственный способ предотвратить это — смириться с неудобством полного отказа от импорта пространства имен `System`!

Если вы полагаете, что это не такая уж большая проблема, позвольте мне представить следующий сценарий. Предположим, что у вас есть приложение, использующее библиотеку из Acme Widgets. Разработчики Acme Widgets думали, что будет удобно предоставить расширяющий метод по имени `WriteLog`, чтобы вы получили в свое распоряжение еще один инструмент отладки при использовании их библиотеки. Будучи хорошими дизайнерами, они определили расширяющий метод в пространстве имен `AcmeWidgetExtensions`. Теперь, двумя версиями позже, вам встретилась библиотека Ace Objects, и вы решили ею воспользоваться. Перед внесением изменений в ваш код, вы включили ссылку на эту сборку в проект, и теперь вдруг весь ваш код перестал собираться; компилятор сообщает об ошибке CS0121, гласящей, что вызовы WriteLog неоднозначны! Внимательное исследование показывает, что разработчики Ace Objects также решили, что будет полезно представить расширяющий метод с тем же именем `WriteLog`. К сожалению, они определили его в пространстве имен `System`, которое импортирует весь ваш код. Вот напасть!

Таким образом, мораль истории заключается в том, что всегда нужно определять ваши расширяющие методы в отдельном пространстве имен, чтобы обеспечить вашим клиентам возможность тонкой настройки при импорте их в свой контекст.

Более того, если вы предоставляете большой набор методов расширения, рассмотрите возможность дальнейшего его подразделения в несколько пространств имен, чтобы предоставить более высокую степень детализации вашим клиентам.

Изменение контракта типа может разрушить расширяющие методы

Когда компилятор ищет имя, соответствующее вызову метода экземпляра, расширяющие методы — это то, что он просматривает в последнюю очередь. Это имеет смысл, потому что если у вас есть класс, который уже реализует метод экземпляра по имени `WriteLog`, вы наверняка не захотите, чтобы расширяющий метод заменил его функциональность. Однако рассмотрим следующий сценарий.

У вас есть приложение, использующее библиотеку Acme Widgets. Чтобы помочь в отладке вашей системы и предоставить богатый механизм протоколирования, вы создали расширяющий метод по имени `WriteLog`, который вы можете использовать для вывода информации о конкретном графическом элементе в файл протокола. Прошло время, и теперь вы решили перейти к версии 2 библиотеки Acme Widgets. А между тем, создатели библиотеки Acme Widgets решили расширить общедоступный контракт некоторых из ее типов, и добавили метод `WriteLog`, потому что перед тем, как вы реализовали собственный расширяющий метод `WriteLog`, вы отправили им запрос на новые средства, описывающий, насколько ценным могло бы стать такое средство. Не зная, что они добавили этот метод к своим типам, вы перекомпилируете свой код. Никаких ошибок не возникает, поскольку так случилось, что сигнатура новых методов экземпляра в точности совпала с сигнатурой вашего расширяющего метода. Но когда следующий раз вы запускаете свое приложение, вы обнаруживаете, что его поведение изменилось, и формат вашего файла протокола стал совершенно другим! Это произошло потому, что теперь компилятор предпочел метод экземпляра старому расширяющему методу. Такие же неприятные вещи могут произойти, если определение типа включает новое свойство с тем же именем, что и у вашего расширяющего метода. Но в этом случае вы получаете ошибку компилятора, поскольку он жалуется на попытку вызвать свойство, как будто это был бы метод.

Единственное реальное решение этой проблемы, если вы когда-либо с ней столкнетесь, состоит в переключении на вызов расширяющего метода через классический синтаксис вызова статического метода вместо синтаксиса вызова расширяющего метода экземпляра. Но если вам настолько не повезло, что переключение произошло молча, как в приведенном выше сценарии, может пройти какое-то время, прежде чем вы осознаете, что вам нужно вызывать расширяющий метод иначе.

Трансформации

Несмотря на то что расширяющие методы — это просто синтаксические сокращения для вызова статических методов с использованием стандартного синтаксиса вызова методов, иногда такое кажущееся упрощение может стимулировать мыслительный процесс, открыв дорогу новым идеям. Например, предположим, что у вас есть коллекция данных. Пусть эта коллекция реализует `IEnumerable<T>`. Теперь

мы хотим применить к каждому элементу списка некую операцию и произвести новый список. Для примера предположим, что у нас есть список целых чисел, и мы хотим трансформировать его в список чисел с плавающей точкой двойной точности, составляющих 1/3 исходного значения. Вы можете подойти к решению этой проблемы так, как показано в следующем примере:

```
using System;
using System.Collections.Generic;
public class TransformExample
{
    static void Main() {
        var intList = new List<int>() { 1, 2, 3, 4, 5 };

        var doubleList = new List<double>();

        // Вычислить новый список.
        foreach( var item in intList ) {
            doubleList.Add( (double) item / 3 );
            Console.WriteLine( item );
        }
        Console.WriteLine();
        // Отобразить новый список.
        foreach( var item in doubleList ) {
            Console.WriteLine( item );
        }
        Console.WriteLine();
    }
}
```

Приведенный код является типичным примером императивного стиля программирования и демонстрирует правильное решение проблемы. К сожалению, это решение не слишком масштабируемо и повторно используемо. Например, представьте, что будет, если вы захотите применить другую операцию к результатам первой, или, может быть, сразу три операции, выстроенные цепочкой. Или же вы решите сделать как можно большую часть этого кода насколько возможно повторно используемой.

В этом примере имеют место как минимум две фундаментальных операции. Первая — итерация по входной коллекции и производство новой коллекции. Вторая операция, совершенно ортогональная первой — деление каждого элемента на 3. Неплохо было бы разнести их, правда? Тогда при правильном кодировании код трансформации мог бы быть повторно использован в самых разных операциях. Поэтому для начала отделим операцию от трансформации и посмотрим, как может выглядеть код:

```
using System;
using System.Collections.Generic;
public class TransformExample
{
    delegate double Operation( int item );
    static List<double> Transform( List<int> input, Operation op ) {
        List<double> result = new List<double>();
```

```
      foreach( var item in input ) {
         result.Add( op(item) );
      }
      return result;
   }
   static double DivideByThree( int n ) {
      return (double)n / 3;
   }
   static void Main() {
      var intList = new List<int>() { 1, 2, 3, 4, 5 };
      // Вычислить новый список.
      var doubleList = Transform( intList, DivideByThree );

      foreach( var item in intList ) {
         Console.WriteLine( item );
      }
      Console.WriteLine();
      // Отобразить новый список.
      foreach( var item in doubleList ) {
         Console.WriteLine( item );
      }
      Console.WriteLine();
   }
}
```

Новый код несколько лучше. Теперь операция вынесена за скобки и передается через делегат статическому методу `Transform`. Как вы можете представить, мы превратили метод `Transform` в расширяющий метод. Но это не все! Мы также может применить обобщения, чтобы сделать код в большей степени повторно используемым. Но, минуточку, это еще не все! Мы же можем использовать итераторы, чтобы заставить метод `Transform` вычислять элементы в "ленивой" манере. Смотрите в следующем фрагменте пример более повторно используемой версии `Transform`:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class MyExtensions
{
   public static IEnumerable<R> Transform<T, R>(
                  this IEnumerable<T> input,
                  Func<T, R> op ) {
      foreach( var item in input ) {
         yield return op( item );
      }
   }
}
public class TransformExample
{
   static double DivideByThree( int n ) {
      return (double)n / 3;
   }
```

```
    static void Main() {
        var intList = new List<int>() { 1, 2, 3, 4, 5 };
        // Вычислить новый список.
        var doubleList =
        intList.Transform( new Func<int, double>(DivideByThree) );
        foreach( var item in intList ) {
            Console.WriteLine( item );
        }
        Console.WriteLine();
        // Отобразить новый список.
        foreach( var item in doubleList ) {
            Console.WriteLine( item );
        }
    }
}
```

Вот теперь другое дело! Для начала обратите внимание, что Transform<T> стал обобщенным расширяющим методом. Более того, он принимает и возвращает экземпляры типа IEnumerable<T>. Теперь Transform<T> может использоваться в любой обобщенной коллекции и принимать делегат, описывающий способ трансформации каждого элемента. Тип Func<TArg0,TResult> определен в пространстве имен System и появился там, начиная с версии .NET 3.0. Поскольку для возврата элементов из Transform<T> применяется итератор, каждый элемент обрабатывается только тогда, когда курсор возвращенного типа IEnumerable<T> перемещается вперед. В этом примере экономия вычислений тривиальна. Однако вы легко можете представить себе ситуацию, когда переданная операция может достаточно долго выполнять обработку каждого элемента входной коллекции. Входная коллекция может содержать длинные строки, и операция, например, может заниматься их шифрованием. Другая причина того, что "ленивое" вычисление здесь настолько удобно, состоит в том, что входная коллекция может представлять собой бесконечную серию. Как? Взгляните на следующий пример, который также включает в себя небольшой анонс лямбда-выражений, которым посвящена 15-я глава:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class MyExtensions
{
    public static IEnumerable<R> Transform<T, R>(
                    this IEnumerable<T> input,
                    Func<T, R> op ) {
        foreach( var item in input ) {
            yield return op( item );
        }
    }
}
public class TransformExample
{
    static IEnumerable<int> CreateInfiniteSeries() {
        int n = 0;
        while( true ) {
            yield return n++;
        }
    }
```

```
    static void Main() {
        var infiniteSeries1 = CreateInfiniteSeries();
        var infiniteSeries2 =
            infiniteSeries1.Transform( x => (double)x / 3 );
        IEnumerator<double> iter =
            infiniteSeries2.GetEnumerator();
        for( int i = 0; i < 25; ++i ) {
            iter.MoveNext();
            Console.WriteLine( iter.Current );
        }
    }
}
```

Насколько это здорово? Конечно, в моем цикле я не могу использовать foreach, иначе программа никогда не завершится, и вам придется прекращать ее принудительно. Забавный синтаксис внутри вызова метода Transform<T> — это лямбда-выражение. Используемое подобным образом лямбда-выражение определяет функцию, которая в данном случае передается в виде делегата. Вы можете воспринимать лямбда-выражения как сжатый синтаксис определения анонимных методов. Если вам не терпится узнать о том, что такое лямбда-выражения, обратитесь в главу 15.

Используемые показанным способом расширяющие методы позволяют реализовать в более полной мере функциональный стиль программирования[2]. В конце концов, только что показанный метод Transform<T> попадает в эту категорию. Фактически вы обнаружите, что большая часть нововведений, появившихся в C# 3.0, облегчают применение парадигмы функционального программирования. К этим средствам относятся расширяющие методы, лямбда-выражения и язык LINQ. Каждое из этих средств делает упор на вычислительную операцию вместо самой структуры вычисления. Преимущества функционального программирования многочисленны, и им можно было бы посвятить целую книгу. Например, функциональное программирование облегчает параллелизм, поскольку переменные обычно никогда не изменяются после начального присваивания; отсюда требуется меньше блокировок синхронизации.

На заметку! Разработчики C++, знакомые с шаблонным метапрограммированием, описанным в блестящей работе Дэвида Абрахамса (David Abrahams) и Алексея Гуртового (Aleksey Gurtovoy) *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* (Boston, MA: Addison-Wesley Professional, 2004 г.), чувствуют себя как дома, когда идет речь об этом стиле функционального программирования. Фактически шаблонное метапрограммирование предоставляет относительно бедную среду функционального программирования, поскольку как только переменной присвоено значение (или *символ* — в терминологии функционального программирования), оно больше никогда не меняется. С другой стороны, C# предлагает гибридную среду, в которой вы вольны реализовать функциональное программирование, если вы того желаете. К тому же те из вас, кто знаком со стандартной библиотекой шаблонов (STL), испытают похожее чувство от этого стиля программирования. STL распространилась в сообществе программистов C++ в начале 90-х годов, и своим появлением стимулировала образ мышления, более склоняющийся в сторону функционального программирования.

---

[2] Чтобы узнать больше о функциональном программировании, запустите поиск по "functional programming" на www.wikipedia.org.

Цепочки операций

Применяя расширяющие методы, построение цепочек становится более естественным процессом. Опять же здесь нет ничего такого, что нельзя было бы сделать в C# 2.0, применяя обычные статические и анонимные методы. Однако благодаря упрощенному синтаксису, связывание операций в цепочки исключает излишнюю суету и может стимулировать новаторское мышление. Начнем с примера из предыдущего раздела, где мы брали список целых чисел и трансформировали его в список действительных чисел двойной точности. На этот раз мы посмотрим, как можно в действительности плавно связывать операции в цепочки. Предположим, что после деления целых чисел на 3 мы хотим вычислить квадрат результата. В следующем коде демонстрируется, как это можно сделать.

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class MyExtensions
{
   public static IEnumerable<R> Transform<T, R>(
                 this IEnumerable<T> input,
                 Func<T, R> op ) {
      foreach( var item in input ) {
         yield return op( item );
      }
   }
}
public class TransformExample
{
   static IEnumerable<int> CreateInfiniteList() {
      int n = 0;
      while( true ) yield return n++;
   }

   static double DivideByThree( int n ) {
      return (double)n / 3;
   }

   static double Square( double r ) {
      return r * r;
   }
   static void Main() {
      var divideByThree =
         new Func<int, double>( DivideByThree );
      var squareNumber =
         new Func<double, double>( Square );

      var result = CreateInfiniteList().
                   Transform( divideByThree ).
                   Transform( squareNumber );
```

```
        var iter = result.GetEnumerator();
        for( int i = 0; i < 25; ++i ) {
            iter.MoveNext();
            Console.WriteLine( iter.Current );
        }
    }
}
```

Ну, разве не изящно? В одном операторе кода я беру бесконечный список целых чисел и применяю деление, за которым следует операция возведения в квадрат, получая в результате "лениво" вычисляемый тип IEnumerable<double>, который и вычисляет каждый элемент по мере необходимости. Функциональное программирование действительно очень удобно, если взглянуть на него с этой точки зрения. Разумеется, вы можете связывать в цепочки столько операций, сколько хотите. Например, вы можете добавить в конец операцию округления. Или же добавить фильтрующую операцию, чтобы учитывались только результаты, отвечающие некоторому критерию. Чтобы сделать это, вы можете создать обобщенный расширяющий метод Filter<T>, подобный Transform<T>, который принимает делегат-предикат в качестве параметра и использует его для фильтрации элементов коллекции.

Уверен, что здесь вы задумались обо всех действительно полезных расширяющих методах, которые вы можете создать для манипулирования данными. Возможно, вы удивитесь, но целый букет подобных методов уже существует. Загляните в класс System.Linq.Queryable. Этот класс предоставляет полный набор расширяющих методов, которые обычно используются вместе с LINQ, о чем я расскажу в главе 16. Главное их отличие в том, что все эти расширяющие методы оперируют типами IQueryable<T>, которые наследуются от IEnumerable<T>.

Также класс System.Linq.Enumerable предлагает ту же функциональность для типов, реализующих IEnumerable<T>.

Пользовательские итераторы

В главе 9 я рассказал об итераторах, которые были добавлены в язык в C# 2.0. Также я описал некоторые способы создания пользовательских итераторов. Расширяющие методы предоставляют еще больше гибкости при создании итераторов для коллекций в очень выразительной манере. По умолчанию каждая коллекция, реализующая IEnumerable или IEnumerable<T>, имеет прямой итератор, поэтому пользовательский итератор может понадобиться для перемещения по коллекции иным способом, чем посредством ее итератора по умолчанию. К тому же вам понадобится создавать пользовательские итераторы для типов, не поддерживающих IEnumerable<T>, как будет показано в следующем разделе, озаглавленном "Заимствование из функционального программирования". Давайте посмотрим, как можно использовать расширяющие методы для реализации пользовательских итераторов на типах, реализующих IEnumerable<T>.

Например, представьте двумерную матрицу, реализованную как тип List<List<int>>. При выполнении некоторых операций на таких матрицах принято использовать итератор, который перемещается по матрице в режиме ведущей строки. Это значит, что итератор сначала перемещается по элементам первой строки, затем второй, и так далее — до тех пор, пока не достигнет конца последней строки.

Вы можете выполнять итерацию по матрице в режиме ведущей строки, как показано ниже:

```
using System;
using System.Collections.Generic;
public class IteratorExample
{
    static void Main() {
        var matrix = new List<List<int>> {
            new List<int> { 1, 2, 3 },
            new List<int> { 4, 5, 6 },
            new List<int> { 7, 8, 9 }
        };
        // Один из способов итерации по матрице.
        foreach( var list in matrix ) {
            foreach( var item in list ) {
                Console.Write( "{0}, ", item );
            }
        }
        Console.WriteLine();
    }
}
```

Да, этот код свою работу выполняет, но его не слишком удобно использовать повторно. Давайте посмотрим, как это можно сделать с помощью расширяющего метода:

```
using System;
using System.Collections.Generic;
public static class CustomIterators
{
    public static IEnumerable<T> GetRowMajorIterator<T>(
                this List<List<T>> matrix ) {
        foreach( var row in matrix ) {
            foreach( var item in row ) {
                yield return item;
            }
        }
    }
}
public class IteratorExample
{
    static void Main() {
        var matrix = new List<List<int>> {
            new List<int> { 1, 2, 3 },
            new List<int> { 4, 5, 6 },
            new List<int> { 7, 8, 9 }
        };
        // Более элегантный способ перечисления элементов.
        foreach( var item in matrix.GetRowMajorIterator() ) {
            Console.Write( "{0}, ", item );
        }
        Console.WriteLine();
    }
}
```

В этой версии я вынес итерацию в расширяющий метод GetRowMajorIterator<T>. В то же время я сделал его обобщенным, чтобы он принимал двумерные вложенные списки, содержащие элементы любого типа, тем самым сделав его более повторно используемым.

Заимствование из функционального программирования

Возможно, вы уже заметили, что многие из новых средств C# 3.0 облегчают применение модели функционального программирования. Возможность реализации функционального программирования была в C# и раньше, но новые средства языка облегчают это синтаксически, делая язык более выразительным. Иногда средства функциональной модели облегчают решение самых разных проблем. Существует несколько языков, которые относятся к категории функциональных, и Lisp — один из них.

Если вы когда-либо программировали на Lisp, то знаете, что список — одна из главных структур этого языка. В C# можно моделировать такой список, используя следующее определение интерфейса:

```
public interface IList<T>
{
   T Head { get; }
   IList<T> Tail { get; }
}
```

Структура этого списка несколько отличается от распространенных реализаций связных списков. Обратите внимание, что вместо одного узла, содержащего значение и указатель на следующий узел, она содержит значение в узле и остальную часть списка. Фактически эта структура по природе своей рекурсивна. Это не удивительно, поскольку рекурсивные технологии являются частью модели функционального программирования. Например, если вы представите список на бумаге, записав его элементы в скобках, то традиционный список может выглядеть примерно так:

```
(1 2 3 4 5 6)
```

С другой стороны, список, определенный с использованием интерфейса IList<T>, может выглядеть следующим образом:

```
(1 (2 (3 (4 (5 (6 (null null)))))))
```

Каждый набор скобок содержит два элемента: значение узла и остальную часть списка внутри вложенной пары скобок. Таким образом, представить список только с одним значением в нем, например, единицей, можно так:

```
(1 (null null))
```

И, конечно, пустой список может быть представлен следующим образом:

```
(null null)
```

В следующем примере кода я создаю пользовательский список по имени MyList<T>, реализующий IList<T>. Способ его построения не слишком эффективен, и я еще вернусь к нему позже.

```
using System;
using System.Collections.Generic;
public interface IList<T>
{
    T Head { get; }
    IList<T> Tail { get; }
}
public class MyList<T> : IList<T>
{
    public static IList<T> CreateList( IEnumerable<T> items ) {
        IEnumerator<T> iter = items.GetEnumerator();
        return CreateList( iter );
    }
    public static IList<T> CreateList( IEnumerator<T> iter ) {
        if( !iter.MoveNext() ) {
            return new MyList<T>( default(T), null );
        }
        return new MyList<T>( iter.Current, CreateList(iter) );
    }
    private MyList( T head, IList<T> tail ) {
        this.head = head;
        this.tail = tail;
    }
    public T Head {
        get {
            return head;
        }
    }
    public IList<T> Tail {
        get {
            return tail;
        }
    }
    private T head;
    private IList<T> tail;
}
public static class CustomIterators
{
    public static IEnumerable<T>
        LinkListIterator<T>( this IList<T> theList ) {
        for( var list = theList;
             list.Tail != null;
             list = list.Tail ) {
            yield return list.Head;
        }
    }
}
public class IteratorExample
{
    static void Main() {
        var listInts = new List<int> { 1, 2, 3, 4 };
```

```
        var linkList =
            MyList<int>.CreateList( listInts );

        foreach( var item in linkList.LinkListIterator() ) {
            Console.Write( "{0}, ", item );
        }
        Console.WriteLine();
    }
}
```

Для начала обратите внимание в Main, что я инициализирую экземпляр MyList<int>, используя List<int>. Статический метод CreateList рекурсивно наполняет MyList<int>, используя эти значения. Как только CreateList завершается, вы имеете экземпляр MyList<int>, который может быть представлен следующим образом:

```
(1 (2 (3 (4 (null null)))))
```

Возможно, вы недоумеваете, почему список представлен не так:

```
(1 (2 (3 (4 null))))
```

Это можно было бы сделать, однако тогда вы обнаружили бы, что с ним неудобно работать — как при составлении списка, так и при его использовании. Если говорить об использовании, вы можете представить, что может возникнуть необходимость выполнить итерацию по одному из этих списков. Для этого вам понадобится пользовательский итератор, который я выделил в примере. Код Main использует этот итератор для отправки всех элементов списка на консоль. Вывод выглядит следующим образом:

```
1, 2, 3, 4,
```

Обратите внимание, что в этом примере метод LinkListIterator<T> создает прямой итератор, исходя из некоторых предположений относительно того, как определять достижение конца списка и как увеличивать курсор во время итерации. Что если вынести эту информацию наружу? Как это сделать? Если идея делегатов возникает в вашем сознании, вы на правильном пути. Взгляните на показанную далее измененную версию расширяющего метода итератора и метода Main:

```
public static class CustomIterators
{
    public static IEnumerable<T>
        GeneralIterator<T>( this IList<T> theList,
                            Func<IList<T>, bool> finalState,
                            Func<IList<T>, IList<T>> incrementer ) {
        while( !finalState(theList) ) {
            yield return theList.Head;
            theList = incrementer( theList );
        }
    }
}
```

```
public class IteratorExample
{
    static void Main() {
        var listInts = new List<int> { 1, 2, 3, 4 };
        var linkList =
            MyList<int>.CreateList( listInts );
        var iterator = linkList.GeneralIterator( delegate( IList<int> list ) {
                                                     return list.Tail == null;
                                                 },
                                                 delegate( IList<int> list ) {
                                                     return list.Tail;
                                                 } );
        foreach( var item in iterator ) {
            Console.Write( "{0}, ", item );
        }
        Console.WriteLine();
    }
}
```

Обратите внимание на то, как метод GeneralIterator<T> принимает еще два делегата, один из которых вызывается для проверки факта достижения курсором конца списка, а второй — для увеличения значения курсора. В методе Main я передаю два делегата в форме анонимных методов. Теперь метод GeneralIterator<T> может быть использован для выполнения итерации к каждому следующему элементу списка просто модификацией делегата, переданного в параметре, увеличивающем значение курсора.

На заметку! Некоторые из вас, возможно, уже знакомы с лямбда-выражениями, которые появились в C# 3.0. В самом деле, используя лямбда-выражения, можно существенно подчистить этот код, применив синтаксис лямбда-выражений вместо анонимных делегатов. Тема лямбда-выражений раскрывается в главе 15.

В качестве финального примера применения расширяющего метода для операций над типом IList<T> рассмотрим, как можно создать расширяющий метод для обращения списка и возврата нового экземпляра IList<T>. Можно предложить несколько способов сделать это, и некоторые из них значительно эффективнее прочих. Однако я хочу показать вам пример, использующий форму рекурсии. Рассмотрим следующую реализацию метода Reverse<T>:

```
public static class CustomIterators
{
    public static IList<T> Reverse<T>( this IList<T> theList ) {
        var reverseList = new List<T>();
        Func<IList<T>, List<T>> reverseFunc = null;
        reverseFunc = delegate(IList<T> list) {
            if( list != null ) {
                reverseFunc( list.Tail );
                if( list.Tail != null ) {
                    reverseList.Add( list.Head );
                }
            }
```

```
            return reverseList;
        };
        return MyList<T>.CreateList( reverseFunc(theList) );
    }
}
```

Если вы никогда не встречали такой стиль кодирования, он наверняка вывернет вам мозги наизнанку. Ключ к работе определяется фактом определения делегата, вызывающего самого себя и захватывающего по пути локальные переменные[3]. В приведенном выше коде анонимный метод присваивается переменной `reverseFunc`. И, как видите, тело анонимного метода вызывает `reverseFunc`, т.е. самого себя! Таким образом, анонимный метод захватывает самого себя. Триггер, запускающий выполнение всей работы, находится в последней строке метода `Reverse<>`. Он инициирует цепочку рекурсивных вызовов анонимного метода и затем передает результирующий `List<T>` методу `CreateList`, тем самым создавая список в обратном порядке.

Те из вас, кто уделяет внимание эффективности, вероятно, отметят неэффективность создания временного экземпляра `List<T>`, который затем передается `CreateList`. Например, если конструктор `MyList<T>` сделать общедоступным, то можно полностью исключить временный `List<T>` и строить новый `MyList<T>`, используя захваченную переменную, как показано ниже:

```
public static class CustomIterators
{
    public static IList<T> Reverse<T>( this IList<T> theList ) {
        var reverseList = new MyList<T>(default(T), null);
        Func<IList<T>, MyList<T>> reverseFunc = null;
        reverseFunc = delegate(IList<T> list) {
            if( list.Tail != null ) {
                reverseList = new MyList<T>( list.Head, reverseList );
                reverseFunc( list.Tail );
            }
            return reverseList;
        };
        return reverseFunc(theList);
    }
}
```

Те из вас, кто ближе знаком с функциональным программированием, возможно, скажут, что расширяющий метод `Reverse<T>` можно подчистить, исключив захватываемую переменную и используя вместо нее стек. Таким образом, мой окончательный пример метода `Reverse<T>` использует только стек в качестве временного хранилища при построении нового обращенного списка:

```
public static class CustomIterators
{
    public static IList<T> Reverse<T>( this IList<T> theList ) {
        Func<IList<T>, IList<T>, IList<T>> reverseFunc = null;
```

[3] Фанаты компьютерных наук любят называть делегат, захватывающий переменные, замыканием (closure), которое представляет собой конструкцию, где функция упаковывается вместе с окружением (таким как переменные).

```
        reverseFunc = delegate(IList<T> list, IList<T> result) {
            if( list.Tail != null ) {
             return reverseFunc( list.Tail, new MyList<T>(list.Head, result) );
            }
            return result;
        };
        return reverseFunc(theList, new MyList<T>(default(T), null));
    }
}
```

На заметку! Этот код использует определение `Func<>`, которое является обобщенным делегатом в пространстве имен `System`. Применение `Func<>` — это сокращение, которое вы можете использовать, чтобы избежать необходимости в отдельном объявлении типов делегатов. Вы используете параметр типа `Func<>` для типов параметров (если есть) и типа возврата делегата.

Класс `MyList<T>`, использованный в предыдущем примере, строит связный список из типа `IEnumerable<T>` целиком перед применением объекта `MyList<T>`. Я использовал `List<T>` в качестве начальных данных, но с тем же успехом мог использовать что угодно, реализующее `IEnumerable<T>`, для наполнения содержимым `MyList<T>`. Но что было бы, если бы `IEnumerable<T>` был бесконечным итератором, подобным тому, что создавался `CreateInfiniteList` в разделе "Цепочки операций"? Если бы вы наполняли `MyList<T>.CreateList` результатом `CreateInfiniteList`, то вам пришлось бы принудительно прерывать программу или ждать, не произойдет ли переполнение памяти при попытке построения `MyList<T>`. Если вы создаете библиотеку общего пользования, содержащую тип `MyList<T>`, который строит себя на основании заданного типа `IEnumerable<T>`, то вам нужно предусмотреть все возможные сценарии, с которыми придется столкнуться. Переданный вам объект `IEnumerable<T>` может потребовать очень много времени на выполнение вычисления каждого элемента перечисления. Например, он может это делать на основе живой информации из базы данных, доступ к которой обходится очень дорого. За примером создания списка в "ленивой" манере, в котором каждый узел создается по мере необходимости, загляните в блестящий блог Веса Дайера (Wes Dyer), озаглавленный "Why all the love for lists?" ("Почему все любят списки?")[4]. Техника "ленивого" вычисления при итерации — фундаментальное средство LINQ, о чем я расскажу в главе 16.

Шаблон Visitor

Шаблон Visitor (Визитер), как он описан в фундаментальной книге "банды четырех" Design Patterns: Elements of Reusable Object-Oriented Software[5], позволяет вам определять новую операцию над группой классов, не изменяя самих этих классов. Расширяющие методы являются удобным средством реализации шаблона Visitor.

---

[4] Этот блог находится по адресу `blogs.msdn.com/wesdyer/`.

[5] Книга Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) Design Patterns: Elements of Reusable Object-Oriented Software (Boston, MA: Addison-Wesley Professional, 1995 г.) включена в список полезных ссылок в конце этой книги.

Например, рассмотрим коллекцию типов, которые могут быть или не быть связанными отношением наследования, и предположим, что вы хотите добавить функциональность для проверки их экземпляров в некоторой точке вашего приложения. Один вариант, хотя и очень привлекательный, заключается в модификации общедоступного контракта всех типов введением нового метода `Validate` в каждый из них. Кто-то может даже прийти к заключению, что простейший путь — ввести новый базовый тип, унаследованный от `System.Object` и реализующий `Validate` как абстрактный метод, с последующим наследованием всех прочих типов от него вместо `System.Object`. И то, и другое не приведет ни к чему иному, кроме как к сплошному кошмару при сопровождении.

Думаю, вы согласитесь, что расширяющий метод или коллекция расширяющих методов лучше справятся с задачей. Имея коллекцию не связанных между собой типов, вы, вероятно, реализуете целую группу расширяющих методов. Но вся красота заключается в том, что вам не нужно изменять ранее определенные типы. Фактически, если они изначально не разработаны вами, вы все равно не можете их изменять. Рассмотрим следующий код:

```
using System;
using ValidatorExtensions;
namespace ValidatorExtensions
{
    public static class Validators
    {
        public static void Validate( this String str ) {
            // Что-то делать для проверки экземпляра String.
            Console.WriteLine( "Экземпляр String с \"" +
                               str +
                               "\" проверен." );
        }

        public static void Validate( this SupplyCabinet cab ) {
            // Что-то делать для проверки экземпляра SupplyCabinet.
            Console.WriteLine( "Экземпляр SupplyCabinet проверен." );
        }
        public static void Validate( this Employee emp ) {
         // Что-то делать для проверки экземпляра Employee.
         Console.WriteLine( "** Сбой при проверке экземпляра Employee! **" );
        }
    }
}
public class SupplyCabinet
{
}
public class Employee
{
}
public class MyApplication
{
    static void Main() {
        String data = "некоторые важные данные";
```

```
        SupplyCabinet supplies = new SupplyCabinet();
        Employee hrLady = new Employee();
        data.Validate();
        supplies.Validate();
        hrLady.Validate();
    }
}
```

Обратите внимание, что для каждого типа объекта, который мы хотим проверить (в данном примере их три), я определил отдельный расширяющий метод `Validate`. Вывод этого приложения показывает, что для каждого экземпляра вызывается правильный метод `Validate`:

```
Экземпляр String с "some important data" проверен.
Экземпляр SupplyCabinet проверен.
** Сбой при проверке экземпляра Employee! **
```

В данном примере важно отметить, что визитеры — в данном случае, расширяющие методы по имени `Validate` — должны трактовать экземпляры, которые они проверяют, как черные ящики. Я имею в виду, что они сами не обладают такими способностями проверки, как настоящие методы экземпляра, поскольку только настоящие методы экземпляра имеют доступ к внутреннему состоянию объекта. Тем не менее, в данном примере имеет смысл проверять экземпляры с точки зрения клиента.

Используя обобщения и ограничения, вы можете слегка расширить предыдущий пример, и представить обобщенную форму расширяющего метода `Validate`, который может быть использован, если экземпляр поддерживает хорошо известный интерфейс. В данном случае этот интерфейс носит имя `IValidator`. Таким образом, будет неплохо создать специальный метод `Validate`, который будет вызван, если тип реализует интерфейс `IValidator`. Рассмотрим следующий код, в котором изменения выделены полужирным:

```
using System;
using ValidatorExtensions;
namespace ValidatorExtensions
{
    public static class Validators
    {
        public static void Validate( this String str ) {
            // Что-то делать для проверки экземпляра String.
            Console.WriteLine( "Экземпляр String с \"" +
                               str +
                               "\" проверен." );
        }
        public static void Validate( this Employee emp ) {
         // Что-то делать для проверки экземпляра Employee.
         Console.WriteLine( "** Сбой при проверке экземпляра Employee! **" );
        }
        public static void Validate<T>( this T obj )
                             where T: IValidator {
            obj.DoValidation();
```

```
            Console.WriteLine( "Проверен экземпляр следующего" +
                               " типа: " +
                               obj.GetType() );
        }
    }
}
public interface IValidator
{
    void DoValidation();
}
public class SupplyCabinet : IValidator
{
    public void DoValidation() {
        Console.WriteLine( "\tПроверка SupplyCabinet" );
    }
}
public class Employee
{
}
public class MyApplication
{
    static void Main() {
        String data = "некоторые важные данные";
        SupplyCabinet supplies = new SupplyCabinet();
        Employee hrLady = new Employee();
        data.Validate();
        supplies.Validate();
        hrLady.Validate();
    }
}
```

Теперь, если экземпляр, на котором мы вызываем Validate, реализует IValidator, и не существует версии Validate, которая принимает в точности его тип в первом параметре, то будет вызвана обобщенная форма Validate, которая затем передается методу DoValidation на экземпляре.

Обратите внимание, что я удалил расширяющий метод, чей первый параметр имел тип SupplyCabinet, чтобы компилятор мог выбрать обобщенную версию. Если бы я оставил его, то код метода Main, как он сформулирован выше, вызвал бы версию, которую я удалил. Однако даже если бы я не удалил необобщенный расширяющий метод, я мог бы заставить компилятор вызвать обобщенную версию, изменив синтаксис в точке вызова, как показано ниже:

```
public class MyApplication
{
    static void Main() {
        String data = "некоторые важные данные";
        SupplyCabinet supplies = new SupplyCabinet();
        Employee hrLady = new Employee();
        data.Validate();
```

```
        // Принудительно вызвать обобщенную версию
        supplies.Validate<SupplyCabinet>();

        hrLady.Validate();
    }
}
```

В методе Main я предоставил компилятору больше информации, чтобы ограничить его в поиске метода Validate только обобщенной формой расширяющего метода, принимающего один обобщенного типа параметр.

Резюме

В этой главе я представил расширяющие методы, включая способы их объявления и вызова, а также описание того, как компилятор реализует их "за кулисами". Вдобавок мы видели, как они служат "синтаксической приправой", не требуя для своей работы никаких изменений исполняющей системы. Расширяющие методы могут вызвать путаницу при неправильном применении, поэтому мы рассмотрели некоторые ловушки, которых следует избегать. Я показал вам, как эти методы могут быть использованы для создания таких полезных вещей, как итераторы (типы IEnumerable<T>) для контейнеров, которые не являются перечислимыми по умолчанию. Даже для типов, имеющих перечислители, вы можете определить альтернативные перечислители, работающие специальным образом. Как вы увидите в главе 15, когда они комбинируются с лямбда-выражениями, расширяющие методы предоставляют определенную степень выразительности, которая чрезвычайно полезна. Показывая способ создания пользовательских итераторов, я предпринял обходной маневр (используя анонимные функции вместо лямбда-выражений), чтобы продемонстрировать вам мир функционального программирования, открытый новыми средствами C# 3.0. Код для этих примеров станет намного яснее, когда вы используете лямбда-выражения вместо анонимных методов.

В следующей главе я представлю вам лямбда-выражения, которые действительно обеспечивают возможность функционального программирования на C# в синтаксически сжатом виде. Вдобавок они позволяют вам конвертировать функциональные выражения либо в код, или в данные в форме IL-кода, либо в дерево выражений, соответственно.

ГЛАВА 15

Лямбда-выражения

Большинство новых средств C# 3.0 открывают программистам на C# мир выразительного функционального программирования. Функциональное программирование в его чистом виде — это методология программирования, построенная поверх неизменяемых переменных (иногда называемых символами), функций, которые могут производить другие функции, и рекурсии; и это лишь несколько его основ. К выдающимся языкам функционального программирования можно отнести Lisp, Haskell, F#[1] и Scheme[2]. Однако функциональное программирование не требует специального функционального языка, и вы можете с успехом реализовать его на традиционных императивных языках, таких как все C-подобные языки (включая C#). Новые средства C# 3.0 трансформируют язык в более выразительный гибридный язык, в котором приемы императивного и функционального программирования сосуществуют в гармонии. Лямбда-выражения — несомненно, самый большой кусок этого пирога функционального программирования.

Введение в лямбда-выражения

Используя лямбда-выражения, вы можете кратко определять функциональные объекты для использования в любое время. C# всегда поддерживал эту возможность через делегаты, посредством которых вы создаете функциональный объект (в форме делегата) и привязываете к нему код обратного вызова во время создания. Лямбда-выражения связывают эти два действия — создание и подключение — в один выразительный оператор кода. Вдобавок вы можете легко ассоциировать среду с функциональными объектами. *Функционал* (functional) — это функция, принимающая функции в своем списке параметров и оперирующая этими функциями, возможно, даже возвращая другую функцию в результате. Например, функционал может принимать две функции, одна из которых выполняет одну математическую операцию, а другая — другую математическую операцию, и возвращать третью

---

[1] F# — выдающийся новый язык функционального программирования для .NET Framework. Подробную информацию об этом языке читайте в книге Роберта Пикеринга (Robert Pickering) *Foundations of F#* (Berkeley, CA: Apress, 2007 г.).

[2] Один из языков, которые я часто использую — C++. Те из вас, кто знаком с метапрограммированием на C++, определенно знакомы и с приемами функционального программирования. Если вы используете C++ и интересуетесь метапрограммированием, загляните в блестящую книгу Дэвида Абрахамса (David Abrahams) и Алексея Гуртового (Aleksey Gurtovoy) *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* (Boston, MA: Addison-Wesley Professional, 2004 г.).

функцию, представляющую комбинацию первых двух. Лямбда-выражения предоставляют более естественный способ создания и вызова функционалов.

В простых синтаксических терминах лямбда-выражение — это синтаксис, посредством которого вы можете объявлять анонимные функции (делегаты) более гладким и выразительным способом. Вообще говоря, нет причин, почему бы нельзя было реализовать технику функционального программирования на C# 2.0[3]. Во-первых, синтаксис лямбда-выражений может потребовать некоторого времени на привыкание к нему. Вообще синтаксис лямбда-выражений очень прямолинеен. Однако при встраивании его в код бывает не совсем легко расшифровать его и привыкнуть им пользоваться.

Лямбда-выражения принимают две формы. Форма, которая наиболее прямо заменяет анонимные методы, представляет собой блок кода, заключенный в фигурные скобки. Я предпочитаю называть это лямбда-операторами. Такие лямбда-операторы — прямая замена анонимных методов. Лямбда-выражения, с другой стороны, предоставляют еще более сокращенный способ объявлять анонимный метод и не требуют ни кода в фигурных скобках, ни оператора `return`. Оба типа лямбда-выражений могут быть преобразованы в делегаты. Однако лямбда-выражения без блоков операторов представляют собой нечто действительно впечатляющее. Вы можете преобразовать их в деревья выражений с помощью типов из пространства имен `System.Linq.Expressions`. Другими словами, функция, описанная в коде, превращается в данные. Тему создания деревьев выражений из лямбда-выражений я раскрою ниже, в разделе "Деревья выражений" настоящей главы.

Лямбда-выражения

Для начала рассмотрим простейшую форму лямбда-выражений; те, что не содержат в себе блока операторов. Как упоминалось в предыдущем разделе, лямбда-выражение — это сокращенный способ объявления простого анонимного метода. Следующее лямбда-выражение может быть использовано в качестве делегата, принимающего один параметр и возвращающего результат выполнения операции над параметром:

```
x => x / 2
```

Это говорит следующее: "взять `x` в качестве параметра и вернуть результат следующей операции в `x`". Обратите внимание, что лямбда-выражение лишено информации о типе. Это не значит, что выражение не имеет типа. Вместо этого компилятор выводит тип аргумента и тип результата из контекста его использования, и отсюда следует, что если вы присваиваете лямбда-выражение делегату, типы определения делегата используются для определения типов внутри лямбда-выражения. Следующий код показывает, что случается, когда лямбда-выражение присваивается типу делегата:

```
using System;
using System.Linq;
public class LambdaTest
{
    static void Main() {
```

[3] Некоторые примеры функционального программирования с анонимными методами приводятся в главе 14.

```
        Func<int, double> expr = x => x / 2;
        int someNumber = 9;
        Console.WriteLine( "Результат: {0}", expr(someNumber) );
    }
}
```

Я выделил лямбда-выражение полужирным, чтобы подчеркнуть его. Тип `Func<>` — это новый вспомогательный тип, представленный в пространстве имен `System`, который вы можете использовать для объявления простых делегатов, принимающих до четырех аргументов и возвращающих результат. В данном случае я объявляю переменную `expr`, которая является делегатом, принимающим `int` и возвращающим `double`. Когда компилятор присваивает лямбда-выражение переменной `expr`, он использует информацию о типе делегата для определения того, что типом `x` должен быть `int`, а типом возврата — `double`.

Теперь, если вы выполните этот код, то заметите, что результат не совсем точен. То есть результат был округлен. Этого следовало ожидать, поскольку результат `x/2` представлен как `int`, который затем приводится к `double`. Вы можете исправить это, специфицируя различные типы в объявлении делегата, как показано ниже:

```
using System;
using System.Linq;
public class LambdaTest
{
    static void Main() {
        Func<double, double> expr = (double x) => x / 2;
        int someNumber = 9;
        Console.WriteLine( "Result: {0}", expr(someNumber) );
    }
}
```

Лямбда-выражение имеет то, что называется явно типизированным списком параметров, и в этом случае `x` объявлена с типом `double`. Также обратите внимание, что тип expr теперь — `Func<double, double>` вместо `Func<int, double>`. Компилятор требует, чтобы всякий раз, когда вы используете типизированный список параметров в лямбда-выражении и присваиваете его делегату, то типы аргументов делегата должны совпадать в точности. Однако поскольку `int` явно преобразуем в `double`, вы можете передать `someNumber` в `expr` во время вызова, как было показано.

На заметку! При использовании типизированных списков параметров обратите внимание, что такой список должен быть заключен в скобки. Скобки также требуются при объявлении делегата, принимающего или больше одного параметра либо вообще не принимающего параметров, как я покажу позже. Фактически вы можете использовать скобки в любое время; они не обязательны в лямбда-выражениях с только одним типизированным параметром.

Когда лямбда-выражение присваивается делегату, тип возврата выражения обычно выводится из типов аргументов. Поэтому в следующем фрагменте кода тип возврата выражения — `double`, поскольку предполагаемый тип параметра **x** — `double`:

```
Func<double, int> expr = (x) => x / 2; // Ошибка компиляции!!!!
```

Однако, поскольку double не конвертируется неявно в int, компилятор выдает ошибку:

```
error CS1662: Cannot convert 'lambda expression' to
  delegate type 'System.Func<double,int>' because some of the return
  types in the block are not implicitly convertible to the delegate return type
```

```
ошибка CS1662: Не удается преобразовать 'лямбда-выражение' к типу
  делегата 'System.Func<double,int>', т.к. некоторые из типов возврата
  в блоке не являются неявно преобразуемыми к возвращаемому типу делегата
```

Исправить это можно, приведя результат тела лямбда-выражения к int:

```
Func<double, int> expr = (x) => (int) x / 2;
```

На заметку! Явные типы в списке параметров лямбда-выражения необходимы, если делегат, которому вы его присваиваете, имеет out- или ref-параметры. Кто-то скажет, что явная фиксация типов параметров внутри лямбда-выражений лишает его элегантности и выразительной мощи. И это определенно затрудняет чтение кода.

Теперь я хочу показать вам простое лямбда-выражение, не принимающее параметров:

```
using System;
using System.Linq;
public class LambdaTest
{
    static void Main() {
        int counter = 0;
        WriteStream( () => counter++ );
        Console.WriteLine( "Финальное значение счетчика: {0}",
                           counter );
    }
    static void WriteStream( Func<int> counter ) {
        for( int i = 0; i < 10; ++i ) {
            Console.Write( "{0}, ", counter() );
        }
        Console.WriteLine();
    }
}
```

Обратите внимание, насколько просто с использованием лямбда-выражения передать функцию в качестве параметра в метод WriteStream. Более того, переданная функция захватывает окружение, внутри которого она выполняется, а именно — значение counter в Main. В старые добрые времена C# 1.0 это было болезненным процессом и приходилось делать нечто вроде следующего:

```
using System;
unsafe public class MyClosure
{
    public MyClosure( int* counter )
    {
        this.counter = counter;
    }
```

```
    public delegate int IncDelegate();
    public IncDelegate GetDelegate() {
        return new IncDelegate( IncrementFunction );
    }
    private int IncrementFunction() {
        return (*counter)++;
    }
    private int* counter;
}
public class LambdaTest
{
    unsafe static void Main() {
        int counter = 0;
        MyClosure closure = new MyClosure( &counter );
        WriteStream( closure.GetDelegate() );
        Console.WriteLine( "Финальное значение счетчика: {0}",
                           counter );
    }
    static void WriteStream( MyClosure.IncDelegate incrementor ) {
        for( int i = 0; i < 10; ++i ) {
            Console.Write( "{0}, ", incrementor() );
        }
        Console.WriteLine();
    }
}
```

Посмотрите, насколько много работы требовалось выполнить, чтобы обойтись без лямбда-выражений. Я выделил полужирным дополнительный код и прочие изменения. Первая задача — создать объект для представления делегата и его окружения. В данном случае среда — это указатель на переменную `counter` в методе `Main`. Я решил использовать класс для инкапсуляции функции и ее окружения. Обратите внимание на использование небезопасного кода в классе `MyClass` для достижения этого. Затем в методе `Main` я создал экземпляр `MyClosure` и передал делегат, созданный вызовом `GetDelegate` методу `WriteStream`.

Какой объем работы! К тому же и понять такой код совсем нелегко.

В C# 2.0 появились анонимные методы, чтобы уменьшить это бремя. Однако они были не настолько функционально выразительны как лямбда-выражения, поскольку все еще придерживались старого императивного стиля программирования и требовали явного указания типов в списке параметров. Вдобавок синтаксис анонимных методов довольно громоздкий. Для наглядности следующий код демонстрирует, как можно было бы реализовать предыдущий пример на основе анонимных методов, чтобы вы увидели отличие в синтаксисе от лямбда-выражений.

```
using System;
public class LambdaTest
{
    static void Main() {
        int counter = 0;
        WriteStream( delegate () {
                         return counter++;
                     } );
```

```
    Console.WriteLine( "Финальное значение счетчика: {0}",
                       counter );
  }
  static void WriteStream( Func<int> counter ) {
     for( int i = 0; i < 10; ++i ) {
        Console.Write( "{0}, ", counter() );
     }
     Console.WriteLine();
  }
}
```

Я выделил отличия между этим и исходным примером с лямбда-выражением. Определенно, он намного яснее, чем способ, которым приходилось пользоваться во времена C# 1.0. Однако он все еще не столь выразителен и краток, как версия с лямбда-выражением. И, наконец, в C# 3.0 мы получили элегантное средство определения потенциально очень сложных функций с применением лямбда-выражений, которые могут быть построены посредством сборки вместе других функций.

На заметку! В предыдущем примере кода вы, вероятно, заметили последствия обращения к переменной `counter` внутри лямбда-выражения. В конце концов, `counter` — это локальная переменная внутри контекста `Main`, однако внутри контекста `WriteStream` на нее ссылаются при вызове делегата. В разделе "Остерегайтесь сюрпризов захваченных переменных" главы 10 я описал, как вы можете достичь того же результата с помощью анонимных методов. В терминологии функционального программирования это называется *замыканием* (closure). По сути, всякий раз, когда лямбда-выражение включает окружающую его среду, в результате получается такое замыкание. Как я покажу в следующем разделе, замыкания могут быть очень полезны. Однако, применяемые неправильно, замыкания чреваты неприятными сюрпризами.

И, наконец, я хочу показать вам пример лямбда-выражения, принимающего более одного параметра:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class LambdaTest
{
   static void Main() {
      var teamMembers = new List<string> {
         "Lou Loomis",
         "Smoke Porterhouse",
         "Danny Noonan",
         "Ty Webb"
      };
      FindByFirstName( teamMembers,
                       "Danny",
                       (x, y) => x.Contains(y) );
   }
   static void FindByFirstName(
                    List<string> members,
                    string firstName,
                    Func<string, string, bool> predicate ) {
```

```
            foreach( var member in members ) {
                if( predicate(member, firstName) ) {
                    Console.WriteLine( member );
                }
            }
        }
    }
```

В данном случае лямбда-выражение используется для создания делегата, принимающего два параметра типа `string` и возвращающего `bool`. Как вы можете видеть, лямбда-выражение представляет симпатичный и краткий способ создания предикатов. В разделе "Вернемся к итераторам и генераторам" далее в главе я представлю новую версию примера из главы 14, демонстрирующую использование лямбда-выражений в качестве предикатов для создания гибких итераторов.

Лямбда-операторы

Все лямбда-выражения, которые я продемонстрировал до сих пор, относились к типу простых выражений. Другой тип лямбда-выражений — те, которые я предпочитаю называть лямбда-операторами. По форме они подобны лямбда-выражениями из предыдущего раздела, но с тем отличием, что построены из составного блока операторов внутри фигурных скобок. По этой причине лямбда с блоками операторов должны иметь оператор `return`. Вообще все лямбда-выражения, показанные в предыдущем разделе, могут быть преобразованы в лямбда с блоком операторов, если их просто окружить фигурными скобками, предварив оператором `return`. Например, следующее лямбда-выражение:

```
(x, y) => x * y
```

может быть переписано в виде блока операторов:

```
(x, y) => { return x * y; }
```

В таком виде лямбда с блоками операторов почти идентичны анонимным методам. И есть одно главное отличие между лямбда с блоками операторов и простыми лямбда-выражениями. Первые могут быть преобразованы только в типы делегатов, в то время как вторые — и в делегаты, и в деревья выражений, посредством семейства типов, сосредоточенных вокруг `System.Linq.Expressions.Expression<T>`. О деревьях выражений мы поговорим в следующем разделе.

> **На заметку!** Большая разница между лямбда с блоками операторов и анонимными методами заключается в том, что анонимные методы должны явно указывать типы параметров, в то время как для лямбда компилятор почти всегда может выводить типы на основе контекста. Сокращенный синтаксис, представленный лямбда-выражениями, стимулирует в большей степени образ мышления и подходы функционального программирования.

Деревья выражений

До сих пор я показывал вам лямбда-выражения, подменяющие функциональность делегатов. Но, остановившись на этом, я оказал бы вам плохую услугу. Дело в том; что компилятор C# 3.0 также обладает способностью преобразовывать

лямбда-выражения в деревья выражений на основе типов из пространства имен System.Linq.Expressions. Позднее, в разделе "Функции как данные", я объясню, чем они хороши. Например, вы уже видели, как можно конвертировать лямбда-выражение в делегат, как показано ниже:

```
Func<int, int> func1 = n => n+1;
```

В этой строке кода выражение преобразуется в делегат, принимающий единственный целочисленный параметр и возвращающий int. Однако взгляните на следующую модификацию:

```
Expression<Func<int, int>> expr = n => n+1;
```

Вот это действительно круто! Лямбда-выражение, вместо того, чтобы конвертироваться в вызываемый делегат, преобразуется в структуру данных, представляющую операцию. Типом переменной expr является Expression<T>, где T заменяется типом делегата, в который может быть преобразовано лямбда-выражение. Компилятор замечает, что вы пытаетесь конвертировать лямбда-выражение в экземпляр Expression<Func<int,int>>, и генерирует весь необходимый код, чтобы это произошло. В некоторый момент позднее вы можете скомпилировать выражение в полезный делегат, как показано в следующем примере:

```
using System;
using System.Linq;
using System.Linq.Expressions;
public class EntryPoint
{
    static void Main() {
        Expression<Func<int, int>> expr = n => n+1;
        Func<int, int> func = expr.Compile();
        for( int i = 0; i < 10; ++i ) {
            Console.WriteLine( func(i) );
        }
    }
}
```

Выделенная полужирным строка показывает шаг, на котором выражение компилируется в делегат. Если вы немножко задумаетесь, то сможете представить себе, как вы могли бы модифицировать это дерево выражений или даже комбинировать несколько деревьев выражений для создания более сложных деревьев выражений перед тем, как компилировать их. Можно даже определить новый язык выражений или реализовать анализатор для существующего языка выражений. Фактически, компилятор работает как анализатор выражений, когда вы присваиваете лямбда-выражение экземпляру типа Expression<T>. "За кулисами" он генерирует код для построения дерева выражений, и если вы используете ILDASM для просмотра сгенерированного кода, то увидите его в действии. Предыдущий пример может быть переписан без использования лямбда-выражений, как показано ниже:

```
using System;
using System.Linq;
using System.Linq.Expressions;
```

```
public class EntryPoint
{
    static void Main() {
        var n = Expression.Parameter( typeof(int), "n" );
        var expr = Expression<Func<int,int>>.Lambda<Func<int,int>>(
                       Expression.Add(n, Expression.Constant(1)),
                       n );

        Func<int, int> func = expr.Compile();

        for( int i = 0; i < 10; ++i ) {
            Console.WriteLine( func(i) );
        }
    }
}
```

Выделенные полужирным строки заменяют единственную строку предшествующего примера, где переменной `expr` присваивается лямбда-выражение `n => n+1`. Думаю, вы согласитесь, что первый пример читать намного легче. Однако этот длинный пример помогает выразить действительную гибкость деревьев выражений. Давайте разобьем на шаги процесс построения выражения. Сначала вы должны представить параметры в списке параметров лямбда-выражения. В данном случае параметр всего один — переменная `n`. Поэтому мы начинаем со следующего:

```
var n = Expression.Parameter( typeof(int), "n" );
```

На заметку! В этих примерах я использую неявно типизированные переменные, чтобы сократить объем ввода и повысить читабельность кода. Напомню, что переменные строго типизированы. Компилятор просто выводит их тип во время компиляции вместо того, чтобы требовать от вас явного указания.

Эта строка кода говорит о том, что нам нужна переменная по имени `n`, относящаяся к типу `int`. Напомню, что в простом лямбда-выражении тип может быть определен на основе предоставленного типа делегата. Теперь нам нужно сконструировать экземпляр `BinaryExpression`, представляющий операцию сложения, как показано ниже:

```
Expression.Add(n, Expression.Constant(1))
```

Здесь я говорю, что выражение `BinaryExpression` должно состоять из прибавления константы — числа `1` — к параметру `n`. Наверное, вы уже ухватили суть. Каркас реализует форму шаблона проектирования Abstract Factory (Абстрактная фабрика) для создания экземпляров элементов выражения. То есть вы не можете создать новый экземпляр `BinaryExpression` или любого другого строительного блока деревьев выражений, и потому должны использовать статические методы класса `Expression` для создания этих экземпляров. Это дает нам, как потребителям, гибкость в выражении того, что мы хотим, и позволяет реализации `Expression` решать, какой тип нам действительно нужен.

Теперь, когда мы имеем `BinaryExpression`, мы должны использовать метод `Expression.Lambda<>` для привязки выражения (в данном случае `n+1`) с параметрами в списке параметров (в данном случае — `n`). Обратите внимание, что в примере

я использую обобщенный метод Lambda<>, так что могу создать тип Expression<Func<int,int>>. Применение обобщенной формы предоставляет компилятору больше информации о типе, чтобы перехватывать любые ошибки, которые я мог бы внести, во время компиляции, не позволяя им нарушить работу приложения во время выполнения.

На заметку! Если бы я использовал не обобщенную версию метода Expression.Lambda, то в результате получился бы экземпляр LambdaExpression. LambdaExpression также реализует метод Compile; однако вместо строго типизированного делегата он возвращает экземпляр типа Delegate. Прежде чем вы сможете вызвать экземпляр Delegate, вы должны привести его к определенному типу делегата, в данном случае — Func<int, int>, или к другому делегату с той же сигнатурой, либо же вы должны будете вызвать DynamicInvoke на делегате. Любой из этих способов может привести к генерации исключения во время выполнения, если обнаружится несоответствие между вашим выражением и типом делегата, который, как вы думаете, оно должно генерировать.

Операции над выражениями

Теперь я хотел бы продемонстрировать пример того, как можно взять дерево выражений, сгенерированное из лямбда-выражения, и модифицировать его для создания нового дерева выражений. В данном случае я возьму выражение (n+1) и превращу его в 2*(n+1):

```
using System;
using System.Linq;
using System.Linq.Expressions;
public class EntryPoint
{
    static void Main() {
        Expression<Func<int,int>> expr = n => n+1;
        // Теперь присвоим expr значение исходного
        // выражения, умноженное на 2.
        expr = Expression<Func<int,int>>.Lambda<Func<int,int>>(
                Expression.Multiply( expr.Body,
                                     Expression.Constant(2) ),
                expr.Parameters );
        Func<int, int> func = expr.Compile();
        for( int i = 0; i < 10; ++i ) {
           Console.WriteLine( func(i) );
        }
    }
}
```

Выделенные полужирным строки показывают стадию, на которой я умножаю исходное лямбда-выражение на 2. Очень важно отметить, что параметры, переданные методу Lambda<>, должны быть именно теми же экземплярами параметров, которые пришли из исходного выражения; т.е. expr.Parameters. Это обязательно. Вы не можете передать методу Lambda<> новый экземпляр ParameterExpression; в противном случае во время выполнения вы получите исключение, подобное описанному ниже, поскольку новый экземпляр ParameterExpression, даже имея то же имя, на самом деле является совершенно другим экземпляром параметра.

```
System.InvalidOperationException: Lambda Parameter not in scope
```

System.InvalidOperationException: Лямбда-параметр не находится в области определения

Существует много классов, унаследованных от класса `Expression`, и много статических методов создания его экземпляров и комбинации с другими выражениями. Я не стану здесь описывать их все. Поэтому я рекомендую обратиться к библиотеке документации MSDN, где содержатся все фантастические подробности о пространстве имен `System.Linq.Expressions`.

Функции как данные

Если вы когда-либо изучали функциональные языки вроде Lisp, то можете заметить сходство между деревьями выражений и тем, как Lisp и подобные ему языки представляют функции как структуры данных. Большинство людей сталкиваются с Lisp в академической среде, и часто изучаемые ими концепции оказываются неприменимыми в реальном мире. Но перед тем как вы решите, что деревья выражений — одно из таких академических упражнений, я хочу показать, насколько они могут быть полезны.

Как вы, возможно, можете предположить, внутри контекста C# 3.0 деревья выражений чрезвычайно полезны в применении к LINQ. Полное представление о LINQ я приведу в главе 16, а пока самый важный факт: LINQ представляет естественный для языка, выразительный синтаксис описания операций над данными, которые невозможно естественным образом смоделировать объектно-ориентированным способом. Например, вы можете создать выражение LINQ для поиска в большом массиве, находящемся в памяти (или другом типе `IEnumerable`), элементов, соответствующих определенному шаблону. LINQ — расширяемый язык и может предоставлять средства оперирования с другими типами хранилищ, такими как XML и реляционные базы данных. Фактически C# 3.0 представляет реализацию LINQ для реляционных баз данных (включая LINQ to SQL, LINQ to Dataset, LINQ to Entities, LINQ to XML и LINQ to Objects), которые все вместе позволяют выполнять операции LINQ на любых типах, поддерживающих `IEnumerable`.

Но как же деревья выражений действуют здесь? Предположим, что вы реализуете LINQ to SQL для запроса к реляционной базе данных. Пользовательская база может находиться на другом конце мира, и может оказаться слишком дорого выполнять простой запрос. К тому же вы не можете представить, насколько сложным может оказаться пользовательское выражение LINQ. Естественно, вы хотите сделать все, что можно для обеспечения максимальной эффективности.

Если выражение LINQ представлено в данных (как дерево выражений), а не в IL (как делегат), то вы можете оперировать с ним. Возможно, у вас есть алгоритм, который может выявлять места, где следует провести оптимизацию, тем самым упрощая выражение. Или, может быть, когда ваша реализация анализирует выражение, вы определяете, что все выражение может быть упаковано, отправлено по сети и полностью выполнено на стороне сервера.

Деревья выражений обеспечивают вам эту важную возможность. Затем, когда вы завершаете операции с данными, вы можете транслировать дерево выражений в окончательную исполняемую операцию посредством механизма типа `LambdaExpression.Compile` и запустить ее. Если бы выражение изначально было

доступно только в виде код IL, ваша гибкость была бы существенно ограничена. Я надеюсь, теперь вы можете оценить действительную мощь деревьев выражений в C# 3.0.

Полезные применения лямбда-выражений

Теперь, после того, как я продемонстрировал, как выглядят лямбда-выражения, давайте рассмотрим некоторые их применения. На самом деле вы можете реализовать большинство следующих примеров на C# 2.0, используя анонимные методы или делегаты. Однако поразительно то, насколько простое синтаксическое дополнение к языку может рассеять туман и открыть богатые возможности в плане выразительности.

Вернемся к итераторам и генераторам

В этой книге я уже пару раз описывал, как вы можете создавать пользовательские итераторы на C#[4]. Теперь я хотел бы продемонстрировать, как можно использовать лямбда-выражения для создания пользовательских итераторов. То, что я хочу здесь подчеркнуть — способ реализации алгоритма в коде, в данном случае алгоритма итерации, который затем превращается в многократно используемый метод, который может применяться почти в любом сценарии.

На заметку! Те из вас, кто программировал на C++ и знаком с применением стандартной библиотеки шаблонов (STL), увидят в этой нотации нечто знакомое. Большинство алгоритмов, определенных в пространстве имен `std` в заголовочном файле `<algorithm>`, требуют для выполнения своей работы предоставления предикатов. Когда STL впервые появилась в начале 90-х годов, она захватила сообщество программистов C++ подобно свежему бризу функционального программирования.

Я хочу показать, как можно выполнять итерацию по обобщенному типу, который может быть или не быть коллекцией в строгом смысле этого слова. Вдобавок вы можете вынести за скобки поведение курсора итерации, а также доступ к текущему значению коллекции. После небольшого размышления то же самое можно сделать почти со всем из метода создания пользовательского итератора, включая тип хранящихся элементов, тип курсора, начальное состояние курсора, конечное его состояние, и способ его продвижения. Все это демонстрируется в следующем примере:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class IteratorExtensions
{
    public static IEnumerable<TItem>
        MakeCustomIterator<TCollection, TCursor, TItem>(
                    this TCollection collection,
```

[4] В главе 9 итераторы представлены через оператор `yield`, а в разделе "Заимствование из функционального программирования" главы 14 рассматриваются пользовательские итераторы.

```
                    TCursor cursor,
                    Func<TCollection, TCursor, TItem> getCurrent,
                    Func<TCursor, bool> isFinished,
                    Func<TCursor, TCursor> advanceCursor) {
        while( !isFinished(cursor) ) {
            yield return getCurrent( collection, cursor );
            cursor = advanceCursor( cursor );
        }
    }
}
public class IteratorExample
{
    static void Main() {
        var matrix = new List<List<double>> {
            new List<double> { 1.0, 1.1, 1.2 },
            new List<double> { 2.0, 2.1, 2.2 },
            new List<double> { 3.0, 3.1, 3.2 }
        };
        var iter = matrix.MakeCustomIterator(
                    new int[] { 0, 0 },
                    (coll, cur) => coll[cur[0]][cur[1]],
                    (cur) => cur[0] > 2 || cur[1] > 2,
                    (cur) => new int[] { cur[0] + 1,
                                         cur[1] + 1 } );
        foreach( var item in iter ) {
            Console.WriteLine( item );
        }
    }
}
```

Смотрите, насколько многократно используемым является MakeCustomIterator<>! Общеизвестно, что для того, чтобы привыкнуть к применению лямбда-синтаксиса, нужно некоторое время, и те, кто привык к императивному стилю кодирования, могут столкнуться с определенными трудностями в его понимании. Обратите внимание, что он принимает три аргумента обобщенного типа. TCollection — это тип коллекции, который в данном примере специфицирован в точке использования как List<List<double>>. TCursor — тип курсора, который в данном случае является простым массивом целых чисел, который может рассматриваться как координаты переменной matrix. А TItem — тип, возвращаемый кодом через оператор yield. Остальные параметры MakeCustomIterator<> — типы делегатов, которые он использует для определения того, как выполнять итерацию по коллекции. Для начала ему нужен способ получения доступа к текущему элементу коллекции, который выражается следующим лямбда-выражением:

```
(coll, cur) => coll[cur[0]][cur[1]]
```

Затем необходим способ определения факта достижения конца коллекции, для чего я применяю следующее лямбда-выражение:

```
(cur) => cur[0] > 2 || cur[1] > 2
```

И, наконец, необходимо знать, как перемещать курсор, что я указываю в следующем лямбда-выражении:

```
(cur) => new int[] { cur[0] + 1, cur[1] + 1 }
```

Другие реализации MakeCustomIterator<> могут принимать первый параметр типа IEnumerable<T>, который в данном примере должен быть IEnumerable<double>. Однако, когда вы накладываете это ограничение, то все, переданное MakeCustomIterator<>, должно реализовывать IEnumerable<>. Переменная matrix реализует IEnumerable<>, но не в той форме, которую легко использовать, поскольку это IEnumerable<List<double>>. Вдобавок вы можете предположить, что коллекция реализует индексатор, как описано в разделе "Индексаторы" главы 4, но тогда это наложит ограничение на повторную применяемость MakeCustomIterator<> и то, какие объекты вы можете использовать в нем. В примере, приведенном выше, индексатор в действительности используется для обращения к конкретному элементу, но его применение вынесено наружу и упаковано в лямбда-выражение, предназначенное для доступа к текущему элементу.

Более того, поскольку операции доступа к текущему элементу коллекции вынесены за скобки, вы можете даже трансформировать данные в исходной переменной matrix в процессе итерации по ней. Например, я мог бы умножить каждое значение на 2 в лямбда-выражении, которое обращается к текущему элементу коллекции, как показано ниже:

```
(coll, cur) => coll[cur[0]][cur[1]] * 2;
```

Можете вы представить, насколько мучительно было реализовать MakeCustomIterator<> с применением делегатов во времена C# 1.0? Именно это я имею в виду, говоря о том, что всего лишь добавление синтаксиса лямбда-выражений в C# 3.0 открывает глаза разработчику на невероятные возможности. Если вы запустите предыдущий пример, то увидите, что я прохожу матрицу по диагонали, что показывает следующий вывод:

```
1
2.1
3.2
```

В качестве финального примера рассмотрим случай, при котором ваш пользовательский итератор даже не выполняет итерации по элементам, а вместо этого использует генератор чисел, как показано ниже:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class IteratorExample
{
    static IEnumerable<T> MakeGenerator<T>( T initialValue,
                                             Func<T, T> advance ) {
        T currentValue = initialValue;
        while( true ) {
            yield return currentValue;
            currentValue = advance( currentValue );
        }
    }
```

```
    static void Main() {
        var iter = MakeGenerator<double>( 1,
                                          x => x * 1.2 );
        var enumerator = iter.GetEnumerator();
        for( int i = 0; i < 10; ++i ) {
            enumerator.MoveNext();
            Console.WriteLine( enumerator.Current );
        }
    }
}
```

После запуска этого кода вы увидите следующий результат:

```
1
1.2
1.44
1.728
2.0736
2.48832
2.985984
3.5831808
4.29981696
5.159780352
```

Вы можете использовать этот метод для бесконечного выполнения, и тогда он остановится только по исключению переполнения памяти либо по принудительному останову. Но главное то, что элементы, по которым выполняется итерация, не существуют в коллекции; вместо этого они генерируются по мере необходимости при каждом перемещении итератора. Вы можете применить эту концепцию многими способами, даже создавая реализацию генератора случайных чисел через итераторы C#.

Замыкания (захват переменной) и мемоизация

В разделе "Остерегайтесь сюрпризов захваченных переменных" главы 10 я описал, как анонимные методы могут захватывать контекст своего лексического окружения. Многие называют этот феномен захватом переменной. На языке функционального программирования это также известно как замыкание (closure)[5]. Ниже показан простой пример замыкания в действии:

```
using System;
using System.Linq;
public class Closures
{
    static void Main() {
        int delta = 1;
        Func<int, int> func = (x) => x + delta;
        int currentVal = 0;
        for( int i = 0; i < 10; ++i ) {
            currentVal = func( currentVal );
```

[5] Более развернутую дискуссию о замыканиях вы найдете по адресу http://en.wikipedia.org/wiki/Closure_%28computer_science%29.

```
            Console.WriteLine( currentVal );
        }
    }
}
```

Переменная delta и делегат func формируют замыкание. Тело выражения ссылается на delta и потому должно иметь доступ к ней при последующем выполнении. Чтобы обеспечить это, компилятор "захватывает" переменную для делегата. "За кулисами" же происходит вот что: тело делегата содержит ссылку на действительную переменную delta. Более того, поскольку захваченная переменная доступна как делегату, так и контексту, содержащему лямбда-выражение, это означает, что захваченная переменная может быть изменена вне контекста и вне связи с делегатом. Такое поведение можно использовать в ваших интересах, но если его не ожидать, оно может вызвать серьезную путаницу.

На заметку! В действительности при формировании замыкания компилятор C# 3.0 берет все эти переменные и упаковывает их в сгенерированный класс. Он также реализует делегат в качестве метода класса. В очень редких случаях вам может понадобиться учитывать это, особенно если обнаружится влияние на производительность при профилировании.

Теперь я хотел бы показать вам хорошее применение замыканий. Одной из основ функционального программирования является то, что сама функция трактуется как первоклассный объект, которым можно манипулировать и оперировать, а также вызывать. Вы уже видели, как лямбда-выражения могут быть преобразованы в деревья выражений, чтобы можно было оперировать ими, производя более или менее сложные выражения. Но один момент, о котором я еще не упомянул — это использование самих функций в качестве строительных блоков для создания новых функций. В качестве быстрой иллюстрации того, что я имею в виду, рассмотрим два лямбда-выражения:

```
x => x * 3
x => x + 3.1415
```

Вы можете создать метод для комбинирования таких лямбда-выражений с целью создания составного лямбда-выражения, как показано ниже:

```
using System;
using System.Linq;
public class Compound
{
    static Func<T, S> Chain<T, R, S>( Func<T, R> func1,
                                      Func<R, S> func2 ) {
        return x => func2( func1(x) );
    }
    static void Main() {
        Func<int, double> func = Chain( (int x) => x * 3,
                                        (int x) => x + 3.1415 );
        Console.WriteLine( func(2) );
    }
}
```

Метод `Chain<>` принимает два делегата и производит третий делегат, комбинируя первые два. В методе `Main` вы можете видеть, как я использую его для производства составного выражения. Делегат, который вы получаете после вызова `Chain<>`, эквивалентен делегату, который вы получаете при преобразовании следующего лямбда-выражения в делегат:

```
x => (x * 3) + 3.1415
```

Метод вроде этого, способный связывать в цепочки произвольные выражения, действительно полезен, но давайте рассмотрим другие способы создания производных функций. Представьте себе операцию, которая требует действительно длительного времени на вычисление. Примерами могут служить операции вычисления факториала или операции вычисления n-го числа Фибоначчи. Пример, который я люблю демонстрировать — обратная константа Фибоначчи, которая выглядит так:

$$\sum_{k=1}^{\infty} \frac{1}{F_k} = 3.35988566\ldots$$

Здесь F_k — число Фибоначчи[6].

Чтобы начать демонстрацию вычислений этой константы, сначала понадобится операция для вычисления n-го числа Фибоначчи:

```
using System;
using System.Linq;
public class Proof
{
    static void Main() {
        Func<int, int> fib = null;
        fib = (x) => x > 1 ? fib(x-1) + fib(x-2) : x;

        for( int i = 30; i < 40; ++i ) {
            Console.WriteLine( fib(i) );
        }
    }
}
```

Первое, что бросается в глаза при взгляде на этот код — формирование процедуры Фибоначчи, т.е. делегата `fib`. Он формирует замыкание на самом себе! Это определенно форма рекурсии, и она делает то, что надо. Однако если вы все-таки запустите этот пример, имея в своем распоряжении суперкомпьютер, то заметите, насколько медленно он работает, даже если все, что он делает — это вычисления с 30-го по 39-е число Фибоначчи! Если это так, нам даже не стоит надеяться продемонстрировать константу Фибоначчи. Такая медлительность обусловлена тем фактом, что каждое число Фибоначчи, которое мы вычисляем, требует немного больше работы, чем вычисление двух предыдущих чисел Фибоначчи, и в результате затрачиваемое время растет лавинообразно.

Эту проблему можно решить, пожертвовав пространством в пользу времени — за счет кэширования чисел Фибоначчи в памяти. Но вместо модификации исходного выражения давайте посмотрим, как можно создать метод, принимающий оригинальные делегаты в качестве параметра и возвращающий новый делегат, который

[6] Weisstein, Eric W. "Reciprocal Fibonacci Constant." From MathWorld — A Wolfram Web Resource. http://mathworld.wolfram.com/ReciprocalFibonacciConstant.html.

заменяет оригинал. Конечная цель — получить возможность заменять первый делегат производным делегатом, не затрагивая код, который его использует. Один из таких приемов называется мемоизацией (memoization)[7]. Это прием, посредством которого функция кэширования возвращает значения, и каждое возвращенное значение ассоциируется с входными параметрами. Это работает только в том случае, если функция не обладает энтропией — в том смысле, что для одних и тех же входных параметров всегда возвращает один и тот же результат. Тогда перед вызовом действительной функции вы сначала проверяете, не вычислялся ли результат для данного параметра ранее, и если да — возвращаете его вместо вызова функции. При очень сложных функциях такая техника требует немного больше пространства памяти, но дает существенный выигрыш в скорости. Рассмотрим пример.

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class Memoizers
{
    public static Func<T,R> Memoize<T,R>( this Func<T,R> func ) {
        var cache = new Dictionary<T,R>();
        return (x) => {
            R result = default(R);
            if( cache.TryGetValue(x, out result) ) {
                return result;
            }
            result = func(x);
            cache[x] = result;
            return result;
        };
    }
}
public class Proof
{
    static void Main() {
        Func<int, int> fib = null;
        fib = (x) => x > 1 ? fib(x-1) + fib(x-2) : x;
        fib = fib.Memoize();
        for( int i = 30; i < 40; ++i ) {
            Console.WriteLine( fib(i) );
        }
    }
}
```

Прежде всего, обратите внимание, что в Main добавился только один дополнительный оператор, в котором я применяю метод Memoize<> к делегату, чтобы произвести новый делегат. Все прочее остается без изменений, так что прозрачная взаимозаменяемость обеспечена. Метод Memoize<> упаковывает оригинальный делегат, который передан через аргумент func с другим замыканием, включающим

[7] Подробнее о мемоизации вы можете прочесть по адресу http://en.wikipedia.org/wiki/Memoization. Кроме того, Вес Дайер (Wes Dyer) отлично описывает мемоизацию в своем блоге http://blogs.msdn.com/wesdyer/archive/2007/01/26/function-memoization.aspx.

экземпляр Dictionary<> для хранения кэшированных значений данного делегата func. В процессе Memoize<> взятие одного делегата и возврат другого реализует кэш, который значительно повышает эффективность. Всякий раз, когда вызывается делегат, он сначала проверяет, нет ли в кэше ранее вычисленного значения.

Внимание! Конечно, мемоизация работает только с функциями, которые детерминировано воспроизводимы — в том смысле, что для одних и тех же параметров гарантируется один и тот же результат. Например, настоящий генератор случайных чисел невозможно подвергнуть мемоизации.

Запустите предыдущий пример и увидите поразительную разницу. Теперь мы вполне можем приступить к вычислению обратной константы Фибоначчи, модифицировав метод Main следующим образом:

```
static void Main() {
   Func<ulong, ulong> fib = null;
   fib = (x) => x > 1 ? fib(x-1) + fib(x-2) : x;
   fib = fib.Memoize();
   Func<ulong, decimal> fibConstant = null;
   fibConstant = (x) => {
      if( x == 1 ) {
         return 1 / ((decimal)fib(x));
      } else {
         return 1 / ((decimal)fib(x)) + fibConstant(x-1);
      }
   };

   fibConstant = fibConstant.Memoize();
   Console.WriteLine( "\n{0}\t{1}\t{2}\t{3}\n",
                      "Номер",
                      "Фибоначчи".PadRight(24),
                      "1/Фибоначчи ".PadRight(24),
                      "Константа Фибоначчи".PadRight(24) );


   for( ulong i = 1; i <= 93; ++i ) {
      Console.WriteLine( "{0:D5}\t{1:D24}\t{2:F24}\t{3:F24}",
                         i,
                         fib(i),
                         (1/(decimal)fib(i)),
                         fibConstant(i) );
   }
}
```

Выделенный полужирным текст показывает делегат, который я создал для вычисления n-й обратной константы Фибоначчи. Вызывая этот делегат все с большим и большим значением x, вы должны заметить, что результат становится все ближе и ближе к обратной константе Фибоначчи. Обратите внимание, что я также осуществил мемоизацию делегата fibConstant. Если этого не делать, можно спровоцировать переполнение стека из-за рекурсии, по мере вызова fibConstant все с большими и большими значениями x. Так что вы можете убедиться, что мемоизация также экономит пространство стека за счет пространства кучи. В каждой

строке вывода код показывает для информации промежуточное значение, но самое интересное значение — в крайней правой колонке. Обратите внимание, что я прекратил вычисление на итерации номер 93. Это потому, что `ulong` на 94-м числе Фибоначчи переполнится. Я мог бы решить эту проблему, используя `BigInteger` из пространства имен `System.Numeric`. Однако это не обязательно, поскольку 93-я итерация обратной константы Фибоначчи, показанная здесь, достаточно близка к цели этого примера.

3.359885666243177553039387

Я выделил полужирным значимые разряды[8]. Думаю, вы согласитесь, что мемоизация чрезвычайно полезна. По этой причине еще много других полезных вещей можно сделать с методами, принимающими функции и производящими другие функции, что я и продемонстрирую в следующем разделе.

Приправа

В предыдущем разделе, посвященном замыканиям, я продемонстрировал, как создать метод, который принимает функцию, переданную в виде делегата, и производит новую функцию. Это очень мощная концепция, и мемоизация, показанная в предыдущем разделе — пример отличного ее применения. В этом разделе я хотел бы продемонстрировать вам технику "приправы" (currying)[9], которая по сути означает создание операции (обычно — метода), принимающей функцию с несколькими параметрами (обычно — делегат) и производящей только один параметр.

На заметку! Если вы — программист C++, знакомый с STL, то, несомненно, использовали операцию "приправы", если имели дело с любой из привязок параметров вроде `Bind1st` и `Bind2nd`.

Предположим, что имеется лямбда-выражение, которое выглядит так:

```
(x, y) => x + y
```

Теперь представьте, что у вас есть список действительных чисел двойной точности, и вы хотите использовать это лямбда-выражение для добавления константного значения к каждому элементу списка, тем самым производя новый список. Было бы здорово создать новый делегат на базе оригинального лямбда-выражения, где одна из переменных стала бы статическим значением. Это понятие называется *привязкой параметра*, и те, кто использовал STL в C++, знакомы с ним. Взгляните на следующий пример, где демонстрируется привязка параметров в действии:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class CurryExtensions
{
```

[8] Вы можете увидеть более точное значение обратной константы Фибоначчи по адресу `http://www.research.att.com/~njas/sequences/A079586`.

[9] Подробнее об этом приеме читайте в `http://en.wikipedia.org/wiki/Currying`.

```
    public static Func<TArg1, TResult>
       Bind2nd<TArg1, TArg2, TResult>(
          this Func<TArg1, TArg2, TResult> func,
          TArg2 constant ) {
       return (x) => func( x, constant );
    }
}

public class BinderExample
{
   static void Main() {
      var mylist = new List<double> { 1.0, 3.4, 5.4, 6.54 };
      var newlist = new List<double>();
      // Здесь - исходное выражение.
      Func<double, double, double> func = (x, y) => x + y;

      // Здесь - "приправленная" функция.
      var funcBound = func.Bind2nd( 3.2 );
      foreach( var item in mylist ) {
         Console.Write( "{0}, ", item );
         newlist.Add( funcBound(item) );
      }
      Console.WriteLine();
      foreach( var item in newlist ) {
         Console.Write( "{0}, ", item );
      }
   }
}
```

"Мясо" этого примера — в расширяющем методе `Bind2nd<>`, который я выделил полужирным. Вы можете видеть, что он создает замыкание и возвращает новый делегат, принимающий только один параметр. Затем, когда вызывается этот новый делегат, он получает только один параметр — первый параметр для исходного делегата, и передает ему константу в качестве второго параметра. Для примера я выполняю итерацию по списку `myList`, при этом строя новый список, содержащийся в переменной `newList`, используя "приправленную" версию оригинального метода, чтобы добавить `3.2` к каждому элементу.

Для сравнения я хочу показать и другой способ "приправы", немного отличающийся от показанного в предыдущем примере:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class CurryExtensions
{
   public static Func<TArg2, Func<TArg1, TResult>>
      Bind2nd<TArg1, TArg2, TResult>(
         this Func<TArg1, TArg2, TResult> func ) {
      return (y) => (x) => func( x, y );
   }
}
```

```
public class BinderExample
{
    static void Main() {
        var mylist = new List<double> { 1.0, 3.4, 5.4, 6.54 };
        var newlist = new List<double>();

        // Здесь - исходное выражение.
        Func<double, double, double> func = (x, y) => x + y;

        // Здесь - "приправленная" функция.
        var funcBound = func.Bind2nd()(3.2);
        foreach( var item in mylist ) {
            Console.Write( "{0}, ", item );
            newlist.Add( funcBound(item) );
        }
        Console.WriteLine();
        foreach( var item in newlist ) {
            Console.Write( "{0}, ", item );
        }
    }
}
```

Я выделил части, отличающиеся от предыдущего примера. В первом примере `Bind2nd<>` возвращал делегат, принимающий один параметр и возвращающий целое число. В данном примере я изменил `Bind2nd<>` так, чтобы он принимал один параметр (значение, привязываемое ко второму параметру исходной функции) и возвращал другой делегат, приправляющий функцию. Обе формы совершенно корректны. Однако приверженцы чистоты стиля могут предпочесть вторую форму первой.

Анонимная рекурсия

В ранее приведенном разделе "Замыкание (захват переменной) и мемоизация" я показал форму рекурсии, использующую замыкания при вычислении чисел Фибоначчи. Для продолжения дискуссии давайте рассмотрим подобное замыкание, которое можно использовать для вычисления факториала числа:

```
Func<int, int> fact = null;
fact = (x) => x > 1 ? x * fact(x-1) : 1;
```

Этот код работает, потому что `fact` формирует замыкание на самом себе и также вызывает себя. То есть вторая строка, где `fact` присваивается лямбда-выражение для вычисления факториала, захватывает сам делегат `fact`. Несмотря на то что такая рекурсия работает, она весьма хрупка, и нужно быть очень осторожным, применяя ее в таком виде.

Напомню, что невзирая на то, что замыкание захватывает переменную для использования внутри анонимного метода, который реализован здесь в виде лямбда-выражения, захваченная переменная остается доступной для изменения вне контекста захватывающего анонимного метода или лямбда-выражения. Например, рассмотрим, что произойдет, если мы поступим следующим образом:

```
Func<int, int> fact = null;
fact = (x) => x > 1 ? x * fact(x-1) : 1;
Func<int, int> newRefToFact = fact;
```

Поскольку объекты в CLR относятся к ссылочным типам, newRefToFact и fact теперь ссылаются на один и тот же делегат. Теперь предположим, что вы сделали нечто вроде следующего:

```
Func<int, int> fact = null;
fact = (x) => x > 1 ? x * fact(x-1) : 1;
Func<int, int> newRefToFact = fact;
fact = (x) => x + 1;
```

Рекурсия разрушена! Заметили, почему? Причина в том, что мы модифицировали захваченную переменную fact. Мы присвоили ей ссылку на новый делегат, основанный на лямбда-выражении (x) => x+1. Но newRefToFact все еще ссылается на лямбда-выражение (x) => x > 1 ? x * fact(x-1) : 1. Однако, когда делегат, на который ссылается newRefToFact, вызывает fact, то он получает новое выражение (x) => x + 1, которое изменяет поведение имеющейся ранее рекурсии.

Существует несколько способов решить эту проблему, но наиболее типичный заключается в использовании анонимной рекурсии[10]. Дело в том, что вы модифицировали лямбда-выражение вычисления факториала для приема другого параметра, являющегося делегатом, который должен быть вызван во время рекурсии. По сути, это удаляет замыкание и преобразует захваченную переменную в параметр делегата. В конечном итоге вы получите нечто вроде следующего:

```
delegate TResult AnonRec<TArg,TResult>( AnonRec<TArg,TResult> f,
                                        TArg arg );
AnonRec<int, int> fact = (f, x) => x > 1 ? x * f(f, x-1) : 1;
```

Суть в том, что вместо рекурсии, полагающейся на захваченную переменную, являющуюся делегатом, вы передаете делегат рекурсии в качестве параметра. В данном примере делегат рекурсии представлен параметром f. Поэтому обратите внимание, что fact не только принимает f как параметр, но вызывает его для рекурсии и затем передает f следующей итерации делегата. По существу, захваченная переменная теперь находится в стеке, поскольку передается каждой рекурсии выражения. Однако, поскольку она в стеке, опасность модификации извне механизма рекурсии исключается.

Чтобы подробнее ознакомиться с этой техникой, настоятельно рекомендую прочесть статью в блоге Веса Дайера (Wes Dyer) "Anonymous Recursion in C#" ("Анонимная рекурсия в C#") по адресу http://blogs.msdn.com/wesdyer. Вес Дьер — один из членов команды C# и ревностный сторонник функционального программирования. В этой статье он демонстрирует, как реализовать Y-комбинатор с фиксированной точкой, обобщающий понятие анонимной рекурсии, показанное ранее[11].

---

[10] Теоретические сведения об анонимной рекурсии вы найдете в статье по адресу http://en.wikipedia.org/wiki/Anonymous_recursion.

[11] Y-комбинаторы с фиксированной точкой описаны по адресу http://en.wikipedia.org/wiki/Fixed_point_combinator.

Резюме

В этой главе я представил синтаксис лямбда-выражений, которые большей частью являются заменой анонимных методов. Фактически, очень жаль, что лямбда-выражения не появились в C# 2.0, потому что тогда не было бы необходимости в анонимных методах. Я продемонстрировал, как вы можете преобразовывать лямбда-выражения без тел операторов в делегаты. Вдобавок вы увидели, как лямбда-выражения без тел операторов конвертируются в деревья выражений на основе типа `Expression<T>`, определенного в пространстве имен `System.Linq.Expression`. Вы можете применять трансформации к деревьям выражений перед их компиляцией в делегат и вызовом. Я завершил главу демонстрацией полезных применений лямбда-выражений. К ним относится создание обобщенных итераторов, мемоизация с использованием замыканий, привязка параметров делегатов с помощью "приправы", а также представление концепции анонимной рекурсии. Почти все эти концепции лежат в основе функционального программирования. Несмотря на то что все эти приемы можно было реализовать в C# на основе анонимных методов, добавление в язык лямбда-синтаксиса сделало их применение более естественным и менее сложным.

Следующая глава посвящена языку LINQ — кульминации всех новых средств C# 3.0. Также я продолжу уделять внимание связанным с ним аспектам функционального программирования.

ГЛАВА 16

LINQ: язык интегрированных запросов

Языки, подобные C (включая C#), императивны по природе — в том смысле, что упор делается на состояние системы и изменениях, которые она претерпевает со временем. Языки запросов данных, такие как SQL, по природе функциональны — в том смысле, что упор делается на операцию, и в этом процессе используется совсем немного неизменных данных (либо они вообще не используются). LINQ (Language Integrated Query — язык интегрированных запросов) заполняет пробел между императивным и функциональным стилями программирования. LINQ — обширная тема, которая достойна того, чтобы посвящать ей и тому, что можно делать с этим языком, целые книги[1]. Есть несколько доступных реализаций LINQ, среди них: LINQ to Objects, LINQ to SQL, LINQ to Dataset и LINQ to XML. Я сосредоточу внимание на LINQ to Objects, поскольку она позволяет получать сообщения LINQ напрямую, не включая каких-либо дополнительных слоев и технологий.

На заметку! Разработка LINQ началась некоторое время назад в Microsoft, и своим появлением эта технология обязана Андерсу Хейлсбергу (Anders Hejlsberg) и Питеру Голду (Peter Gold). Идея состояла в создании более естественного и интегрированного в язык способа доступа к данным из такого языка, как C#. Однако в то же время было нежелательно реализовывать его таким способом, который бы дестабилизировал реализацию компилятора C# и привносил путаницу в язык. Поэтому имело смысл реализовать некоторые строительные блоки на языке, чтобы обеспечить функциональность и выразительность LINQ. Отсюда и появились такие средства, как лямбда-выражения, анонимные типы, расширяющие методы и неявно типизированные переменные. Все они — не только великолепные средства сами по себе, но к тому же послужили основой для LINQ.

LINQ очень хорошо выполняет работу, позволяя программистам сосредоточиться на бизнес-логике и меньше тратить время на кодирование рутинных деталей, которые обычно ассоциируются с кодом доступа к данным. Если у вас есть опыт построения работающих с данными приложений, подумайте о том, насколько час-

---

[1] За более исчерпывающим описанием LINQ советую обратиться к книге Джозефа Ратца-мл. (Joseph C. Rattz, Jr.) *Pro LINQ: Language Integrated Query in C# 2008* (Berkeley, CA: Apress, 2007 г.).

то вам приходилось снова и снова писать один и тот же код с небольшими вариациями. LINQ избавляет вас от этого.

Мост к данным

На протяжении настоящей книги я подчеркиваю, что почти все новые средства, представленные в C# 3.0, стимулируют применение модели функционального программирования. На то имеется веская причина, потому что запросы данных — обычно функциональный процесс. Например, оператор SQL сообщает серверу в точности то, что вы хотите, и что нужно делать. Он не описывает объектов и структур, а также их взаимоотношений — динамических и статических, т.е. всего того, чем вам приходится заниматься, проектируя новое приложение на объектно-ориентированном языке. Поэтому функциональное программирование здесь играет ключевую роль, и любые приемы, знакомые вам по таким функциональным языкам программирования, как Lisp, Scheme или F#, здесь пригодятся.

Выражения запросов

На первый взгляд, выражения запросов LINQ выглядят очень похоже на SQL. Но не допускайте ошибки! LINQ — это не SQL. Прежде всего, LINQ строго типизирован. В конце концов, C# — строго типизированный язык, поэтому то же касается и LINQ. В язык LINQ были добавлены 8 новых ключевых слов для построения выражений запросов. Однако их реализация со стороны компилятора замечательно проста. Выражения запросов LINQ транслируются в цепочки вызовов расширяющих методов на последовательности или коллекции. Этот набор расширяющих методов четко определен и носит название *стандартных операций запросов*.

На заметку! Модель LINQ довольно расширяема. Если компилятор просто транслирует выражения запросов в серии вызовов расширяющих методов, из этого следует, что вы можете представить собственные реализации этих расширяющих методов. Фактически, так оно и есть. Например, класс `System.Linq.Enumerable` предоставляет реализации этих методов для LINQ to Objects, в то время как `System.Linq.Queryable` представляет реализации этих методов для LINQ to SQL.

Давайте перейдем прямо к делу и посмотрим, как выглядят запросы. В следующем примере создается коллекция объектов `Employee`, а затем выполняется простой запрос:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Decimal Salary { get; set; }
    public DateTime StartDate { get; set; }
}
```

```
public class SimpleQuery
{
    static void Main() {
        // Создадим базу данных сотрудников.
        var employees = new List<Employee> {
            new Employee {
                FirstName = "Joe",
                LastName = "Bob",
                Salary = 94000,
                StartDate = DateTime.Parse("1/4/1992") },
            new Employee {
                FirstName = "Jane",
                LastName = "Doe",
                Salary = 123000,
                StartDate = DateTime.Parse("4/12/1998") },
            new Employee {
                FirstName = "Milton",
                LastName = "Waddams",
                Salary = 1000000,
                StartDate = DateTime.Parse("12/3/1969") }
        };
        var query = from employee in employees
                    where employee.Salary > 100000
                    orderby employee.LastName, employee.FirstName
                    select new { LastName = employee.LastName,
                                 FirstName = employee.FirstName };
        Console.WriteLine( "Высокооплачиваемые сотрудники:" );
        foreach( var item in query ) {
            Console.WriteLine( "{0}, {1}",
                               item.LastName,
                               item.FirstName );
        }
    }
}
```

Прежде всего, в 99% случаев вам придется импортировать пространство имен `System.Linq`, что я покажу в следующем разделе, озаглавленном "Стандартные операции запросов". В этом примере я выделил выражение запроса полужирным, чтобы оно сразу бросалось в глаза. Если вы впервые видите выражение LINQ, оно покажется довольно-таки необычным. В конце концов, язык C# происходит от C++ и Java, а синтаксис LINQ совсем не похож на эти языки.

На заметку! Те из вас, кто знаком с SQL, вероятно, сразу отметят, что этот запрос напоминает вам нечто хорошо знакомое. В SQL конструкция `select` обычно стоит в начале выражения. Есть несколько причин к тому, что в C# имеет смысл обратное. Одна причина — облегчение работы Intellisence. В данном примере, если бы `select` стояла в начале, средству Intellisence пришлось бы сложнее определять свойства, имеющиеся у `employee`, поскольку тип `employee` был бы еще не известен.

До выражения запроса я создал простой список экземпляров `Employee`, чтобы были данные, с которыми можно работать. Каждое выражение запроса начинает-

ся с конструкции from, объявляющей то, что называется *переменной диапазона*. Конструкция from нашего примера очень похожа на оператор foreach в том, что осуществляет итерацию по коллекции employees, и на каждом шаге сохраняет каждый элемент коллекции в переменной employee. После конструкции from запрос состоит из последовательности конструкций, где можно использовать различные операции запроса для фильтрации данных, представленных переменной диапазона. В моем примере, как видите, я применил конструкции where и orderby. И, наконец, выражение закрывается операцией *проекции*. Когда вы выполняете проекцию в выражении запроса, то обычно создаете другую коллекцию информации, или одиночную порцию информации, представляющую трансформированную версию коллекции, по которой выполняется итерация посредством переменной диапазона. В предыдущем примере мне нужно было получить в результате только имя и фамилия сотрудников.

Конечный результат построения выражения запроса сосредоточен в том, что называется *переменной запроса*, в данном примере — query. В конце концов, можете ли вы представить тип запроса? Если интересно, можете отправить query.GetType на консоль и увидите следующий тип:

```
System.Linq.Enumerable+<SelectIterator>d__b`2[Employee,
<>f__AnonymousType0`2[System.String,System.String]]
```

Другая вещь, которую следует отметить — это использование анонимных типов в конструкции select. Я хотел, чтобы запрос создал трансформации исходных данных в коллекцию структур, где каждый экземпляр содержит свойство FirstName, и ничего более. Конечно, мне нужно иметь уже определенную структуру до выполнения запроса, когда конструкция select создает экземпляры этого типа, однако, такой подход в некоторой степени нивелирует удобство и выразительность запроса LINQ.

И что более важно, о чем я скажу чуть позже, в разделе "Добродетель лени" — выражение запроса не выполняется в точке присвоения переменной запроса. Вместо этого переменная запроса в данном примере реализует общую форму IEnumerable, и последующее применение foreach на переменной query производит результат этого примера.

Вернемся к расширяющим методам и лямбда-выражениям

Прежде чем приступить к более подробному разбору элементов выражения LINQ, я хотел бы показать альтернативный способ выполнения той же работы. Фактически, это более или менее то, что делает компилятор "за кулисами".

Синтаксис LINQ выглядит очень чуждым в преимущественно императивном языке, каковым является C#. Легко прийти к заключению, что язык C# претерпел существенные модификации для реализации LINQ. В действительности компилятор просто трансформирует выражение LINQ в последовательность вызовов расширяющих методов, принимающих лямбда-выражения.

Если вы заглянете в пространство имен System.Linq, то увидите, что там есть два интересных статических класса, переполненных расширяющими методами — Enumerable и Queryable. Enumerable определяет коллекцию обобщенных расширяющих методов, полезных для типов IEnumerable, тогда как Queryable определяет ту же коллекцию обобщенных расширяющих методов, применимых к типам IQueryable. Если вы взглянете на имена этих расширяющих методов, то

увидите, что они совпадают с конструкциями выражений запросов. Это не случайно, поскольку расширяющие методы реализуют стандартные операции запросов, которые упоминались в предыдущем разделе. Фактически выражение запроса из предыдущего примера может быть заменено следующим кодом:

```
var query = employees
     .Where( emp => emp.Salary > 100000 )
     .OrderBy( emp => emp.LastName )
     .OrderBy( emp => emp.FirstName )
     .Select( emp => new {LastName = emp.LastName,
                          FirstName = emp.FirstName} );
```

Обратите внимание, что это просто цепочка вызовов расширяющих методов на IEnumerable, который реализован employees. Фактически, вы можете пойти на шаг дальше и изменить оператор внутри, исключив синтаксис расширенных методов и просто вызывая их как статические методы, что показано ниже:

```
var query =
    Enumerable.Select(
     Enumerable.OrderBy(
      Enumerable.OrderBy(
       Enumerable.Where(
        employees, emp => emp.Salary > 100000),
       emp => emp.LastName ),
       emp => emp.FirstName ),
       emp => new {LastName = emp.LastName,
                   FirstName = emp.FirstName} );
```

Но зачем это может понадобиться? Я просто показал это в целях иллюстрации, чтобы вы знали, что в действительности происходит "за кулисами". Те из вас, кто действительно является приверженцем анонимных методов C# 2.0, может продвинуться еще дальше и заменить лямбда-выражения анонимными методами. Излишне говорить, что расширяющие методы Enumerable и Queryable очень полезны и вне контекста LINQ.

Стандартные операции запросов

LINQ построен на использовании стандартных операций запросов, которые представляют собой методы, предназначенные для операций с последовательностями, подобными коллекциям, которые реализуют интерфейсы IEnumerable и IQueryable. Как описывалось ранее, когда компилятор C# встречает выражение запроса, то преобразует в последовательность или цепочку запросов расширяющих методов, реализующих это поведение.

У такого подхода есть два преимущества. Одно состоит в том, что вы обычно можете выполнять те же действия, что и выражение запроса LINQ, явно вызывая расширяющие методы. Результирующий код не так легко читать, как код с выражениями запросов. Однако бывают случаи, когда нужна функциональность расширяющих методов, а полное выражение запроса может быть излишним.

Самое большое преимущество такого подхода связано с расширяемостью LINQ. То есть вы можете определить собственный набор расширяющих методов, а компилятор сгенерирует их вызовы при компиляции выражения запроса LINQ. Например,

предположим, что вы не импортировали пространство имен System.Linq и вместо этого решили предоставить собственную реализацию Where и Select. Это можно сделать так:

```
using System;
using System.Collections.Generic;
public static class MySqoSet
{
    public static IEnumerable<T> Where<T> (
            this IEnumerable<T> source,
            System.Func<T,bool> predicate ) {
        Console.WriteLine( "Вызвана моя собственная реализация Where." );
        return System.Linq.Enumerable.Where( source,
                                             predicate );
    }
    public static IEnumerable<R> Select<T,R> (
             this IEnumerable<T> source,
             System.Func<T,R> selector ) {
        Console.WriteLine( "Вызвана моя собственная реализация Select." );
        return System.Linq.Enumerable.Select( source,
                                              selector );
    }
}
public class CustomSqo
{
    static void Main() {
        int[] numbers = { 1, 2, 3, 4 };
        var query = from x in numbers
                    where x % 2 == 0
                    select x * 2;
        foreach( var item in query ) {
            Console.WriteLine( item );
        }
    }
}
```

Обратите внимание, что мне не понадобилось импортировать пространство имен System.Linq. Помимо дополнительного удобства это подтверждает мои слова о том, что отказ от импорта пространства имен System.Linq предотвращает автоматическое нахождение компилятором расширяющих методов System.Linq.Enumerable. В статическом классе MySqoSet я предоставил собственную реализацию стандартных операций запросов Where и Select, которые просто протоколируют сообщение и затем пересылают его операциям из Enumerable. Если вы запустите этот пример, то получите следующий вывод:

```
Вызвана моя собственная реализация Where.
Вызвана моя собственная реализация Select.
4
8
```

Вы можете немного усложнить это упражнение и представить, что хотите использовать LINQ с коллекцией, не поддерживающей IEnumerable. Хотя обычно вы будете в своих коллекциях реализовывать IEnumerable, для наглядности предположим, что она вместо этого поддерживает пользовательский интерфейс

IMyEnumarable. В данном случае вы можете обеспечить собственный набор операций запросов, работающих с IMyEnumarable вместо IEnumarable. Однако есть один минус. Если ваши типы не наследуются от IEnumarable, то вы не сможете использовать LINQ, потому что конструкция from требует источника данных, реализующего IEnumerable. Тем не менее, достичь того же эффекта можно, вызвав стандартные операции запросов на вашем типе IMyEnumarable. Я покажу пример этого в следующем разделе, озаглавленном “Приемы функционального программирования”, где воспользуюсь примером из главы 14.

Ключевые слова запросов C#

C# 2008 представил небольшой набор новых ключевых слов для создания выражений запросов LINQ: from, join, where, group, into, let, orderby и select. В следующих разделах я расскажу об их применении.

Конструкция from и переменные диапазона

Каждый запрос начинается со слова from. Конструкция from — это генератор, который также определяет переменную диапазона — локальную переменную, используемую для представления каждого элемента входной коллекции, по мере применения к нему выражения запроса. Конструкция from похожа на оператор foreach в стиле императивного программирования, а переменная диапазона идентична по своему предназначению переменной итерации в том же foreach.

Выражение запроса может содержать более одной конструкции from. В таком случае у вас имеется более одной переменной диапазона, что является аналогом вложённых операторов foreach. В следующем примере используется несколько конструкций from для генерации хорошо известной вам из начальной школы таблицы умножения, хотя и не в табличном формате, да и с некоторыми повторами:

```
using System;
using System.Linq;
public class MultTable
{
    static void Main() {
        var query = from x in Enumerable.Range(0,10)
                    from y in Enumerable.Range(0,10)
                    select new {
                        X = x,
                        Y = y,
                        Product = x * y
                    };
        foreach( var item in query ) {
            Console.WriteLine( "{0} * {1} = {2}",
                               item.X,
                               item.Y,
                               item.Product );
        }
    }
}
```

Напомню, что выражения LINQ компилируются в строго типизированный код. Так каковы же типы x и y в данном примере? Компилятор выводит типы этих двух переменных диапазона на основе типа аргумента интерфейса IEnumerable<T>, возвращенного Range. Поскольку Range возвращает тип IEnumerable<int>, типом x и y является int. Теперь вы можете недоумевать, что произойдет, если вы хотите применить выражение запроса к коллекции, поддерживающей не обобщенный интерфейс IEnumerable? В таких случаях вы должны явно специфицировать тип переменной диапазона, как показано ниже:

```
using System;
using System.Linq;
using System.Collections;
public class NonGenericLinq
{
    static void Main() {
        ArrayList numbers = new ArrayList();
        numbers.Add( 1 );
        numbers.Add( 2 );

        var query = from int n in numbers
                    select n * 2;
        foreach( var item in query ) {
            Console.WriteLine( item );
        }
    }
}
```

Здесь вы можете видеть, что я явно указываю тип переменной диапазона n — int. Во время выполнения осуществляется приведение, которое может закончиться неудачей с генерацией исключения InvalidCastException. Поэтому лучше стремиться к использованию обобщенного, строго типизированного интерфейса IEnumerable<T> вместо IEnumerable, так что ошибки подобного рода можно будет перехватить ко время компиляции, а не во время выполнения.

На заметку! Как я все время подчеркиваю на протяжении этой книги, компилятор — ваш лучший друг. Старайтесь использовать максимум его возможностей для перехвата ошибок кодирования во время компиляции, а не во время выполнения. Строго типизированные языки, такие как C#, полагаются на компилятор в проверке целостности операций, выполняемых над типами, которые определены в вашем коде. Если вы отбросите конкретный тип, и будете работать с общими типами вроде System.Object, а не конкретными типами объектов, то тем самым откажетесь от одной из самых мощных возможностей компилятора. Тогда, если в вашем коде появится ошибка, связанная с типами, и служба контроля качества не обнаружит ее до выпуска программы в свет, можно поручиться, что ваш заказчик даст вам знать об этом, возможно, даже не в самой вежливой форме!

Конструкция join

Вслед за конструкцией from вы можете использовать join для установки соотношения данных из двух различных источников. Операции объединения (join) обычно не обязательны в средах, где объекты связываются в иерархии и другие ассоциативные отношения. Однако в мире реляционных баз данных обычно не суще-

ствует жестких связей между элементами двух разных коллекций, или таблиц (помимо эквивалентности между элементами внутри каждой записи). Этак операция эквивалентности определяется, когда вы создаете конструкцию `join`. Рассмотрим следующий пример:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class EmployeeId
{
   public string Id { get; set; }
   public string Name { get; set; }
}
public class EmployeeNationality
{
   public string Id { get; set; }
   public string Nationality { get; set; }
}
public class JoinExample
{
   static void Main() {
      // Построить коллекцию сотрудников.
      var employees = new List<EmployeeId>() {
      new EmployeeId{ Id = "111-11-1111",
      Name = "Ed Glasser" },
      new EmployeeId{ Id = "222-22-2222",
      Name = "Spaulding Smails" },
      new EmployeeId{ Id = "333-33-3333",
      Name = "Ivan Ivanov" },
      new EmployeeId{ Id = "444-44-4444",
      Name = "Vasya Pupkin" }
      };
      // Построить коллекцию национальностей.
      var empNationalities = new List<EmployeeNationality>() {
         new EmployeeNationality{ Id = "111-11-1111",
                              Nationality = "American" },
         new EmployeeNationality{ Id = "333-33-3333",
                              Nationality = "Russian" },
         new EmployeeNationality{ Id = "222-22-2222",
                              Nationality = "Irish" },
         new EmployeeNationality{ Id = "444-44-4444",
                              Nationality = "Russian" }
      };
      // Построить запрос.
      var query = from emp in employees
               join n in empNationalities
                  on emp.Id equals n.Id
               orderby n.Nationality descending
               select new {
                  Id = emp.Id,
                  Name = emp.Name,
                  Nationality = n.Nationality
               };
```

```
        foreach( var person in query ) {
            Console.WriteLine( "{0}, {1}, \t{2}",
                               person.Id,
                               person.Name,
                               person.Nationality );
        }
    }
}
```

В примере присутствуют две коллекции. Первая содержит базу данных сотрудников с их идентификационными номерами. Вторая хранит базу данных национальностей сотрудников, где каждый сотрудник идентифицируется только его идентификатором. Чтобы не усложнять пример, каждая часть данных относится к типу string. Теперь я хочу получить список имен всех сотрудников вместе с их национальностями, при этом отсортировав список по национальностям в убывающем порядке. Конструкция join здесь нужна потому, что необходимая информация содержится в более чем одном источнике данных. Она позволяет объединить информацию из двух источников данных, и LINQ делает это в два счета! В выражении запроса я выделил полужирным конструкцию join. Для каждого элемента, на которые ссылается переменная диапазона emp, она находит элемент в коллекции empNationalities, где Id эквивалентно Id, на который ссылается emp. Затем конструкция проекции select берет данные из обеих коллекций при построении результата и проектирует эти данные на анонимный тип. Таким образом, результатом запроса является единственная коллекция, в которой каждый элемент из employee и empNationalities сплавляется в один. Если запустить этот пример, вы получите следующий результат:

```
333-33-3333, Ivan Ivanov, Russian
444-44-4444, Vasya Pupkin, Russian
222-22-2222, Spaulding Smails, Irish
111-11-1111, Ed Glasser, American
```

Когда ваш запрос включает операцию join, компилятор "за кулисами" преобразует ее в вызов расширяющего метода Join, если только за ней не следует конструкция into. Если присутствует into, то компилятор использует расширяющий метод GroupJoin, который также группирует результаты. Подробную информацию о более эзотерических вещах, которые вы можете делать с помощью конструкций join и into, ищите в документации MSDN по LINQ или же в книге Джозефа Ратца-мл. (Joseph C. Rattz, Jr.) *Pro LINQ: Language Integrated Query in C# 2008* (Berkeley, CA: Apress, 2007 г.).

На заметку! Нет причин, по которым нельзя было бы иметь несколько конструкций join внутри запроса, чтобы сразу связать данные из трех разных коллекций. В предыдущем примере вы могли бы иметь коллекцию, представляющую языки, на которых говорит каждая нация, и соединить каждый элемент из коллекции empNationalities с элементами в коллекции языков. Чтобы сделать это, достаточно было бы за одной конструкцией join установить другую.

Конструкция where и фильтры

За одной или более конструкциями from или join, если они есть, обычно размещается одна или более конструкций фильтра. Фильтры состоят из ключевого слова where, за которым следует выражение предиката. Конструкция where транслируется в вызов расширяющего метода Where как лямбда-выражение. Вызовы Enumerable.Where, которые используются, если вы выполняете запрос на типе IEnumerable, преобразуют лямбда-выражение в делегат. И наоборот, вызовы Queryable.Where, которые используются, если вы выполняете запрос к коллекции через интерфейс IQueryable, преобразуют лямбда-выражения в деревья выражений[2]. Дополнительные сведения о деревьях выражений в LINQ вы найдете в разделе "Еще раз о деревьях выражений" далее в главе.

Конструкция orderby

Конструкция orderby используется для сортировки последовательности — результата запроса. Вслед за ключевым словом orderby идет элемент, по значению которого вы хотите сортировать — обычно это общее свойство переменной диапазона. Вы можете сортировать как в возрастающем, так и в убывающем порядке, и если вы не специфицируете ни ключевого слова ascending, ни descending, по умолчанию принимается порядок по возрастанию (ascending). После конструкции orderby вы можете указывать неограниченный набор подсортировок, просто отделяя каждый элемент сортировки запятой, как показано ниже:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class Employee
{
   public string LastName { get; set; }
   public string FirstName { get; set; }
   public string Nationality { get; set; }
}
public class OrderByExample
{
   static void Main() {
      var employees = new List<Employee>() {
         new Employee {
            LastName = "Glasser", FirstName = "Ed",
            Nationality = "American"
         },
         new Employee {
            LastName = "Pupkin", FirstName = "Vasya",
            Nationality = "Russian"
         },
```

[2] В главе 15 я показал, как лямбда-выражения, присвоенные переменным — экземплярам делегата, преобразуются в исполняемый код IL, в то время как лямбда-выражения, присвоенные Expression<T>, конвертируются в деревья выражений, тем самым описывая выражения в данных вместо исполняемого кода.

```
            new Employee {
                LastName = "Smails", FirstName = "Spaulding",
                Nationality = "Irish"
            },
            new Employee {
                LastName = "Ivanov", FirstName = "Ivan",
                Nationality = "Russian"
            }
        };
        var query = from emp in employees
                    orderby emp.Nationality,
                            emp.LastName descending,
                            emp.FirstName descending
                    select emp;
        foreach( var item in query ) {
            Console.WriteLine( "{0},\t{1},\t{2}",
                               item.LastName,
                               item.FirstName,
                               item.Nationality );
        }
    }
}
```

Обратите внимание, что поскольку конструкция `select` просто возвращает переменную диапазона, это целое выражение запроса — не что иное, как операция сортировки. Несомненно, это — удобный способ сортировки сущностей в C#. В данном примере сначала осуществляется сортировка в порядке возрастания по `Nationality`, затем, во втором выражении в конструкции `orderby`, результат сортируется по убыванию в каждой группе национальности по `LastName` и, наконец, каждая из этих групп сортируется по `FirstName` в порядке убывания.

Во время компиляции компилятор транслирует первое выражение в конструкции `orderby` в вызов расширяющего метода `OrderBy` стандартной операции запроса. Все последующие вторичные выражения сортировки транслируются в цепочку вызовов расширяющих методов `ThenBy`.

Конструкция `select` и проекция

В запросе LINQ конструкция `select` используется для производства конечного результата запроса. Она называется *проектором*, потому что проектирует, или транслирует, данные внутри запроса в форму, удобную для применения. Если в выражении запроса присутствуют фильтрующие конструкции `where`, они должны предшествовать `select`. Компилятор преобразует конструкцию `select` в вызов расширяющего метода `Select`. Тело конструкции `select` конвертируется в лямбда-выражение, которое передается в метод `Select`, использующий его для производства каждого элемента результирующего набора.

Анонимные типы здесь чрезвычайно удобны, и вы будете правы, предположив, что средство анонимных типов произошло от операции `select` в процессе разработки LINQ. Чтобы увидеть, чем же удобны анонимные типы в данном случае, рассмотрим следующий пример:

```
using System;
using System.Linq;
public class Result
{
    public Result( int input, int output ) {
        Input = input;
        Output = output;
    }
    public int Input { get; set; }
    public int Output { get; set; }
}
public class Projector
{
    static void Main() {
        int[] numbers = { 1, 2, 3, 4 };
        var query = from x in numbers
                    select new Result( x, x*2 );
        foreach( var item in query ) {
            Console.WriteLine( "Input = {0}, Output = {1}",
                               item.Input,
                               item.Output );
        }
    }
}
```

Этот код работает. Однако посмотрите, как я объявил новый тип `Result`, чтобы сохранить результаты запроса. Что, если в будущем я захочу изменить результат, чтобы он включал `x`, `x*2` и `x*3`? Сначала мне придется модифицировать определение класса `Result`, чтобы адаптировать его к этим изменениям. Ох! Намного легче просто воспользоваться анонимными типами, как показано ниже:

```
using System;
using System.Linq;
public class Projector
{
    static void Main() {
        int[] numbers = { 1, 2, 3, 4 };
        var query = from x in numbers
                    select new {
                        Input = x,
                        Output = x*2 };

        foreach( var item in query ) {
            Console.WriteLine( "Input = {0}, Output = {1}",
                               item.Input,
                               item.Output );
        }
    }
}
```

Вот теперь намного лучше! Теперь я могу взять и добавить новое свойство к результирующему типу, назвав его `Output2`, например, и это не потребует никаких

изменений нигде, кроме анонимного типа, чей экземпляр создается внутри выражения запроса. Существующий код продолжит работать, и любой, кто захочет использовать новое свойство Output2, сможет это сделать.

Конечно, бывают случаи, когда вы хотите использовать предопределенные типы в конструкции select. Однако чем чаще вы станете использовать анонимные типы, тем большую гибкость вы обеспечите на будущее.

Конструкция let

Конструкция let представляет новый локальный идентификатор, на который можно впоследствии ссылаться в остальной части запроса. Думайте о ней как о локальной переменной, видимой только внутри выражения запроса, подобно тому, как обычная локальная переменная внутри нормального блока кода видима только внутри этого блока. Рассмотрим следующий пример:

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Employee
{
    public string LastName { get; set; }
    public string FirstName { get; set; }
}

public class OrderByExample
{
    static void Main() {
        var employees = new List<Employee>() {
            new Employee {
                LastName = "Glasser", FirstName = "Ed"
            },
            new Employee {
                LastName = "Pupkin", FirstName = "Vasya"
            },
            new Employee {
                LastName = "Smails", FirstName = "Spaulding"
            },
            new Employee {
                LastName = "Ivanov", FirstName = "Ivan"
            }
        };
        var query = from emp in employees
                    let fullName = emp.FirstName +
                                   " " + emp.LastName
                    orderby fullName
                    select fullName;
        foreach( var item in query ) {
            Console.WriteLine( item );
        }
    }
}
```

В данном примере я хотел отсортировать имена в порядке возрастания, но при этом сортируя по полному имени, сформированному соединением вместе FirstName и LastName. Я ввожу эту конструкцию посредством применения let для определения переменной fullName.

Другое замечательное качество локальных идентификаторов, создаваемых конструкцией let, заключается в том, что они ссылаются на коллекции, потому вы можете использовать переменную в качестве ввода для другой конструкции from, чтобы создать новую, производную переменную диапазона. В одном из предшествующих разделов, озаглавленном "Конструкция from и переменные диапазона", я привел пример использования нескольких конструкций from для генерации таблицы умножения. Ниже показана небольшая вариация этого примера с использованием конструкции let:

```
using System;
using System.Linq;
public class MultTable
{
   static void Main() {
      var query = from x in Enumerable.Range(0,10)
                  let innerRange = Enumerable.Range(0, 10)
                     from y in innerRange
                     select new {
                        X = x,
                        Y = y,
                        Product = x * y
                     };
      foreach( var item in query ) {
         Console.WriteLine( "{0} * {1} = {2}",
                            item.X,
                            item.Y,
                            item.Product );
      }
   }
}
```

Я выделил полужирным изменения, отличающие этот пример от прежнего. Обратите внимание, что я добавил новый промежуточный идентификатор по имени innerRange, и затем вслед за этим выполнил итерацию по коллекции с помощью конструкции from.

Конструкция group

Выражение запроса может иметь необязательную конструкцию group, которая является мощным средством для разделения ввода запроса. Конструкция group — это проектор, поскольку проектирует данные на коллекцию интерфейсов IGrouping. Интерфейс IGrouping определен в пространстве имен System.Linq и наследует IEnumerable. Поэтому вы можете применять интерфейс IGrouping везде, где можно использовать интерфейс IEnumerable. IGrouping включает свойство по имени Key, которое является объектом, описывающим подмножество. Каждый результирующий набор формируется применением операции эквивалентности ме-

жду Key и фрагментом входных данных либо данных, производных от входных. Давайте рассмотрим пример, принимающий последовательность целых числен и разбивающий их на множества четных и нечетных чисел[3].

```
using System;
using System.Linq;
public class GroupExample
{
    static void Main() {
        int[] numbers = {
            0, 1, 2, 3, 4, 5, 6, 7, 8, 9
        };
        // Разделение чисел на четные и нечетные.
        var query = from x in numbers
                    group x by x % 2;
        foreach( var group in query ) {
            Console.WriteLine( "mod2 == {0}", group.Key );
            foreach( var number in group ) {
                Console.Write( "{0}, ", number );
            }
            Console.WriteLine( "\n" );
        }
    }
}
```

Прежде всего, обратите внимание, что в запросе отсутствует конструкция select. Конечным результатом запроса является последовательность из двух экземпляров IGrouping. Первый экземпляр — результирующая последовательность, содержащая четные числа, а вторая содержит нечетные числа, что показывает следующий вывод:

```
mod2 == 0
0, 2, 4, 6, 8,
mod2 == 1
1, 3, 5, 7, 9,
```

Первый оператор foreach выполняет итерацию по двум группам или, точнее, по двум экземплярам IGrouping. И поскольку каждый IGrouping реализует IEnumerable, здесь присутствует вложенный цикл foreach, осуществляющий итерацию по всем элементам группы. Как видите, этот простой запрос проходит по всем элементам исходной коллекции данных numbers и производит две результирующих группы. Внутренне компилятор транслирует каждую конструкцию group в вызов стандартной операции запроса GroupBy.

Конструкция group может также разделять входную коллекцию, используя множественные ключи, также известные, как составные ключи. Я предпочитаю воспринимать это как сортировку по одному ключу, состоящему из множества порций данных. Чтобы выполнить такую группировку, вы можете воспользоваться анонимным типом для представления множественных ключей в запросе, как показано в следующем примере:

---

[3] Говоря о конструкции group, я использую слово *раздел* (partition) в контексте теории множеств. То есть набор разделов пространства S — это набор разделенных подмножеств, чье объединение производит S.

```
using System;
using System.Linq;
using System.Collections.Generic;
public class Employee
{
   public string LastName { get; set; }
   public string FirstName { get; set; }
   public string Nationality { get; set; }
}
public class OrderByExample
{
   static void Main() {
      var employees = new List<Employee>() {
         new Employee {
            LastName = "Jones", FirstName = "Ed",
            Nationality = "American"
         },
         new Employee {
            LastName = "Ivanov", FirstName = "Vasya",
            Nationality = "Russian"
         },
         new Employee {
            LastName = "Jones", FirstName = "Tom",
            Nationality = "Welsh"
         },
         new Employee {
            LastName = "Smails", FirstName = "Spaulding",
            Nationality = "Irish"
         },
         new Employee {
            LastName = "Ivanov", FirstName = "Ivan",
            Nationality = "Russian"
         }
      };

      var query = from emp in employees
                  group emp by new {
                     Nationality = emp.Nationality,
                     LastName = emp.LastName
                  };
      foreach( var group in query ) {
         Console.WriteLine( group.Key );
         foreach( var employee in group ) {
            Console.WriteLine( employee.FirstName );
         }
         Console.WriteLine();
      }
   }
}
```

Обратите внимание на анонимный тип внутри конструкции `group`. Он говорит о том, что я хочу разделить входную коллекцию на группы, в которых одинаковы комбинации `Nationality` и `LastName`. В данном примере каждая группа

в конечном итоге получает по одной сущности, за исключением одной — той, где Natinality имеет значение Russian, а LastName — Ivanov. По сути, это работает следующим образом: строится экземпляр анонимного типа и проверяется эквивалентность экземпляра ключа ключу существующей группы. Если эквивалентность имеет место, элемент идет в эту группу. Если нет, создается новая группа с экземпляром анонимного типа в качестве ключа.

Группировка полезна сама по себе. Однако что вы хотите делать далее с каждой из групп внутри запроса, трактуя результирующее разбиение на группы в качестве промежуточного шага? И вот здесь вы используете ключевое слово into, описанное в следующем разделе.

Конструкция into и продолжение

Ключевое слово into подобно ключевому слову let в том, что оно определяет локальный по отношению к контексту запроса идентификатор. Используя конструкцию into, вы сообщаете запросу, что хотите присвоить результат операции group или join идентификатору, который может быть использован в запросе позднее. На языке запросов это называется *продолжением* (continuation), поскольку конструкция group — не финальный проектор запроса. Однако конструкция into работает как генератор, во многом подобно конструкциям from, и идентификатор, представленный into, подобен переменной диапазона в конструкции from. Рассмотрим некоторые примеры:

```
using System;
using System.Linq;
public class GroupExample
{
    static void Main() {
        int[] numbers = {
            0, 1, 2, 3, 4, 5, 6, 7, 8, 9
        };
        // Разделение чисел на четные и нечетные.
        var query = from x in numbers
                    group x by x % 2 into partition
                    where partition.Key == 0
                    select new {
                        Key = partition.Key,
                        Count = partition.Count(),
                        Group = partition
                    };
        foreach( var item in query ) {
            Console.WriteLine( "mod2 == {0}", item.Key );
            Console.WriteLine( "Count == {0}", item.Count );
            foreach( var number in item.Group ) {
                Console.Write( "{0}, ", number );
            }
            Console.WriteLine( "\n" );
        }
    }
}
```

В этом запросе продолжение, т.е. часть запроса, следующая после конструкции into, фильтрует серии групп, где Key равно 0, используя конструкцию where. Это фильтрует группу четных чисел. Затем я проектирую эту группу на анонимный тип, производя счетчик элементов в группе, сопровождаемый значением свойства Key и элементами группы. Таким образом, вывод на консоль включает лишь одну группу.

Но что, если я захочу добавить счетчик в каждую группу раздела? Как я уже говорил, конструкция into — это генератор. Поэтому я могут получить желаемый результат, изменив запрос следующим образом:

```
var query = from x in numbers
            group x by x % 2 into partition
            select new {
               Key = partition.Key,
               Count = partition.Count(),
               Group = partition
            };
```

При выполнении этой версии запроса получается следующий желаемый вывод:

```
mod2 == 0
Count == 5
0, 2, 4, 6, 8,
mod2 == 1
Count == 5
1, 3, 5, 7, 9,
```

Добродетель лени

Когда вы строите выражение запроса LINQ и присваиваете его переменной запроса, в этом операторе выполняется очень мало кода. Данные становятся доступными только во время итерации по этой переменной запроса, которая выполняет запрос по одному разу для каждого элемента в результирующем наборе. Поэтому, например, если результирующий набор состоит из 100 элементов, и вы можете выполнять итерацию только по десяти из них, вам не приходится платить за вычисление остальных 90 элементов результирующего набора.

На заметку! Вы можете использовать расширяющий метод Take, чтобы сократить количество элементов в результирующем наборе; однако он все равно производит перечислитель, использующий прием отложенного выполнения. Столь же полезны методы TakeWhile, Skip и SkipWhile.

Выгоды этого подхода на основе отложенного выполнения многочисленны. Прежде всего, операции, описанные в выражении запроса, могут оказаться достаточно дорогими. Поскольку эти операции предоставляются пользователем, и проектировщики LINQ не имеют никакой возможности прогнозировать сложность этих операций, лучше получать каждый элемент лишь по мере необходимости. К тому же данные могут быть расположены в базе данных в другом полушарии. В этом случае вы определенно предпочтете "ленивое вычисление" на своей сторо-

не. И, наконец, переменная диапазона может в принципе выполнять итерацию по бесконечной последовательности. В следующем разделе я приведу пример этого.

Поощрение лени итераторами C#

Внутренне переменная запроса реализуется посредством итераторов C# с использованием ключевого слова yield. В главе 9 я объяснял, что код, содержащий операторы yield, в действительности компилируется в объект итератора. Поэтому, когда вы присваиваете выражение LINQ переменной запроса, единственный код, выполняющийся при этом — это код конструктора объекта итератора. Итератор может зависеть от других вложенных объектов, и они также инициализируются. Вы получаете результат выражения LINQ, как только запускаете итерацию по переменной запроса с помощью оператора foreach либо с использованием интерфейса IEnumerable.

В качестве примера давайте взглянем на слегка модифицированный код примера из предшествующего раздела "Выражения запросов". Для удобства приведем только самый значимый код:

```
var query = from employee in employees
         where employee.Salary > 100000
         select new { LastName = employee.LastName,
                      FirstName = employee.FirstName };
Console.WriteLine( "Высокооплачиваемые сотрудники:" );
foreach( var item in query ) {
   Console.WriteLine( "{0}, {1}",
                      item.LastName,
                      item.FirstName );
```

Обратите внимание, что единственное отличие заключается в том, что я удалил оператор orderby из оригинального выражения LINQ; зачем — объясню в следующем разделе. Напомню, что запрос транслируется в цепочку связанных вызовов расширяющих методов на переменной employees. Каждый из этих методов возвращает объект, реализующий IEnumerable<T>. В действительности эти объекты — итераторы, созданные оператором yield.

Давайте рассмотрим, что происходит, когда вы начинаете итерацию по результатам в блоке foreach. Чтобы получить следующий результат, сначала конструкция from извлекает следующий элемент из коллекции employees и заставляет переменную диапазона employee ссылаться на него. Затем, "за кулисами", конструкция where передает следующий элемент, на который ссылается переменная диапазона, расширяющему методу Where. Если он отклоняется фильтром, выполнение передается обратно конструкции from для получения следующего элемента в коллекции. Выполнение продолжается в цикле до тех пор, пока employees полностью не опустошается, или элемент employees проходит предикат конструкции where. Затем конструкция select проектирует элемент на нужный формат, создавая и возвращая экземпляр анонимного типа. Как только элемент возвращен из конструкции select, ее работа завершена до тех пор, пока курсор переменной запроса не будет перемещен на следующем шаге итерации.

На заметку! Выражения запросов LINQ могут использоваться повторно. Например, предположим, что вы начали итерацию по результатам выражения запроса. Теперь представьте, что переменная диапазона проходит итерацию только по нескольким элементам входной коллекции, и переменная, ссылающаяся на коллекцию, изменяется, указывая на другую коллекцию. Вы можете продолжать выполнять итерацию по тому же запросу и получать изменения в новой входной коллекции без необходимости заново переопределять запрос. Как такое возможно? Подсказка: подумайте о замыканиях и захвате переменных, а также о том, что происходит, если захваченная переменная модифицируется вне контекста замыкания.

Ниспровержение лени

В предыдущем разделе я удалил конструкцию `orderby` из выражения запроса, и вы могли удивиться, почему. Дело в том, что некоторые операции запросов разрушают "ленивое вычисление". В конце концов, может ли `orderby` выполнить свою работу, не имея полных результатов предшествующих конструкций? Конечно, нет, и потому `orderby` заставляет предшествующие ему конструкции выполнить всю итерацию до конца.

На заметку! `orderby` — не единственная конструкция, которая отменяет "ленивое", или отложенное, выполнение выражений запроса. `group...by` и `join` делают то же самое. Вдобавок всякий раз, когда выполняется вызов расширяющего метода на переменной запроса, порождающей одиночное значение, такого как `Count`, вы вынуждаете весь запрос пройти всю итерацию до конца.

Оригинальное выражение запроса, использованное в разделе "Выражения запросов", выглядело так:

```
var query = from employee in employees
            where employee.Salary > 100000
            orderby employee.LastName, employee.FirstName
            select new { LastName = employee.LastName,
                         FirstName = employee.FirstName };
Console.WriteLine( "Высокооплачиваемые сотрудники:" );
foreach( var item in query ) {
    Console.WriteLine( "{0}, {1}",
                       item.LastName,
                       item.FirstName );
}
```

Я выделил полужирным конструкцию `orderby`, чтобы обратить ваше внимание. Когда вы запрашиваете следующий элемент в результирующем наборе, конструкция `from` посылает следующий элемент `employees` в фильтр `where`. Если элемент проходит фильтр, то далее отправляется конструкции `orderby`. Однако теперь `orderby` нужно видеть остальную часть входной информации, прошедшей фильтр, поэтому она возвращает выполнение обратно конструкции `from`, чтобы получить следующий элемент, прошедший фильтрацию. Этот цикл продолжается до тех пор, пока в коллекции `employees` не остается элементов. Затем, после упорядочивания элементов на основе указанного критерия, первый элемент упорядоченного набора отправляется проектору `select`. Когда `foreach` запрашивает следующий элемент результирующего набора, вычисление начинается с конструкции `orderby`, посколь-

ку все результаты предыдущих конструкций уже кэшированы. Берется следующий элемент из внутреннего кэша и передается проектору select. Это продолжается до тех пор, пока потребитель переменной диапазона не переберет все результаты, опустошая кэш, сформированный orderby.

Ранее я упоминал случай, когда переменная диапазона в выражении выполняет итерацию по бесконечному циклу. Рассмотрим следующий пример:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class InfiniteList
{
    static IEnumerable<int> AllIntegers() {
        int count = 0;
        while( true ) {
            yield return count++;
        }
    }
    static void Main() {
        var query = from number in AllIntegers()
                    select number * 2 + 1;
        foreach( var item in query.Take(10) ) {
            Console.WriteLine( item );
        }
    }
}
```

Обратите внимание на выделенное полужирным выражение запроса. Оно осуществляет вызов AllIntegers, который просто является итератором по всем целым числам, начиная с нуля. Конструкция select проектирует эти целые числа на все нечетные. Затем с использованием Take и цикла foreach отображаются первые десять нечетных чисел. Отметьте, что если бы я не использовал Take, программа работала бы бесконечно, если только вы не компилировали ее с включенной опцией /checked+ для обнаружения переполнений.

На заметку! Методы, создающие итераторы для бесконечных множеств, подобные AllIntegers из предыдущего примера, иногда называют потоками (streams). Классы Queryable и Enumerable также содержат полезные методы, генерирующие конечные коллекции. К ним относится Empty, возвращающий пустое множество элементов, Range, возвращающий последовательность чисел, и Repeat, который генерирует повторяющийся поток константных объектов на основе переданного ему объекта и количества, которые он должен вернуть. Думаю, Repeat должен выполнять бесконечную итерацию, если ему передать отрицательное число.

Рассмотрим, что случится, если слегка модифицировать выражение запроса, как показано ниже:

```
var query = from number in AllIntegers()
            orderby number descending
            select number * 2 + 1;
```

Если вы попытаетесь выполнить итерацию по этой переменной запроса хотя бы один раз, чтобы получить первый результат, вам нужно приготовиться к тому,

чтобы прервать эту программу. Дело в том, что конструкция orderby потребует полного прохода по всем элементам, прежде чем приступит к сортировке. В данном случае этот проход никогда не кончится.

Даже если ваша переменная диапазона не выполняет итерацию по бесконечному набору, конструкции, предшествующие orderby, могут быть очень дороги в выполнении. Поэтому мораль этой истории в том, что следует быть осторожными, применяя в выражениях запросов orderby, group...by и join, поскольку они могут отрицательно повлиять на производительность.

Немедленное выполнение запросов

Иногда возникает необходимость выполнить весь запрос незамедлительно. Возможно, вы хотите локально кэшировать результаты запроса в памяти или минимизировать длительность блокировки базы данных SQL. Это можно сделать двумя способами. Можно немедленно после запроса запустить цикл foreach, который выполнит итерацию по переменной запроса, помещая результат в List<T>. Но это слишком императивно! Не лучше ли сделать это функционально? Вы можете вызвать расширяющий метод ToList на переменной запроса, что даст тот же результат за один-единственный вызов метода. Как и с примером orderby из предыдущего раздела, будьте осторожны при вызове ToList на запросе, возвращающем бесконечный результирующий набор. Существует также расширяющий метод ToArray, предназначенный для преобразования результатов в массив. Ниже, в разделе "Замена операторов foreach", я приведу интересный пример применения ToArray.

Наряду с ToList, существуют и другие расширяющие методы, известные под общим названием агрегатных операций, которые форсируют немедленное выполнение всего запроса. К ним относятся такие методы, как Count, Sum, Max, Min, Average, Last, Reverse, а также прочие методы, которые должны выполнять весь запрос для производства результата.

Еще раз о деревьях выражений

В главе 15 я описал, как лямбда-выражения могут быть преобразованы в деревья выражений. Также я кратко упомянул о том, что это очень полезно для LINQ to SQL.

Когда вы используете LINQ to SQL, тела конструкций LINQ, который сводятся к лямбда-выражениям, представляются деревьями выражений. Эти деревья выражений затем используются для преобразования всего выражения в оператор SQL для запуска на сервере. Когда вы выполняете LINQ to Objects, как это делаю я на протяжении всей настоящей главы, то вместо этого лямбда-выражения преобразуются в делегаты в форме кода IL. Ясно, что это неприемлемо для LINQ to SQL. Можете вы представить, как трудно было бы конвертировать IL в SQL?

Как вы знаете, конструкции LINQ превращаются в вызовы расширяющих методов, реализованных либо в System.Linq.Enumerable, либо в System.Linq.Queryable. Но когда какой набор методов расширения используется? Если вы заглянете в документацию по методам Enumerable, то увидите там, что предикаты конвертируются в делегаты, поскольку все эти методы принимают параметр типа, основанного на обобщенном типе делегата Func<>. Однако расширяющие методы Queryable,

имеющие те же имена, что и у Enumerable, все преобразуют лямбда-выражения в деревья выражений, поскольку принимают параметр типа Expression<T>. Ясно, что LINQ to SQL использует расширяющие методы из Queryable.

На заметку! Кстати, когда вы используете расширяющие методы Enumerable, вы можете передавать им как лямбда-выражения, так и анонимные функции, поскольку они принимают делегат в списке своих параметров. Однако расширяющие методы Queryable могут принимать только лямбда-выражения, поскольку анонимные функции не могут быть преобразованы в деревья выражений.

Приемы функционального программирования

В последующих разделах я хотел бы поговорить немного подробнее о концепциях функционального программирования, которые получили распространение вместе с новыми средствами C# 3.0. Как вы вскоре убедитесь, некоторые проблемы решаются при разумном использовании делегатов, созданных из лямбда-выражений, которые добавляют общеизвестный дополнительный уровень посредничества. Также я покажу, как вы можете заменить многие конструкции императивного стиля программирования, такие как циклы for и foreach, на конструкции в функциональном стиле.

Пользовательские стандартные операции запросов и "ленивое вычисление"

В этом разделе я вернусь к примеру, представленному в главе 14, где было показано, как реализовать однонаправленный связный список в стиле Lisp наряду с некоторыми расширяющими методами, позволяющими работать с этим списком. Первичный интерфейс этого списка выглядит так:

```
public interface IList<T>
{
    T Head { get; }
    IList<T> Tail { get; }
}
```

В главе 14 также была продемонстрирована возможная реализация коллекции на основе этого типа. Повторю ее здесь для удобства:

```
public class MyList<T> : IList<T>
{
    public static IList<T> CreateList( IEnumerable<T> items ) {
        IEnumerator<T> iter = items.GetEnumerator();
        return CreateList( iter );
    }
    public static IList<T> CreateList( IEnumerator<T> iter ) {
        if( !iter.MoveNext() ) {
            return new MyList<T>( default(T), null );
        }
        return new MyList<T>( iter.Current, CreateList(iter) );
    }
```

```
    private MyList( T head, IList<T> tail ) {
        this.head = head;
        this.tail = tail;
    }
    public T Head {
        get {
            return head;
        }
    }
    public IList<T> Tail {
        get {
            return tail;
        }
    }
    private T head;
    private IList<T> tail;
}
```

Обратите внимание, как я представил два метода `CreateList`, чтобы можно было создавать экземпляр `MyList` на основе типа `IEnumerable` или `IEnumerator`. Теперь предположим, что мы хотим реализовать стандартные операции запросов `Where` и `Select`. На основе реализации `MyList` эти операции могут быть закодированы так:

```
public static class MyListExtensions
{
public static IEnumerable<T>
    GeneralIterator<T>( this IList<T> theList,
                        Func<IList<T>, bool> finalState,
                        Func<IList<T>, IList<T>> incrementer ) {
        while( !finalState(theList) ) {
            yield return theList.Head;
            theList = incrementer( theList );
        }
    }
    public static IList<T> Where<T>( this IList<T> theList,
                                     Func<T, bool> predicate ) {
        Func<IList<T>, IList<T>> whereFunc = null;

        whereFunc = list => {
            IList<T> result = new MyList<T>(default(T), null);
            if( list.Tail != null ) {
                if( predicate(list.Head) ) {
                    result = new MyList<T>( list.Head, whereFunc(list.Tail) );
                } else {
                    result = whereFunc( list.Tail );
                }
            }
            return result;
        };
        return whereFunc( theList );
    }
```

```
    public static IList<R> Select<T,R>( this IList<T> theList,
                                        Func<T,R> selector ) {
        Func<IList<T>, IList<R>> selectorFunc = null;
        selectorFunc = list => {
            IList<R> result = new MyList<R>(default(R), null);

            if( list.Tail != null ) {
                result = new MyList<R>( selector(list.Head),
                                        selectorFunc(list.Tail) );
            }

            return result;
        };
        return selectorFunc( theList );
    }
}
```

Каждый из этих методов — Where и Select — использует встроенное лямбда-выражение, конвертируемое в делегат для выполнения работы.

На заметку! В главе 14 был показан простой прием, но поскольку лямбда-выражения там еще не были представлены, использовались анонимные методы. Конечно, лямбда-выражения существенно проясняют синтаксис.

В обоих методах встроенное лямбда-выражение используется для выполнения простого рекурсивного вычисления, чтобы получить желаемый результат. Конечный результат рекурсии производит продукт, который нам нужно получить от каждого из этих методов.

Метод GeneralIterator в предыдущем примере служит для создания итератора, реализующего IEnumerable на экземплярах объекта MyList. Это практически то же самое, что было показано в примере главы 14.

И, наконец, вы можете собрать все это вместе и выполнить показанный ниже код:

```
public class SqoExample
{
    static void Main() {
        var listInts = new List<int> { 5, 2, 9, 4, 3, 1 };
        var linkList =
            MyList<int>.CreateList( listInts );

        // Теперь - сортировка.
        var linkList2 = linkList.Where( x => x > 3 ).Select( x => x * 2 );
        var iterator2 = linkList2.GeneralIterator( list => list.Tail == null,
        list => list.Tail );

        foreach( var item in iterator2 ) {
            Console.Write( "{0}, ", item );
        }
        Console.WriteLine();
    }
}
```

Конечно, вам придется импортировать соответствующие пространства имен, чтобы код компилировался. Эти пространства имен следующие: `System`, `System.Linq` и `System.Collections.Generic`. Если запустить этот код на выполнение, вы получите следующий результат:

```
10, 18, 8
```

Однако в этом примере присутствуют некоторые очень важные моменты и проблемы, на которые нужно обратить внимание. Прежде всего, обратите внимание, что мой запрос не был написан с использованием выражения запроса LINQ, даже несмотря на то, что я применяю стандартные операции запросов `Where` и `Select`. Поскольку интерфейс `IList` не реализует `IEnumerable`, здесь невозможно использовать `foreach` или `from`. Поэтому приходится применять расширяющий метод `GeneralIterator`, чтобы получить интерфейс `IEnumerable` на `IList`, а затем использовать его в конструкции `from` выражения запроса LINQ. В этом случае не понадобилось бы реализовывать пользовательские методы `Where` и `Select`. Однако результаты вашего запроса будут в форме `IEnumerable`, а не `IList`, поэтому вам придется обратно преобразовать этот результат запроса в `IList`. Хотя эти преобразования возможны, давайте для примера предположим, что было выдвинуто требование, чтобы стандартные операции запросов принимали тип `IList` и возвращали тип `IList`. В соответствии с этим требованием, невозможно использовать выражения запросов LINQ, и мы должны вызывать стандартные операции запросов непосредственно.

На заметку! Вы можете видеть всю мощь многослойного дизайна и реализации LINQ. Даже когда ваша пользовательская коллекция не реализует `IEnumerable`, вы все равно можете выполнять операции, применяя пользовательские стандартные операции запросов, даже несмотря на то, что невозможно применить выражения запроса LINQ.

Одна главная проблема связана с реализацией `MyList` и расширяющих методов в классе `MyListExtensions`, как он был показан до сих пор. Они чудовищно неэффективны! Одним из приемов функционального программирования, повсеместно применяемым в реализации LINQ, является "ленивое вычисление". В разделе, озаглавленном "Добродетель лени", я показал, как в точке создания выражения запроса LINQ выполняется очень мало кода, а все операции выполняются только по мере необходимости, когда вы осуществляете итерацию по результатам запроса. Показанные реализации `Where` и `Select` для `IList` не следуют этой методике. Например, когда вызывается `Where`, полный входной список обрабатывается перед тем, как потребителю будет возвращен какой-либо результат. Это плохо, потому что вдруг `IList` окажется бесконечным списком? Тогда вызов `Where` никогда не вернет управления.

На заметку! При разработке реализаций стандартных операций запросов или любых других методов, где желательно "ленивое вычисление", я предпочитаю использовать бесконечный список для ввода, поскольку это отличный тест правильности работы моего "ленивого" кода. Конечно, как показано в разделе "Ниспровержение лени", есть определенные операции, которые просто невозможно закодировать в методике "ленивого выполнения".

Теперь давайте определим наше внимание на новую реализацию пользовательских стандартных операций запроса для предыдущего примера, на этот раз используя "ленивое вычисление". Начнем с рассмотрения операции Where. Как можно иначе реализовать ее для "ленивого вычисления"? Она принимает IList и возвращает новый IList, поэтому как сделать так, чтобы Where возвращала только по одному элементу за раз? Решение состоит в реализации класса MyList. Рассмотрим типичную реализацию IEnumerator. Она содержит внутренний курсор, указывающий на элемент, который возвращает свойство IEnumerable.Current, а также имеет метод MoveNext для перехода к следующему элементу. Метод IEnumerable.MoveNext — ключ к извлечению каждого значения по мере необходимости. Когда вы вызываете MoveNext, то тем самым вызываете операцию, производящую следующий результат, но только по мере необходимости, таким образом реализуя "ленивое вычисление".

Я упоминал "Фундаментальную теорему программной инженерии" Эндрю Кенига (Andrew Koenig's "Fundamental Theorem of Software Engineering"), которая гласит, что все проблемы могут быть решены вводом дополнительного слоя посредничества[4]. Хотя это на самом деле вовсе и не теорема, такое утверждение истинно и очень полезно. В языке C такая форма посредничества обычно принимает вид указателя. В C++ и других объектно-ориентированных языках дополнительный слой посредничества обычно принимает форму класса, иногда называемого классом-оболочкой. В функциональном программировании такой дополнительный уровень посредничества — обычно функция в форме делегата.

Как же мы можем решить эту проблему в MyList добавлением пресловутого дополнительного промежуточного слоя? На самом деле это в основе достаточно просто. Не вычисляйте IList, который является IList.Tail, пока он не будет запрошен. Рассмотрим следующие изменения в реализации MyList:

```
public class MyList<T> : IList<T>
{
    public static IList<T> CreateList( IEnumerable<T> items ) {
        IEnumerator<T> iter = items.GetEnumerator();
        return CreateList( iter );
    }
    public static IList<T> CreateList( IEnumerator<T> iter ) {
        Func<IList<T>> tailGenerator = null;
        tailGenerator = () => {
            if( !iter.MoveNext() ) {
                return new MyList<T>( default(T), null );
            }
            return new MyList<T>( iter.Current, tailGenerator );
        };
        return tailGenerator();
    }
    public MyList( T head, Func<IList<T>> tailGenerator ) {
        this.head = head;
        this.tailGenerator = tailGenerator;
    }
```

[4] Впервые я столкнулся с так называемой фундаментальной теоремой программной инженерии Эндрю Кенига в его блестящей книге, написанной в соавторстве с Барбари Му (Barbary Moo) *Ruminations on C++* (Boston, MA: Addison-Wesley Professional, 1996 г.).

```
    public T Head {
        get {
            return head;
        }
    }

    public IList<T> Tail {
        get {
            if( tailGenerator == null ) {
                return null;
            } else if( tail == null ) {
                tail = tailGenerator();
            }
            return tail;
        }
    }
    private T head;
    private Func<IList<T>> tailGenerator;
    private IList<T> tail = null;
}
```

В этом коде я выделил полужирным наиболее интересные части. Обратите внимание, что конструктор по-прежнему принимает элемент, присваиваемый head, но вместо получения хвоста IList в качестве второго аргумента он принимает делегат, который знает, как вычислить tail. Вот это и есть наш слой посредничества! К тому же посмотрите, как метод доступа get свойства Tail затем использует этот делегат по мере необходимости для вычисления tail, когда поступит соответствующий запрос. И, наконец, статический метод CreateList, который строит IList из IEnumerator, должен запускать делегат, который просто извлекает следующий элемент IEnumerator. Поэтому, даже если вы инициализируете MyList с IEnumerable, то тип IEnumerable полностью не потребляется во время создания, как то было в примере из главы 14. Это — определенный плюс, потому что переданный IEnumerable может ссылаться на бесконечный поток объектов.

Теперь давайте обратим внимание на модификации, необходимые для стандартных операций запросов, чтобы они могли работать на этой новой реализации MyList:

```
public static class MyListExtensions
{
    public static IEnumerable<T>
        GeneralIterator<T>( this IList<T> theList,
                            Func<IList<T>,bool> finalState,
                            Func<IList<T>,IList<T>> incrementer ) {
        while( !finalState(theList) ) {
            yield return theList.Head;
            theList = incrementer( theList );
        }
    }
    public static IList<T> Where<T>( this IList<T> theList,
                                     Func<T, bool> predicate ) {
```

```
        Func<IList<T>> whereTailFunc = null;
        whereTailFunc = () => {
            IList<T> result = null;
            if( theList.Tail == null ) {
                result = new MyList<T>( default(T), null );
            }

            if( predicate(theList.Head) ) {
                result = new MyList<T>( theList.Head,
                                        whereTailFunc );
            }
            theList = theList.Tail;
            if( result == null ) {
                result = whereTailFunc();
            }

            return result;
        };
        return whereTailFunc();
    }

    public static IList<R> Select<T,R>( this IList<T> theList,
                                        Func<T,R> selector ) {
        Func<IList<R>> selectorTailFunc = null;
        selectorTailFunc = () => {
            IList<R> result = null;
            if( theList.Tail == null ) {
                result = new MyList<R>( default(R), null );
            } else {
                result = new MyList<R>( selector(theList.Head),
                                        selectorTailFunc );
            }
            theList = theList.Tail;
            return result;
        };
        return selectorTailFunc();
    }
}
```

Теперь реализации `Where` и `Select` строят делегат, который знает, как вычислять следующий элемент в результирующем наборе и передать этот делегат новому экземпляру `MyList`, который они возвращают. Таким образом, мы достигли "ленивого вычисления". Обратите внимание, что каждое лямбда-выражение в каждом методе формирует замыкание, использующее переданную информацию для формирования рекурсивного кода, генерирующего следующий элемент списка. Теперь давайте протестируем наше "ленивое вычисление", введя бесконечный связный список значений.

Прежде чем вы проверите "ленивое вычисление" на бесконечном списке, вам нужно либо выполнять итерацию по результатам применения цикла `for` (поскольку цикл `foreach` пытается выполнить итерацию до несуществующего конца), либо вместо использования цикла `for` реализовать стандартную операцию запроса

Take, которая возвращает заданное количество элементов из списка. Ниже показана одна из возможных реализаций Take, использующая нашу новую "ленивую" реализацию MyList.

```
public static class MyListExtensions
{
    public static IList<T> Take<T>( this IList<T> theList,
                                    int count ) {
        Func<IList<T>> takeTailFunc = null;
        takeTailFunc = () => {
            IList<T> result = null;
            if( theList.Tail == null || count-- == 0 ) {
                result = new MyList<T>( default(T), null );
            } else {
                result = new MyList<T>( theList.Head,
                                        takeTailFunc );
            }
            theList = theList.Tail;
            return result;
        };
        return takeTailFunc();
    }
}
```

Эта реализация Take очень похожа на Select, за исключением того, что замыкание, формируемое лямбда-выражением, присвоенным takeTailFunc, также захватывает параметр count.

На заметку! Применение Take представляет собой более функциональный подход программирования, чем использование цикла for для подсчета первых нескольких элементов коллекции.

Вооружившись методом Take, мы можем проверить работу "ленивого вычисления" в следующем коде:

```
public class SqoExample
{
    static IList<T> CreateInfiniteList<T>( T item ) {
        Func<IList<T>> tailGenerator = null;

        tailGenerator = () => {
            return new MyList<T>( item, tailGenerator );
        };
        return tailGenerator();
    }
    static void Main() {
        var infiniteList = CreateInfiniteList<int>( 21 );
        var linkList = infiniteList.Where( x => x > 3 )
                                   .Select( x => x * 2 )
                                   .Take( 10 );
        var iterator = linkList.GeneralIterator(
                                    list => list.Tail == null,
                                    list => list.Tail );
```

```
        foreach( var item in iterator ) {
            Console.Write( "{0}, ", item );
        }
        Console.WriteLine();
    }
}
```

Метод Main использует метод CreateInfiniteList для создания бесконечного потока IList, возвращающего константу 21. Вслед за созданием infiniteList идет цепочка вызовов наших пользовательских стандартных операций запроса. Обратите внимание, что финальный метод в цепочке — Take, в котором я только запрашиваю первые 10 элементов результирующего набора. Без этого вызова следующий цикл foreach работал бы бесконечно. Поскольку метод Main в действительности доходит до своего завершения, это доказывает, что "ленивое вычисление", закодированное в новом MyList, и новые реализации Where, Select и Take работают, как ожидалось. Если бы с ними было что-то не так, выполнение бы зависло в бесконечном цикле.

Замена операторов foreach

Как и большинство новых средств C# 3.0, LINQ придает языку вкус функционального программирования, который, будучи правильно примененным, может оставить сладкое послевкусие. Поскольку функциональное программирование с годами заслужило репутацию менее эффективного в отношении потребления памяти и ресурсов процессора, может быть, что неправильное применение LINQ действительно может привести к снижению эффективности. Как почти со всеми средствами, используемыми при разработке программного обеспечения, умеренность — ключ к успеху. Применяя их взвешенно, с умом, вы будете поражены, сколько проблем могут быть решены другим, часто более ясным способом, с использованием LINQ и функционального программирования, по сравнению с тем, чего можно добиться привычными средствами императивного программирования, характерными для C-образных языков, таких как C#, C++ и Java.

Во многих примерах настоящей книги я посылаю список элементов на консоль, чтобы проиллюстрировать результат работы примера. Обычно я использовал вызов метода Console.WriteLine внутри оператора foreach для итерации по результатам, когда таким результатом является коллекция. Теперь я хотел бы показать вам, как это можно сделать иначе средствами LINQ:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class Extensions
{
    public static string Join( this string str,
    IEnumerable<string> list ) {
        return string.Join( str, list.ToArray() );
    }
}
```

```
public class Test
{
    static void Main() {
        var numbers = new int[] { 5, 8, 3, 4 };

        Console.WriteLine(
            string.Join(", ",
                        (from x in numbers
                        orderby x
                        select x.ToString()).ToArray()) );
    }
}
```

Интересную часть кода я выделил полужирным. В одном операторе я посылаю все элементы коллекции numbers на консоль, разделяя их запятыми и сортируя в порядке возрастания. Разве не здорово? Это работает так, что мое выражение запроса вычисляется немедленно, поскольку я вызываю расширяющий метод ToArray на нем, чтобы преобразовать результаты запроса в массив. Поэтому здесь исчезает типичная конструкция foreach. Статический метод String.Join не следует путать с LINQ-конструкцией join или расширяющим методом Join, который вы получаете при использовании пространства имен System.Linq. Этот метод берет первый параметр — строку (в данном случае содержащую запятую), посредством которой затем соединяет элементы массива строк, выстраивая одну длинную строку, в которой содержатся все элементы массива, разделенные запятыми. Затем результат, полученный от String.Join, просто отправляется в Console.WriteLine.

На заметку! По моему мнению, LINQ для C# стал тем, чем является стандартная библиотека шаблонов STL для C++. Когда STL появилась впервые в начале 90-х годов, она действительно заставила программистов C++ мыслить более функционально. Это определенно было подобно глотку свежего воздуха. LINQ создал тот же эффект для C#, и я верю, что со временем вы увидите все более и более изощренные применения приемов функционального программирования на основе LINQ. Например, если программист C++ эффективно использует STL, то у него редко возникает потребность в цикле for, поскольку STL предоставляет алгоритмы, позволяющие передать функцию в алгоритм вместе с коллекцией, над которой нужно провести операцию, и применяющие эту функцию к каждому элементу коллекции. Эта техника удивительно эффективна. Одна из причин состоит в том, что циклы for — это место, где часто допускаются непреднамеренные ошибки диапазона, разумеется, ключевое слово C# foreach также помогает избежать этой проблемы.

Хорошенько подумав, вы, вероятно, сможете заменить почти любой блок foreach в программе выражением запроса LINQ.

Резюме

LINQ представляет собой вершину большинства новых средств C# 3.0. Или, если посмотреть иначе, большинство новых средств C# 3.0 произрастают из LINQ. В этой главе я показал базовый синтаксис запросов LINQ, включая то, как выражения запросов LINQ в конечном итоге компилируются в цепочку вызовов расширяющих методов, известных как стандартные операции запроса. Затем я описал

все новые ключевые слова C#, предназначенные для построения выражений LINQ. Хотя вы не обязаны использовать выражения запросов LINQ, и можете отдать предпочтение непосредственному вызову расширяющих методов, они определенно повышают читабельность кода. Однако я также рассказал, как при реализации стандартных операций запросов на типах коллекций, которые не реализуют `IEnumerable`, вы можете не иметь возможности применения выражений запросов LINQ.

Затем я раскрыл полезность "ленивого вычисления", или отложенного исполнения, которое интенсивно используется библиотекой, предоставляющей стандартные операции LINQ в типах `IEnumerable` и `IQueryable`. И, наконец, я завершил главу объяснением того, как применять концепцию "ленивого вычисления" при определении ваших собственных пользовательских реализаций стандартных операций запросов.

LINQ — настолько обширная тема, что невозможно охватить все нюансы в одной главе. Например, вы заметили, что я рассказал только о LINQ to Objects, но не затронул LINQ to SQL, LINQ to XML и LINQ to DataSet. Языку LINQ посвящены целые книги. Я настоятельно рекомендую почаще обращаться к документации MSDN по LINQ. Дополнительно стоит ознакомиться с книгами Клаудио Феррачатти (Claudio Ferracchiati) *LINQ for Visual C# 2005* и Джозефа Ратца-мл. (Joseph C. Rattz, Jr.) *Pro LINQ: Language Integrated Query in C# 2008* (обе книги выпущены издательством Apress).

ПРИЛОЖЕНИЕ

Справочная информация

Ниже приведен перечень использованных в книге источников, которые рекомендуются для дополнительного изучения.

1. Abrams, Brad. .NET Framework Standard Library Annotated Reference,Volumes 1 and 2. Boston, MA: Addison-Wesley Professional, 2004, 2005.
2. Alexandrescu, Andrei. Modern C++ Design: Generic Programming and Design Patterns Applied. Boston, MA: Addison-Wesley Professional, 2001.
3. Archer, Tom, and Andrew Whitechapel. Inside C#, Second Edition. Redmond, WA: Microsoft Press, 2002.
4. Box, Don. Essential COM. Boston, MA: Addison-Wesley Professional, 1997.
5. Box, Don, with Chris Sells. Essential .NET,Volume 1: The Common Language Runtime. Boston, MA: Addison-Wesley Professional, 2003.
6. Box, Don with Anders Hejlsberg. "The LINQ Project." `http://msdn2.microsoft.com/en-us/library/aa479865.aspx`, September 2005.
7. Brown, Keith. The .NET Developer's Guide to Windows Security. Boston, MA: Addison-Wesley Professional, 2004.
8. Coplien, James O. Advanced C++ Programming Styles and Idioms. Boston, MA: Addison-Wesley Professional, 1991.
9. Cwalina, Krzysztof, and Brad Abrams. Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries. Boston, MA: Addison-Wesley Professional, 2005.
10. Ecma International. Standard ECMA-334: C# Language Specification, Third Edition. June 2005.
11. Ecma International. Standard ECMA-335: Common Language Infrastructure (CLI), Third Edition. June 2005.
12. Ecma International. Standard ECMA-372: C++/CLI Language Specification. December 2005.
13. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley Professional, 1995.

14. Hejlsberg, Anders, Scott Wiltamuth, and Peter Golde. The C# Programming Language. Boston, MA: Addison-Wesley Professional, 2003.

15. Horton, Anson. "The Evolution of LINQ and It's Impact On the Design of C#," MSDN Magazine, June 2007.

16. Kaplan, Michael, and Cathy Wissink. "Custom Cultures: Extend Your Code's Global Reach with New Features in the .NET Framework 2.0," MSDN Magazine,October 2005.

17. LaMacchia, Brian A., Sebastian Lange, Matthew Lyons, Rudi Martin, and Kevin T. Price. .NET Framework Security. Upper Saddle River, NJ: Pearson Education, 2002.

18. Meyers, Scott. Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs. Boston, MA: Addison-Wesley Professional, 1997.

19. Meyers, Scott. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Boston, MA: Addison-Wesley Professional, 1995.

20. Microsoft Corporation. "C# Language Specification Version 3.0." September 2007.

21. Miller, Jim, and Susann Ragsdale. The Common Language Infrastructure Annotated Standard. Boston, MA: Addison-Wesley Professional, 2003.

22. Nathan, Adam. .NET and COM: The Complete Interoperability Guide. Indianapolis, IN: Sams, 2002.

23. Richter, Jeffrey. Applied Microsoft .NET Framework Programming. Redmond, WA: Microsoft Press, 2002.

24. Robbins, John. "Unhandled Exceptions and Tracing in the .NET Framework 2.0," MSDN Magazine, July 2005.

25. Russinovich, Mark E., and David A. Solomon. Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003,Windows XP, and Windows 2000. Redmond, WA: Microsoft Press, 2004.

26. Schmidt, Douglas C. "Monitor Object: An Object Behavioral Pattern for Concurrent Programming," Department of Computer Science and Engineering, Washington University, St. Louis, MO, April 2005.

27. Stroustrup, Bjarne. The Design and Evolution of C++. Boston, MA: Addison-Wesley Professional, 1994.

28. Sutter, Herb. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Exception-Safety Solutions. Boston, MA: Addison-Wesley Professional, 1999.

29. Sutter, Herb. Exception C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions. Boston, MA: Addison-Wesley Professional, 2004.

30. Sutter, Herb. More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions. Boston, MA: Addison-Wesley Professional, 2001.

31. Toub, Stephen. "High Availability: Keep Your Code Running with the Reliability Features of the .NET Framework," MSDN Magazine, October 2005.

32. Troelsen, Andrew. Pro C# 2005 and the .NET 2.0 Platform. Berkeley, CA: Apress, 2005.

33. Vermeulen, Allan. "An Asynchronous Design Pattern," Dr.Dobb's Journal, June 1996.

Блоги

http://blogs.msdn.com/brada/

http://blogs.msdn.com/cbrumme/

http://blogs.msdn.com/kcwalina/

http://blogs.msdn.com/maoni/

http://blogs.msdn.com/ricom/

http://pluralsight.com/blogs/dbox/

http://pluralsight.com/blogs/hsutter/

http://www.sellsbrothers.com/news/

http://blogs.msdn.com/wesdyer/

http://blogs.msdn.com/charlie/default.aspx

Предметный указатель

A

Active Template Library (ATL), 182; 347
aspect-oriented programming (AOP), 46
aspect-oriented software development (AOSD), 46

B

Base Class Library (BCL), 185

C

Code access security (CAS), 293
Common Language Infrastructure (CLI), 23
Common Language Runtime (CLR), 22; 31
Common Language Specification (CLS), 43
Common Type System (CTS), 43
Constrained Execution Region (CER), 218

D

Document Object Model (DOM), 29

F

Free Threaded Marshaller (FTM), 386

G

Garbage Collector (GC), 24; 64
Global Assembly Cache (GAC), 35

H

I

Interface Description Language (IDL), 37; 162
Intermediate Language Disassembler (ILDASM), 32

J

Just In Time (JIT), 22

L

Language Integrated Query (LINQ), 28

M

Microsoft Transaction Server (MTS), 46
Multi-threaded apartment (MTA), 385

N

.NET 1.1, 201
.NET 2.0, 201
.NET Framework, 239

P

Portable Executable (PE), 23

R

Resource Acquisition Is Initialization (RAII), 139; 226

S

Service-oriented architecture (SOA), 160
Single-threaded apartment (STA), 385
Standard Template Library (STL), 27

T

Thread-local storage (TLS), 381

V

Virtual Execution System (VES), 23; 31

W

Web Services Description Language (WSDL), 162
Windows Management Instrumentation (WMI), 29

А

Аргумент
  ref, 145
  -значение, 145
Атрибут, 66

Б

Библиотека
  ATL, 182; 347
  BCL, 185
  .NET, 237
  TLB, 37

В

Выражение, 40
  лямбда, 509; 536

Г

Герметизация (closure), *См.* Захват переменных, 27
Глобализация, 237

Д

Делегат, 300
  использование, 301
  несвязанный, 306

обобщенный, 338
одиночный, 302
привязка параметров, 321
создание, 301
экземпляр делегата, 301
Делегирование, 153

З

Запрос, 537
Захват переменной, 523

И

Индексатор, 97
Инициализатор, 67
выражение инициализатора, 133
объекта, 77; 115
Инкапсуляция, 77
Интерфейс, 83; 111; 160; 325
ICloneable, 429
ICollection, 275
ICollection<T>, 277
IComparable, 127; 268
IConvertable, 462
ICustomFormatter, 245
IDisposable, 140; 434
IEnumerable, 275; 537
IList, 278
IList<T>, 278
IQueryable, 537
IUIControl, 160; 169
наследование интерфейсов, 163
обобщенный, 336
определение, 161
правила сопоставления членов интерфейсов, 172
реализация, 165
неявная, 165
явная, 166; 176
Исключение, 199
генерация, 200
повторная, 204
необработанное, 201
неопределенного поведения, 139
обработка, 140; 200
сгенерированное в блоке finally, 207
сгенерированное в статическом конструкторе, 209
сгенерированное в финализаторе, 207
создание пользовательских классов исключений, 224
трансляция, 204
Итератор, 289
блок итератора, 282
двунаправленный, 294
обратный, 294
пользовательский, 497
прямой, 294

К

Класс
Array, 264
Convert, 172; 461
Exception, 224
HashSet, 279
Interlocked, 387
Monitor, 393
Object, 125
Semaphore, 407
StringBuilder, 251
ThreadPool, 412
Timer, 421
WaitHandle, 409
абстрактный, 93
базовый, 64
вложенный, 94
герметизированный, 92
листовой, 92
определение, 63; 65
производный, 64
статический, 102
частичный, 100
Ключевое слово
abstract, 149
base, 70; 91; 110; 135
catch, 202
class, 63; 100
finally, 143; 202; 207; 230
foreach, 552; 564
from, 539
get, 74
group, 547
into, 550
join, 540
let, 546
new, 127; 151
orderby, 543
out, 144; 147
params, 148
partial, 100; 101
ref, 145
sealed, 92; 152
select, 544
set, 74
struct, 65
this, 70; 98; 108; 135
throw, 200
try, 143; 202; 211; 230
using, 143; 230
var, 113

virtual, 149; 150
where, 543
yield, 290; 552
Код
нейтральный к исключениям, 211
управляемый, 22
Коллекция, 275
синхронизация коллекций, 277
Компилятор
csc.exe, 26
Компиляция
JIT, 22
единица компиляции, 27
Конструктор, 70; 106
статический, 70; 130
экземпляра, 70; 133
Контракт
интерфейсный, 181
программный, 179
реализованный классом, 179
Кэш сборки
глобальный, 35

Л

Литерал
строковый, 236
Лямбда
-выражение, 27; 509; 536
-оператор, 515
-функция, 27

М

Манифест сборки, 33
Массив, 264
params, 148
ковариантность, 268
конвертируемость, 268
многомерный, 271
зубчатый, 273
прямоугольный, 271
неявно типизированный, 265
поиск, 268
синхронизация, 269
сортировка, 268
Мемоизация, 523
Метаданные, 36
Метод, 70
Copy, 235
Dispose, 142
Finalize, 137
Paint, 160
абстрактный, 149
анонимный, 314
асинхронный вызов, 413
виртуальный, 149
обобщенный, 336
перегрузка методов, 148
расширяющий, 482; 491; 536
статический, 70; 71
частичный, 101
экземпляра, 70; 71
Многопоточность, 369
Модификатор
abstract, 103; 149
internal, 82
new, 150; 153
out, 148
override, 150
params, 148
private, 82
protected, 82
protected internal, 82
public, 82
ref, 148
sealed, 82; 103; 152
static, 103
virtual, 149
доступа, 82
параметра, 148
поля, 66

Н

Набор, 279
Наследование, 64; 85; 93; 153; 347

О

Обобщения (generics), 54; 328; 347
Объект, 64
Mutex, 406
блокирующий, 401
одноразовый (disposable), 140
уничтожение, 137
детерминированное, 139
Ограничения (constraints), 348
на неклассовых типах, 353
обобщений, 350
Оператор
в C#, 41
лямбда, 515
Операция
!, 190
%, 190
&, 190
*, 187
+, 187
++, 187
-, 187
--, 187
/, 187
<, 190

<<, 190
<=, 190
>, 190
>=, 190
>>, 190
^, 190
|, 190
~, 190
as, 52
false, 190
is, 52
true, 190
бинарная, 187; 190
булевская, 195
запроса, 537
перегрузка операций, 186
сложения, 189
преобразования, 193
приоритет, 40
сравнения, 187; 191
унарная, 187; 190
Определения классов, 63

П

Перегрузка операций, 186
сложения, 189
Переменная
запроса, 536
ссылочного типа, 64
Перечисления, 45
Поле, 66
модификатор, 66
internal, 66
new, 66
private, 66
protected, 66
public, 66
readonly, 66; 67
static, 66
volatile, 66; 69
экземпляра, 66
Полиморфизм, 64; 86
Поток
завершение, 376
запуск, 370
локальное хранилище, 381
переднего плана, 380
синхронизация между потоками, 386
состояния потока, 373
управляемый, 369
фоновый, 380
Преобразование
неявное, 86
типов, 50
упаковывающее, 52
Приоритеты операций, 40
Программа
ILDASM, 32
Проектор, 544
Пространство имен, 56
определение, 57
Пул
внутренний, 236

Р

Распаковка, 120
Рекурсия
анонимная, 530
Рефлексия, 37

С

Сборка (assembly), 33
глобальный кэш сборок, 35
загрузка, 35
локальная, 35
манифест сборки, 33
приватная, 35
строгое (полное) именование, 35
частичное именование, 35
Сборка мусора, 24; 137
Сборщик мусора, 64
Свойство, 72
автореализуемое, 75
объявление, 73
только для записи, 74
только для чтения, 74
Семафор, 407
Сервер транзакций Microsoft, 46
Событие, 309; 408
AutoResetEvent, 408
ManualResetEvent, 408
синхронизации, 409
Специализация, 86
Спецификатор формата, 237
Список, 278
Строка, 235
группирование строк, 255
замена, 259
поиск с помощью регулярных выражений, 253
работа со строками, 235
из внешних источников, 249
сравнение строк, 247
строковый литерал, 236
форматная, 241; 243

Т

Таймер, 421
Тип, 42
встроенный, 42

значений, 43; 65; 105; 171; 176
преобразования типов, 50
упаковывающее, 52
ссылочный, 43; 46
Трансформация, 491

У

Упаковка, 122
Управляемый код, 22

Ф

Финализатор, 111; 137; 207; 437
критичный, 218
Формат
PE (Portable Executable), 23
Форматная строка, 241; 243
Фрейм стека, 199
Функционал, 509
Функция
анонимная, 300
лямбда, 27

Ш

Шаблон
NVI, 425
Strategy, 325
Visitor, 504

Э

Экземпляр
String, 235
делегата, 301

Я

Язык
C#, 22
встроенные типы, 42
выражения, 41
операторы, 41
перечисления, 45
синтаксис, 39
C++, 23
CLS, 43
IDL, 37; 162
IL (Intermediate Language), 22
LINQ, 28; 533
WSDL, 162

Научно-популярное издание

Трей Нэш

C# 2008: ускоренный курс для профессионалов

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 25.01.2008. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 46,44. Уч.-изд. л. 33,21.
Тираж 3000 экз. Заказ № 6661

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Автор книги
Accelerated C# 2005

Один из соавторов
Accelerated VB 2005
Accelerated VB 2008

Трей Нэш в настоящее время разрабатывает программное обеспечение в одной из лидирующих на рынке программных компаний, занимающихся вопросами безопасности. До этого он в течение пяти лет работал на Macromedia Inc., специализируясь на технологии COM/DCOM с использованием C/C++/ATL, до тех пор, пока не грянула революция .NET.

НА WEB-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с Web-сайта издательства по адресу: http://www.williamspublishing.com.

www.williamspublishing.com

Apress®

www.apress.com

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

C# 2008 УСКОРЕННЫЙ КУРС ДЛЯ ПРОФЕССИОНАЛОВ

Уважаемый читатель!
Вы держите в руках руководство по созданию эффективного кода на C#. Эта книга максимально сфокусирована на языке C# 3.0. В ней показано, как пишутся программы, которые характеризуются надежностью, устойчивостью к ошибкам и готовностью быть помещенными в широко доступные библиотеки.
Я не буду отнимать ваше драгоценное время на бесконечное обсуждение библиотек. Взамен я предлагаю хорошо организованный и легко читаемый текст, посвященный C# 3.0 и хорошо испытанным и правильным идиомам, шаблонам и принципам проектирования, которые появлялись в течение всего времени существования .NET Framework. На многочисленных коротких примерах я продемонстрирую регулярное использование общих шаблонов проектирования в .NET Framework и покажу, как применять их в собственных разработках.
В этой книге детально рассматриваются все новые средства C# 3.0, включая расширяющие методы, лямбда-выражения, язык LINQ и многие другие. Все эти замечательные нововведения поощряют использование модели функционального программирования в рамках того, что ранее было в основном императивным языком программирования. После того как вы начнете применять их, вы очень быстро ощутите, насколько расширились ваши возможности в построении решений.
Еще одной областью интересов разработчиков на C# является написание безопасного к исключениям и устойчивого к ошибкам кода. Среда .NET Framework поддерживает множество возможностей, включая ограниченные области выполнения, которые помогают защищать состояние приложения в случае асинхронного исключения. Все эти возможности также рассматриваются в книге.
Успешного вам программирования, и помните, что определение контракта перед реализацией, стремление к нейтральному в отношении исключений коду и экономное использование ресурсов — это ключи к владениям гуру в C#.
Трей Нэш

Категория: программирование/язык программирования C#

Предмет рассмотрения: C# 3.0

Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-8459-1377-7