C#

ДЛЯ ПРОФЕССИОНАЛОВ

ТОНКОСТИ ПРОГРАММИРОВАНИЯ

Третье издание

Новый перевод

Джон Скит

Предисловие **Эрика Липперта**

Содержание

Предисловие 19

Благодарности 20

Об этой книге 21

Кто должен читать эту книгу? 21
Дорожная карта 22
Терминология, оформление и загружаемый код 24
От издательства 24

Об авторе 25

Часть I. Подготовка к путешествию 27

Глава 1. Изменение стиля разработки в C# 28

1.1. Простой тип данных 29
1.1.1. Тип `Product` в C# 1 29
1.1.2. Строго типизированные коллекции в C# 2 31
1.1.3. Автоматически реализуемые свойства в C# 3 32
1.1.4. Именованные аргументы в C# 4 32
1.2. Сортировка и фильтрация 34
1.2.1. Сортировка товаров по названию 34
1.2.2. Запрашивание коллекций 37
1.3. Обработка отсутствия данных 39
1.3.1. Представление неизвестной цены 40
1.3.2. Необязательные параметры и стандартные значения 41
1.4. Введение в LINQ 42
1.4.1. Выражения запросов и внутренние запросы 42
1.4.2. Запрашивание файла XML 43
1.4.3. LINQ to SQL 44
1.5. COM и динамическая типизация 45
1.5.1. Упрощение взаимодействия с COM 45
1.5.2. Взаимодействие с динамическим языком 46
1.6. Более простое написание асинхронного кода 47
1.7. Разделение платформы .NET 49
1.7.1. Язык C# 49
1.7.2. Исполняющая среда 50
1.7.3. Библиотеки инфраструктуры 50
1.8. Как сделать код фантастическим 51
1.8.1. Представление полных программ в виде набора фрагментов 51
1.8.2. Учебный код не является производственным 52
1.8.3. Спецификация языка как лучший друг 53
1.9. Резюме 53

Глава 2. Язык C# как основа всех основ 54

2.1. Делегаты 55
2.1.1. Рецепт для простых делегатов 55
2.1.2. Объединение и удаление делегатов 60
2.1.3. Краткое введение в события 61
2.1.4. Резюме по делегатам 62

2.2. Характеристики системы типов 63
2.2.1. Место C# в мире систем типов 63
2.2.2. Когда возможности системы типов C# 1 оказываются недостаточными? 67
2.2.3. Резюме по характеристикам системы типов 70
2.3. Типы значений и ссылочные типы 70
2.3.1. Значения и ссылки в реальном мире 70
2.3.2. Основные положения типов значений и ссылочных типов 71
2.3.3. Развенчание мифов 73
2.3.4. Упаковка и распаковка 75
2.3.5. Резюме по типам значений и ссылочным типам 76
2.4. За рамками C# 1: новые возможности на прочной основе 76
2.4.1. Средства, связанные с делегатами 77
2.4.2. Средства, связанные с системой типов 79
2.4.3. Средства, связанные с типами значений 81
2.5. Резюме 82

Часть II. C# 2: решение проблем, присущих C# 1 83

Глава 3. Параметризованная типизация с использованием обобщений 85

3.1. Необходимость в обобщениях 86
3.2. Простые обобщения для повседневного использования 88
3.2.1. Обучение на примерах: обобщенный словарь 88
3.2.2. Обобщенные типы и параметры типов 90
3.2.3. Обобщенные методы и чтение обобщенных объявлений 93
3.3. Дополнительные сведения 97
3.3.1. Ограничения типов 97
3.3.2. Выведение типов для аргументов типов в обобщенных методах 103
3.3.3. Реализация обобщений 104
3.4. Дополнительные темы, связанные с обобщениями 110
3.4.1. Статические поля и статические конструкторы 111
3.4.2. Обработка обобщений JIT-компилятором 112
3.4.3. Обобщенная итерация 114
3.4.4. Рефлексия и обобщения 117
3.5. Недостатки обобщений в C# и сравнение с другими языками 121
3.5.1. Отсутствие обобщенной вариантности 122
3.5.2. Отсутствие ограничений операций или "числового" ограничения 127
3.5.3. Отсутствие обобщенных свойств, индексаторов и других членов типа 128
3.5.4. Сравнение с шаблонами C++ 129
3.5.5. Сравнение с обобщениями Java 130
3.6. Резюме 132

Глава 4. Типы, допускающие значения `null` 133

4.1. Что делать, когда значение просто отсутствует? 134
4.1.1. Почему переменные типов значений не могут быть установлены в `null` 134
4.1.2. Шаблоны для представления значений `null` в C# 1 135
4.2. Типы `System.Nullable<T>` и `System.Nullable` 137
4.2.1. Введение в `Nullable<T>` 137
4.2.2. Упаковка и распаковка типа `Nullable<T>` 140
4.2.3. Равенство экземпляров типа `Nullable<T>` 142
4.2.4. Поддержка необобщенного класса `Nullable` 142

4.3. Синтаксический сахар C# 2 для работы с типами, допускающими null 143
4.3.1. Модификатор ? 144
4.3.2. Присваивание и сравнение с null 145
4.3.3. Преобразования и операции над типами, допускающими null 147
4.3.4. Булевская логика, допускающая значение null 150
4.3.5. Использование операции as с типами, допускающими null 152
4.3.6. Операция объединения с null 153
4.4. Новаторское использование типов, допускающих null 156
4.4.1. Проба выполнения операции без использования выходных параметров 156
4.4.2. Безболезненные сравнения с использованием операции объединения с null 158
4.5. Резюме 161

Глава 5. Оперативно о делегатах 162

5.1. Прощание с неуклюжим синтаксисом для делегатов 163
5.2. Преобразования групп методов 165
5.3. Ковариантность и контравариантность 166
5.3.1. Контравариантность для параметров делегата 167
5.3.2. Ковариантность возвращаемых типов делегатов 168
5.3.3. Небольшой риск несовместимости 169
5.4. Встраивание действий делегатов с помощью анонимных методов 170
5.4.1. Начинаем с простого: действие над одним параметром 171
5.4.2. Возвращение значений из анонимных методов 173
5.4.3. Игнорирование параметров делегата 175
5.5. Захватывание переменных в анонимных методах 177
5.5.1. Определение замыканий и различных типов переменных 177
5.5.2. Исследование поведения захваченных переменных 178
5.5.3. Смысл захваченных переменных 180
5.5.4. Продленное время жизни захваченных переменных 180
5.5.5. Создание экземпляров локальных переменных 182
5.5.6. Смесь разделяемых и отдельных переменных 184
5.5.1. Руководящие принципы и резюме по захваченным переменным 186
5.6. Резюме 187

Глава 6. Простой способ реализации итераторов 188

6.1. C# 1: сложность написанных вручную итераторов 189
6.2. C# 2: простые итераторы с операторами yield 192
6.2.1. Появление итераторных блоков и оператора yield return 192
6.2.2. Визуализация рабочего потока итератора 194
6.2.3. Расширенный поток выполнения итератора 196
6.2.4. Индивидуальные особенности реализации 200
6.3. Реальные примеры использования итераторов 201
6.3.1. Итерация по датам в расписании 201
6.3.2. Итерация по строкам в файле 203
6.3.3. Ленивая фильтрация элементов с использованием итераторного блока и предиката 206
6.4. Написание псевдосинхронного кода с помощью библиотеки Concurrency and Coordination Runtime 208
6.5. Резюме 210

Глава 7. Заключительные штрихи C# 2: финальные возможности 212
7.1. Частичные типы 213
7.1.1. Создание типа с помощью нескольких файлов 214
7.1.2. Использование частичных типов 216
7.1.3. Частичные методы (только C# 3) 218
7.2. Статические классы 220
7.3. Отдельные модификаторы доступа для средств получения/установки свойств 222
7.4. Псевдонимы пространств имен 223
7.4.1. Уточнение псевдонимов пространств имен 224
7.4.2. Псевдоним глобального пространства имен 225
7.4.3. Внешние псевдонимы 226
7.5. Директивы `pragma` 228
7.5.1. Директивы `#pragma warning` 228
7.5.2. Директивы `#pragma checksum` 229
7.6. Буферы фиксированного размера в небезопасном коде 230
7.7. Открытие внутренних членов для избранных сборок 232
7.7.1. Дружественные сборки в простом случае 232
7.7.2. Причины использования атрибута `InternalsVisibleTo` 233
7.7.3. Атрибут `InternalsVisibleTo` и подписанные сборки 234
7.8. Резюме 235

Часть III. C# 3: революционные изменения в доступе к данным 237

Глава 8. Отбрасывание мелочей с помощью интеллектуального компилятора 239
8.1. Автоматически реализуемые свойства 240
8.2. Неявная типизация локальных переменных 243
8.2.1. Использование ключевого слова `var` для объявления локальной переменной 243
8.2.2. Ограничения неявной типизации 245
8.2.3. Доводы за и против неявной типизации 246
8.2.4. Рекомендации 248
8.3. Упрощенная инициализация 249
8.3.1. Определение нескольких демонстрационных типов 249
8.3.2. Установка простых свойств 250
8.3.3. Установка свойств встроенных объектов 251
8.3.4. Инициализаторы коллекций 252
8.3.5. Использование средств инициализации 255
8.4. Неявно типизированные массивы 256
8.5. Анонимные типы 258
8.5.1. Знакомство с анонимными типами 258
8.5.2. Члены анонимного типа 260
8.5.3. Инициализаторы проекций 261
8.5.4. В чем смысл существования анонимных типов? 262
8.6. Резюме 264

Глава 9. Лямбда-выражения и деревья выражений 265
9.1. Лямбда-выражения как делегаты 267
9.1.1. Подготовительные работы: знакомство с типами делегатов `Func<...>` 267
9.1.2. Первая трансформация в лямбда-выражение 268
9.1.3. Использование одиночного выражения в качестве тела 269
9.1.4. Списки неявно типизированных параметров 269
9.1.5. Сокращение для единственного параметра 270

9.2. Простые примеры использования типа List<T> и событий 272
9.2.1. Фильтрация, сортировка и действия на списках 272
9.2.2. Регистрация внутри обработчика событий 273
9.3. Деревья выражений 275
9.3.1. Построение деревьев выражений программным образом 275
9.3.2. Компиляция деревьев выражений в делегаты 276
9.3.3. Преобразование лямбда-выражений C# в деревья выражений 278
9.3.4. Деревья выражений являются основой LINQ 281
9.3.5. Использование деревьев выражений за рамками LINQ 283
9.4. Изменения в выведении типов и распознавании перегруженных версий 285
9.4.1. Причины внесения изменений: упрощение вызова обобщенных методов 285
9.4.2. Выведение возвращаемых типов анонимных функций 286
9.4.3. Двухэтапное выведение типов 288
9.4.4. Выбор правильного перегруженного метода 292
9.4.5. Итоги по выведению типов и распознаванию перегруженных версий 294
9.5. Резюме 294

Глава 10. Расширяющие методы 295

10.1. Ситуация до появления вспомогательных методов 296
10.2. Синтаксис расширяющих методов 298
10.2.1. Объявление расширяющих методов 299
10.2.2. Вызов расширяющих методов 300
10.2.3. Обнаружение расширяющих методов 301
10.2.4. Вызов метода на ссылке null 303
10.3. Расширяющие методы в .NET 3.5 305
10.3.1. Первые шаги в работе с классом Enumerable 305
10.3.2. Фильтрация с помощью метода Where() и соединение обращений к методам в цепочку 307
10.3.3. Антракт: разве мы не видели метод Where() раньше? 308
10.3.4. Проецирование с использованием метода Select() и анонимных типов 309
10.3.5. Сортировка с использованием метода OrderBy() 310
10.3.6. Бизнес-примеры, предусматривающие соединение вызовов в цепочки 312
10.4. Идеи и руководство по использованию 313
10.4.1. "Расширение мира" и совершенствование интерфейсов 313
10.4.2. Текучие интерфейсы 314
10.4.3. Разумное использование расширяющих методов 316
10.5. Резюме 317

Глава 11. Выражения запросов и LINQ to Objects 319

11.1. Введение в LINQ 320
11.1.1. Фундаментальные концепции LINQ 320
11.1.2. Определение эталонной модели данных 325
11.2. Простое начало: выборка элементов 326
11.2.1. Превращение начального источника в выборку 327
11.2.2. Трансляция компилятором как основа выражений запросов 327
11.2.3. Переменные диапазонов и нетривиальные проекции 330
11.2.4. Cast(), OfType() и явно типизированные переменные диапазонов 332
11.3. Фильтрация и упорядочение последовательности 335
11.3.1. Фильтрация с использованием конструкции where 335
11.3.2. Вырожденные выражения запросов 336
11.3.3. Упорядочение с использованием конструкции orderby 337

11.4. Конструкции let и прозрачные идентификаторы 339
11.4.1. Добавление промежуточных вычислений с помощью конструкции let 339
11.4.2. Прозрачные идентификаторы 340
11.5. Соединения 342
11.5.1. Внутренние соединения с использованием конструкций join 342
11.5.2. Групповые соединения с использованием конструкций join...into 346
11.5.3. Перекрестные соединения и выравнивание последовательностей с использованием нескольких конструкций from 349
11.6. Группирование и продолжение 353
11.6.1. Группирование с помощью конструкции group...by 353
11.6.2. Продолжение запроса 356
11.7. Выбор между выражениями запросов и точечной нотацией 359
11.7.1. Операции, которые требуют точечной нотации 359
11.7.2. Использование выражений запросов в ситуациях, когда точечная нотация может быть проще 360
11.7.3. Ситуации, когда выражения запросов блестящи 361
11.8. Резюме 362

Глава 12. LINQ за рамками коллекций 363

12.1. Запрашивание базы данных с помощью LINQ to SQL 364
12.1.1. Начало работы: база данных и модель 365
12.1.2. Начальные запросы 367
12.1.3. Запросы, в которых задействованы соединения 370
12.2. Трансляция с использованием IQueryable и IQueryProvider 372
12.2.1. Введение в IQueryable<T> и связанные интерфейсы 372
12.2.2. Имитация: реализация интерфейсов для регистрации вызовов 374
12.2.3. Интеграция выражений: расширяющие методы из класса Queryable 376
12.2.4. Имитированный поставщик запросов в действии 378
12.2.5. Итоги по интерфейсу IQueryable 379
12.3. API-интерфейсы, дружественные к LINQ, и LINQ to XML 380
12.3.1. Основные типы в LINQ to XML 380
12.3.2. Декларативное конструирование 382
12.3.3. Запросы для одиночных узлов 385
12.3.4. Выравнивающие операции запросов 386
12.3.5. Работа в гармонии с LINQ 388
12.4. Замена LINQ to Objects технологией Parallel LINQ 388
12.4.1. Отображение множества Мандельброта с помощью одного потока 389
12.4.2. Введение в ParallelEnumerable, ParallelQuery и AsParallel() 390
12.4.3. Подстройка параллельных запросов 392
12.5. Инвертирование модели запросов с помощью LINQ to Rx 394
12.5.1. IObservable<T> и IObserver<T> 394
12.5.2. Простое начало (снова) 396
12.5.3. Запрашивание наблюдаемых объектов 397
12.5.4. Какой в этом смысл? 400
12.6. Расширение LINQ to Objects 400
12.6.1. Руководство по проектированию и реализации 401
12.6.2. Пример расширения: выборка случайного элемента 402
12.7. Резюме 404

Часть IV. C# 4: изящная игра с другими 407

Глава 13. Небольшие изменения, направленные на упрощение кода 408

13.1. Необязательные параметры и именованные аргументы 409
13.1.1. Необязательные параметры 409
13.1.2. Именованные аргументы 416
13.1.3. Объединение двух средств 420
13.2. Модернизация взаимодействия с COM 424
13.2.1. Ужасы автоматизации Word до выхода C# 4 425
13.2.2. Реванш необязательных параметров и именованных аргументов 426
13.2.3. Ситуации, когда параметр `ref` в действительности таковым не является 427
13.2.4. Вызов именованных индексаторов 428
13.2.5. Связывание основных сборок взаимодействия 429
13.3. Обобщенная вариантность для интерфейсов и делегатов 432
13.3.1. Типы вариантности: ковариантность и контравариантность 432
13.3.2. Использование вариантности в интерфейсах 434
13.3.3. Использование вариантности в делегатах 437
13.3.4. Сложные ситуации 438
13.3.5. Ограничения и замечания 440
13.4. Мелкие изменения в блокировке и событиях, подобных полям 443
13.4.1. Надежная блокировка 443
13.4.2. Изменения в событиях, подобных полям 445
13.5. Резюме 446

Глава 14. Динамическое связывание в статическом языке 447

14.1. Что? Когда? Почему? Как? 449
14.1.1. Что такое динамическая типизация? 449
14.1.2. Когда динамическая типизация удобна и почему? 450
14.1.3. Как в C# 4 поддерживается динамическая типизация? 451
14.2. Пятиминутное руководство по `dynamic` 452
14.3. Примеры применения динамической типизации 455
14.3.1. COM в общем и Microsoft Office в частности 455
14.3.2. Динамические языки, подобные IronPython 457
14.3.3. Динамическая типизация в полностью управляемом коде 462
14.4. Заглядывая за кулисы 468
14.4.1. Введение в DLR 468
14.4.2. Основные концепции DLR 470
14.4.3. Как компилятор C# обрабатывает динамическое поведение 474
14.4.4. Компилятор C# становится еще интеллектуальнее 478
14.4.5. Ограничения, накладываемые на динамический код 481
14.5. Реализация динамического поведения 484
14.5.1. Использование `ExpandoObject` 484
14.5.2. Использование `DynamicObject` 488
14.5.3. Реализация интерфейса `IDynamicMetaObjectProvider` 495
14.6. Резюме 499

Часть V. C# 5: упрощение асинхронности 501

Глава 15. Асинхронность с помощью `async/await` 502

15.1. Введение в асинхронные функции 504
15.1.1. Первые встречи с асинхронностью 504
15.1.2. Разбор первого примера 506

15.2. Обдумывание асинхронности 507
15.2.1. Фундаментальные основы асинхронного выполнения 508
15.2.2. Моделирование асинхронных методов 510
15.3. Синтаксис и семантика 511
15.3.1. Объявление асинхронного метода 511
15.3.2. Возвращаемые типы асинхронных методов 512
15.3.3. Шаблон ожидания 513
15.3.4. Поток выражений `await` 516
15.3.5. Возвращение значений из асинхронных методов 521
15.3.6. Исключения 521
15.4. Асинхронные анонимные функции 530
15.5. Детали реализации: трансформация компилятора 532
15.5.1. Обзор сгенерированного кода 533
15.5.2. Структура каркасного метода 536
15.5.3. Структура конечного автомата 537
15.5.4. Одна точка входа для управления всем 539
15.5.5. Поток управления для выражений `await` 540
15.5.6. Отслеживание стека 542
15.5.7. Дополнительные сведения 543
15.6. Практическое использование `async/await` 544
15.6.1. Асинхронный шаблон, основанный на задачах 544
15.6.2. Объединение асинхронных операций 548
15.6.3. Модульное тестирование асинхронного кода 552
15.6.4. Возвращение к шаблону ожидания 555
15.6.5. Асинхронные операции в WinRT 556
15.7. Резюме 557

Глава 16. Дополнительные средства C# 5 и заключительные размышления 559

16.1. Изменения в захваченных переменных внутри циклов `foreach` 560
16.2. Атрибуты информации о вызывающем компоненте 560
16.2.1. Базовое поведение 561
16.2.2. Регистрация в журнале 563
16.2.3. Реализация интерфейса `INotifyPropertyChanged` 563
16.2.4. Использование атрибутов информации о вызывающем компоненте без .NET 4.5 565
16.3. Заключительные размышления 565

Приложение A. Стандартные операции запросов LINQ 567

A.1. Агрегирование 567
A.2. Конкатенация 568
A.3. Преобразование 569
A.4. Операции элементов 571
A.5. Эквивалентность 572
A.6. Генерация 573
A.7. Группирование 573
A.8. Соединения 574
A.9. Разделение 575
A.10. Проецирование 576
A.11. Квантификаторы 577
A.12. Фильтрация 578
A.13. Операции, основанные на множествах 578
A.14. Сортировка 579

Приложение Б. Обобщенные коллекции в .NET 581
Б.1. Интерфейсы 581
Б.2. Списки 583
Б.2.1. `List<T>` 583
Б.2.2. Массивы 584
Б.2.3. `LinkedList<T>` 585
Б.2.4. `Collection<T>`, `BindingList<T>`, `ObservableCollection<T>` и `KeyedCollection<TKey, TItem>` 586
Б.2.5. `ReadOnlyCollection<T>` и `ReadOnlyObservableCollection<T>` 587
Б.3. Словари 587
Б.3.1. `Dictionary<TKey, TValue>` 587
Б.3.2. `SortedList<TKey, TValue>` и `SortedDictionary<TKey, TValue>` 588
Б.3.3. `ReadOnlyDictionary<TKey, TValue>` 589
Б.4. Множества 589
Б.4.1. `HashSet<T>` 590
Б.4.2. `SortedSet<T>` (.NET 4) 590
Б.5. `Queue<T>` и `Stack<T>` 591
Б.5.1. `Queue<T>` 591
Б.5.2. `Stack<T>` 591
Б.6. Параллельные коллекции (.NET 4) 592
Б.6.1. `IProducerConsumerCollection<T>` и `BlockingCollection<T>` 592
Б.6.2. `ConcurrentBag<T>`, `ConcurrentQueue<T>` и `ConcurrentStack<T>` 593
Б.6.3. `ConcurrentDictionary<TKey, TValue>` 593
Б.7. Интерфейсы, допускающие только чтение (.NET 4.5) 593
Б.8. Резюме 595

Приложение В. Итоговые сведения по версиям 596
В.1. Главные выпуски инфраструктуры для настольных приложений 596
В.2. Средства языка C# 597
В.2.1. C# 2.0 597
В.2.2. C# 3.0 598
В.2.3. C# 4.0 598
В.2.4. C# 5.0 598
В.3. Средства библиотек инфраструктуры 598
В.3.1. .NET 2.0 598
В.3.2. .NET 3.0 599
В.3.3. .NET 3.5 599
В.3.4. .NET 4 600
В.3.5. .NET 4.5 601
В.4. Средства исполняющей среды (CLR) 601
В.4.1. CLR 2.0 601
В.4.2. CLR 4.0 601
В.5. Связанные инфраструктуры 602
В.5.1. Compact Framework 602
В.5.2. Silverlight 603
В.5.3. Micro Framework 603
В.5.4. Windows Runtime (WinRT) 604
В.6. Резюме 604

Предметный указатель 605

Отзывы о втором издании

Шедевр о C#.

Кирилл Осенков, команда Microsoft C# Team

Если вы собираетесь профессионально овладеть C#, то эта книга обязательна к прочтению.

Тайсон С. Максвелл,
старший инженер по программному обеспечению, Raytheon

Мы держим пари, что это будет лучшей книгой по C# 4.0.

Никандер Брюггеман и Маргрит Брюггеман,
консультанты по .NET, Lois & Clark IT Services

Полезный и увлекательный взгляд на эволюцию C# 4.

Джо Албахари,
автор инструмента *LINQPad*
и книги *C# 5.0. Полный справочник программиста*

Одна из лучших книг по C#, которые мне приходилось читать.

Алексей Нудельман,
исполнительный директор, C# Computing, LLC

Эта книга обязательна к прочтению всеми профессиональными разработчиками на C#.

Стюарт Каборн,
старший разработчик, BNP Paribas

Очень специализированный и высокопрофессиональный ресурс по обновлениям языка во всех основных версиях C#. Эту книгу должен иметь каждый опытный разработчик, желающий быть в курсе о новых средствах языка C#.

Шон Рейли,
программист/аналитик, Point2 Technologies

Зачем читать основы снова и снова? Джон сосредоточен на пережевывании нового материала!

Кейт Хилл, архитектор программного обеспечения,
Agilent Technologies

Все, что вы не понимаете, но должны знать о C#.

Джаред Парсонс, старший инженер
по программному обеспечению, Microsoft

Отзывы о первом издании

Попросту говоря, это, пожалуй, лучшая книга по программированию, которую я читал.

Крейг Пелки, автор,
System iNetwork

Я занимался разработкой на языке C# с самого его появления, но даже для меня эта книга преподнесла несколько сюрпризов. Я был особенно впечатлен великолепным изложением делегатов, анонимных методов, ковариантности и контравариантности. Даже если вы бывалый разработчик, эта книга научит вас чему-то совершенно новому о языке C#... Материал действительно настолько глубок, что никакая другая книга с этой не сравнится.

Адам Дж. Вольф,
Southeast Valley .NET User Group

Я читал всю книгу с удовольствием; она хорошо написана, а примеры в ней легки для понимания. Меня на самом деле увлекла тема лямбда-выражений и действительно понравилась посвященная им глава.

Хосе Роландо Гайя Пас,
разработчик веб-приложений, CSW Solutions

Эта книга воплощает отличные знания автором внутренней работы C# и выступает для читателей хорошо написанным, лаконичным и практичным руководством.

Джим Холмс,
автор Windows Developer Power Tools

Каждый термин используется надлежащим образом и в правильном контексте, каждый пример является ярким и содержит минимум кода, который демонстрирует всю широту средства... редкое удовольствие.

Френк Дженнин,
обозреватель Amazon UK

Если вы занимались разработкой на C# в течение нескольких лет и желаете узнать внутреннюю работу, то эта книга, несомненно, для вас.

Голо Роден, автор, лектор и инструктор
по .NET и связанным технологиям

Лучшая книга по C#, которую мне приходилось читать.

Крис Маллинс, C# MVP

Моим сыновьям, Тому, Робину и Уильяму

Вступление

Существуют два типа пианистов.

Есть пианисты, которые играют не потому, что им это нравится, а потому, что их родители заставляли брать уроки. А есть те, кто играют на пианино потому, что им нравится сама музыка. Их не нужно заставлять; наоборот, иногда совершенно не известно, когда они остановятся.

Ко второму типу относятся те, для которых игра на пианино является хобби. Другие играют, зарабатывая на жизнь. Это требует более высокого уровня самоотдачи, мастерства и таланта. Они могут иметь определенную свободу выбора музыкального жанра и стиля игры, но в целом их выбор регулируется требованиями работодателя и вкусами аудитории.

Во втором типе пианистов есть те, кто играет главным образом за деньги. Но есть и профессионалы, которые хотели бы играть для публики, даже если им не платят. Они с наслаждением используют свое мастерство и талант, даря музыку другим. Если же они могут получить удовольствие, но и плату за выступление, то так еще лучше.

Во втором типе можно выделить тех, кто являются самоучками, кто играет на слух, кто обладает большим талантом и возможностями, но не в состоянии донести свое интуитивное понимание другим, кроме как через саму музыку. А есть и те, кто официально обучались теории и практике. Они могут объяснить, какие приемы композитор применяет для достижения нужного эмоционального эффекта, и пользуются своими знаниями для формирования собственной интерпретации произведения.

Ко второму типу относятся те, кто никогда не заглядывал внутрь пианино. Но есть и те, кто в восторге от искусных анкерных механизмов, приподнимающих фетровые глушители за мгновение до того, как молоточки ударят по струнам. Они владеют выравнивателями клавиш и осевыми ключами. Они получают удовольствие и испытывают гордость от осознания того, что способны понимать работу механизмов инструмента, имеющего от 5 до 10 тысяч подвижных частей.

Наконец, во втором типе пианистов есть те, кто удовлетворены уровнем своего мастерства и используют свой талант ради удовольствия и выгоды, которую он приносит. А есть те, кто является не только музыкантами, теоретиками и техническими специалистами; они каким-то образом находят время для передачи своих знаний другим в качестве наставников.

Я понятия не имею, является ли Джон Скит пианистом или каким-то другим музыкантом. Но исходя из общения с ним по электронной почте на протяжении многих лет как с одним из наиболее ценных специалистов в команде проектировщиков C#, отслеживания его блога и чтения каждого слова в его книгах, по крайней мере, троекратно,

становится ясным, что Джон — разработчик программного обеспечения второго типа: он полон энтузиазма, информирован, талантлив, любознателен, умеет анализировать и к тому же обучает других.

C# является весьма прагматичным и быстро развивающимся языком. Я надеюсь, что благодаря добавлению всеобъемлющих запросов, более мощного выведения типов, компактного синтаксиса для анонимных функций и тому подобного, мы получаем в свое распоряжение совершенно новый стиль программирования, в то же время по-прежнему оставаясь верными статически типизированному, компонентно-ориентированному подходу, который обеспечил успех C#.

Многие из этих новых стилистических элементов обладают парадоксальным качеством выглядеть очень старыми (лямбда-выражения возвращают нас к фундаментальным основам вычислительной техники, заложенным в первой половине двадцатого века) и в то же самое время казаться новыми и незнакомыми разработчикам, которые используют более современный объектно-ориентированный подход.

Джон ухватил идею всего этого. Настоящая книга идеальна для профессиональных разработчиков, которые нуждаются в понимании того, *что* и *как* работает в последнем выпуске C#. Однако она предназначена также и для разработчиков, стремящихся обогатить свой опыт знанием причин, по которым были выбраны те или иные принципы проектирования языка.

Возможность извлечения максимально пользы от всех новых возможностей требует нового мышления о данных, функциях и отношении между ними. Это мало чем отличается от попытки играть джаз после многих лет обучения классической музыке — или наоборот. В любом случае я с нетерпением жду появления функциональных сочинений, которые придумает следующее поколение программистов на C#. Желаю успешного сочинительства, и благодарю за выбор для этого клавиши C#.

Эрик Липперт,
архитектор по анализу языка C#
в компании Coverity

Предисловие

Вот так-так! При написании этого предисловия я начал с предисловия ко второму изданию, которое начиналось со слов о том, как много времени прошло с тех пор, когда я писал предисловие к первому изданию. Второе издание теперь стало далеким воспоминанием и выглядит как совершенно другая жизнь. Я не уверен, говорит ли это о темпе современной жизни или о моей памяти, но в любом случае оно поучительно.

Со времен выхода первого издания и даже со второго характер разработки чрезвычайно изменился. Это было обусловлено многими факторами, наиболее очевидным из которых стал, пожалуй, возросший объем мобильных устройств. Однако многие проблемы остались теми же. Писать соответствующим образом интернационализированные приложения по-прежнему трудно. Все так же сложно изящно обрабатывать ошибки во всех ситуациях. Все еще довольно нелегко писать корректные многопоточные приложения, хотя эта задача значительно упростилась за счет улучшений, годами вносимых в язык и библиотеки.

В контексте этого предисловия важнее всего то, что я считаю, что разработчики по-прежнему должны знать используемый ими язык на том уровне, на котором они уверены в его поведении. Разработчики могут не знать тонкие детали каждого применяемого обращения к API-интерфейсу или даже непонятные краевые случаи в языке, которые не придется использовать[1], но основная часть языка должна быть подобна верному другу, на поведение которого разработчик может положиться.

Я уверен, что в дополнение к букве языка, на котором вы разрабатываете программное обеспечение, большое преимущество дает понимание его духа. Когда временами вы обнаруживаете, что безуспешно стараетесь победить какую-то проблему, то при совершении попыток заставить свой код работать так, как на это рассчитывали проектировщики языка, опыт принесет немалую пользу.

---

[1] Я должен сделать одно признание: я очень мало знаю о небезопасном коде и указателях в C#. Я просто никогда не испытывал в них потребности.

Благодарности

Может показаться, что написание третьего издания должно быть несложным делом — ведь все изменения сосредоточены в двух новых главах. На самом деле написание содержимого глав 15 и 16 "с чистого листа" было легкой частью. Намного труднее было незначительно корректировать текст в остальных главах, проверяя любые аспекты, которые были актуальны несколько лет назад, но теперь утратили смысл, и в целом удостоверяться в том, что вся книга соответствует тем высоким стандартам, которые, как я полагаю, предъявляются читателями. К счастью, мне посчастливилось иметь дело с людьми, которые поддерживали меня и позволили сделать книгу точной и сжатой.

Самое важное то, что моя семья вела себя замечательно как никогда. Моя жена Холли сама является детским автором, поэтому наши дети привыкли к тому, что мы иногда замыкаемся в себе, чтобы вложиться в редакционные сроки, но они все время оставались бодрыми и энергичными. Холли спокойно относилась ко всему этому, и я признателен ей за то, что она ни разу не напомнила мне о том, сколько книг она начала с нуля и благополучно завершила за то время, пока я трудился над этим третьим изданием. Официальные рецензенты перечислены позже, но я хотел бы персонально поблагодарить всех, кто заказал ранние копии третьего издания, искал опечатки и предлагал изменения, постоянно спрашивая, когда выйдет книга. Сам факт наличия читателей, с нетерпением ожидавших получения в свои руки окончательной книги, был крупным источником вдохновения.

Я всегда ладил с командой сотрудников в издательстве Manning, и было настоящим удовольствием работать с несколькими давними друзьями по первому изданию, а также с новоприбывшими. Майк Стивенс и Джефф Блейл профессионально организовали процесс принятия решений о том, что менять из предыдущего издания, а что оставить как есть. Они все расставили по своим местам. Энди Керолл и Кэти Теннант обеспечили, соответственно, квалифицированное техническое редактирование и корректуру, никогда не выражая недовольство стилем моего английского, придирчивостью или общей неясностью. Производственная группа, как всегда, делала свою магию за кулисами, но я все равно благодарен им: Дотти Марсико, Джанет Вейл, Марии Тюдор и Мэри Пирджис. Наконец, я хотел бы поблагодарить издателя, Марьяна Бейса, за то, что позволил мне написать третье издание и показал интересные перспективы на будущее.

Независимая рецензия чрезвычайно важна, и не только для обеспечения правильности технических деталей книги, но также для соблюдения равновесия и нужной интонации. Иногда получаемые комментарии оказывали влияние на форму всей книги; в других случаях я вносил в ответ весьма специфические изменения. В любом случае приветствовались любые отзывы. Итак, я благодарю следующих рецензентов за то, что они сделали эту книгу лучше для всех нас: Энди Криша, Баса Пеннингса, Брета Коллофа, Чарльза М. Гросса, Дрора Хелпера, Дастина Лейна, Ивана Тодоровича, Джона Пэриша, Себастьяна Мартина Агилара, Тиаана Гелденхайса и Тимо Бреденурта.

Особенно хочу поблагодарить Стивена Тауба и Стивена Клири, чьи ранние рецензии по главе 15 были просто бесценными. Асинхронность — такая тема, которую необычайно сложно описать ясно и точно, и их экспертные заключения существенно повлияли главу.

Разумеется, не будь команды проектировщиков C#, не появилась бы и эта книга. Их преданность языку при проектировании, реализации и тестировании достойна подражания, и я с нетерпением жду, что же они придумают в следующий раз. С тех пор как было опубликовано второе издание, Эрик Липперт покинул команду проектировщиков C# ради нового сказочного приключения, но я весьма признателен ему за то, что он по-прежнему смог выступить в качестве технического рецензента в этом третьем издании. Я также благодарен ему за вступление, которое он первоначально написал для первого издания и которое включено в настоящее издание еще раз. Я ссылаюсь на мысли Эрика по разнообразным вопросам повсеместно в книге, и если вы еще не читаете его блог (`http://ericlippert.com`), то самое время начать делать это.

Об этой книге

Эта книга посвящена языку C#, начиная с версии 2 и далее — вот так все просто. Я очень мало раскрываю версию C# 1, а библиотеки .NET Framework и общеязыковую исполняющую среду (Common Language Runtime — CLR) описываю, только когда они касаются языка. Это обдуманное решение, в результате которого получилась книга, несколько отличающаяся от большинства виденных мною книг по C# и .NET.

За счет предположения, что читатель располагает разумным объемом знаний C# 1, я избегаю траты сотен страниц на представление материала, который, как я считаю, большинство читателей уже понимает. Это обеспечивает мне пространство для раскрытия деталей более поздних версий C#, из-за которых, как я надеюсь, вы читаете данную книгу. Когда я писал первое издание этой книги, даже версия C# 2 была относительно незнакома некоторым читателям. К настоящему времени почти все разработчики на C# имеют определенный опыт использования средств, введенных в C# 2, но я все равно сохранил этот материал в третьем издании, поскольку он является чрезвычайно фундаментальным для того, что появилось в дальнейшем.

Кто должен читать эту книгу?

Эта книга ориентирована на разработчиков, в какой-то степени уже знающих язык C#. Абсолютно ценно, если вы будете хорошо знать C# 1, но очень немногое относительно последующих версий. Существует не так много читателей, которые находятся в лучшем положении, но я уверен, что все еще есть немало разработчиков, которые могут выиграть от более глубокого исследования версий C# 2 и C# 3, даже если они уже применяют их какое-то время, а многие разработчики даже не пользовались версией C# 4 или C# 5.

Если же вы вообще не знаете C#, то, вероятно, эта книга не для вас. Вы могли бы стараться изо всех сил, пытаясь понять аспекты, которые вам не знакомы, но это не может считаться эффективным способом изучения. Гораздо лучше начать с другой книги, а затем постепенно добавить в свою коллекцию и данную книгу. Доступно широкое многообразие книг, раскрывающих язык C# с нуля и написанных в самых разных стилях. Рекомендуется обратиться к одной из следующих книг (или даже ко всем им): *C# 5.0. Справочник. Полное описание языка* (ИД "Вильямс", 2013 г.), *Язык программирования C# 5.0 и платформа .NET 4.5* (ИД "Вильямс", 2013 г.) и *C# 5.0 и платформа .NET 4.5 для профессионалов* ("Диалектика", 2013 г.).

Я не собираюсь заявлять, что чтение этой книги сделает из вас классного кодировщика. Инженерия программного обеспечения предполагает намного большее, чем одно лишь знание синтаксиса языка, который приходится применять. Хотя я даю некоторые руководящие рекомендации, но, в конечном счете, интуиция и инстинкты при разработке присутствуют в намного большей степени, чем многие бы из нас хотели. Однако я могу утверждать, что если вы прочитаете и поймете эту книгу, то должны чувствовать себя комфортно с C# и свободно следовать своим инстинктам без особого опасения. Речь идет не о том, что можно написать код, который никто другой не поймет, поскольку в нем используются неизвестные закоулки языка, а о наличии уверенности в том, что вы знаете доступные варианты и то, к какому пути следования вас подталкивают идиомы C#.

Дорожная карта

Структура книги проста. Есть пять частей и три приложения. Первая часть служит введением, включая повторение тем, связанных с C# 1, которые важны для понимания более поздних версий языка и часто вызывают путаницу. Во второй части раскрываются новые средства версии C# 2, в третьей рассматривается версия C# 3 и т.д.

Бывают случаи, когда организация материала подобным образом означает, что мы будем возвращаться к какой-то теме пару раз — в частности, делегаты были усовершенствованы в C# 2 и затем еще раз в C# 3, — но в моем безрассудстве смысл все же присутствует. Я предвижу, что некоторые читатели будут применять в разных проектах разные версии языка; например, на работе вы можете использовать C# 4, а дома экспериментировать с C# 5. Это значит, что удобно пояснять, что к какой версии относится. Кроме того, такая организация способствует ощущению контекста и эволюции — она отражает то, каким образом язык развивался с течением времени.

В **главе 1** устанавливается сцена, для чего берется простой фрагмент кода C# 1 и затем развивается, чтобы продемонстрировать, каким образом последующие версии позволяют исходному коду становиться более читабельным и мощным. Мы взглянем на исторический контекст, в котором расширялся язык C#, и технический контекст, в котором он действует в качестве завершенной платформы; C# как язык построен на библиотеках платформы и мощной исполняющей среде для превращения абстракции в реальность.

В **главе 2** мы снова будем иметь дело с C# 1, рассматривая три специфичных аспекта: делегаты, характеристики системы типов и разницу между типами значений и ссылочными типами. Эти темы часто понимаются разработчиками на C# 1 в стиле "лишь относительно хорошо", но поскольку язык C# развивался и значительно усовершенствовал их, для освоения большинства новых средств требуется глубокое понимание основ.

В **главе 3** обсуждается крупнейшее средство C# 2, потенциально самое трудное для освоения: обобщения. Методы и типы могут быть написаны обобщенным образом, с параметрами типов, указанными вместо реальных типов, которые задаются в вызывающем коде. Поначалу обобщения будут казаться не менее запутанными, чем это описание, но после того, как вы их поймете, вы непременно удивитесь, как в принципе могли обходиться без них ранее. Если вам когда-либо хотелось представлять целочисленное значение, равное `null`, то **глава 4** как раз для вас. В ней рассматриваются типы, допускающие `null`: средство, построенное на основе обобщений, которое использует в своих интересах поддержку со стороны языка, исполняющей среды и инфраструктуры.

В **главе 5** описаны усовершенствования делегатов в C# 2. До сих пор делегаты можно было применять только для обработки событий, таких как щелчки на кнопках. В C# 2 упростилось создание делегатов, а библиотечная поддержка сделала их более удобными в ситуациях, отличных от обработки событий.

В **главе 6** будут исследоваться итераторы и легкий способ их реализации в C# 2. Итераторные блоки используют лишь немногие разработчики, но поскольку технология LINQ to Objects построена на основе итераторов, они будут становиться все более и более важными. Ленивая природа их выполнения также является ключевой частью LINQ.

В **главе 7** представлено несколько мелких средств, введенных в C# 2, каждое из которых делает жизнь чуть более приятной. Проектировщики языка сгладили несколько шероховатостей в C# 1, сделав возможным более гибкое взаимодействие с генераторами кода, улучшив поддержку вспомогательных классов, обеспечив более гибкий доступ к свойствам и т.д.

В **главе 8** снова раскрывается несколько относительно простых средств, но на этот раз из версии C# 3. Почти весь новый синтаксис приспособлен к общей цели техноло-

гии LINQ, но строительные блоки также полезны и сами по себе. Благодаря анонимным типам, автоматически реализуемым свойствам, неявно типизированным локальным переменным и существенно улучшенной поддержке инициализации, C# 3 становится намного более мощным языком для выражения необходимого поведения.

В **главе 9** рассматривается первая крупная тема, связанная с C# 3 — лямбда-выражения. Не довольствуясь и без того достаточно лаконичным синтаксисом, который обсуждался в главе 5, проектировщики языка еще больше упростили создание делегатов по сравнению с C# 2. Лямбда-выражения способны на большее — они могут быть преобразованы в деревья выражений, которые являются мощным способом представления кода в виде данных.

В **главе 10** будут исследоваться расширяющие методы, которые позволяют заставить компилятор считать, что методы, объявленные в одном типе, на самом деле принадлежат другому типу. На первый взгляд они выглядят кошмаром в плане читабельности, но если хорошо подумать, то это исключительно мощное средство, к тому же являющееся жизненно важным для LINQ.

Глава 11 объединяет три предыдущих главы в форме выражений запросов — лаконичным, но мощным способом запрашивания данных. Первоначально мы сосредоточим внимание на LINQ to Objects, но вы увидите, что шаблон выражений запросов применяется так, что позволяет легко подключать других поставщиков данных.

В **главе 12** предлагается краткий обзор разнообразных сценариев использования LINQ. Сначала мы взглянем на преимущества выражений запросов, скомбинированных с деревьями выражений, демонстрируя способность LINQ to SQL преобразовывать то, что выглядит как нормальный код C#, в операторы SQL. Затем мы перейдем к рассмотрению способов проектирования библиотек для хорошего сочетания с LINQ, взяв в качестве примера LINQ to XML. Двумя альтернативными подходами к внутрипроцессным запросам являются Parallel LINQ и Reactive Extensions, а в завершение главы приводятся рассуждения о том, как можно расширить LINQ to Objects собственными операциями LINQ.

Рассмотрение версии C# 4 начинается в **главе 13**, в которой раскрываются именованные аргументы и необязательные параметры, улучшения во взаимодействии с COM и обобщенная вариантность. В некотором смысле все они являются совершенно разными средствами, но именованные аргументы и необязательные параметры способствуют взаимодействию с COM, а также более специфичным возможностям, которые доступны только при работе с объектами COM.

В **главе 14** описана единственное крупное средство C# 4: динамическая типизация. Возможность динамической привязки членов во время выполнения вместо статической на этапе компиляции является серьезным концептуальным отклонением для языка C#, но она применяется выборочно — выполняться динамически будет только код, в котором задействовано динамическое значение.

Глава 15 посвящена асинхронности. В C# 5 содержится только одно главное средство — возможность написания асинхронных функций. Это единственное средство одновременно сложно в понимании и элегантно в использовании. Наконец-то стало возможным написание кода, который не выглядит подобно спагетти.

В **главе 16** рассматриваются оставшиеся средства C# 5 (оба они крошечные) и приводятся некоторые мысли о будущем.

Приложения содержат справочный материал. В **приложении А** описаны стандартные операции запросов LINQ с несколькими примерами. В **приложении Б** рассматриваются основные обобщенные классы и интерфейсы коллекций. В **приложении В** предоставляется краткое описание разных версий .NET, включая разновидности наподобие Compact Framework и Silverlight.

Терминология, оформление и загружаемый код

Большая часть терминологии, применяемой в книге, объясняется по ходу дела, но есть несколько определений, о которых полезно упомянуть здесь. Я использую понятия C# 1, C# 2, C# 3, C# 4 и C# 5 вполне очевидным образом, но в других книгах и на веб-сайтах вы можете встретить варианты C# 1.0, C# 2.0, C# 3.0, C# 4.0 и C# 5.0. Как по мне, указание дополнительной конструкции ".0" избыточно, поэтому я ее опускаю — надеюсь, смысл понятен.

Я позаимствовал пару терминов из книги по C#, написанной Марком Михаэлисом. Во избежание путаницы между *исполняющей средой* (runtime), как в "общеязыковой исполняющей среде", и моментом времени, как в "переопределение происходит во время выполнения", для последней концепции Марк применяет понятие *время выполнения* (execution time), обычно сопоставляя его с *этапом компиляции* (compile time). Мне кажется, что это совершенно здравая идея, которая, как я надеюсь, будет принята широким сообществом. Следуя его примеру, я внес свою лепту, используя данные термины в этой книге.

Я часто ссылаюсь на "спецификацию языка" или просто "спецификацию" — если только не указано иное, это означает спецификацию языка C#. Тем не менее, доступно множество версий этой спецификации, частично из-за наличия разных версий самого языка, а частично из-за процесса стандартизации. Все упоминаемые номера разделов относятся к спецификации по языку C# 5.0, написанной в Microsoft.

В настоящей книге содержатся многочисленные фрагменты кода, которые представлены с помощью моноширинного шрифта; этот же шрифт также применяется в выводе кода из листингов. Некоторые листинги сопровождаются аннотациями, а иногда отдельные части кода выделены полужирным, чтобы обозначить изменение, улучшение или добавление. Почти весь код приведен в форме фрагментов, позволяющих ему оставаться компактным, но по-прежнему выполняемым, правда, в надлежащей среде. Такой средой является Snippy — специальный инструмент, который представлен в разделе 1.8. Инструмент Snippy доступен для загрузки наряду с кодом всех примеров, рассмотренных в книге (в форме фрагментов, в виде полноценных решений Visual Studio или чаще в обоих видах), на веб-сайте книги `csharpindepth.com`, а также на веб-сайте издательства.

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 веб-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:	127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины:	03150, Киев, а/я 152

Об авторе

Справедливости ради отмечу, что я не являюсь типичным разработчиком на C#. В последние пять лет я почти все время работал с C# просто ради удовольствия — фактически это было навязчивым хобби. Моя работа связана с написанием серверного кода на Java в Google London, и я могу смело утверждать, что ничто так не помогает по-настоящему оценить новые средства языка, как необходимость написания кода на языке, в котором эти средства отсутствуют, но который достаточно похож на первоначальный язык, чтобы постоянно напоминать об их отсутствии.

Я старался поддерживать контакт с теми разработчиками, которые испытывали трудности с языком C#, отслеживая информацию на веб-сайте Stack Overflow, публикуя статьи о странностях в своем блоге и периодически беседуя о C# и связанных темах везде, где публика была готова меня слушать. Вдобавок я активно разрабатываю API-интерфейс для .NET с открытым кодом для работы с датами и временем под названием Noda Time (`http://nodatime.org`). Короче говоря, C# по-прежнему энергично струится в моих сосудах.

При всех эти странностях и, невзирая на мой вызывающий удивление статус "микрознаменитости" благодаря веб-сайту Stack Overflow, во многих других отношениях я остаюсь совершенно обычным разработчиком. Я пишу много кода, который заставляет меня поморщиться, когда я спустя время к нему возвращаюсь. Я не всегда сначала пишу модульные тесты, а иногда вообще обхожусь без них. Я не так уж редко допускаю ошибки. Раздел, посвященный выведению типов, в спецификации по C# по-прежнему выглядит для меня запутанным, а некоторые сценарии использования шаблонов Java ставят меня в тупик. Я отнюдь не выдающийся программист.

Но вот как должны обстоять дела в идеальном случае. На следующих нескольких сотнях страниц я постараюсь представить себя иначе: я буду предлагать полезные советы, как если бы сам всегда им следовал, и выражать неодобрение вульгарным сокращениям, как будто никогда даже не помышлял об их применении. Не верьте ни единому моему слову. Правда заключается в том, что я, скорее всего, такой же, как и вы. Так уж вышло, что я немного больше вас знаю о том, как работает C#, вот и все... но такое положение дел просуществует только до момента, когда вы закончите чтение этой книги.

Об иллюстрации на обложке

Надпись для иллюстрации на обложке этой книги гласит: "Музыкант". Иллюстрация была взята из альбома с коллекцией костюмов Османской империи, напечатанного 1 января 1802 года Уильямом Миллером на Старой Бонд-стрит в Лондоне. Титульная страница утеряна, и к настоящему времени мы не смогли ее разыскать. В оглавлении альбома рисунки обозначены на английском и французском языке, и для каждого рисунка указаны фамилии двух художников, трудившихся над ней. Не сомневаюсь, что они были бы весьма удивлены, обнаружив, что их произведение украшает обложку книги по программированию, вышедшей спустя две сотни лет.

Альбом с этой коллекцией был приобретен редактором из Manning на блошином рынке антиквариата "Garage" на 26-й западной улице в Манхэттене. Продавцом был американец, живущий в Анкаре, и сделка произошла как раз, когда он собирал раскладку, завершив торговлю. У редактора Manning не было с собой достаточной суммы наличными, а кредитную карту и чек продавец вежливо отклонил. С учетом того, что он вечером улетал домой в Анкару, ситуация становилась безнадежной. Каким было решение? Вполне достаточно оказалось старомодного устного соглашения, скрепленного рукопожатием. Продавец просто предложил перевести ему деньги через банк, и редактор ушел с клочком бумаги с банковской информацией и альбомом под мышкой. Само собой разумеется, мы перечислили деньги сразу же на следующий день, но все еще остаемся признательными и восхищенными доверием, которое этот неизвестный человек оказал одному из нас. Это напоминает нам то, что могло происходить только в старые добрые времена.

Мы в Manning оцениваем изобретательность, инициативность и, конечно же, юмористичность книг на темы, связанные с компьютерами, на основе богатого разнообразия региональной жизни два столетия тому назад, воскрешая из небытия рисунки из этой коллекции.

Часть I

Подготовка к путешествию

Каждый читатель этой книги обладает собственным уровнем мастерства и возлагает на нее свои надежды. Возможно, вы эксперт, ищущий возможность заполнить пробелы, пусть и малые, в существующих знаниях. Или вы считаете себя рядовым разработчиком с небольшим опытом использования обобщений и лямбда-выражений, но желаете лучше понять, как они работают. А может быть, вы довольно хорошо владеете версиями C# 2 и C# 3, но не имеете опыта применения C# 4 или C# 5.

Как автор, я не могу считать всех читателей одинаковыми, да и не хочу, если бы даже мог. Однако я надеюсь, что все читатели обладают двумя общими чертами: стремлением к более глубокому пониманию C# как языка и наличием хотя бы базовых знаний C# 1. Обо всем остальном я позабочусь сам.

Потенциально широкий диапазон уровней квалификации и является главной причиной существования этой части книги. Возможно, вы уже знаете, чего ожидать от последующих версий C#, либо наоборот — все это может оказаться для вас совершенно новым. Вы можете иметь основательные знания C# 1 — или же отставать в каких-то деталях, важность которых значительно возрастает при освоении более поздних версий языка. К концу части I все это деление на уровни я отброшу в сторону, но вы должны подойти к изучению остального материала книги с уверенностью и пониманием того, что будет происходить позже.

В первых двух главах мы будем устремлять взгляд как вперед, так и назад. Одной из ключевых тем этой книги является эволюция. Перед введением любого средства в язык члены команды проектировщиков C# тщательно взвешивают, хорошо ли вписывается это средство в контекст того, что уже существует, и намечают цели на будущее. Это обеспечивает ощущение согласованности языка даже в разгар перемен. Чтобы понять, как и почему язык развивается, необходимо видеть, откуда он пришел и куда двигается.

В главе 1 представлен вид с высоты птичьего полета на оставшиеся части книги, с кратким взглядом на некоторые важнейшие особенности C#, появившиеся после версии 1. Я продемонстрирую развитие кода со времен C# 1, применяя новые средства поодиночке до тех пор, пока код не станет совершенно неузнаваемым со своего скромного начального вида. Мы также ознакомимся с терминологией, используемой в дальнейших материалах книги, и форматом кода примеров.

Глава 2 в большой степени сосредоточена на C# 1. Если вы эксперт в C# 1, можете пропустить эту главу, но учтите, что в ней затрагиваются области C# 1, которые зачастую понимаются неправильно. Вместо попытки объяснить язык в целом внимание в главе сосредоточено на возможностях, которые являются фундаментальными для последних версий C#. Получив такую прочную основу, можно смело переходить к рассмотрению C# 2 во второй части этой книги.

Глава 1

Изменение стиля разработки в C#

В этой главе...

- Развивающийся пример
- Композиция .NET
- Использование кода в этой книге
- Спецификация языка C#

Знаете, что мне по-настоящему нравится в динамических языках, таких как Python, Ruby и Groovy? Они отбрасывают из кода все незначительное, оставляя только его сущность — части, которые действительно что-то *делают*. Скучная формальность уступает дорогу средствам наподобие генераторов, лямбда-выражений и списковых включений.

Интересно отметить, что некоторые средства, стремящиеся придать динамическим языкам легковесный характер, ничего не делают для обеспечения своей динамичности. Разумеется, определенные средства это делают — например, утиная (неявная) типизация и "магия", используемая в Active Record — но статически типизированные языки не *обязательно* должны быть неуклюжими и тяжеловесными.

Давайте обратимся к C#. В некотором смысле язык C# 1 можно рассматривать как улучшенную версию языка Java образца примерно 2001 года. Все сходства были слишком очевидными, но в C# имелось несколько дополнений: свойства как основополагающая характеристика языка, делегаты и события, циклы `foreach`, операторы `using`, явное переопределение методов, перегрузка операций и специальные типы значений — и это далеко не полный перечень. Понятно, что предпочтения относительно языка — личное дело каждого, однако я воспринимал C# 1 как шаг вперед от Java, когда только начал пользоваться им.

В тех пор все становилось только лучше. В каждую новую версию добавлялись важные средства, снижающие инстинктивный страх разработчиков, причем всегда в тщательно продуманной манере и с минимальной обратной несовместимостью. Даже

до появления в C# 4 возможности применения динамической типизации там, где она действительно удобна, многие средства, традиционно связываемые с динамическими и функциональными языками, нашли свое отражение в C#, позволяя получать код, который проще в написании и сопровождении. Аналогично, хотя средства, касающиеся асинхронной обработки, в C# 5 не точно совпадают с такими средствами в F#, мне кажется, что последние оказали определенное влияние.

В настоящей книге я последовательно проведу вас по всем изменениям, предоставляя достаточно деталей, чтобы вы спокойно воспринимали те удивительные вещи, которые компилятор C# теперь готов предложить. Тем не менее, обо всех них пойдет речь позже — в этой главе я в бешеном темпе постараюсь осветить столько, сколько смогу, едва переводя дыхание. Я объясню, что имеется в виду при сопоставлении C# как языка и .NET как платформы, и предоставлю несколько важных замечаний относительно кода примеров для остальных частей книги. После этого можно углубляться в детали.

В этой одной главе мы не собираемся охватить абсолютно все изменения, внесенные в язык C#. Тем не менее, мы рассмотрим обобщения, свойства с различными модификаторами доступа, типы, допускающие значения `null`, анонимные методы, автоматически реализуемые свойства, усовершенствованные инициализаторы коллекций и объектов, лямбда-выражения, расширяющие методы (часто называемые также методами расширения), неявную типизацию, выражения запросов LINQ, именованные аргументы, необязательные параметры, упрощенное взаимодействие с COM, динамическую типизацию и асинхронные функции. Это проведет нас по всему пути от C# 1 до последней версии, C# 5. Итак, приступим.

1.1. Простой тип данных

В этой главе я позволю компилятору C# предпринимать удивительные вещи, не говоря о том, *как* он это делает, а лишь кратко указывая на то, *что* он делает или *почему*. Это единственный раз, когда я не буду объяснять, каким образом все работает, или пытаться охватить сразу более одного шага. На самом деле все наоборот — план заключается в том, чтобы произвести впечатление, а не обучать. Если при чтении этого раздела у вас не возникло хотя бы легкого волнения по поводу того, что можно делать с помощью C#, то возможно эта книга не для вас. Однако, скорее всего, вы готовы ринуться в исследование деталей работы этих магических трюков, о которых пойдет речь в остальных частях книги.

Пример, который будет использоваться, несколько надуман — в нем преследуется цель втиснуть максимально возможное количество новых средств в как можно более короткий фрагмент кода. Это избитый пример, но зато он должен быть знаком. Да, речь идет о примере с товаром, наименованием и ценой, представляющим собой аналог программы "Hello, world" в области электронной коммерции. Мы посмотрим, как решаются разнообразные задачи, и каким образом новые версии C# позволяют упростить эти решения и сделать их более элегантными. Вы не увидите преимуществ C# 5 вплоть до конца главы, но это отнюдь не уменьшит их важность.

1.1.1. Тип `Product` в C# 1

Мы начнем с определения типа, представляющего товар, и затем приступим к манипуляциям с ним. Здесь пока что нет ничего выдающегося — просто инкапсуляция пары свойств. Чтобы упростить дальнейшую демонстрацию, мы создадим список предопределенных товаров.

В листинге 1.1 показано, как можно было бы реализовать тип на C# 1. После этого мы посмотрим, как переписать этот код во всех последующих версиях C#. Мы будем следовать такому шаблону и в остальных фрагментах кода. Учитывая, что я пишу это в 2013 году, скорее всего, код выглядит для вас довольно знакомым, но всегда полезно оглянуться назад и оценить, насколько существенно продвинулся язык в своем развитии.

Листинг 1.1. Тип `Product` (C# 1)

```
using System.Collections;
public class Product
{
  string name;
  public string Name { get { return name; } }

  decimal price;
  public decimal Price { get { return price; } }

  public Product(string name, decimal price)
  {
    this.name = name;
    this.price = price;
  }

  public static ArrayList GetSampleProducts()
  {
    ArrayList list = new ArrayList();
    list.Add(new Product("West Side Story", 9.99m));
    list.Add(new Product("Assassins", 14.99m));
    list.Add(new Product("Frogs", 13.99m));
    list.Add(new Product("Sweeney Todd", 10.99m));
    return list;
  }

  public override string ToString()
  {
    return string.Format("{0}: {1}", name, price);
  }
}
```

В листинге 1.1 нет ничего такого, что было бы трудно понять — в конце концов, это всего лишь код C# 1. Тем не менее, здесь демонстрируются три ограничения.

- На этапе компиляции тип `ArrayList` не имеет каких-либо сведений о том, что в нем содержится. В список, создаваемый внутри метода `GetSampleProducts()`, может быть случайно добавлена, скажем, строка, и компилятор не заметит этого.
- Мы предоставили открытые средства получения (`get`) для свойств, а это значит, что соответствующие им средства установки (`set`), если бы они понадобились, пришлось бы также делать открытыми.
- Присутствует масса незначительных деталей при создании свойств и переменных — код, который усложняет довольно простую задачу инкапсуляции строки и десятичного значения.

Давайте посмотрим, каким образом C# 2 поможет улучшить ситуацию.

1.1.2. Строго типизированные коллекции в C# 2

Начальный набор изменений (показанных в листинге 1.2) касается первых двух ограничений, упомянутых выше, и включает наиболее важное нововведение версии C# 2 — обобщения. Новые фрагменты кода выделены полужирным.

Листинг 1.2. Строго типизированные коллекции и закрытые средства установки (C# 2)

```
public class Product
{
  string name;
  public string Name
  {
    get { return name; }
    private set { name = value; }
  }
  decimal price;
  public decimal Price
  {
    get { return price; }
    private set { price = value; }
  }
  public Product(string name, decimal price)
  {
    Name = name;
    Price = price;
  }
  public static List<Product> GetSampleProducts()
  {
    List<Product> list = new List<Product>();
    list.Add(new Product("West Side Story", 9.99m));
    list.Add(new Product("Assassins", 14.99m));
    list.Add(new Product("Frogs", 13.99m));
    list.Add(new Product("Sweeney Todd", 10.99m));
    return list;
  }
  public override string ToString()
  {
    return string.Format("{0}: {1}", name, price);
  }
}
```

Теперь мы имеем свойства с закрытыми средствами установки (которые используются внутри конструктора), и без особого труда можно догадаться, что тип `List<Product>` сообщает компилятору о том, что список содержит товары. Попытка добавления в список значения другого типа приведет к ошибке компиляции, к тому же не придется выполнять приведение результатов после извлечения их из списка.

Изменения, внесенные в C# 2, позволили обойти два из трех указанных выше ограничений, а разрешить оставшееся поможет версия C# 3.

1.1.3. Автоматически реализуемые свойства в C# 3

Мы начнем с исследования ряда довольно простых средств, появившихся C# 3. Автоматически реализуемые свойства и упрощенная инициализация, продемонстрированные в листинге 1.3, относительно тривиальны по сравнению с лямбда-выражениями и прочими средствами, однако они помогают сделать код намного проще.

Листинг 1.3. Автоматически реализуемые свойства и упрощенная инициализация (C# 3)

```
using System.Collections.Generic;
class Product
{
  public string Name { get; private set; }
  public decimal Price { get; private set; }

  public Product(string name, decimal price)
  {
    Name = name;
    Price = price;
  }

  Product() {}

  public static List<Product> GetSampleProducts()
  {
    return new List<Product>
    {
       new Product { Name="West Side Story", Price = 9.99m },
       new Product { Name="Assassins", Price=14.99m },
       new Product { Name="Frogs", Price=13.99m },
       new Product { Name="Sweeney Todd", Price=10.99m}
    };
  }

  public override string ToString()
  {
    return string.Format("{0}: {1}", Name, Price);
  }
}
```

Как видите, со свойствами не связан какой-либо код (или видимые переменные), а жестко закодированный список строится совершенно по-другому. Поскольку переменные `name` и `price` теперь отсутствуют, в классе приходится повсеместно применять свойства, улучшая согласованность. В коде предусмотрен закрытый конструктор без параметров, предназначенный для выполнения новой инициализации с помощью свойств. (Этот конструктор вызывается для каждого элемента списка перед установкой свойств.)

В приведенном выше примере можно было бы полностью удалить открытый конструктор, но тогда во внешнем коде не удалось бы создавать другие экземпляры товаров.

1.1.4. Именованные аргументы в C# 4

При рассмотрении случая с C# 4 мы возвратимся к первоначальному коду со свойствами и конструктором, чтобы сделать их полностью неизменяемыми. Тип, имеющий только закрытые средства установки, не разрешено изменять *открытым* образом, но это можно сделать более очевидным, если устранить также и возможность закрытого

изменения[1]. К сожалению, сокращений для свойств, допускающих только чтение, не предусмотрено, но C# 4 позволяет указывать имена аргументов при вызове конструктора (листинг 1.4), что обеспечивает прозрачность инициализаторов C# 3 и отсутствие изменяемости.

Листинг 1.4. Использование именованных аргументов для получения прозрачного кода инициализации (C# 4)

```
using System.Collections.Generic;
public class Product
{
  readonly string name;
  public string Name { get { return name; } }

  readonly decimal price;
  public decimal Price { get { return price; } }

  public Product(string name, decimal price)
  {
    this.name = name;
    this.price = price;
  }
  public static List<Product> GetSampleProducts()
  {
    return new List<Product>
    {
      new Product( name: "West Side Story", price: 9.99m),
      new Product( name: "Assassins", price: 14.99m),
      new Product( name: "Frogs", price: 13.99m),
      new Product( name: "Sweeney Todd", price: 10.99m)
    };
  }
  public override string ToString()
  {
    return string.Format("{0}: {1}", name, price);
  }
}
```

В этом примере преимущества явного указания имен аргументов относительно невелики, но в случае, когда метод или конструктор принимает множество параметров, такая возможность позволяет сделать код намного яснее — особенно, если параметры имеют один и тот же тип или необходимо передавать значения `null` для некоторых аргументов. Разумеется, вы самостоятельно решаете, когда применять это средство, указывая имена аргументов только в случае, если это упрощает понимание кода.

На рис. 1.1 показаны итоги эволюции типа `Product` к настоящему моменту. Аналогичные блок-схемы будут предлагаться после решения каждой задачи, что поможет вам оценить, каким образом эволюция C# улучшает код. Обратите внимание, что во всех блок-схемах не учитывается C# 5; причина в том, что главная особенность C# 5 (асинхронные функции) касается области, которая в действительности развивалась не в терминах языковой поддержки. Вскоре мы кратко рассмотрим это средство.

[1] В коде C# 1 также можно было обеспечить неизменяемость; это не было сделано лишь для того, чтобы упростить демонстрацию изменений в C# 2 и C# 3.

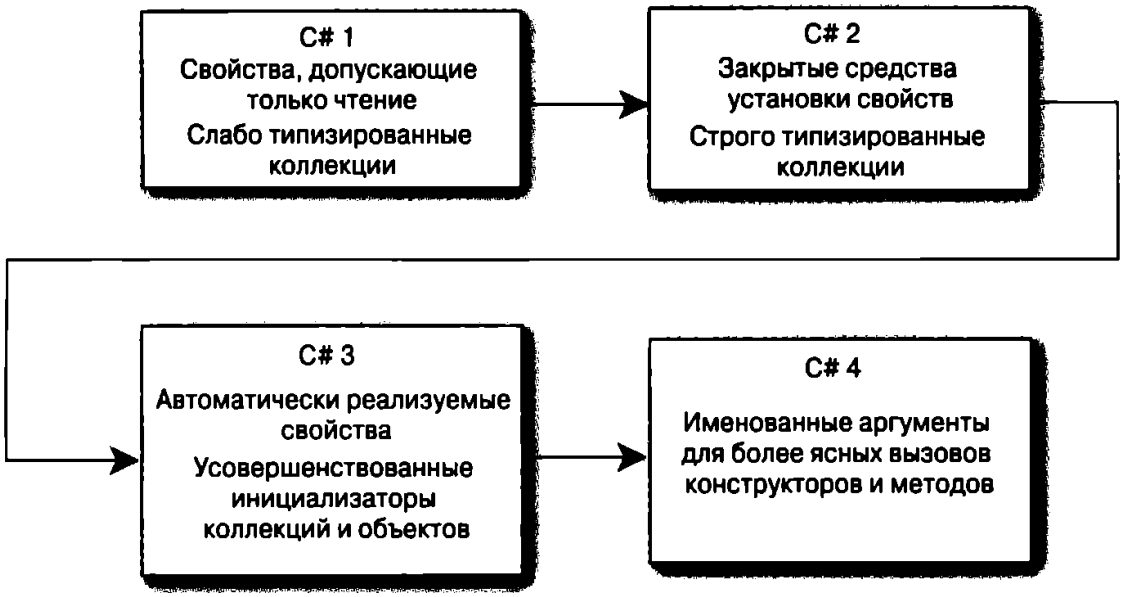

Рис. 1.1. Эволюция типа `Product`, демонстрирующая улучшенную инкапсуляцию, более строгую типизацию и простоту инициализации

Пока что изменения были относительно небольшими. Но на самом деле добавление обобщений (синтаксиса `List<Product>`) является, пожалуй, самой важной частью C# 2, хотя до сих пор пользу от них можно было оценить только частично. Еще не было показано ничего такого, что заставило бы сердце биться чаще, но мы ведь только начали. Следующая задача связана с выводом списка товаров в алфавитном порядке.

1.2. Сортировка и фильтрация

В этом разделе тип `Product` изменяться не будет. Взамен мы отсортируем примеры товаров по названию и затем найдем самые дорогостоящие. Обе задачи не являются трудоемкими, но мы покажем, насколько их решение становилось проще с течением времени.

1.2.1. Сортировка товаров по названию

Простейший способ вывода списка в определенном порядке предусматривает его сортировку и проход по нему с отображением элементов. В .NET 1.1 для этого используется метод `ArrayList.Sort()` и дополнительно предоставляется реализация интерфейса `IComparer`, позволяющая задать отдельное сравнение. Можно было бы обеспечить реализацию типом `Product` интерфейса `IComparable`, но это позволило бы определять только один порядок сортировки, а совсем несложно предположить, что на каком-то этапе понадобится сортировать по цене, а также по названию.

В листинге 1.5 определяется реализация `IComparer`, после чего выполняется сортировка списка и его отображение.

Листинг 1.5. Сортировка `ArrayList` с применением `IComparer` (C# 1)

```
class ProductNameComparer : IComparer
{
  public int Compare(object x, object y)
  {
```

```
        Product first = (Product)x;
        Product second = (Product)y;
        return first.Name.CompareTo(second.Name);
    }
}
...
ArrayList products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
    Console.WriteLine (product);
}
```

Первое, что обнаруживается в листинге 1.5 — это определение дополнительного типа для помощи в сортировке. Ничего страшного в нем нет, если не считать необходимость в написании излишне большого объема кода в ситуации, когда сортировка нужна только в одном месте. Далее обратите внимание на несколько приведений в методе `Compare()`. Приведения являются способом сообщения компилятору о том, что вам известно больше информации, нежели ему, и обычно означают возможность неправильного предположения. Если список `ArrayList`, возвращаемый из метода `GetSampleProducts()`, в действительности содержит строку, работа кода нарушится в том месте, где предпринимается попытка приведения этой строки к типу `Product`.

Приведение также присутствует в коде, отображающем отсортированный список. Оно не очевидно, т.к. компилятор добавляет его автоматически, в результате чего цикл `foreach` неявно приводит каждый элемент списка к типу `Product`. Опять-таки, это приведение может дать сбой во время выполнения, и здесь снова на выручку приходят обобщения из C# 2. В листинге 1.6 показан предыдущий код с *единственным* внесенным изменением, которое связано с использованием обобщений.

Листинг 1.6. Сортировка `List<Product>` с применением `IComparer<Product>` (C# 2)

```
class ProductNameComparer : IComparer<Product>
{
    public int Compare(Product x, Product y)
    {
        return x.Name.CompareTo(y.Name);
    }
}
...
List<Product> products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
    Console.WriteLine(product);
}
```

Код компаратора в листинге 1.6 проще, поскольку он сразу получает объекты товаров. Приведение здесь не нужно. Аналогично исчезает невидимое приведение в цикле `foreach`. Компилятор по-прежнему должен учитывать возможность преобразования исходного типа последовательности в целевой тип переменной, но в этом случае оба типа относятся к `Product`, и никакого специального кода преобразования генерироваться не будет.

Хотя улучшение заметно, но было бы неплохо сортировать товары, просто указывая нужное сравнение и не реализуя для этого интерфейс. Именно это и делается в листинге 1.7, в котором с помощью делегата методу `Sort()` сообщается, каким образом сравнивать два товара.

Листинг 1.7. Сортировка `List<Product>` с использованием делегата `Comparison<Product>` (C# 2)

```
List<Product> products = Product.GetSampleProducts();
products.Sort(delegate(Product x, Product y)
  { return x.Name.CompareTo(y.Name); }
);
foreach (Product product in products)
{
  Console.WriteLine(product);
}
```

Обратите внимание на отсутствие типа `ProductNameComparer`. Выделенный полужирным оператор создает экземпляр делегата, который предоставляется методу `Sort()` для выполнения сравнений. Вы изучите это средство (*анонимные методы*) в главе 5.

Итак, все ограничения, касающиеся кода на C# 1, ликвидированы. Однако это не означает, что в C# 3 нельзя добиться дополнительных улучшений. Для начала заменим анонимный метод еще более компактным способом создания экземпляра делегата (листинг 1.8).

Листинг 1.8. Сортировка с применением делегата `Comparison<Product>`, получаемого из лямбда-выражения (C# 3)

```
List<Product> products = Product.GetSampleProducts();
products.Sort((x, y) => x.Name.CompareTo(y.Name));
foreach (Product product in products)
{
  Console.WriteLine(product);
}
```

Синтаксис выглядит еще более странно (из-за наличия *лямбда-выражения*), но он по-прежнему создает делегат `Comparison<Product>`, как в листинге 1.7, и на этот раз с меньшей суетой. Нам не пришлось использовать ключевое слово `delegate` для определения делегата или даже указывать типы его параметров.

Более того, в C# 3 можно легко вывести названия товаров по порядку, не модифицируя исходный список товаров. В листинге 1.9 показано, как это делается с помощью метода `OrderBy()`.

Листинг 1.9. Упорядочение `List<Product>` с применением расширяющего метода (C# 3)

```
List<Product> products = Product.GetSampleProducts();
foreach (Product product in products.OrderBy(p => p.Name) )
{
  Console.WriteLine (product);
}
```

В этом листинге код выглядит так, будто производится вызов метода `OrderBy()` на списке, но если заглянуть в документацию MSDN, можно выяснить, что такой метод в классе `List<Product>` не существует. Обращение к нему становится возможным благодаря наличию *расширяющего метода*, механизм которого более подробно рассматривается в главе 10. На самом деле список больше не сортируется "на месте", а производится извлечение его элементов в определенном порядке. Иногда требуется изменять действительный список, но временами упорядочение без побочных эффектов оказывается предпочтительнее.

Важными характеристиками этого кода являются компактность и читабельность (разумеется, при условии, что понятен синтаксис). Нам необходим список, упорядоченный по названию, и это в точности то, что выражает код. Он не указывает на необходимость сортировки путем сравнения названия одного товара с названием другого товара, как это делалось в коде C# 2, или сортировки с использованием экземпляра другого типа, которому известен способ сравнения товаров друг с другом. Код просто обеспечивает упорядочивание по названию. Такая простота в смысле выразительности является одним из ключевых преимуществ C# 3. Когда отдельные порции запрашиваются и обрабатываются настолько просто, крупные трансформации могут оставаться компактными и читабельными в рамках одного фрагмента кода. Это, в свою очередь, содействует взгляду на мир, более ориентированному на данные.

В этом разделе была продемонстрирована дополнительная мощь C# 2 и C# 3, с множеством пока еще необъясненного синтаксиса, но даже без понимания деталей можно заметить продвижение в направлении более ясного и простого кода. Такая эволюция отражена на рис. 1.2.

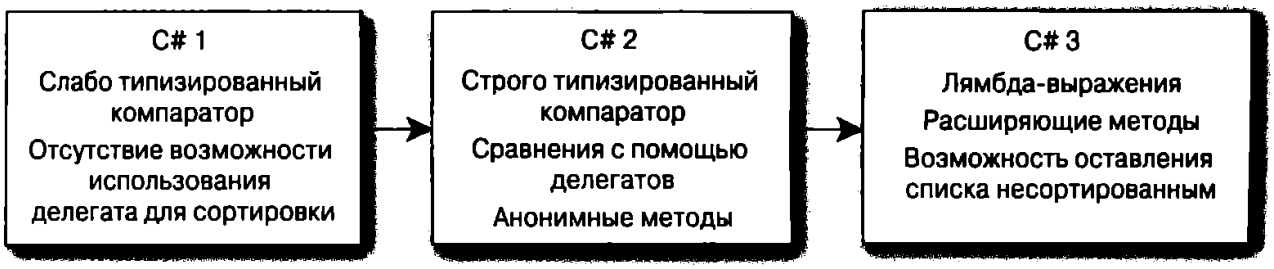

Рис. 1.2. Средства, позволяющие упростить сортировку в C# 2 и C# 3

Это все, что можно было сказать о сортировке[2]. Давайте перейдем к другой форме манипулирования данными — выполнению запросов.

1.2.2. Запрашивание коллекций

Следующая задача заключается в нахождении всех элементов списка, которые соответствуют определенному критерию — в данном случае с ценой выше $10. Как показано в листинге 1.10, в C# 1 необходимо организовать цикл по списку с проверкой каждого элемента и его выводом, если он подходит.

[2] В C# 4 предлагается средство, имеющее отношение к сортировке, которое называется *обобщенной вариантностью*, но рассмотрение примера потребовало бы слишком длинных объяснений. Соответствующие подробности приведены в конце главы 13.

Листинг 1.10. Цикл, проверка и вывод (C# 1)

```
ArrayList products = Product.GetSampleProducts();
foreach (Product product in products)
{
  if (product.Price > 10m)
  {
    Console.WriteLine(product);
  }
}
```

Понять этот код *несложно*. Однако полезно помнить о том, насколько тесно переплетены эти три задачи — организация цикла с помощью `foreach`, проверка соответствия критерию посредством `if` и отображение товара с применением `Console.WriteLine()`. Зависимости между ними очевидны из-за их вложенности друг в друга.

В листинге 1.11 видно, что C# 2 позволяет несколько выправить ситуацию.

Листинг 1.11. Отделение проверки от вывода (C# 2)

```
List<Product> products = Product.GetSampleProducts();

Predicate<Product> test = delegate(Product p) { return p.Price > 10m; };
List<Product> matches = products.FindAll(test);

Action<Product> print = Console.WriteLine;
matches.ForEach(print);
```

Переменная `test` инициализируется с использованием средства анонимных методов, которое использовалось в предыдущем разделе. Переменная `print` инициализируется с применением другого средства, появившегося в C# 2, которое называется *преобразованиями групп методов* и упрощает создание делегатов из существующих методов.

Я не буду утверждать, что этот код проще, чем код C# 1, но он определенно намного мощнее[3].

В частности, прием разделения двух видов ответственности вроде этого делает *очень* простым изменение проверяемого условия и предпринимаемого действия независимым образом. Необходимые переменные делегатов (`test` и `print`) можно было бы передавать методу, и этот же метод мог бы в конечном итоге проверять совершенно отличающиеся условия и выполнять абсолютно разные действия. Естественно, всю проверку и вывод можно поместить в один оператор, что и сделано в листинге 1.12.

Листинг 1.12. Проверка и вывод в одном операторе (C# 2)

```
List<Product> products = Product.GetSampleProducts();
products.FindAll(delegate(Product p) { return p.Price > 10;})
        .ForEach(Console.WriteLine);
```

В определенных отношениях эта версия лучше, но `delegate(Product p)` создает помехи своими фигурными скобками. Они способствуют беспорядку в коде, что вредит его читабельности. В случаях, когда нужно применять одну и ту же проверку и то же

[3] В некотором смысле это не совсем так. В C# 1 можно было бы определить соответствующие делегаты и обращаться к ним внутри цикла. Методы `FindAll()` и `ForEach()` в .NET 2.0 просто содействуют соблюдению принципа разделения ответственности.

самое действие, я отдаю предпочтение версии C# 1. (Хоть это очевидно, но полезно помнить, что код C# 1 можно использовать и с компиляторами последующих версий. Вряд ли стоит заводить бульдозер для посадки топинамбура, а в последнем листинге примерно такое излишество и было допущено.)

В листинге 1.13 можно видеть, что C# 3 существенно улучшает ситуацию, устраняя незначительные аспекты, которые окружают *логику* делегата.

Листинг 1.13. Выполнение проверки с помощью лямбда-выражения (C# 3)

```
List<Product> products = Product.GetSampleProducts();
foreach (Product product in products.Where(p => p.Price > 10))
{
  Console.WriteLine(product);
}
```

Сочетание лямбда-выражения, содержащего проверку, и хорошо именованного метода дает возможность *буквально* читать код вслух и понимать его без раздумываний. По-прежнему сохраняется гибкость C# 2 — аргумент метода `Where()` может поступать из переменной, и при желании можно применять тип `Action<Product>` вместо жестко закодированного вызова `Console.WriteLine()`.

Эта задача подчеркнула то, что уже известно из задачи с сортировкой: анонимные методы упрощают написание делегатов, а лямбда-выражения еще больше сокращают код. В обоих случаях такая краткость означает, что операцию запроса или сортировки можно поместить внутрь первой части цикла `foreach`, не теряя ясности кода.

На рис. 1.3 приведена сводка по рассмотренным выше изменениям. В версии C# 4 не предлагается ничего такого, что могло бы дополнительно упростить решение этой задачи.

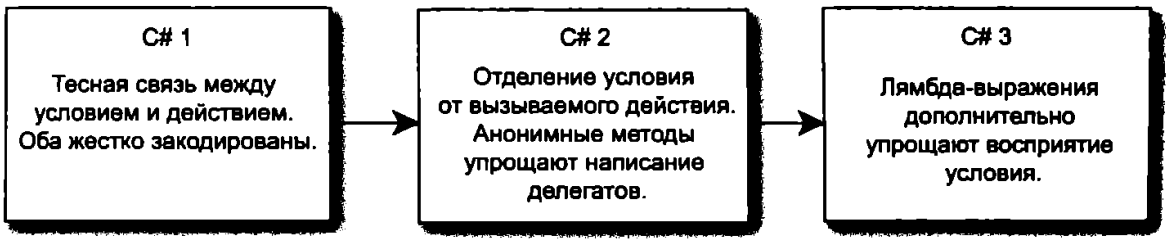

Рис. 1.3. Анонимные методы и лямбда-выражения в C# 2 и C# 3 способствуют разделению ответственности и улучшению читабельности кода

Теперь, когда был отображен отсортированный список, давайте подумаем об изменении первоначальных предположений относительно данных. Что произойдет, если цена товара не всегда известна? Как решить эту проблему в классе `Product`?

1.3. Обработка отсутствия данных

Мы рассмотрим две формы отсутствия данных. Сначала мы обсудим сценарий, при котором информация просто не доступна, а затем посмотрим, как *устранять* информацию из вызовов методов, используя стандартные значения.

1.3.1. Представление неизвестной цены

В этот раз кода будет немного, но я уверен, что проблема вам знакома, особенно если вы долго имели дело с базами данных. Представьте, что список товаров содержит не только товары, продаваемые в настоящий момент, но и те, которые еще не поступили. В ряде случаев их цена может быть неизвестна. Если бы тип `decimal` был ссылочным, то для представления неизвестной цены можно было бы применить `null`, но поскольку `decimal` является типом значения, так поступать нельзя. Как тогда представить неизвестную цену в C# 1?

Существуют три распространенных альтернативы.

- Создание оболочки ссылочного типа для типа `decimal`.
- Поддержка отдельного булевского флага, который будет указывать, известна ли цена.
- Использование для представления неизвестной цены "магического значения" (например, `decimal.MinValue`).

Надеюсь, что вы согласитесь с тем, что ни одна из указанных альтернатив не выглядит особо привлекательной. Фокус в том, что проблему можно решить путем добавления единственного символа к объявлениям переменных и свойств. В .NET 2.0 решение упростилось за счет появления структуры `Nullable<T>`, и в C# 2 предлагается дополнительный "синтаксический сахар", который позволяет изменить объявление свойства следующим образом:

```
decimal? price;
public decimal? Price
{
  get { return price; }
  private set { price = value; }
}
```

Тип параметра конструктора изменен на `decimal?`, в результате чего в качестве аргумента можно передавать значение `null`, а также применять в коде класса оператор вроде `Price = null;`. Смысл значения `null` изменяется со "специальной ссылки, которая не указывает на какой-либо объект" на "специальное значение любого типа, допускающего `null`, которое позволяет представить отсутствие полезных данных", при этом все ссылочные типы и все типы, основанные на `Nullable<T>`, считаются *типами, допускающими null*.

Код получается намного более выразительным, чем при любом другом решении. Остальная часть кода функционирует без изменений — товар с неизвестной ценой будет трактоваться как дешевле, чем $10, из-за особенностей обработки значений типов, допускающих `null`, в операциях сравнения "больше". Для проверки, известна ли цена, можно сравнить ее с `null` или воспользоваться свойством `HasValue`, поэтому вывод всех товаров с неизвестными ценами в C# 3 осуществляется с помощью кода, приведенного в листинге 1.14.

Листинг 1.14. Отображение товаров с неизвестными ценами (C# 3)

```
List<Product> products = Product.GetSampleProducts();
foreach (Product product in products.Where(p => p.Price == null))
{
  Console.WriteLine(product.Name);
}
```

Код C# 2 будет похож на код из листинга 1.12, но только придется выполнить проверку на предмет null в анонимном методе:

```
List<Product> products = Product.GetSampleProducts();
products.FindAll(delegate(Product p) { return p.Price == null; })
        .ForEach(Console.WriteLine);
```

Версия C# 3 не предлагает в данном случае каких-либо изменений, но в C# 4 реализовано средство, которое имеет к этому отношение, по крайней мере, косвенное.

1.3.2. Необязательные параметры и стандартные значения

Иногда указывать все необходимые значения для метода нежелательно, например, когда отдельный параметр почти всегда имеет одно и то же значение. Традиционное решение предусматривало перегрузку такого метода, однако в C# 4 появились *необязательные параметры*, которые позволяют поступать проще.

В версии типа Product на C# 4 имеется конструктор, принимающий название и цену товара. Цене можно назначить десятичный тип, допускающий null, как в C# 2 и C# 3, но давайте предположим, что цены *большинства* товаров не известны. Было бы неплохо иметь возможность инициализировать товар примерно так:

```
Product p = new Product("Unreleased product");
```

До появления C# 4 для этой цели в классе Product пришлось бы определить новую перегруженную версию конструктора. Версия C# 4 позволяет объявить для параметра price стандартное значение (в этом случае null):

```
public Product(string name, decimal? price = null)
{
  this.name = name;
  this.price = price;
}
```

При объявлении необязательного параметра всегда должно указываться константное значение. Оно не обязательно должно быть равно null; просто в данной ситуации null оказалось подходящим стандартным значением. Требование о том, что стандартное значение является константой, применимо к параметру любого типа, хотя для ссылочных типов, отличных от строк, вы ограничены null как единственным доступным константным значением.

На рис. 1.4 представлена сводка по рассмотренной ранее эволюции обработки отсутствующих данных в разных версиях C#.

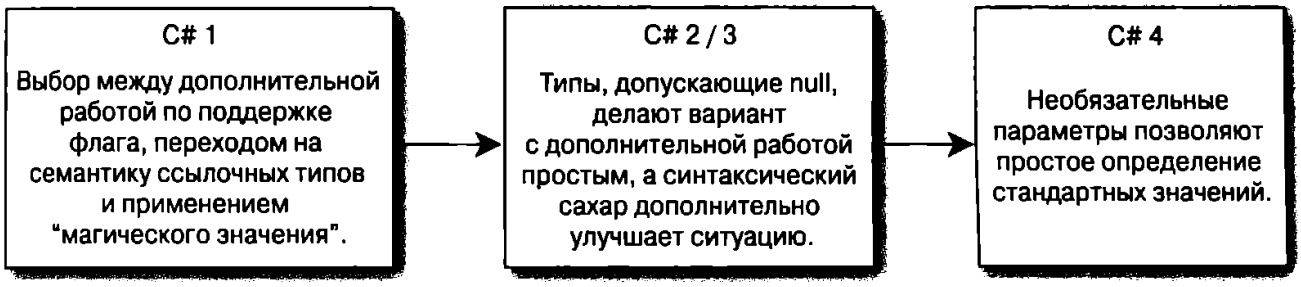

Рис. 1.4. Доступные способы работы с отсутствующими данными

Рассмотренные до сих пор средства полезны, но ничего особенного собой не представляют. А теперь перейдем к кое-чему более интересному: LINQ.

1.4. Введение в LINQ

Язык интегрированных запросов (Language Integrated Query — LINQ) является центральной частью изменений, внесенных в C# 3. Как должно быть понятно по названию, язык LINQ предназначен для запросов — его цель заключается в том, чтобы обеспечить простоту написания запросов ко многим источникам данных с согласованным синтаксисом и возможностями, а также в пригодном для чтения и компоновки стиле.

В то время как средства C# 2 в большей степени направлены на решение проблем версии C# 1, а не на что-то сенсационное, в C# 3 почти все построено в пользу LINQ, и результат оказывается довольно специфическим. Мне приходилось видеть в других языках средства, которые охватывают *некоторые* области, покрываемые LINQ, но ни одно из них не было настолько всесторонним и гибким.

1.4.1. Выражения запросов и внутренние запросы

Если вы сталкивались с кодом LINQ ранее, то возможно видели *выражения запросов*, которые позволяют использовать декларативный стиль для создания запросов к разнообразным источникам данных. Причина, по которой выражения запросов не применялись до сих пор в этой главе, связана с тем, что *без* использования дополнительного синтаксиса все рассмотренные примеры были проще. Это не говорит о том, что LINQ вообще нельзя было применять — например, код в листинге 1.15 реализует функциональность, эквивалентную полученной в листинге 1.13.

Листинг 1.15. Первые действия с выражениями запросов: фильтрация коллекции

```
List<Product> products = Product.GetSampleProducts();
var filtered = from Product p in products
               where p.Price > 10
               select p;
foreach (Product product in filtered)
{
  Console.WriteLine(product);
}
```

Лично я нахожу ранее приведенные листинги более легкими для восприятия — единственное преимущество этого выражения запроса связано с упрощением конструкции `where`. Здесь присутствует одно дополнительное средство — *неявно типизированные локальные переменные*, которые объявлены с использованием контекстного ключевого слова `var`. Это позволяет компилятору выводить тип переменной на основе значения, которое ей первоначально было присвоено; в данном случае типом `filtered` станет `IEnumerable<Product>`. Ключевое слово `var` будет довольно часто применяться в остальных примерах этой главы; это особенно удобно в книгах, где экономия пространства, занимаемого листингами, в большом почете.

Но если выражения запросов настолько плохи, то почему вокруг них и LINQ в целом поднят такой шум? Хотя выражения запросов не особенно полезны для решения простых задач, они *очень* эффективны в более сложных ситуациях, когда код, основанный на эквивалентных вызовах методов, было бы трудно читать (особенно в версии C#

1 или C# 2). Давайте несколько усложним задачу, добавив еще один тип Supplier, который представляет поставщика.

Каждый поставщик имеет название Name (типа string) и идентификатор SupplierID (типа int). Кроме того, к типу Product добавлено свойство SupplierID и должным образом скорректирован пример данных. Надо признать, что способ назначения каждому товару поставщика нельзя назвать объектно-ориентированным — он намного ближе к тому, как данные будут представлены в базе данных. Это упрощает демонстрацию рассматриваемого средства в настоящий момент, но в главе 12 будет показано, что LINQ позволяет использовать также и более естественную модель.

Теперь давайте взглянем на код в листинге 1.16, который выполняет соединение примеров товаров с примерами поставщиков (очевидно на основе идентификатора поставщика), применяет к товарам тот же самый фильтр по цене, что и ранее, сортирует по названию поставщика, а затем по наименованию товара и, наконец, выводит названия поставщика и товара для каждого соответствия. Это весьма приличный объем работы, который в ранних версиях C# реализовать было совсем нелегко. Однако в LINQ решение довольно тривиально.

Листинг 1.16. Соединение, фильтрация, упорядочение и проецирование (C# 3)

```
List<Product> products = Product.GetSampleProducts();
List<Supplier> suppliers = Supplier.GetSampleSuppliers();
var filtered = from p in products
               join s in suppliers
                 on p.SupplierID equals s.SupplierID
               where p.Price > 10
               orderby s.Name, p.Name
               select new { SupplierName = s.Name, ProductName = p.Name };
foreach (var v in filtered)
{
  Console.WriteLine("Supplier={0}; Product={1}",
                    v.SupplierName, v.ProductName);
}
```

Вы можете обнаружить, что код в листинге 1.16 удивительно похож на SQL. И действительно, первая реакция многих разработчиков на язык LINQ (до его внимательного исследования) — его откладывание в сторону как попытки простого встраивания возможностей SQL внутрь языка ради взаимодействия с базами данных. К счастью, хотя язык LINQ позаимствовал синтаксис и некоторые идеи из SQL, работа с ним не требует наличия базы данных. Показанный до сих пор код вообще не касается базы данных. На самом деле можно было бы запрашивать данные из любых источников, например, XML.

1.4.2. Запрашивание файла XML

Предположим, что вместо жесткого кодирования поставщиков и товаров используется следующий файл XML:

```
<?xml version="1.0"?>
<Data>
  <Products>
    <Product Name="West Side Story" Price="9.99" SupplierID="1" />
    <Product Name="Assassins" Price="14.99" SupplierID="2" />
```

```
    <Product Name="Frogs" Price="13.99" SupplierID="1" />
    <Product Name="Sweeney Todd" Price="10.99" SupplierID="3" />
  </Products>
  <Suppliers>
    <Supplier Name="Solely Sondheim" SupplierID="1" />
    <Supplier Name="CD-by-CD-by-Sondheim" SupplierID="2" />
    <Supplier Name="Barbershop CDs" SupplierID="3" />
  </Suppliers>
</Data>
```

Файл довольно прост, но как лучше всего извлекать из него данные? Каким образом его запрашивать? Как выполнять на нем соединение? Несомненно, это должно быть сложнее того, что делалось в листинге 1.16, правильно? В листинге 1.17 можно оценить объем работы, необходимой для выполнения запроса с помощью LINQ to XML.

Листинг 1.17. Сложная обработка файла XML с помощью LINQ to XML (C# 3)

```
XDocument doc = XDocument.Load("data.xml");
var filtered = from p in doc.Descendants("Product")
               join s in doc.Descendants("Supplier")
                 on (int)p.Attribute("SupplierID")
                 equals (int)s.Attribute("SupplierID")
               where (decimal)p.Attribute("Price") > 10
               orderby (string)s.Attribute("Name"),
                       (string)p.Attribute("Name")
               select new
               {
                   SupplierName = (string)s.Attribute("Name"),
                   ProductName = (string)p.Attribute("Name")
               };
foreach (var v in filtered)
{
   Console.WriteLine("Supplier={0}; Product={1}",
                     v.SupplierName, v.ProductName);
}
```

Подход не настолько прямолинеен, т.к. системе необходимо сообщить о том, каким образом должны восприниматься данные (в плане того, какие атрибуты в качестве каких типов должны применяться), но об этом речь пойдет чуть позже. В частности, существует очевидная связь между частями двух листингов. Не будь ограничений на длину печатной строки в книге, вы бы легко заметили построчное соответствие между этими двумя запросами.

Все еще под впечатлением? Или недостаточно убедительно? Давайте теперь поместим данные туда, где они с большой вероятностью должны находиться — в базу данных.

1.4.3. LINQ to SQL

Чтобы сообщить LINQ to SQL сведения о том, какие данные ожидать в таблицах, понадобится проделать определенную работу, но она довольно прямолинейна и по большей части может быть автоматизирована. Мы перейдем прямо к коду запроса, который показан в листинге 1.18. Определение класса `LinqDemoDataContext` можно найти в загружаемом исходном коде.

Листинг 1.18. Применение выражения запроса к базе данных SQL (C# 3)

```
using (LinqDemoDataContext db = new LinqDemoDataContext())
{
  var filtered = from p in db.Products
                 join s in db.Suppliers
                   on p.SupplierID equals s.SupplierID
                 where p.Price > 10
                 orderby s.Name, p.Name
                 select new { SupplierName = s.Name, ProductName = p.Name };
  foreach (var v in filtered)
  {
    Console.WriteLine("Supplier={0}; Product={1}",
                      v.SupplierName, v.ProductName);
  }
}
```

Теперь код должен выглядеть очень знакомо. Все, что находится ниже строки с конструкцией `join`, было без изменений скопировано из листинга 1.16.

Несмотря на впечатляющий вид, возникает вопрос, касающийся производительности: зачем извлекать все данные из базы и затем применять к ним указанные запросы и упорядочение .NET? Почему бы ни поручить эту работу базе данных? Ведь это то, что база данных умеет хорошо делать, не так ли? Безусловно, это так — но точно такую же работу выполняет и LINQ to SQL. Код в листинге 1.18 выдает запрос к базе данных, который по существу является запросом, транслируемым в SQL. Хотя запрос *выражен* в коде C#, он *выполняется* как запрос SQL.

Позже вы увидите, что для реализации такого соединения существует подход, в большей степени ориентированный на отношения, когда схеме и сущностям известно отношение между поставщиками и товарами. Тем не менее, результат будет тем же самым, что просто демонстрирует сходные черты LINQ to Objects (язык LINQ, работающий с коллекциями в памяти) и LINQ to SQL.

Язык LINQ обладает исключительной гибкостью — можно построить собственный поставщик для взаимодействия с веб-службой или транслировать запрос в какое-то специализированное представление. В главе 13 будет показано, насколько в действительности широко понятие LINQ и как оно может выйти за рамки запрашивания коллекций.

1.5. COM и динамическая типизация

Следующими будут рассматриваться некоторые средства, специфичные для версии C# 4. В то время как язык LINQ был основным центром внимания C# 3, способность к взаимодействию является одной из главных тем в C# 4. Она включает работу как со старой технологией COM, так и с динамическими языками, которые поддерживаются *исполняющей средой динамического языка* (Dynamic Language Runtime — DLR). Начнем с экспорта списка товаров в электронную таблицу Excel.

1.5.1. Упрощение взаимодействия с COM

Обеспечить доступность данных в Excel можно разными способами, но использование COM для управления этим предоставляет самую высокую эффективность и гиб-

кость. К сожалению, в предшествующих версиях C# работать с COM было довольно-таки трудно; даже язык VB обладал гораздо лучшей поддержкой такого рода. В C# 4 эта ситуация в основном была исправлена.

В листинге 1.19 приведен код для сохранения данных в новой электронной таблице.

Листинг 1.19. Сохранение данных в таблице Excel с помощью COM (C# 4)

```
var app = new Application { Visible = false };
Workbook workbook = app.Workbooks.Add();
Worksheet worksheet = app.ActiveSheet;
int row = 1;
foreach (var product in Product.GetSampleProducts()
                                .Where(p => p.Price != null))
{
  worksheet.Cells[row, 1].Value = product.Name;
  worksheet.Cells[row, 2].Value = product.Price;
  row++;
}
workbook.SaveAs(Filename: "demo.xls",
                FileFormat: XlFileFormat.xlWorkbookNormal);
app.Application.Quit();
```

Хотя код может выглядеть не настолько хорошо, как хотелось бы, он намного лучше, чем в случае применения более ранних версий C#. На самом деле вам уже известны некоторые продемонстрированные здесь средства C# 4, но есть и несколько других, менее очевидных средств. Ниже представлен их список.

- В вызове метода `SaveAs()` применяются именованные аргументы.
- В ряде вызовов не указаны аргументы для необязательных параметров — например, методу `SaveAs()` обычно можно передавать до 10 дополнительных аргументов!
- В C# 4 допускается встраивание нужных частей основной сборки взаимодействия (primary interop assembly — PIA) в вызывающий код, поэтому отпадает необходимость в отдельном развертывании сборки PIA.
- В C# 3 присваивание переменной `worksheet` не прошло бы без приведения, поскольку типом свойства `ActiveSheet` является `object`. В случае использования средства встраивания сборки PIA типом `ActiveSheet` становится `dynamic`, который открывает совершенно иные возможности.

Кроме того, при работе с COM в C# 4 поддерживаются именованные индексаторы — средство, которое в этом примере не применялось.

Давайте теперь рассмотрим последнее средство — динамическую типизацию в C# с использованием типа `dynamic`.

1.5.2. Взаимодействие с динамическим языком

Динамическая типизация является настолько обширной темой, что для ее освещения выделена целая глава 14. Здесь будет показан лишь один небольшой пример того, что она может делать.

Предположим, что информация о товарах хранится не в базе данных, XML-файле или памяти. Она доступна через какую-то веб-службу, но для обращения к ней имеется только код на языке Python. В этом коде благодаря динамической природе Python

выполняется построение результатов без объявления типа, содержащего все свойства, которые необходимы для доступа к каждому элементу внутри результатов. Вместо этого результаты позволяют вам запрашивать любое свойство и выясняют, что вы имели в виду, во время выполнения. В языке вроде Python такая ситуация вполне обычна. Но как получить доступ к результатам из кода C#?

Ответом является `dynamic` — новый тип[4], с которым компилятор C# разрешает работать динамически. Когда выражение имеет тип `dynamic`, можно вызывать на нем методы, обращаться к свойствам, передавать его в качестве аргумента методам и т.д. — большая часть нормального процесса привязки происходит во время выполнения, а не на этапе компиляции. Вдобавок значение `dynamic` можно неявно преобразовывать в любой другой тип (вот почему сработало приведение переменной `worksheet` в листинге 1.19) и предпринимать другие интересные действия.

Это поведение может также оказаться удобным даже в рамках чистого кода C#, в котором взаимодействие с COM не применяется, но нужно работать с другими языками. В листинге 1.20 показано, каким образом получить список товаров от IronPython и вывести его. Здесь также присутствует весь код настройки для запуска кода Python внутри того же самого процесса.

Листинг 1.20. Запуск IronPython и динамическое извлечение свойств (C# 4)

```
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.ExecuteFile("FindProducts.py");
dynamic products = scope.GetVariable("products");
foreach (dynamic product in products)
{
  Console.WriteLine("{0}: {1}", product.ProductName, product.Price);
}
```

Переменные `products` и `product` объявлены как `dynamic`, поэтому компилятор позволит пройти по списку товаров и вывести значения свойств, хотя ему не известно, будет ли это работать. Если допустить опечатку, к примеру, указав `product.Name` вместо `product.ProductName`, то она обнаружится только во время выполнения.

Это совершенно противоположно остальным частям C#, которые являются статически типизированными. Однако динамическая типизация вступает в игру, только когда присутствуют выражения с типом `dynamic`; большая часть кода C#, скорее всего, останется статически типизированной.

1.6. Более простое написание асинхронного кода

Наконец, давайте рассмотрим крупное средство версии C# 5 — асинхронные функции, которые позволяют приостанавливать выполнение кода, не блокируя поток.

Это по-настоящему обширная тема, и здесь будет описан только небольшой ее фрагмент. Как вам уже должно быть известно, в отношении многопоточности инфраструктуры Windows Forms существуют два золотых правила: вы не должны блокировать поток пользовательского интерфейса и не должны получать доступ к элементам пользовательского интерфейса из любого другого потока, за исключением нескольких хорошо определенных способов.

[4] Во всяком случае, отчасти. С точки зрения компилятора C# это тип, но среде CLR о нем ничего не известно.

В листинге 1.21 приведен единственный метод, который обрабатывает щелчок на кнопке в приложении Windows Forms и отображает информацию о товаре с заданным идентификатором.

Листинг 1.21. Отображение информации о товарах в приложении Windows Forms с использованием асинхронной функции

```
private async void CheckProduct(object sender, EventArgs e)
{
  try
  {
    productCheckButton.Enabled = false;
    string id = idInput.Text;

    Task<Product> productLookup = directory.LookupProductAsync(id);
    Task<int> stockLookup = warehouse.LookupStockLevelAsync(id);
    Product product = await productLookup;
    if (product == null)
    {
      return;
    }
    nameValue.Text = product.Name;
    priceValue.Text = product.Price.ToString("c");

    int stock = await stockLookup;
    stockValue.Text = stock.ToString();
  }
  finally
  {
    productCheckButton.Enabled = true;
  }
}
```

Полный код метода несколько длиннее, чем показано в листинге 1.21, т.к. он дополнительно отображает сообщения о состоянии и очищает результаты в самом начале, но этот листинг содержит все важные части. Несколько фрагментов выделены полужирным — метод имеет новый модификатор `async` и определены два выражения `await`.

Игнорирование указанных аспектов в данный момент даст возможность понять общее направление кода. Он начинается с просмотра каталога товаров и склада в поисках информации о товаре и текущем его запасе. Затем метод ожидает до тех пор, пока не получит информацию о товаре, и завершается, если каталог не имеет записи для заданного идентификатора. В противном случае он заполняет элементы пользовательского интерфейса, предназначенные для названия и цены, и после этого ожидает сведений о запасе товара на складе, чтобы также отобразить их.

Просмотры каталога товаров и склада являются асинхронными, однако они могли бы быть операциями в базе данных или обращениями к веб-службам. Все это не имеет значения — во время ожидания результатов поток пользовательского интерфейса в действительности не блокируется, хотя код внутри метода *выполняется* в этом потоке. После возвращения результатов метод продолжает работу с того места, где он был приостановлен. Этот пример также демонстрирует, что нормальное управление потоком выполнения (`try/finally`) действует в точности так, как ожидалось. Что действительно удивляет в этом методе — это достижение им нужного вида асинхроннос-

ти без традиционной возни с запуском других потоков либо созданием экземпляров `BackgroundWorker`, вызовом метода `Control.BeginInvoke()` или присоединением обратных вызовов к асинхронным событиям. Конечно, аспектов для обдумывания по-прежнему немало — асинхронность не стала *легкой* из-за применения ключевых слов `async/await`, но оказалась менее утомительной и теперь требует написания намного меньшего объема стереотипного кода, отвлекающего от внутренней сложности, которую вы пытаетесь контролировать.

Еще не закружилась голова? Расслабьтесь, дальше изложение пойдет не в таком высоком темпе. В частности, будут объяснены краевые случаи, предоставлены дополнительные сведения о том, *почему* появились разнообразные средства, и предложены некоторые указания относительно того, где их уместно использовать.

До сих пор были показаны средства языка C#. Часть этих средств требует библиотечной поддержки, а часть — поддержки времени выполнения. Давайте проясним, что имеется в виду.

1.7. Разделение платформы .NET

Сразу после своего появления понятие .NET применялось в качестве обобщенного термина для обозначения широкого спектра технологий, предлагаемых Microsoft. Например, идентификатор Windows Live ID назывался *паспортом* .NET (.NET Passport), несмотря на отсутствие четкой связи между ним и тем, что в текущий момент известно как .NET. К счастью, с тех пор все несколько упорядочилось. В этом разделе мы рассмотрим различные части .NET.

Во многих местах этой книги я ссылаюсь на три вида средств: средства C# как *языка*, средства исполняющей среды, предоставляющей своего рода "механизм", и средства библиотек инфраструктуры .NET. Внимание в этой книге в основном сосредоточено на языке C#, а средства исполняющей среды и библиотек инфраструктуры будут обсуждаться, только когда они имеют отношение к самому языку C#. Средства часто будут пересекаться, но важно понимать, где проходят их границы.

1.7.1. Язык C#

Язык C# определен своей спецификацией, которая описывает формат исходного кода C#, включая синтаксис и поведение. В ней не затрагивается платформа, на которой будут выполняться результаты компиляции, кроме нескольких ключевых точек взаимодействия. Например, языку C# требуется тип по имени `System.IDisposable`, который содержит метод под названием `Dispose()`. Это необходимо для определения оператора `using`. Аналогично, платформе нужна возможность поддержки (в той или иной форме) типов значений и ссылочных типов наряду со сборкой мусора.

Теоретически может существовать компилятор C# для любой целевой платформы, которая поддерживает обязательные средства. К примеру, компилятор C# может вполне законно генерировать вывод в форме, отличной от *промежуточного языка* (Intermediate Language — IL), которая является общепринятой на момент написания этой книги. Исполняющая среда будет интерпретировать вывод из компилятора C# или преобразовывать его полностью в машинный код за один шаг, не выполняя JIT-компиляцию. Хотя такие варианты встречаются сравнительно редко, они все-таки существуют; например, в Micro Framework используется интерпретатор, такой как Mono (`http://mono-project.net`).

С другой стороны, полная компиляция применяется в NGen и Xamarin.iOS (`http://xamarin.com/ios`) — платформе для построения приложений, ориентированных на iPhone и другие устройства с операционной системой iOS.

1.7.2. Исполняющая среда

Аспект платформы .NET, связанный с исполняющей средой, реализован в виде относительно небольшого объема кода, который отвечает за то, что программы, представленные на IL, выполняются в соответствие со спецификацией *общеязыковой инфраструктуры* (Common Language Infrastructure — CLI), описанной в документах ECMA-335 и ISO/IEC 23271 (части I — III). Часть CLI, касающаяся исполняющей среды, называется *общеязыковой исполняющей средой* (Common Language Runtime — CLR). Ссылки на CLR в оставшихся главах книги будут относиться к ее реализации от Microsoft.

Некоторые элементы языка C# никогда не появляются на уровне исполняющей среды, но другие пересекают ее границу. Например, на уровне исполняющей среды не определены перечислители, вдобавок интерфейс `IDisposable` мало что значит здесь, однако массивы и делегаты для исполняющей среды важны.

1.7.3. Библиотеки инфраструктуры

Библиотеки предлагают код, который доступен вашим программам. Библиотеки инфраструктуры в .NET в значительной степени построены на самом языке IL с участием машинного кода только там, где это необходимо. Это является достоинством исполняющей среды: написанный вами код не рассчитывает на то, чтобы быть второстепенным классом — он может обеспечить такой же уровень мощи и производительности, как и библиотеки, которыми он пользуется. Объем кода в библиотеках намного превышает объем кода исполняющей среды, примерно как автомобиль намного больше его двигателя.

Библиотеки инфраструктуры частично стандартизированы. В части IV спецификации CLI предлагается несколько разных профилей (*компактный* (compact) и *ядро* (kernel)) и библиотек. Раздел IV состоит из двух частей — общего описания библиотек, включающего идентификацию обязательных библиотек в каждом профиле, и детальных сведений о самих библиотеках в формате XML. Именно такая форма документации генерируется в случае использования комментариев XML в коде C#.

Многие средства внутри .NET *не* находятся в библиотеках. В случае написания программы, которая работает *только* с библиотеками, причем корректно, вы обнаружите, что код безупречно функционирует под управлением любой реализации — Mono, .NET или какой-нибудь другой. Однако на практике едва ли не каждая программа любого размера будет взаимодействовать с библиотеками, которые не были стандартизированы — например, Windows Forms или ASP.NET. Проект Mono имеет собственные библиотеки, не являющиеся частью .NET, такие как GTK#, и в нем реализованы многие нестандартизированые библиотеки.

Понятие *.NET* относится к сочетанию исполняющей среды и библиотек, предоставляемых Microsoft, и также охватывает компиляторы для языков C# и VB.NET. Это можно рассматривать как полноценную *платформу разработки*, построенную на основе Windows. Для каждого аспекта .NET поддерживаются отдельные версии, что может стать источником путаницы. В приложении B приведены краткие сведения о том, когда вышла та или иная версия, и какими возможностями она обладает.

Если все это понятно, то осталось обсудить последний аспект перед тем, как погрузиться в исследования C#.

1.8. Как сделать код фантастическим

Возможно, заголовок данного раздела вводит в заблуждение. Сам по себе этот раздел *не* поможет сделать ваш код фантастическим или хотя бы сколько-нибудь похожим на "глоток свежего воздуха". Однако он служит целям практически всей этой книги — именно поэтому вы должны прочитать его сейчас. Такие вопросы затрагивались во введении, но многие читатели имеют привычку пропускать введение, переходя непосредственно к главам книги. С учетом этого следующий материал дается максимально сжато.

1.8.1. Представление полных программ в виде набора фрагментов

Одной из проблем, возникающих при написании книги по языку программирования (отличному от сценарных языков), является довольно быстрый рост размеров полных программ — тех, которые читатель может ввести, скомпилировать и запустить, не прибегая к постороннему исходному коду. Я хочу обойти эту проблему и предоставлять код, который можно было бы легко набирать и затем экспериментировать с ним, поскольку считаю, что *опробование* какого-то средства гораздо лучше, чем просто чтение о нем.

Благодаря ссылкам на сборки и правильным директивам `using`, можно многого добиться даже с небольшим объемом кода C#, но прежде чем дело дойдет до написания первой строки *полезного* кода, приходится заниматься такими малоинтересными вещами, как указание директив `using`, объявление класса и определение метода `Main()`. Мои примеры в большинстве случаев представлены в форме *фрагментов*, в которых игнорируются незначительные аспекты, препятствующие простоте программ, а основное внимание уделяется важным частям. Эти фрагменты могут напрямую запускаться с помощью построенного мною небольшого инструмента, который называется *Snippy*.

Если фрагмент не содержит многоточия (`...`), то весь код должен рассматриваться в качестве тела метода `Main()` программы. Если же многоточие присутствует, то весь код, который находится до него, считается объявлениями методов и вложенных типов, а код после многоточия относится к методу `Main()`. Например, взгляните на следующий фрагмент:

```
static string Reverse(string input)
{
  char[] chars = input.ToCharArray();
  Array.Reverse(chars);
  return new string(chars);
}
...
Console.WriteLine(Reverse("dlrow olleH"));
```

Инструмент Snippy расширит его так, как показано ниже:

```
using System;
public class Snippet
{
   static string Reverse(string input)
   {
     char[] chars = input.ToCharArray();
     Array.Reverse(chars);
     return new string(chars);
   }
   [STAThread]
   static void Main()
```

```
    {
        Console.WriteLine(Reverse("dlrow olleH"));
    }
}
```

В действительности инструмент Snippy включает гораздо больше директив `using`, но расширенная версия уже стала длиннее. Следует отметить, что содержащийся класс всегда называется `Snippet`, а все типы, объявленные внутри фрагмента, будут вложенными в этот класс.

Дополнительные сведения об использовании Snippy можно найти на веб-сайте книги (`http://csharpindepth.com/Snippy.aspx`), где также доступны все примеры в виде фрагментов и расширенных версий в рамках решений Visual Studio. Кроме того, понадобится также поддержка LINQPad (`http://www.linqpad.net`) — простого инструмента, разработанного Джозефом Албахари, который особенно удобен при исследовании LINQ.

А теперь давайте посмотрим, что не так с приведенным выше кодом.

1.8.2. Учебный код не является производственным

Было бы прекрасно, если бы вы могли взять код всех примеров и просто применить его в своих приложениях безо всякой модификации, но я настоятельно рекомендую не делать этого. Большинство примеров предназначено для демонстрации отдельных аспектов — обычно в этом и заключается их цель. Большая часть фрагментов не содержит кода проверки допустимости аргументов, модификаторов доступа, модульных тестов или документации. Вдобавок фрагменты могут привести к отказу в случае использования за пределами контекста, для которого они планировались.

Например, рассмотрим тело показанного ранее метода, который изменяет порядок следования символов в строке на противоположный. Этот код неоднократно встречается в книге:

```
char[] chars = input.ToCharArray();
Array.Reverse(chars);
return new string(chars);
```

Не принимая во внимание проверку допустимости аргументов, этот код успешно обратит последовательность кодовых позиций UTF-16 внутри строки, но в некоторых случаях недостаточно хорошо. Скажем, если отображаемый глиф образован из буквы *e*, за которой следует комбинированный символ, представляющий ударение, то последовательность кодовых позиций меняться не должна; в конечном итоге ударение окажется ошибочным символом. Или предположим, что строка содержит символ не из базовой многоязычной плоскости, а сформированный из суррогатной пары — изменение порядка следования кодовых позиций даст в результате строку, которая будет недопустимой в контексте кодировки UTF-16. Устранение указанных проблем приведет к намного более сложному коду, отвлекающему внимание от демонстрируемого аспекта.

Пользуйтесь свободно кодом из книги, но имейте в виду сказанное здесь. Будет гораздо лучше, если этот код послужит источником вдохновения, а не просто окажется дословно скопированным в расчете на то, что он удовлетворит конкретным требованиям.

В конце концов, доступны другие книги, посвященные отдельным темам.

1.8.3. Спецификация языка как лучший друг

Хотя я приложил максимум усилий, чтобы соблюсти точность излагаемого материала, было бы удивительно, если бы книга вообще не содержала ошибок. Кроме того, могут возникать вопросы, не раскрытые в настоящей книге. В конечном итоге основным источником, в котором описано поведение C#, является спецификация языка.

Существуют две важных формы спецификации — международный стандарт от ECMA и спецификация от Microsoft. На момент написания книги спецификация ECMA (ECMA-334 и ISO/IEC 23270) охватывала только версию C# 2, несмотря на свое четвертое издание. Совершенно неясно, будет ли она обновлена и когда, но версия спецификации от Microsoft уже завершена и свободно доступна. На веб-сайте этой книги имеются ссылки на все доступные версии обоих вариантов спецификации (`http://csharpindepth.com/Articles/Chapter1/Specifications.aspx`), а среда Visual Studio поставляется с собственной копией спецификации[5]. При упоминании разделов спецификации в данной книге используется нумерация из спецификации Microsoft C# 5, даже если речь идет о предшествующих версиях языка. Настоятельно рекомендуется загрузить эту версию спецификации, чтобы всегда иметь ее под рукой, когда нужно срочно проверить какой-нибудь странный краевой случай.

Одна из моих целей заключается в том, чтобы сделать спецификацию *по большей части* излишней для разработчиков, и предоставить вариант, более ориентированный на разработчиков, в котором раскрывается все, что можно встретить при повседневном кодировании, без учета деталей, требуемых при построении компиляторов. Несмотря на сказанное, спецификация вполне читабельна и не должна отпугивать вас. Если вы сочтете ее интересной, то доступны аннотированные версии для C# 3 и C# 4. Обе версии содержат полезные комментарии от членов команды разработчиков C# и других участников. (Скромно отмечу, что я и сам являюсь одним из участников версии для C# 4.)

1.9. Резюме

В настоящей главе были представлены (пока без объяснений) средства, которые более подробно рассматриваются в оставшихся главах книги. Есть еще немало того, что здесь не обсуждалось, а со многими средствами, показанными до сих пор, связаны дополнительные возможности. Будем надеяться, что материал этой главы разжег интерес к остальным частям книги.

Хотя большая часть главы была посвящена описанию средств, мы также взглянули на ряд областей, которые должны помочь в получении максимальной пользы от книги в целом. Я прояснил, что понимаю под языком, исполняющей средой и библиотеками, а также объяснил, как будет представлен в книге код примеров.

Есть еще одна область, которую необходимо раскрыть, прежде чем погружаться в детали возможностей C# 2, и это язык C# 1. Очевидно, что я как автор понятия не имею об уровне ваших знаний C# 1, однако мне известен ряд областей C#, которые часто вызывают концептуальные проблемы. Некоторые из таких областей являются критически важными для понимания последующих версий C#, поэтому в главе 2 они будут рассматриваться более подробно.

---

5 Точное местоположение спецификации зависит от системы, но в случае установки Visual Studio 2012 Professional спецификация находится в каталоге `C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC#\Specifications\1033`.

Глава 2

Язык C# как основа всех основ

В этой главе...

- Делегаты
- Характеристики системы типов
- Типы значений и ссылочные типы

Эта глава не является кратким справочником по всему языку C# 1. По правде говоря, я не смог бы отдать должное *каждой* теме языка C#, если бы пришлось его описывать целиком в единственной главе. Я писал эту книгу в расчете на то, что вы располагаете хотя бы базовыми знаниями C# 1. Разумеется, оценка уровня базовых знаний в какой-то мере субъективна, но я предполагаю, что вы, по крайней мере, способны успешно пройти собеседование при приеме на работу в качестве младшего разработчика на C#, ответив на соответствующие вопросы технического характера. Вы можете обладать и большим опытом, но это тот минимальный уровень знаний, которым необходимо располагать.

В данной главе мы сосредоточим внимание на трех областях языка C# 1, которые особенно важны для понимания средств, появившихся в последующих версиях. Это должно несколько поднять наименьший общий знаменатель, что позволит сделать немного больше предположений позже в книге. Учитывая, что это *является* наименьшим общим знаменателем, вы можете обнаружить, что хорошо понимаете все концепции, изложенные в главе. Если вы уверены, что обойдетесь без чтения этой главы, можете смело пропустить ее. Когда позже окажется, что не все так просто, вы всегда сможете вернуться к этой главе. При желании можно ознакомиться с приведенными в конце каждого раздела резюме, в которых описаны все важные аспекты, и если что-то покажется непонятным, то внимательно почитать соответствующий раздел.

Мы начнем с рассмотрения делегатов, затем сравним систему типов C# с рядом других возможностей и, наконец, выясним отличия между типами значений и ссылочными типами. Для каждой темы будут описаны лежащие в основе идеи и поведение, а также определены термины, которыми можно будет пользоваться впоследствии. После объяснения, как работает C# 1, будет представлен краткий обзор связи многих новых средств из последующих версий языка с темами, затронутыми в настоящей главе.

2.1. Делегаты

Уверен, что вы интуитивно понимаете, чем является делегат, хотя, может быть, и не в состоянии четко сформулировать это. Если вы знакомы с языком C и хотите описать делегаты другому программисту на C, то, несомненно, всплывет термин *указатель на функцию*. В сущности, делегаты предоставляют определенный уровень косвенности: вместо непосредственного указания поведения, которое должно быть выполнено, его каким-то образом "упаковывают" в объект. Такой объект затем может использоваться подобно любому другому, и одной поддерживаемой им операцией является выполнение инкапсулированного действия. В качестве альтернативы тип делегата можно считать интерфейсом с единственным методом, а экземпляр делегата — объектом класса, который реализует этот интерфейс.

Если это вам кажется абракадаброй, возможно, поможет пример. Он несколько необычен, однако охватывает все аспекты, которые характеризуют делегаты. Обдумайте свое завещание — т.е. последнюю волю. Например, это может быть набор инструкций: "оплатить счета, пожертвовать на благотворительность, оставить остальное имущество кошке". Завещание пишется *перед* смертью и оставляется в подходящем безопасном месте. *После* смерти ваш поверенный будет (как вы надеетесь) действовать согласно этим инструкциям.

Делегат в C# действует подобно завещанию в реальном мире — он позволяет указать последовательность действий для выполнения в надлежащее время. Делегаты обычно используются, когда коду, который желает выполнить действия, ничего не известно о том, какими должны быть эти действия. Например, единственной причиной, по которой класс `Thread` знает, что именно запускать в новом потоке, когда вы его стартуете, является предоставление конструктору экземпляра делегата `ThreadStart` или `ParameterizedThreadStart`.

Рассмотрение делегатов начинается с четырех абсолютных основ, без которых все остальное теряет смысл.

2.1.1. Рецепт для простых делегатов

Чтобы делегат мог что-то делать, должны быть удовлетворены четыре условия.

- Должен быть объявлен тип делегата.
- Код, предназначенный для выполнения, должен содержаться внутри метода.
- Должен быть создан экземпляр делегата.
- Экземпляр делегата должен быть вызван.

Давайте рассмотрим шаги этого рецепта по очереди.

Объявление типа делегата

Фактически *тип делегата* — это список типов параметров и возвращаемый тип. Тип делегата указывает вид действия, который может быть представлен экземплярами этого типа. Например, взгляните на тип делегата, объявленный следующим образом:

```
delegate void StringProcessor(string input);
```

Этот код говорит о том, что для создания экземпляра `StringProcessor` понадобится метод с одним параметром (типа `string`) и возвращаемым типом `void` (т.е. метод ничего не возвращает).

Важно понимать, что StringProcessor действительно является типом, производным от класса System.MulticastDelegate, который, в свою очередь, унаследован от System.Delegate. Он имеет методы, можно создавать его экземпляры, а также передавать ссылки на эти экземпляры. Очевидно, с этим связан ряд особенностей, но если вы зададитесь вопросом, что происходит в данной конкретной ситуации, то сначала подумайте о том, что было бы в случае применения обычного ссылочного типа.

Источник путаницы: неоднозначный термин *делегат*

Делегаты иногда понимаются неправильно, поскольку слово *делегат* часто используется для описания и *типа делегата*, и *экземпляра делегата*. Различие между ними точно такое же, как между другим типом и его экземплярами — например, сам по себе тип string отличается от отдельной последовательности символов. В этой главе будут повсеместно применяться термины *тип делегата* и *экземпляр делегата*, чтобы максимально прояснить, о чем идет речь в конкретный момент.

При рассмотрении следующего шага будет использоваться тип делегата StringProcessor.

Поиск подходящего метода для действия экземпляра делегата

Этот шаг предусматривает поиск (или написание) метода, который выполняет необходимое действие, а также имеет ту же самую сигнатуру, что и применяемый тип делегата. Идея в том, чтобы при вызове экземпляра делегата все используемые параметры совпадали, и была возможность обработки возвращаемого значения (если оно есть) так, как планировалось — в общем, подобно вызову обычного метода.

Рассмотрим следующие пять сигнатур методов в качестве кандидатов на применение с экземпляром StringProcessor:

```
void PrintString(string x)
void PrintInteger(int x)
void PrintTwoStrings(string x, string y)
int GetStringLength(string x)
void PrintObject(object x)
```

Первый метод удовлетворяет всем условиям, поэтому его можно использовать для создания экземпляра делегата. Второй метод принимает один параметр, но не string, так что он несовместим с типом StringProcessor. Третий метод имеет первый параметр корректного типа, но также располагает еще одним параметром, следовательно, является несовместимым. Четвертый метод принимает параметр подходящего типа, однако возвращаемый тип отличается от void. (Если в типе делегата предусмотрен возвращаемый тип, то возвращаемый тип метода также должен ему соответствовать.)

Пятый метод интересен — при любом обращении к экземпляру StringProcessor можно вызывать метод PrintObject() с теми же аргументами, т.к. тип string является производным от object. Имело бы смысл применять его для создания экземпляра StringProcessor, но в C# 1 типы параметров делегата должны совпадать *точно*[1].

В C# 2 ситуация изменилась — об этом речь пойдет в главе 5. Четвертый метод в какой-то мере подобен, поскольку нежелательное возвращаемое значение всегда можно

[1] В дополнение к типам у параметров должны также совпадать и модификаторы, т.е. in (стандартный), out или ref. Хотя параметры out и ref в делегатах используются довольно редко.

проигнорировать. Однако возвращаемые типы `void` и отличные от `void` в настоящее время всегда считаются несовместимыми. Частично это объясняется тем, что другим аспектам системы (особенно JIT) необходимо знать, оставлять ли значение в стеке в качестве возвращаемого после выполнении метода[2].

Давайте представим, что уже есть тело метода для совместимой сигнатуры (`PrintString()`), и перейдем к следующему шагу — созданию самого экземпляра делегата.

Создание экземпляра делегата

При наличии типа делегата и метода с подходящей сигнатурой можно создать экземпляр этого типа делегата, указав, что данный метод должен быть выполнен при обращении к экземпляру делегата. Официальной терминологии для такой ситуации не предусмотрено, но в настоящей книге это будет называться *действием* экземпляра делегата.

Точная форма выражения, применяемого для создания экземпляра делегата, зависит от того, какой метод использует действие — статический или метод экземпляра. Предположим, что `PrintString()` является статическим методом внутри типа по имени `StaticMethods`, а также методом экземпляра в типе под названием `InstanceMethods`. Ниже приведены два примера создания экземпляра `StringProcessor`:

```
StringProcessor proc1, proc2;
proc1 = new StringProcessor(StaticMethods.PrintString);
InstanceMethods instance = new InstanceMethods();
proc2 = new StringProcessor(instance.PrintString);
```

Когда действие выражено статическим методом, необходимо указать только имя типа, а если действие представляет собой метод экземпляра, то понадобится экземпляр этого типа (или производного от него типа), как делается обычно. Этот объект называется целью действия, и при обращении к экземпляру делегата метод будет вызван как раз на данном объекте. Если действие определено внутри того же самого класса (как это часто бывает, особенно при написании обработчиков событий в коде пользовательского интерфейса), уточнять его не придется — для методов экземпляра[3] неявно применяется ссылка `this`. Опять-таки, эти правила действуют, как если бы метод вызывался напрямую.

Потенциальная проблема сборки мусора

Следует помнить о том, что экземпляр делегата будет препятствовать уничтожению своей цели сборщиком мусора, если сам экземпляр делегата не может быть уничтожен. Это может привести к очевидным утечкам памяти, особенно когда какой-то недолговечный объект подписывается на событие, возникающее в долговечном объекте, используя самого себя в качестве цели. Долговечный объект косвенно удерживает ссылку на недолговечный объект, продлевая его существование.

---

[2] Слово *стек* здесь преднамеренно применяется нечетко во избежание привлечения не относящихся к делу деталей. За дополнительными сведениями обращайтесь к статье "The void is invariant" ("Тип `void` является инвариантным") в блоге Эрика Липперта (`http://mng.bz/4g58`).

[3] Разумеется, если действие является методом экземпляра и предпринимается попытка создать экземпляр делегата внутри статического метода, по-прежнему необходимо предоставить ссылку на цель.

Создавать экземпляр делегата не имеет смысла, если он не будет вызван в определенный момент. Давайте взглянет на последний шаг — вызов.

Вызов экземпляра делегата

Экземпляр делегата вызывается очень просто[4]: нужно лишь вызвать метод на экземпляре делегата. Сам метод называется `Invoke()`. Он всегда присутствует в типе делегата и имеет такой же список параметров и возвращаемый тип, как в объявлении типа делегата. В рассматриваемом примере метод выглядит следующим образом:

```
void Invoke(string input)
```

Вызов `Invoke()` выполнит действие экземпляра делегата, передавая ему любые аргументы, которые были указаны при вызове `Invoke()`, и вернет возвращаемое значение действия (если его типом не является `void`).

Несмотря на простоту, C# еще более упрощает дело; с переменной[5] типа делегата можно обращаться так, как если бы это был сам метод. Лучше всего представить происходящее в виде цепочки событий, возникающих в разное время, как показано на рис. 2.1.

Рис. 2.1. Обработка вызова экземпляра делегата, который использует сокращенный синтаксис C#

Как видите, все получилось легко. Все компоненты на месте, так что можно приступать к исследованию взаимодействия между ними.

Полноценный пример и изложение мотивов

Наблюдать все это в действии лучше всего в полноценном примере — наконец-то хоть что-нибудь удастся запустить! Поскольку здесь вовлечено слишком много разных мелочей, на этот раз приведен полный исходный код, а не фрагменты. В листинге 2.1 нет ничего особо удивительного — это просто конкретный код, который мы обсудим.

---

[4] Во всяком случае, это касается синхронного вызова. Чтобы вызвать экземпляр делегата асинхронно, можно воспользоваться методами `BeginInvoke()` и `EndInvoke()`, однако данная тема выходит за рамки настоящей главы.

[5] Или выражение любого другого вида, но обычно применяется переменная.

Листинг 2.1. Использование делегатов разнообразными простыми способами

```
using System;
delegate void StringProcessor(string input);      ←❶ Объявление типа делегата
class Person
{
  string name;
  public Person(string name) { this.name = name; }
  public void Say(string message)      ←❷ Объявление совместимого метода экземпляра
  {
    Console.WriteLine("{0} says: {1}", name, message);
  }
}
class Background
{
  public static void Note(string note) ←❸ Объявление совместимого статического метода
  {
    Console.WriteLine("({0})", note);
  }
}
class SimpleDelegateUse
{
  static void Main()
  {
    Person jon = new Person("Jon");
    Person tom = new Person("Tom");
    StringProcessor jonsVoice, tomsVoice, background;
    jonsVoice = new StringProcessor(jon.Say);
    tomsVoice = new StringProcessor(tom.Say);
    background = new StringProcessor(Background.Note);
    jonsVoice("Hello, son.");
    tomsVoice.Invoke("Hello, Daddy!");
    background("An airplane flies past.");
  }
}
```

❹ Создание трех экземпляров делегата

❺ Вызов экземпляров делегата

Вначале объявляется тип делегата ❶. Затем создаются два метода (❷ и ❸), совместимые с типом делегата. Один из них является методом экземпляра (`Person.Say()`), а другой — статическим методом (`Background.Note()`), что позволит продемонстрировать отличия в их применении, когда создаются экземпляры делегата ❹. В листинге 2.1 созданы два экземпляра класса `Person`, чтобы можно было увидеть разницу в целях делегата.

Обращение к `jonsVoice` ❺ приводит к вызову метода `Say()` на объекте `Person` с именем `Jon`; аналогично обращение к `tomsVoice` предусматривает использование объекта `Person` с именем `Tom`. Ради интереса в этом коде реализованы два показанных ранее способа вызова экземпляров делегата — явный вызов метода `Invoke()` и применение сокращения C#. Обычно будет использоваться сокращение.

Вывод, получаемый в результате выполнения кода в листинге 2.1, довольно очевиден:

```
Jon says: Hello, son.
Tom says: Hello, Daddy!
(An airplane flies past.)
```

По правде говоря, листинг 2.1 содержит слишком много кода, как для получения вывода, состоящего всего из трех строк. Даже если нужно работать с классами `Person` и `Background`, нет никакой реальной необходимости в применении здесь делегатов. Так в чем тогда смысл? Почему бы просто не вызывать методы напрямую? Ответ скрыт в исходном примере с поверенным, исполняющим завещание — одно лишь желание, чтобы что-то произошло, не означает, что вы окажетесь в нужное время в нужном месте, чтобы сделать это возможным. Иногда приходится выдавать инструкции — *делегировать* ответственность, что и было сделано.

Я должен подчеркнуть, что в мире программного обеспечения объекты не оставляют завещаний. Часто объект, который создал экземпляр делегата, благополучно существует и при вызове этого экземпляра. Но бывают случаи, когда необходимо предоставить код, который должен быть выполнен в такой момент, когда у вас не будет возможности (или желания) изменить код, функционирующий в данный момент. Если нужно, чтобы в результате щелчка на кнопке что-то произошло, не следует изменять код самой *кнопки*. Кнопке просто потребуется сообщить о необходимости вызова одного из заранее определенных методов, который и предпримет соответствующее действие. Это сводится к добавлению уровня косвенности — характерной черты объектно-ориентированного программирования. Как видите, это увеличивает сложность (взгляните, насколько много строк кода пришлось записать, чтобы получить такой крошечный вывод!), но также повышает гибкость.

Теперь, когда вы знаете, как работают простые делегаты, мы кратко обсудим вопрос объединения делегатов для выполнения целой группы действий вместо одного.

2.1.2. Объединение и удаление делегатов

Все рассмотренные до сих пор экземпляры делегатов имели единственное действие. В реальности все несколько сложнее: экземпляр делегата располагает списком связанных с ним действий, который называется *списком вызова*. Ответственность за создание новых экземпляров делегата путем слияния списков вызова двух экземпляров, а также за удаление списка вызова одного экземпляра из другого экземпляра возлагается на статические методы `Combine()` и `Remove()` типа `System.Delegate`.

Делегаты являются неизменяемыми

После того как экземпляр делегата создан, в нем уже ничего не может быть изменено. Это позволяет безопасно передавать ссылки на экземпляры делегата и объединять их с другими, не беспокоясь о согласованности, безопасности в отношении потоков или внешних попытках изменения их действий. Это похоже на строки, которые также являются неизменяемыми, и метод `Delegate.Combine()` подобен `String.Concat()` — оба они объединяют существующие экземпляры, чтобы сформировать новый экземпляр, никак не изменяя исходные объекты. В случае экземпляров делегата исходные списки вызова сливаются вместе. Обратите внимание, что попытка объединения `null` с экземпляром делегата приводит к тому, что значение `null` трактуется как экземпляр делегата с пустым списком вызова.

Вы редко встретите явный вызов метода `Delegate.Combine()` в коде C#, т.к. обычно используются операции + и +=. На рис. 2.2 показан процесс преобразования, в котором задействованы переменные x и y одного и того же (или совместимого) типа делегата. Все это делается компилятором C#.

Как видите, это простое преобразование позволяет писать намного более лаконичный код. Помимо возможности объединения экземпляров делегата, с помощью метода `Delegate.Remove()` можно удалять один экземпляр из другого, для чего в C# применяются вполне очевидные операции - и -=.

Метод `Delegate.Remove(source, value)` создает новый делегат, получающий список вызова `source`, из которого был удален список `value`. Если результатом оказывается пустой список вызова, возвращается `null`.

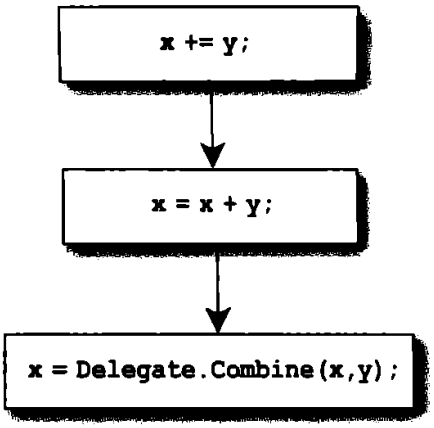

Рис. 2.2. Процесс преобразования, применяемый в отношении сокращенного синтаксиса для объединения экземпляров делегатов

После вызова экземпляра делегата все действия выполняются по очереди. Если возвращаемый тип в сигнатуре делегата отличается от `void`, то метод `Invoke()` вернет значение, которое было возвращено *последним* выполненным действием. Экземпляры делегата, возвращающего не `void`, с более чем одним действием в списке вызова встречаются редко, поскольку возвращаемые значения всех других действий не будут доступны, если только вызывающий код явно не выполняет действия по одному за раз, используя метод `Delegate.GetInvocationList()` для извлечения списка действий.

Если любое действие в списке вызова генерирует исключение, это предотвращает выполнение последующих действий. Например, когда вызывается экземпляр делегата со списком вызова `[a, b, c]`, и действие `b` генерирует исключение, это исключение распространяется немедленно, так что действие `c` выполняться не будет.

Объединение и удаление экземпляров делегатов особенно полезно при работе с событиями. Теперь, когда вы понимаете, что собой представляют объединение и удаление, можно поговорить о событиях.

2.1.3. Краткое введение в события

Возможно, вы интуитивно понимаете общее *назначение* событий, особенно если занимались построением пользовательских интерфейсов. Идея в том, что событие позволяет коду реагировать, когда что-либо происходит — например, сохранить файл по щелчку на определенной кнопке. В этом случае событием является сам щелчок на кнопке, а действием — сохранение файла. Тем не менее, понимание мотива, положенного в основу концепции — это не то же самое, что и понимание способа определения событий в терминах языка C#.

Разработчики часто путают события и экземпляры делегатов, или события и поля, объявленные с типом делегатов. Отличие очень важно: *события не являются полями*. Причина этой путаницы связана с тем, что в C# предлагается сокращение в форме *событий, подобных полям*. Вскоре мы рассмотрим их, но сначала давайте разберемся, из чего состоят события с точки зрения компилятора C#.

События удобно считать похожими на свойства. Прежде всего, они оба объявляются с определенным типом — событие должно иметь тип делегата.

Когда вы пользуетесь свойствами, это *выглядит* так, как будто вы извлекаете или присваиваете значения непосредственно полям, однако на самом деле вызываются методы (получения и установки). Реализация свойств допускает выполнение любой работы внутри этих методов. Так уж вышло, что большинство свойств реализовано с

применением простых поддерживающих полей, при этом в методах установки иногда осуществляется проверка допустимости значений, а также предпринимаются меры по обеспечению безопасности потоков.

Аналогично, подписание или отмена подписки на событие *выглядит* так, как будто в отношении поля, имеющего тип делегата, используются операции += и -=. На этот раз тоже в действительности вызываются методы (добавления и удаления) [6]. Все, что можно делать с событием — подписываться на него (добавлять обработчик события) или отменять подписку (удалять обработчик события). Внутри методов события предпринимаются какие-то полезные действия, такие как обнаружение добавляемых и удаляемых обработчиков события или обеспечение их доступности в рамках класса.

Основная причина существования событий очень похожа на причину существования свойств — они добавляют уровень инкапсуляции, реализуя шаблон "публикация/подписка" (см. мою статью "Delegates and Events" ("Делегаты и события") по адресу `http://csharpindepth.com/Articles/Chapter2/Events.aspx`). Точно так же как вы не хотите, чтобы другой код имел возможность устанавливать значения полей, не позволяя их владельцу хотя бы проверить новые значения, часто нежелательно, чтобы внешний по отношению к классу код мог произвольно изменять (или вызывать) обработчики для события. Разумеется, в класс *могут* быть добавлены методы для предоставления дополнительного доступа — скажем, для очистки списка обработчиков события или для инициирования события (другими словами, для вызова его обработчиков). Например, метод `BackgroundWorker.OnProgressChanged()` просто вызывает обработчики события `ProgressChanged`. Но если вы откроете доступ только к самому событию, то внешний код будет иметь только возможность добавления и удаления обработчиков.

События, подобные полям, существенно упрощают реализацию всего этого — понадобится единственное объявление. Компилятор превращает это объявление в событие со стандартными реализациями методов добавления/удаления и закрытое поле того же самого типа. Коду внутри класса видно это поле, а коду за пределами класса видно только событие. Это *выглядит* как возможность обращения к событию, но на самом деле для вызова обработчиков события осуществляется обращение к экземпляру делегата, ссылка на который хранится в поле.

Подробные сведения о событиях в этой главе не предоставляются — в последних версиях языка C# механизм событий не изменился [7], но здесь нужно было указать на разницу между экземплярами делегатов и событиями во избежание недоразумений в будущем.

2.1.4. Резюме по делегатам

Давайте подведем итоги того, что было рассмотрено о делегатах.

- Делегаты инкапсулируют действие с определенным возвращаемым значением и набором параметров, что похоже на интерфейс с единственным методом.
- Сигнатура типа, описанного с помощью объявления типа делегата, определяет методы, которые могут использоваться для создания экземпляров делегата, и сигнатуру для вызова.

---

[6] Имена этих методов не фиксированы, иначе в типе можно было бы иметь только одно событие. Компилятор создает два метода с именами, которые не могут применяться в другом месте, и включает специальный фрагмент метаданных, чтобы остальные типы могли знать, что существует событие с заданным именем, а также названия его методов добавления/удаления.

[7] В версии C# 4 были внесены небольшие изменения в события, подобные полям. Об этом пойдет речь в разделе 4.2.

- Создание экземпляра делегата требует метод и (для методов экземпляра) целевой объект, на котором этот метод вызывается.
- Экземпляры делегата являются неизменяемыми.
- Каждый экземпляр делегата содержит список вызова, т.е. список действий.
- Экземпляры делегата могут объединяться друг с другом и удаляться друг из друга.
- События не являются экземплярами делегатов — это всего лишь пары методов добавления/удаления (подобно средствам получения/установки свойств).

Делегаты представляют собой просто одно специфичное средство C# и .NET – мелкая деталь по большому счету. В оставшихся разделах этого главы речь пойдет о гораздо более широких темах. Первым делом мы выясним значение утверждения о том, что язык C# является *статически типизированным*, и последствия, вытекающие из этого.

2.2. Характеристики системы типов

Почти каждый язык программирования располагает системой типов какого-то вида. С течением времени системы типов классифицировались как строгие и слабые, безопасные и небезопасные, статические и динамические; встречалась также масса более экзотических вариантов. Важность понимания системы типов, с которой приходится работать, должна быть очевидной. К тому же вполне разумно ожидать, что знание категорий, к которым относится язык, предоставит много информации в этом направлении. Но поскольку разные люди применяют отличающуюся терминологию, недоразумения практически неизбежны. Чтобы свести путаницу к минимуму, для каждого термина я постараюсь объяснить *в точности*, что под ним понимаю.

Важно отметить, что материал настоящего раздела применим только к *безопасному* коду, которым является весь код C#, не помещенный явно в небезопасный контекст. Как должно быть ясно по названию, код внутри небезопасного контекста может выполнять различные действия, которые невозможно реализовать с помощью безопасного кода, и это может нарушать требования нормальной безопасности типов, хотя система типов по-прежнему безопасна во многих других отношениях. Большинству разработчиков вряд ли придется когда-либо писать небезопасный код, и если принимать во внимание только безопасный код, то характеристики системы типов становятся более простыми в описании и понимании.

В этом разделе показано, какие ограничения применяются, а какие — нет, в языке C# 1 при определении ряда терминов, представляющих указанное поведение. После этого мы рассмотрим несколько действий, которые нельзя реализовать с помощью C# 1 — сначала с точки зрения того, что *невозможно* сообщать компилятору, а затем с точки зрения того, что *не следует* сообщать компилятору.

Давайте начнем с описания того, что язык C# 1 делает, и терминологии, обычно используемой для представления такого рода поведения.

2.2.1. Место C# в мире систем типов

Начать проще с утверждения и затем прояснить, что оно означает и какими могут быть альтернативы:

Система типов C# 1 является статической, явной и безопасной.

Возможно, вы ожидали встретить в этом утверждении также и слово *строгой*, и я был не прочь его включить. Но хотя большинство людей могут вполне согласиться с тем, что язык обладает перечисленными выше характеристиками, решение о том, считать ли язык *строго типизированным*, часто вызывает горячие споры, поскольку бытующие определения существенно отличаются. Некоторые черты (предотвращение любых преобразований, явных или неявных) определенно отдаляют C# от *статически типизированного* языка, тогда как другие делают его довольно близким к этому (или даже таковым), что относится и к версии C# 1. В большинстве прочитанных мною статей и книг, в которых C# был описан как строго типизированный язык, понятие "строгая типизация" фактически использовалось для обозначения статической типизации.

Давайте последовательно пройдемся по определениям терминов и прольем на них некоторый свет.

Сравнение статической и динамической типизации

Язык C# 1 является *статически типизированным*: каждая переменная имеет отдельный тип, который известен на этапе компиляции[8]. Разрешены только операции, определенные для данного типа, и это контролируется компилятором. Рассмотрим следующий пример:

```
object o = "hello";
Console.WriteLine(o.Length);
```

Глядя на код, вполне очевидно, что значение `o` является строкой, а тип `string` имеет свойство `Length`, но компилятор считает значение `o` как относящееся к типу `object`. Если нужно обратиться к свойству `Length`, придется сообщить компилятору о том, что значение `o` ссылается на строку:

```
object o = "hello";
Console.WriteLine(((string)o).Length);
```

После этого компилятор способен найти свойство `Length` класса `System.String`. Он проверяет корректность вызова, генерирует соответствующий код IL и выясняет тип более крупного выражения. Тип выражения на этапе компиляции также известен как его статический тип, поэтому можно сказать так: "статическим типом `o` является `System.Object`".

Почему типизацию называют статической?

При описании этого вида типизации применяется слово *статическая* из-за того, что анализ доступных операций производится с использованием *неизменных* данных: типов выражений на этапе компиляции. Предположим, что переменная объявлена с типом `Stream`; этот тип переменной не изменяется, даже если значение переменной варьируется между ссылкой на `MemoryStream`, ссылкой на `FileStream` или вообще не указывает на существующий поток (посредством ссылки `null`). Даже в рамках статических систем типов может поддерживаться определенное динамическое поведение; фактическая реализация, выполняемая в результате вызова виртуального метода, будет зависеть от значения, на котором производится вызов.

[8] Это относится также и к большинству выражений, однако не ко всем. Определенные выражения не имеют типа, например, вызовы методов `void`, но это не влияет на статус C# 1 как статически типизированного языка. Чтобы избежать путаницы, в этом разделе используется слово *переменная*.

Идея неизменной информации является также мотивом, лежащим в основе модификатора `static`, но в целом проще считать статический член принадлежащим самому типу, а не какому-то отдельному экземпляру этого типа. Для большинства практических целей такие два случая использования этого слова можно рассматривать как несвязанные друг с другом.

Альтернативой статической типизации является *динамическая типизация*, которая может принимать множество форм. Сущность динамической типизации в том, что переменные просто имеют значения — они не ограничены конкретными типами, поэтому компилятор не может выполнять упомянутые ранее проверки. Взамен исполняющая среда пытается распознать выражения способами, соответствующими применяемым в них значениям. Например, если бы язык C# 1 *был* динамически типизированным, следующий код оказался бы допустимым:

```
o = "hello";
Console.WriteLine(o.Length);
o = new string[] {"hi", "there"};
Console.WriteLine(o.Length);
```

Этот код привел бы к обращению к двум совершенно не связанным друг с другом свойствам `Length` — `String.Length` и `Array.Length` — за счет динамического исследования типов во время выполнения. Подобно многим аспектам систем типов, существуют различные уровни динамической типизации. Некоторые языки позволяют указывать типы там, где необходимо (возможно, по-прежнему трактуя их динамически кроме случая присваивания), но разрешают использовать нетипизированные переменные в других местах.

Хотя в этом описании неоднократно упоминался язык C# 1, полностью статическая типизация в языке распространяется вплоть до версии C# 3 включительно. Позже вы увидите, что в версии C# 4 появилась некоторая динамическая типизация, хотя в подавляющем большинстве кода приложений на C# 4 будет применяться все та же статическая типизация.

Сравнение явной и неявной типизации

Разница между *явной* и *неявной* типизацией важна только в статически типизированных языках. При явной типизации тип каждой переменной должен быть явно указан в ее объявлении. Неявная типизация позволяет компилятору выводить тип переменной на основе ее использования. Например, язык мог бы предписывать, что типом переменной является тип выражения, которое применяется для присваивания ей начального значения.

Рассмотрим гипотетический язык, в котором для обозначения выведения типа используется ключевое слово `var`[9]. В табл. 2.1 показано, как код в таком языке можно было бы записать на C# 1. Код в левой колонке *не* разрешен в C# 1, а в правой колонке представлен эквивалентный допустимый код.

Надеюсь, теперь понятно, почему это касается только ситуаций со статической типизацией: при явной и неявной типизации тип переменной *известен* на этапе компиляции,

[9] Ладно, не настолько гипотетический. В разделе 8.2 описаны возможности неявно типизированных локальных переменных C# 3.

даже если он не указан явным образом. В динамическом контексте на этапе компиляции переменная не имеет типа, который можно было бы задать или вывести.

Таблица 2.1. Пример, демонстрирующий отличия между неявной и явной типизацией

Недопустимый код C# 1 — неявная типизация	Допустимый код C# 1 — явная типизация
`var s = "hello";`	`string s = "hello";`
`var x = s.Length;`	`int x = s.Length;`
`var twiceX = x * 2;`	`int twiceX = x * 2;`

Сравнение безопасной и небезопасной к типам системы

Простейший способ описания системы, безопасной к типам, предполагает рассмотрение ее противоположности. Некоторые языки (особенно C и C++) позволяют выполнять ряд по-настоящему удивительных действий. В подходящих ситуациях они проявляют свою мощь, но такие ситуации встречаются относительно редко. При неправильном обращении эти окольные действия могут порядком навредить. Одним из них является нарушение системы типов.

Прибегнув к определенной магии, в таких языках можно заставить трактовать значение одного типа так, как если бы оно было значением *совершенно* другого типа, не применяя никаких преобразований. Здесь не имеется в виду вызов метода, у которого оказалось то же самое имя, как было показано ранее в примере с динамической типизацией. Речь идет о коде, который просматривает низкоуровневые байты внутри значения и интерпретирует их "некорректным" способом. В листинге 2.2 приведен простой пример кода на языке C, помогающий понять сказанное.

Листинг 2.2. Демонстрация небезопасной к типам системы с помощью кода C

```
#include <stdio.h>
int main(int argc, char** argv)
{
  char *first_arg = argv[1];
  int *first_arg_as_int = (int *)first_arg;
  printf ("%d", *first_arg_as_int);
}
```

Если вы скомпилируете код листинга 2.2 и запустите его с простым аргументом `"hello"`, то увидите вывод значения `1819043176` — во всяком случае, в системе с архитектурой, имеющей порядок следования байтов от младшего к старшему, компилятором, трактующим `int` как 32 бита и `char` как 8 битов, и представлением текста с использованием кодировки ASCII или UTF-8. Код считает указатель на `char` как указатель на `int`, поэтому его разыменование возвращает первые 4 байта текста, рассматривая их как число.

На самом деле, этот крошечный пример не идет ни в какое сравнение с другими потенциальными нарушениями — выполнение приведения между совершенно несвязанными структурами очень легко может привести к полному хаосу. Нельзя сказать, что подобное происходит в реальной жизни очень часто, однако некоторые элементы системы типов языка C часто требуют от вас указания компилятору о том, что он должен делать, не оставляя ему выбора, кроме как доверять вам даже во время выполнения.

К счастью, ничего такого в C# не случается. Да, существует немало доступных преобразований, однако данные определенного типа не удастся выдать за данные другого типа. Можно *попытаться* с помощью приведения предоставить компилятору такую дополнительную (и некорректную) информацию, но если компилятор решит, что выполнить такое приведение на самом деле невозможно, он сообщит об ошибке — а если оно теоретически разрешено, но в действительности некорректно во время выполнения, то среда CLR сгенерирует исключение.

Теперь, когда вы немного узнали о том, как C# 1 вписывается в более общие рамки систем типов, имеет смысл упомянуть о некоторых недостатках, связанных с выбранными решениями. Это вовсе не говорит о том, что решения оказались неправильными — просто им присущи ограничения. Зачастую проектировщики языков должны выбирать между разными подходами, которые добавляют различные ограничения или приводят к другим нежелательным последствиям. Начнем со случая, когда вы *хотите* предоставить компилятору дополнительную информацию, но не существует способа сделать это.

2.2.2. Когда возможности системы типов C# 1 оказываются недостаточными?

Существуют две распространенных ситуации, когда может требоваться предоставление дополнительной информации коду, вызывающему метод, или, возможно, ограничение вызывающего кода относительно того, что должно предоставляться в аргументах. Первая ситуация затрагивает коллекции, а вторая — наследование и переопределение методов или реализацию интерфейсов. Мы рассмотрим их по очереди.

Строго типизированные и слабо типизированные коллекции

Несмотря на избегание применения терминов *строгая* и *слабая* в отношении системы типов C# в целом, они будут использоваться при обсуждении коллекций. В таком контексте эти термины встречаются везде с очень малыми возможностями для двусмысленности. Вообще говоря, в .NET 1.1 есть три встроенных типа коллекций:

- массивы (строго типизированные) в языке и исполняющей среде;
- слабо типизированные коллекции в пространстве имен `System.Collections`;
- строго типизированные коллекции в пространстве имен `System.Collections.Specialized`.

Массивы строго типизированы[10], поэтому на этапе компиляции невозможно установить элемент массива `string[]` в экземпляр типа `FileStream`, например. Однако массивы ссылочных типов также поддерживают *ковариантность*, которая поддерживает неявное преобразование одного типа массива в другой, а также преобразование между типами элементов. Во время выполнения происходят проверки, гарантирующие, что сохраняться будет только допустимый тип ссылки, как показано в листинге 2.3.

Листинг 2.3. Демонстрация ковариантности массивов и проверок во время выполнения

```
string[] strings = new string[5];
object[] objects = strings;      <-- ❶ Применение ковариантного преобразования
objects[0] = new Button();       <-- ❷ Попытка сохранения объекта Button
```

[10] Во всяком случае, язык позволяет им быть таковыми. Тем не менее, для слабо типизированного доступа к массивам можно использовать тип `Array`.

Если вы запустите код в листинге 2.3, то получите исключение `ArrayTypeMismatchException` ❷. Причина в том, что преобразование из `string[]` в `object[]` ❶ возвращает исходную ссылку — и `strings`, и `objects` ссылаются на тот же самый массив. Самому массиву известно, что он строковый, поэтому он будет отвергать попытки сохранения в нем ссылок на объекты, отличные от строк. Ковариантность массивов иногда полезна, однако она достается за счет обеспечения безопасности типов во время выполнения, а не на этапе компиляции.

Сравним это с ситуацией, когда применяются слабо типизированные коллекции, такие как `ArrayList` и `Hashtable`. В API-интерфейсе этих коллекций в качестве типа для ключей и значений используется `object`. При написании метода, который принимает, например, `ArrayList`, нет никаких способов удостовериться на этапе компиляции в том, что вызывающий код будет передавать список строк. Это можно документировать, и если вы приведете каждый элемент списка к типу `string`, то исполняющая среда обеспечит его, но вы не получите безопасности типов на этапе компиляции. Более того, в случае возвращения `ArrayList` в документации можно отразить тот факт, что список будет содержать только строки, однако вызывающий код будет вынужден доверять этому и предусматривать приведения при доступе к элементам списка.

Наконец, рассмотрим строго типизированные коллекции, подобные `StringCollection`. Они предоставляют строго типизированный API-интерфейс, поэтому можно с уверенностью утверждать, что коллекция `StringCollection`, применяемая в качестве параметра или возвращаемого значения, будет содержать только строки, и выполнять приведение при извлечении элементов из коллекции не понадобится. Звучит идеально, но на самом деле есть две проблемы. Во-первых, указанная коллекция реализует интерфейс `IList`, так что можно по-прежнему *пытаться* добавить в нее нестроковые элементы (хотя это потерпит неудачу во время выполнения). Во-вторых, коллекция работает только со строками. Доступны и другие специализированные коллекции, однако все они не ушли слишком далеко в данном отношении. Существует тип `CollectionBase`, который можно использовать для построения собственных строго типизированных коллекций, но это означает создание нового типа коллекции для каждого типа элемента, что также не идеально.

Теперь, когда вы ознакомились с проблемой, касающейся коллекций, давайте рассмотрим затруднение, которое может возникнуть при переопределении методов и реализации интерфейсов. Это связано с идеей ковариантности, которая уже была показана для массивов.

Отсутствие ковариантности возвращаемых типов

Интерфейс `ICloneable` относится к числу простейших в рамках инфраструктуры. Он имеет единственный метод `Clone()`, который должен возвращать копию объекта, на котором этот метод вызывается. А теперь, оставив в стороне вопрос, какой должна быть эта копия — глубокой или поверхностной, взглянем на сигнатуру метода `Clone()`:

```
object Clone()
```

Разумеется, сигнатура проста — как уже упоминалось, этот метод должен возвращать копию объекта, на котором он был вызван. Это означает необходимость возвращения объекта того же самого типа или, в крайнем случае, совместимого с ним типа (когда значение меняется в зависимости от типа).

Был бы смысл иметь возможность переопределения этого метода с применением сигнатуры, которая дает более точное описание того, что метод в действительности возвращает.

Например, в классе `Person` было бы неплохо реализовать интерфейс `ICloneable` со следующим методом:

```
public Person Clone()
```

Это не приводит к каким-либо нарушениям — код, ожидающий любой объект типа `object`, по-прежнему будет работать. Такая возможность называется *ковариантностью возвращаемых типов*, но, к сожалению, механизмы реализации интерфейсов и переопределения методов ее не поддерживают. Вместо этого для достижения желаемого результата в случае интерфейсов используется обходной путь — *явная реализация интерфейса*:

```
public Person Clone()
{
  [Реализация]
}
object ICloneable.Clone()      ← Явная реализация интерфейса
{
  return Clone();              ← Вызов метода, не относящегося к интерфейсу
}
```

Любой код, который вызывает `Clone()` на выражении со статическим типом `Person`, будет взаимодействовать с первым методом; если же типом выражения является `ICloneable`, то будет вызван второй метод. Хоть это и работает, но выглядит безобразно. Ситуация находит зеркальное отражение в параметрах, когда имеется интерфейсный или виртуальный метод с сигнатурой вроде `void Process(string x)`, и кажется вполне логичным наличие возможности реализации или переопределения данного метода с применением менее требовательной сигнатуры, такой как `void Process(object x)`. Это называется *контравариантностью типов параметров* и не поддерживается подобно ковариантности возвращаемых типов. Здесь придется использовать тот же самый обходной путь для интерфейсов и нормальнкю перегрузку для виртуальных методов. Такие сложности работу не останавливают, однако вызывают раздражение.

Разумеется, разработчикам на C# 1 долгое время приходилось мириться со всеми этими проблемами, а разработчики на Java пребывали в аналогичной ситуации гораздо дольше. Хотя безопасность типов на этапе компиляции в целом является великолепной характеристикой, я что-то не припомню частых случаев помещения кем-либо в коллекцию элемента неподходящего типа. Меня вполне устраивает отсутствие ковариантности и контравариантности. Однако существуют такие понятия, как элегантность и четкость выражения кодом своих целей, желательно *без* необходимости в предоставлении пояснительных комментариев. Даже если ошибки не возникают, навязывание документированного контракта о том, что коллекция *должна* содержать только строки (например), может оказаться затратным и хрупким с учетом изменяемости коллекций. Такого рода контракт должна обеспечивать сама система типов.

Позже вы увидите, что язык C# 2 в этом плане небезупречен, хотя в него было внесено много улучшений. Еще больше изменений появилось в версии C# 4, но даже в ней ковариантность возвращаемых типов и контравариантность типов параметров отсутствует[11].

[11] В C# 4 была введена ограниченная *обобщенная* ковариантность и контравариантность, но это не совсем то же самое.

2.2.3. Резюме по характеристикам системы типов

В этом разделе вы узнали о ряде отличий между системами типов и, в частности, ознакомились с характеристиками, применяемыми к C# 1.

- Язык C# 1 является статически типизированным — компилятору известно, какие члены типов допускается использовать.
- Язык C# 1 является явным — вы должны указывать тип для каждой переменной.
- Язык C# 1 является безопасным — нельзя трактовать один тип, как если бы он был другим, если настоящее преобразование не доступно.
- Статическая типизация не позволяет отдельной коллекции быть строго типизированным списком строк или списком целочисленных значений и требует в этом случае большой объем дублированного кода для разных типов элементов.
- Переопределение методов и реализация интерфейсов не допускает ковариантности или контравариантности.

В следующем разделе раскрывается один из наиболее фундаментальных аспектов системы типов C# помимо ее высокоуровневых характеристик — отличия между структурами и классами.

2.3. Типы значений и ссылочные типы

Важность темы, рассматриваемой в этом разделе, трудно преувеличить. Все, что делается в .NET, относится либо к типу значения, либо к ссылочному типу, и вполне возможно долгое время заниматься разработкой, имея только смутное представление о разнице между этими понятиями. Хуже того, распространено множество мифов, еще больше запутывающих ситуацию. К сожалению, очень легко сделать краткое, но некорректное утверждение, которое довольно близко к истине, чтобы выглядеть правдоподобным, но достаточно неточно, чтобы ввести в заблуждение. Однако относительно сложно предоставить лаконичное и вместе с тем точное описание.

В этом разделе не рассматриваются низкоуровневая поддержка типов, маршализация между доменами приложений, взаимодействие с машинным кодом и другие аналогичные вопросы. Взамен предлагается взглянуть на абсолютные основы этой темы (применительно к C# 1), которые критически важны для понимания последующих версий C#.

Мы начнем с ознакомления с фундаментальными отличиями между типами значений и ссылочными типами, которые можно естественно наблюдать в реальном мире, а также в .NET.

2.3.1. Значения и ссылки в реальном мире

Предположим, что вы читаете фантастически интересную статью и хотите, чтобы ваш друг тоже прочитал ее. Далее представим, что она хранится в виде документа в открытом домене, просто чтобы избежать любых обвинений в нарушении авторского права. Что потребуется предоставить другу для того, чтобы он также смог прочитать его? Это зависит целиком от того, что вы читаете.

Для начала рассмотрим случай, когда вы располагаете напечатанным текстом документа. Чтобы предоставить другу копию, вам придется фотокопировать все страницы и затем передать ему. После этого ваш друг будет иметь собственную полную копию документа. В такой ситуации вы имеете дело с поведением *типа значения*. Вся информация

находится непосредственно в ваших руках — никуда ходить за ней не нужно. После того, как вы сделали копию, ваша копия информации совершенно не зависит от копии вашего друга. Вы можете добавлять пометки на свои страницы, но страницы вашего друга останутся незатронутыми.

Сравните сказанное с ситуацией, когда вы читаете эту статью на какой-то веб-странице. На этот раз другу понадобится предоставить только URL веб-страницы. Это является поведением *ссылочного типа*, при котором URL занимает место ссылки. Чтобы прочитать документ, нужно перейти по ссылке, поместив URL в адресную строку браузера и загрузив веб-страницу. Если по какой-то причине веб-страница изменилась (например, это страница в Википедии, к которой вы добавляли свои заметки), то при следующей ее загрузке и вы, и ваш друг увидите изменения.

Описанные отличия в реальном мире иллюстрируют основную разницу между типами значений и ссылочными типами в C# и .NET. Большинство типов в .NET являются ссылочными типами, и, скорее всего, вы будете создавать *намного* больше ссылочных типов, чем типов значений. Наиболее распространенные случаи — это классы (объявляемые с использованием ключевого слова `class`), которые представляют собой ссылочные типы, и структуры (объявляемые с применением ключевого слова `struct`), которые являются типами значений. Ниже перечислены другие примеры.

- Типы массивов являются ссылочными типами, даже если типы элементов относятся к типам значений (поэтому `int[]` будет ссылочным типом, хотя `int` — тип значения).
- Перечисления (объявляемые с помощью ключевого слова `enum`) являются типами значений.
- Типы делегатов (объявляемые с помощью ключевого слова `delegate`) являются ссылочными типами.
- Типы интерфейсов (объявляемые с помощью ключевого слова `interface`) являются ссылочными типами, но они могут быть реализованы типами значений.

Получив базовое представление о ссылочных типах и типах значений, можно переходить к рассмотрению нескольких самых важных деталей.

2.3.2. Основные положения типов значений и ссылочных типов

Имея дело с типами значений и ссылочными типами, важно понять одну ключевую концепцию — что собой представляет значение конкретного выражения. Чтобы излишне не усложнять, в качестве наиболее распространенных примеров будут использоваться переменные, однако то же самое применимо также к свойствам, вызовам методов, индексаторам и другим выражениям.

Как обсуждалось в разделе 2.2.1, с большинством выражений связаны статические типы. Результатом выражения типа значения является простое значение. Например, значение выражения 2 + 3 равно 5. Однако значением выражения *ссылочного* типа является ссылка — это *не* объект, на который производится ссылка. Значением выражения `String.Empty` будет *не* пустая строка, а ссылка на пустую строку. В повседневных обсуждениях и даже в документации мы склонны размывать это отличие. Например, я могу описать метод `String.Concat()` как возвращающий "строку, которая представляет собой результат конкатенации всех параметров". Использование здесь точной терминологии требует много времени и отвлекает внимание, к тому же никаких проблем не возникает до тех пор, пока все понимают, что возвращается только *ссылка*.

Для дальнейшей демонстрации предположим, что необходим тип Point, представляющий точку и хранящий два целочисленных значения, x и y. Он мог бы иметь конструктор, который принимает два значения. Тип можно было бы реализовать как структуру или как класс.

На рис. 2.3 представлен результат выполнения следующих строк кода:

```
Point p1 = new Point(10, 20);
Point p2 = p1;
```

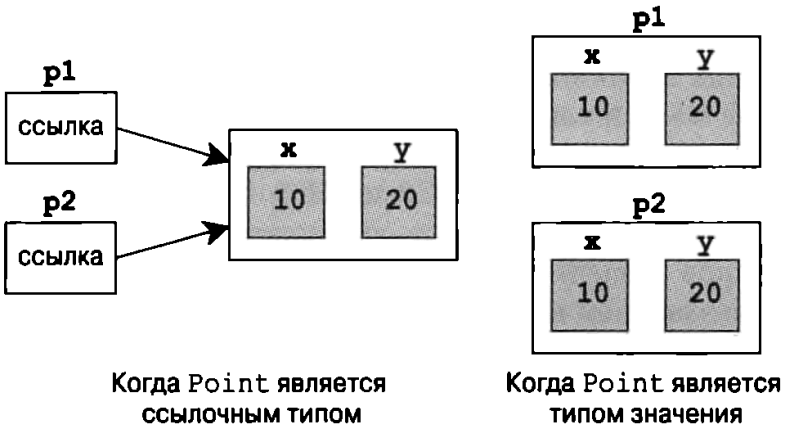

Рис. 2.3. Сравнение поведения типа значения и ссылочного типа в отношении присваивания

В левой части рис. 2.3 отражена ситуация, когда Point является классом (ссылочный тип), а в правой части — когда тип Point определен как структура (тип значения).

В обоих случаях после присваивания p1 и p2 получают одинаковые значения. Но в случае, когда Point имеет ссылочный тип, это значение является ссылкой: p1 и p2 ссылаются на один и тот же объект. Когда Point имеет тип значения, значением p1 являются полные данные о точке — значения x и y. Присваивание p1 переменной p2 копирует все эти данные.

Значения переменных хранятся там, где они объявлены. Значения локальных переменных всегда хранятся в стеке[12], а значения переменных экземпляра — там, где хранится сам экземпляр. Экземпляры ссылочных типов (объекты) всегда хранятся в куче, как и статические переменные.

Другое отличие между этими двумя разновидностями типа связано с тем, что от типов значений нельзя наследовать. В результате значение не нуждается в какой-то дополнительной информации о том, к какому типу оно *в действительности* относится. Сравните это со ссылочными типами, в которых каждый объект содержит в своем начале блок данных, идентифицирующих тип этого объекта, и ряд другой информации. Тип объекта изменить невозможно — когда осуществляется простое приведение, исполняющая среда всего лишь берет ссылку, проверяет, указывает ли она на допустимый объект заданного типа, и возвращает ссылку, если она корректна, или генерирует исключение в противном случае. Самой ссылке не известен тип объекта, поэтому одна и та же ссылка может использоваться для нескольких переменных разных типов. Например, взгляните на следующий код:

```
Stream stream = new MemoryStream();
MemoryStream memoryStream = (MemoryStream) stream;
```

[12] Это верно только для C# 1. Как будет показано далее в книге, в последующих версиях языка локальные переменные при определенных обстоятельствах могут оказываться в куче.

В первой строке кода создается новый объект `MemoryStream` и устанавливается значение переменной `stream` для ссылки на этот объект. Во второй строке кода выполняется проверка, ссылается ли значение переменной `stream` на объект `MemoryStream` (или производного от него класса), и устанавливается значение переменной `memoryStream`, совпадающее со значением `stream`.

Как только вы хорошо поймете эти базовые аспекты, можете применять их при обдумывании ряда ложных утверждений, касающихся типов значений и ссылочных типов.

2.3.3. Развенчание мифов

Разнообразные мифы регулярно передаются из уст в уста. Я уверен, что ложная информация почти всегда распространяется без злого умысла и неведения относительно допущенных неточностей, но вряд ли это может чем-то помочь. В этом разделе я коснусь наиболее известных мифов и объясню истинное состояние дел.

Миф №1: структуры являются легковесными классами

Этот миф преподносится во многих формах. Некоторые люди уверены, что типы значений не могут или не должны иметь методы либо обладать другим важным поведением, а должны использоваться как простые типы для передачи данных, располагающие только открытыми полями или простыми свойствами. Хорошим контрпримером может служить тип `DateTime`: ему рационально быть типом значения в том смысле, что он является фундаментальной единицей вроде числа или символа, и в то же время вполне уместной можно считать возможность выполнения вычислений на основе его значения. Взглянув на все это с другой стороны, следует отметить, что типы для передачи данных часто должны быть ссылочными типами. Такое решение необходимо принимать на основе желаемой семантики для типа (т.е. типа значения или ссылочного типа), а не на простоте типа.

Другие люди верят в то, что типы значений обеспечивают лучшую производительность. Правда заключается в том, что в *некоторых* случаях типы значений обеспечивают более высокую производительность — например, для них не нужна сборка мусора, когда они не упакованы, они не сопряжены с накладными расходами, связанными с идентификацией типов, и они не требуют разыменования. Однако в других случаях более производительными оказываются ссылочные типы: передача параметров, присваивание значений переменным, возвращение значений и подобные операции требуют копирования только 4 или 8 байтов (в зависимости от того, под управлением какой версии CLR выполняется приложение — 32-разрядной или 64-разрядной), а не *всех* данных. Если бы массив `ArrayList` был "чистым" типом значения, то передача выражения `ArrayList` методу приводила бы к копированию всех его данных! Практически во всех случаях решение о выборе между типом значения или ссылочным типом никак не влияет на производительность. Узкие места возникают, в основном, не там, где они предположительно могли бы быть, и перед тем, как принимать проектное решение, основанное на производительности, потребуется провести соответствующие измерения для разных вариантов.

Следует отметить, что комбинация описанных выше заблуждений также не сработает. Совершенно не играет роли, сколько методов определено в типе (будь то класс или структура) — на объем памяти, занимаемой каждым экземпляром, это не влияет. (Существует разница в затратах памяти на сам код, но это происходит однократно, а не для каждого экземпляра.)

Миф №2: ссылочные типы находятся в куче, а типы значений — в стеке

Этот миф не повторяет разве что ленивый. Первая часть утверждения корректна - экземпляр ссылочного типа всегда создается в куче. Проблемы кроются во второй части. Как уже упоминалось, значение переменной хранится там, где она объявлена, поэтому если есть класс с переменной экземпляра типа `int`, то значение этой переменной для любого объекта будет всегда находиться в том же месте, что и остальные данные объекта — в куче. В стеке размещаются только локальные переменные (переменные, объявленные внутри метода) и параметры метода. В C# 2 и последующих версиях даже некоторые локальные переменные не находятся в стеке, как будет показано при рассмотрении анонимных методов в главе 5.

Актуальны ли эти концепции сейчас?

Довольно спорно утверждать, что при написании управляемого кода вы должны возлагать заботу об эффективном использовании памяти на исполняющую среду. На самом деле спецификация языка не предоставляет никаких гарантий относительно того, где будут храниться те или иные данные. Будущая реализация исполняющей среды может быть в состоянии создавать некоторые объекты в стеке, если ей известно, что это удастся, или же компилятор C# сможет генерировать код, в котором стек почти не используется.

Следующий миф обычно связывают с вопросами терминологии.

Миф №3: по умолчанию в C# объекты передаются по ссылке

Пожалуй, это наиболее широко распространенный миф. Люди, которые утверждают подобное, часто (хотя и не всегда) знают действительное поведение C#, но не понимают, что на самом деле означает процедура "передачи по ссылке". К сожалению, это запутывает тех, кому известно ее значение.

Формальное определение *передачи по ссылке* выглядит относительно сложно. В нем применяются различные термины из области вычислительной техники, такие как *l-значение*, но важно понимать, что в случае передачи переменной по ссылке вызываемый метод может изменять *значение переменной вызывающего кода* за счет модификации значения своего параметра. А теперь вспомните, что значение переменной ссылочного типа является *ссылкой*, а не самим объектом. *Содержимое* объекта, на который ссылается параметр, можно изменить без необходимости в передаче самого параметра по ссылке. Например, в следующем методе модифицируется содержимое объекта `StringBuilder`, но выражение в вызывающем коде будет по-прежнему ссылаться на тот же самый объект, что и ранее:

```
void AppendHello(StringBuilder builder)
{
  builder.Append("hello");
}
```

Когда этот метод вызывается, значение параметра (ссылка на объект `StringBuilder`) передается по значению. Если вы измените значение переменной `builder` внутри метода (например, с помощью оператора `builder = null;`), *это* изменение, вопреки мифу, не будет видно в вызывающем коде.

Интересно отметить, что в мифическом утверждении неточной является не только часть "по ссылке", но также и часть "объекты передаются". Сами объекты *никогда* не передаются, ни по ссылке, ни по значению. При работе со ссылочным типом выполняется либо передача переменной по ссылке, либо передача значения аргумента (ссылки) по значению. Помимо всего прочего, это дает ответ на вопрос о том, что происходит в случае указания `null` для аргумента, передаваемого по значению — если бы передавались объекты, то неминуемо возникла бы проблема, поскольку `null` означает отсутствие объекта, подлежащего передаче! Вместо этого ссылка `null` передается по значению таким же образом, как и любая другая ссылка.

Если после приведенного краткого объяснения еще остались вопросы, можете почитать мою статью "Parameter passing in C#" ("Передача параметров в C#"), доступную по адресу `http://yoda.arachsys.com/csharp/parameters.html`, в которой эта тема рассматривается более подробно.

Существуют и другие мифы кроме указанных выше. Упаковка и распаковка привносят свою долю в общее непонимание, которую я постараюсь устранить.

2.3.4. Упаковка и распаковка

Иногда значение типа значения просто не нужно, а нужна ссылка. Подобное случается по разным причинам, и к счастью, в C# и .NET предлагается механизм под названием *упаковка*, который позволяет создавать объект из значения переменной типа значения и использовать ссылку на этот новый объект. Прежде чем обратиться к примеру, давайте проанализируем два важных факта:

- значением переменной ссылочного типа всегда является ссылка;
- значением переменной типа значения всегда является значение этого типа.

С учетом этих двух фактов следующие три строки кода на первый взгляд бессмысленны:

```
int i = 5;
object o = i;
int j = (int) o;
```

Есть две переменных: `i` — переменная типа значения, а `o` — переменная ссылочного типа. Какой смысл в присваивании значения `i` переменной `o`? Значение `o` должно быть ссылкой, но число `5` — это не ссылка, а целочисленное значение. В действительности здесь происходит упаковка: исполняющая среда создает объект (в области кучи, т.к. это обычный объект), который содержит значение (`5`). Затем значением `o` становится ссылка на этот новый объект. Значением в объекте является *копия* исходного значения — модификация значения `i` не приводит к изменению значения внутри упаковки.

В третьей строке кода выполняется обратная операция — *распаковка*. Вы должны сообщить компилятору, в какой тип необходимо распаковать объект. В случае указания неподходящего типа (например, если было упаковано значение типа `uint` или `long`, либо вообще никакого) генерируется исключение `InvalidCastException`. При распаковке значение, находящееся внутри упаковки, копируется; после присваивания дальнейшая связь между переменной `j` и объектом отсутствует.

Так в двух словах можно описать упаковку и распаковку. Осталось только выяснить, когда они происходят. Распаковка обычно очевидна, поскольку в коде присутствует приведение. Упаковка может быть менее заметной. Выше был показан простой случай, однако упаковка может также выполняться при вызове методов `ToString()`, `Equals()` и `GetHashCode()` для значения типа, в котором отсутствуют переопределенные версии

этих методов[13]. Кроме того, упаковка происходит, если значение применяется в качестве выражения с типом интерфейса, т.е. значение присваивается переменной, имеющей тип интерфейса, или передается как значение параметра, для которого указан тип интерфейса. Например, оператор `IComparable x = 5;` будет упаковывать число 5.

Об упаковке и распаковке полезно знать, поскольку с ними связаны потенциальные потери производительности. Влияние одиночной операции упаковки или распаковки ничтожно, но выполнение сотен тысяч таких операций приводит не только к возрастанию расходов в плане производительности, но также и к созданию *огромного* количества объектов, что явно добавит работы сборщику мусора. Хотя подобное падение производительности обычно не является проблемой, об этом полезно знать, а при появлении поводов для беспокойства — проводить измерение влияния операций упаковки и распаковки.

2.3.5. Резюме по типам значений и ссылочным типам

В этом разделе были рассмотрены отличия между типами значений и ссылочными типами, а также несколько распространенных мифов о них. Ниже перечислены ключевые моменты.

- Значением выражения ссылочного типа (например, переменной) является ссылка, а не объект.
- Ссылки похожи на URL — они представляют собой небольшие порции данных, которые позволяют получать доступ к реальной информации.
- Значением выражения типа значения являются действительные данные.
- Временами типы значений оказываются эффективнее ссылочных типов, а временами — наоборот.
- Объекты ссылочных типов всегда находятся в куче, но значения типов значений в зависимости от контекста могут быть либо стеке, либо в куче.
- Когда ссылочный тип используется для параметра метода, по умолчанию аргумент будет передаваться по значению, но самим значением является ссылка.
- Когда необходимо поведение, характерное для ссылочного типа, значения типов значений упаковываются; распаковка представляет собой обратный процесс.

Теперь, когда кратко упомянуты все аспекты языка C# 1, которые вы должны освоить, наступило время забежать вперед и выяснить, каким образом каждое средство совершенствовалось в последующих версиях C#.

2.4. За рамками C# 1: новые возможности на прочной основе

Три темы, раскрытые в этой главе, жизненно важны для всех версий C#. Почти все новые средства связаны хотя бы с одной из них, и эти темы изменяют особенности работы с языком. Прежде чем завершить главу, давайте выясним, как новые возможности соотносятся со старыми. Хотя особые детали приводиться не будут (по причине соблю-

[13] Упаковка будет *всегда* происходить при вызове метода `GetType()` на переменной типа значения, т.к. этот метод не может быть переопределен. Если вы имеете дело с неупакованной формой, то уже должны знать точный тип, поэтому можете просто использовать вместо `GetType()` операцию `typeof`.

дения краткости), все же полезно получить представление о том, куда мы двигаемся, до того, как погрузиться в рассмотрение мельчайших подробностей. Мы будем описывать средства в том же самом порядке, в котором это делалось ранее, начиная с делегатов.

2.4.1. Средства, связанные с делегатами

Делегаты всех видов получили поддержку в C# 2, и даже еще большее развитие в C# 3. Большинство связанных с ними средств являются не нововведениями CLR, а более искусными трюками компилятора, которые улучшают работу с делегатами внутри языка. Изменения затрагивают не только синтаксис, который *можно* использовать, но также внешний вид и поведение самого кода C#. Со временем C# обретает более функциональный подход.

Синтаксис C# 1, применяемый для создания экземпляра делегата, довольно неуклюж. Прежде всего, даже если необходимо предпринять довольно простое действие, приходится писать целый отдельный метод, чтобы создать для него экземпляр делегата. В C# 2 ситуация была исправлена с помощью анонимных методов и появления более простого синтаксиса в случаях, когда по-прежнему нужно использовать обычный метод для предоставления действия делегату. Создавать экземпляры делегатов можно также с применением методов, имеющих совместимые сигнатуры — сигнатура метода больше не должна быть точно такой же, как указанная в объявлении делегата.

Все эти улучшения продемонстрированы в листинге 2.4.

Листинг 2.4. Улучшения в создании экземпляров делегатов, внесенные в C# 2

```
static void HandleDemoEvent(object sender, EventArgs e)
{
  Console.WriteLine ("Handled by HandleDemoEvent");
                  // Обработан методом HandleDemoEvent
}
...
EventHandler handler;
handler = new EventHandler(HandleDemoEvent);      ❶ Указание типа делегата и метода
handler(null, EventArgs.Empty);

handler = HandleDemoEvent;                        ❷ Неявное преобразование в экземпляр делегата
handler(null, EventArgs.Empty);
handler = delegate(object sender, EventArgs e)    ❸ Указание действия с помощью анонимного метода
{
  Console.WriteLine ("Handled anonymously");
                  // Обработан анонимным методом
};
handler(null, EventArgs.Empty);
handler = delegate                                ❹ Использование сокращения посредством анонимного метода
{
  Console.WriteLine ("Handled anonymously again");
                  // Снова обработан анонимным методом
};
handler(null, EventArgs.Empty);
MouseEventHandler mouseHandler = HandleDemoEvent; ❺ Использование контравариантности делегатов
mouseHandler(null,
  new MouseEventArgs(MouseButtons.None, 0, 0, 0, 0));
```

В первой части главного кода ❶ в целях сравнения приведен код на C# 1. Все остальные делегаты используют новые средства C# 2. Преобразования групп методов ❷ существенно улучшает читабельность кода подписки на события — оператор вроде `saveButton.Click += SaveDocument;` намного проще для восприятия и не отвлекает внимание на маловажные детали. Синтаксис анонимных методов ❸ немного громоздкий, но он позволяет действию оставаться в точке создания, вместо того чтобы находиться в другом методе, который еще придется отыскать, чтобы понять, что в нем происходит. При использовании анонимных методов доступно сокращение ❹, но такая форма может применяться, только когда не нужны параметры. Анонимные методы обладают также и другими мощными возможностями, но мы рассмотрим их позже.

Последний создаваемый экземпляр делегата ❺ является экземпляром `MouseEventHandler`, а не просто `EventHandler`, но метод `HandleDemoEvent()` по-прежнему используется благодаря *контравариантности*, которая обеспечивает совместимость параметров. *Ковариантность* определяет совместимость возвращаемых типов. Ковариантность и контравариантность более подробно рассматриваются в главе 5. Самую большую выгоду от них получают, пожалуй, обработчики событий, т.к. оказывается, что рекомендация от Microsoft о необходимости следования всех типов делегатов, применяемых в событиях, одному и тому же соглашению имеет намного больше смысла. В C# 1 не играло роли, что два разных обработчика событий выглядели очень похожими — для создания экземпляра делегата требовался метод с *точно* совпадающей сигнатурой. В C# 2 один метод можно использовать для обработки множества разных видов событий, особенно если предназначение метода явно не зависит от событий (например, метод реализует регистрацию в журнале).

В C# 3 для создания экземпляров типов делегатов предлагается специальный синтаксис, использующий *лямбда-выражения*. Для его демонстрации будет применяться новый тип делегата. Когда в .NET 2.0 среда CLR получила обобщения, стали доступными обобщенные типы делегатов, которые используются при многих обращениях к API-интерфейсу в обобщенных коллекциях. В .NET 3.5 дополнительно появилась группа обобщенных типов делегатов под названием `Func`, которые принимают параметры указанных типов и возвращают значение другого заданного типа. В листинге 2.5 демонстрируется использование типа делегата `Func` и лямбда-выражений.

Листинг 2.5. Лямбда-выражения подобны улучшенным анонимным методам

```
Func<int,int,string> func = (x, y) => (x * y).ToString();
Console.WriteLine(func(5, 20));
```

Здесь `Func<int,int,string>` представляет собой тип делегата, который принимает два целочисленных значения и возвращает строку. Лямбда-выражение в листинге 2.5 указывает, что экземпляр делегата (содержащийся в `func`) должен умножать два целых числа и вызывать метод `ToString()`. Такой синтаксис намного проще, чем в случае применения анонимных методов, и он также обладает другими преимуществами, которые касаются количества выведений типов, обеспечиваемых компилятором. Лямбда-выражения критически важны для LINQ, и вы должны сделать их основной частью набора инструментов для работы с языком. Тем не менее, они не ограничиваются работой только с LINQ — любой случай использования анонимного метода в C# 2 можно заменить лямбда-выражением в C# 3, почти всегда получая более короткий код.

Подводя итоги, ниже перечислены новые средства, связанные с делегатами.

- Обобщения (обобщенные типы делегатов) — C# 2.
- Выражения для создания экземпляров делегатов — C# 2.
- Анонимные методы — C# 2.
- Ковариантность и контравариантность делегатов — C# 2.
- Лямбда-выражение — C# 3.

Кроме того, в C# 4 для делегатов разрешена *обобщенная* ковариантность и контравариантность, как было указано выше. На самом деле обобщения являются одним из принципиальных улучшений системы типов, что и является темой следующего раздела.

2.4.2. Средства, связанные с системой типов

Главным новым средством C# 2, связанным с системой типов, является появление обобщений. Они в значительной мере устраняют проблемы, указанные в разделе 2.2.2 о строго типизированных коллекциях, хотя обобщенные типы удобны также и в других ситуациях. Это элегантное средство позволяет решить реальную проблему и, несмотря на ряд недостатков, в целом работает хорошо. Вы уже видели немало примеров его применения, и в следующей главе данное средство рассматривается более подробно. Что касается системы типов, то обобщения можно считать самой важной возможностью, появившейся в C# 2, и в оставшихся главах книги обобщенные типы будут встречаться повсеместно.

В C# 2 не решены проблемы ковариантности возвращаемых типов и контравариантности параметров при переопределении членов или реализации интерфейсов. Однако в этой версии был улучшен способ создания экземпляров делегатов в определенных ситуациях, как было показано в разделе 2.4.1.

В C# 3 появилось множество новых концепций, связанных с системой типов, наиболее заметными из которых являются *анонимные типы*, *неявно типизированные локальные переменные* и *расширяющие методы*. Сами анонимные типы в основном существуют ради LINQ, где удобно иметь возможность эффективного создания типа для передачи данных с множеством свойств, предназначенных только для чтения, не требуя фактического написания кода для них. Тем не менее, ничто не препятствует использованию анонимных типов за рамками LINQ, если это упрощает решение каких-либо задач. В листинге 2.6 новые средства демонстрируются в действии.

Листинг 2.6. Применение анонимных типов и неявной типизации

```
var jon = new { Name = "Jon", Age = 31 };
var tom = new { Name = "Tom", Age = 4 };
Console.WriteLine ("{0} is {1}", jon.Name, jon.Age);
Console.WriteLine ("{0} is {1}", tom.Name, tom.Age);
```

В первых двух строках кода показана неявная типизация (использование ключевого слова `var`) и инициализаторы анонимных объектов (конструкция `new { ... }`), которые создают экземпляры анонимных типов.

На данном этапе следует обратить внимание на два момента, которые ранее у многих вызывали напрасные волнения. Первый момент — язык C# 3 по-прежнему остается статически типизированным. Компилятор C# объявил переменные `jon` и `tom` как относящиеся к определенному типу, а при взаимодействии со свойствами этих объектов

они ведут себя подобно нормальным свойствам — никакого динамического поиска не происходит. Просто вы (как автор исходного кода) не можете сообщить компилятору, какой тип использовать в объявлении переменной, поэтому компилятор сгенерирует его самостоятельно. Свойства также являются статически типизированными — свойство `Age` имеет тип `int`, а свойство `Name` — тип `string`.

Второй момент связан с тем, что здесь не создаются два разных анонимных типа. Переменные `jon` и `tom` имеют один и тот же тип, т.к. на основе имен свойств, их типов и порядка следования компилятор выясняет, что может быть сгенерирован только один тип, который и будет назначен обеим переменным. Это делается в рамках каждой сборки и упрощает написание кода, давая возможность присваивать значение одной переменной другой переменной (например, к предыдущему коду вполне допустимо добавить оператор `jon = tom;`) и аналогичные операции.

Расширяющие методы также предназначены для LINQ, но могут применяться и за его пределами. Каждый раз представляйте ситуацию, когда нужно, чтобы какой-то тип инфраструктуры располагал определенным методом, для реализации которого вы должны написать статический вспомогательный метод. Например, чтобы создать новую строку за счет обращения порядка символов в существующей строке, можно реализовать статический метод `StringUtil.Reverse()`. Фактически средство расширяющих методов позволяет вызывать этот статический метод, как если бы он был определен в самом типе `string`, так что разрешено записывать следующий код:

```
string x = "dlrow olleH".Reverse();
```

Расширяющие методы также позволяют добавлять методы с реализациями к интерфейсам, и LINQ в большой степени полагается на это, позволяя вызывать на интерфейсе `IEnumerable<T>` все виды методов, которые ранее не существовали.

В C# 4 с системой типов связаны две возможности. Одна относительно небольшая возможность представляет собой ковариантность и контравариантность для обобщенных делегатов и интерфейсов. Она присутствует в CLR, начиная с выхода .NET 2.0, но только с появлением C# 4 и обновлением обобщенных типов в *библиотеке базовых классов* (Base Class Library — BCL) эта возможность стала доступна для использования разработчиками на C#. Другим, существенно более крупным средством, хотя и не востребованным многими программистами, является динамическая типизация в C#.

Вспомните введение в статическую типизацию, где предпринималась попытка обратиться к свойству `Length` массива и строки через одну и ту же переменную. Так вот, в C# 4 это работает — естественно, когда оно нужно. В листинге 2.7 приведен тот же самый код за исключением объявления переменной, но теперь это допустимый код C# 4.

Листинг 2.7. Динамическая типизация в C# 4

```
dynamic o = "hello";
Console.WriteLine(o.Length);
o = new string[] {"hi", "there"};
Console.WriteLine(o.Length);
```

За счет объявления переменной `o` как принадлежащей к статическому типу `dynamic` (именно так — статическому), компилятор обрабатывает почти все действия с `o` по-другому, откладывая решения по связыванию (вроде того, что должно означать `Length`) до этапа выполнения.

Понятно, что мы будем рассматривать динамическую типизацию намного подробнее, но сейчас следует подчеркнуть, что по большей части C# 4 по-прежнему остается

статически типизированным языком. Если вы не используете тип `dynamic` (который действует как статический тип, обозначающий динамическое значение), то все будет работать точно так же, как и ранее. Многим разработчикам на языке C# динамическая типизация требуется очень редко, поэтому в остальное время они могут ее спокойно игнорировать. Когда динамическая типизация *нужна*, применять ее довольно легко, к тому же она позволит взаимодействовать с кодом, который написан на динамических языках, работающих под управлением исполняющей среды динамического языка (Dynamic Language Runtime — DLR). Я просто не советую начинать использовать C# как главным образом динамический язык. Если необходимо именно это, обратитесь к языку IronPython или ему подобному; языки, изначально спроектированные для поддержки динамической типизации, наверняка имеют меньшее количество нестыковок.

Ниже приведен краткий перечень средств с указанием версии C#, в которой они появились.

- Обобщения — C# 2.
- Ограниченная ковариантность и контравариантность делегатов — C# 2.
- Анонимные типы — C# 3.
- Неявная типизация — C# 3.
- Расширяющие методы — C# 3.
- Ограниченная ковариантность и контравариантность обобщений — C# 4.
- Динамическая типизация — C# 4.

После рассмотрения такого довольно разнообразного набора возможностей, связанных с системой типов, давайте взглянем на средства, которые были добавлены к одной специфической части типизации в .NET — типам значений.

2.4.3. Средства, связанные с типами значений

Здесь мы обсудим два средства, введенные в C# 2. Первое снова касается обобщений, в частности, коллекций. Одна из распространенных жалоб относительно применения типов значений в .NET 1.1 была связана с тем, что из-за определения всех универсальных API-интерфейсов в терминах типа `object` каждая операция по добавлению в коллекцию значения типа структуры вызывала упаковку, а при извлечении значение требовалось распаковывать. Хотя накладные расходы упаковки незначительны для одного вызова, они могут существенно повлиять на производительность при наличии коллекций, доступ к которым производится часто. Кроме того, увеличивается объем необходимой памяти, требуемой для упакованных объектов. Обобщения исправляют недостатки, связанные с производительностью и памятью, за счет использования *реального* типа вместо универсального. Например, в .NET 1.1 было бы весьма неосмотрительно читать файл и сохранять каждый его байт в виде элемента `ArrayList`, но в .NET 2.0 это вполне можно реализовать с помощью `List<byte>`.

Второе средство решает еще одну распространенную причину нареканий, особенно когда речь идет о работе с базами данных — невозможность присваивания `null` переменной, имеющей тип значения. К примеру, концепция значения `null` типа `int` не существует, несмотря на то, что целочисленное поле в таблице *базы данных* вполне может принимать значение `null`. Это затрудняет моделирование таблицы базы данных с помощью статически типизированного класса, приводя к появлению неуклюжего кода в той или иной форме. Типы, допускающие `null`, появились в .NET 2.0, а в C# 2 был включен дополнительный синтаксис для их простого использования.

В листинге 2.8 представлен небольшой пример.

Листинг 2.8. Демонстрация разнообразных возможностей типов, допускающих null

```
int? x = null;              <-- 1 Объявление и установка переменной, допускающей значение null
x = 5;
if (x != null)              <-- 2 Проверка наличия реального значения
{
  int y = x.Value;          <-- 3 Получение реального значения
  Console.WriteLine (y);
}
int z = x ?? 10;            <-- 4 Использование операции объединения с null
```

В листинге 2.8 продемонстрировано несколько возможностей типов, допускающих `null`, и сокращение, предоставляемое в языке C# для работы с ними. Мы подробно рассмотрим каждую возможность в главе 4, а пока важно отметить, что все стало намного более простым и ясным, чем обходные приемы, которые приходилось применять в прошлом.

На этот раз список усовершенствований невелик, однако он включает важные средства с точки зрения производительности и элегантности представления.

- Обобщения — C# 2.
- Типы, допускающие `null` — C# 2.

2.5. Резюме

По большей части в этой главе были описаны возможности C# 1. Цель заключалась не в предоставлении полного описания каких-либо тем, а в сжатом изложении наиболее важных основ, что позволит рассматривать появившиеся впоследствии средства, не отвлекаясь на эти основы.

Все раскрытые здесь темы являются фундаментальными для C# и .NET, но мне часто приходилось сталкиваться с их неправильным толкованием. Хотя каждый отдельный аспект в главе не обсуждался особо подробно, надеюсь, предложенные сведения поспособствуют более ясному пониманию остального материала этой книги.

Все три основные темы, кратко затронутые в главе, были значительно расширены со времен версии C# 1, и некоторые средства охватывают сразу несколько тем. В частности, добавление обобщений повлияло почти на каждую область, рассмотренную в настоящей главе — пожалуй, это наиболее широко используемое средство в версии C# 2. Теперь, когда все подготовительные действия завершены, можно приступать к более детальным исследованиям обобщений в следующей главе.

Часть II

C# 2: решение проблем, присущих C# 1

В части I мы кратко коснулись нескольких средств языка C# 2. Наступило время перейти к подробностям. Вы увидите, что в C# 2 устранены различные проблемы, которые возникали у разработчиков, использующих C# 1, и узнаете, как C# 2 повышает удобство работы с существующими средствами за счет их упрощения. Хотя это все еще нелегкое дело, но разработка на C# 2 намного приятнее, чем на C# 1.

Новые средства C# 2 в определенной степени независимы. Это не говорит о том, что они вообще не связаны; многие средства основаны или, по крайней мере, взаимодействуют с тем крупным вкладом, который привнесли в язык обобщения. Однако различные темы, которые будут рассматриваться в следующих пяти главах, не объединены в единое средство.

В первых четырех главах этой части описаны самые важные новые возможности. Будут раскрыты перечисленные ниже темы.

- Обобщения. Будучи наиболее важным новым средством в C# 2 (и, конечно же, в среде CLR для .NET 2.0), обобщения делают возможной параметризацию типов и методов на основе типов, с которыми они взаимодействуют.
- Типы, допускающие значения `null`. Типы значений, такие как `int` и `DateTime`, не поддерживают концепцию "отсутствия значения"; типы, допускающие `null`, позволяют представлять отсутствие содержательного значения.
- Делегаты. Несмотря на то что делегаты на уровне среды CLR не изменились, в C# 2 с ними стало намного легче работать. Появление анонимных методов в дополнение к нескольким простым сокращениям приводит к переходу на более функциональный стиль программирования — и эта тенденция продолжается в C# 3.
- Итераторы. Хотя использование итераторов всегда являлось простым в языке C# благодаря оператору `foreach`, в C# 1 была затруднена их реализация. Компилятор C# 2 благополучно строит "за кулисами" конечный автомат, скрывая множество сопряженных с этим сложностей.

После описания этих главных и сложных новых средств языка C# 2 в главах, специально выделенных для них, в главе 7 рассмотрение завершается представлением нескольких более простых возможностей. Более простых не означает менее полезных; например, частичные типы критически важны для улучшенной поддержки визуального конструирования в среде Visual Studio 2005 и последующих версиях. Это же средство также полезно для другого генерируемого кода. Подобным образом в наши дни разработчики на C# принимают за данное возможность написания свойства с открытым методом получения и закрытым методом установки, хотя это появилось только в версии C# 2.

Когда вышло первое издание этой книги, многие разработчики вообще еще не применяли C# 2. В 2013 году у меня сложилось впечатление, что редко когда удается найти разработчика, в текущий момент использующего C#, который бы не опробовал, пусть даже поверхностно, версию C# 2, возможно версию C# 3 и уж наверняка версию C# 4. Темы, раскрываемые в этой части, являются фундаментом для понимания того, как работают более поздние версии языка C#; в частности, изучить LINQ, не имея представления об обобщениях и итераторах, будет довольно сложно. Материал главы, посвященной итераторам, также связан с асинхронными методами C# 5; на первый взгляд эти два средства сильно отличаются друг от друга, однако оба они предполагают построение компилятором конечных автоматов для изменения обычного потока выполнения.

Если вы имели дело с C# 2 и последующими версиями на протяжении какого-то времени, то можете обнаружить, что многие материалы этой части вам уже знакомы, но я не сомневаюсь, что вы все равно извлечете пользу, получив более глубокие знания представляемых деталей.

Глава 3

Параметризованная типизация с использованием обобщений

В этой главе...

- Выведение типов для обобщенных методов
- Ограничения типов
- Рефлексия и обобщения
- Поведение среды CLR
- Ограничения обобщений
- Сравнение с другими языками

Приведу правдивую историю[1]: недавно мы с женой вышли за еженедельными покупками в ближайший продовольственный магазин. Прямо перед выходом жена спросила меня, захватил ли я список. Я подтвердил, что список у меня *есть*, после чего мы выдвинулись. Наше недоразумение обнаружилось только в магазине. Жена спрашивала о списке *необходимых покупок*, а у меня оказался список важных средств языка C# 2. Продавец довольно странно посмотрел на нас, когда мы спросили, можем ли мы купить какие-нибудь анонимные методы.

[1] Под этим я понимаю "пригодную для целей введения в материал главы", но не обязательно точную.

Если бы мы только могли выражать свои вопросы яснее! Если бы у жены была возможность сказать, что от меня требуется список необходимых покупок! Если бы мы имели обобщения... и так далее.

Для большинства разработчиков обобщения являются самым важным новым средством C# 2. Они улучшают производительность, делают код более выразительным и переносят множество проверок, связанных с безопасностью, с этапа выполнения на этап компиляции. По существу обобщения позволяют *параметризировать* типы и методы. Подобно тому, как при вызове обычных методов часто указываются параметры, сообщающие им о том, какие *значения* использовать, обобщенные типы и методы имеют параметры типов, с помощью которых задаются *типы*, предназначенные для применения. Поначалу все это может сбить с толку — и если тема обобщений совершенно нова для вас, то будет над чем поломать голову, — но как только вы ухватите основную идею, обобщения быстро вам понравятся.

В этой главе будет показано, как использовать обобщенные типы и методы, предоставленные другими (либо самой инфраструктурой, либо библиотеками от независимых разработчиков), и как писать собственные типы и методы подобного рода. Попутно с помощью вызовов API-интерфейса рефлексии мы также выясним, как работают обобщения, и ознакомимся с тем, каким образом обобщения обрабатываются средой CLR. В завершение главы будут описаны некоторые наиболее часто встречающиеся ограничения обобщений и возможные способы их обхода, а также приведено сравнение обобщений в C# и аналогичных средств в других языках.

Однако в первую очередь необходимо понять проблемы, которые привели к возникновению обобщений.

3.1. Необходимость в обобщениях

Если у вас есть под рукой какой-нибудь код C# 1, подсчитайте количество приведений в нем — особенно интересно будет сделать это для кода, в котором интенсивно используются коллекции. Не забывайте, что почти каждое применение оператора `foreach` содержит неявное приведение. При работе с разными типами данных приведения неизбежны. С их помощью компилятору сообщается о том, что все в порядке, и нужно лишь трактовать то или иное выражение как принадлежащее *конкретному* типу. Использование практически любого API-интерфейса, имеющего `object` в качестве типа параметра или возвращаемого типа, скорее всего, в какой-то момент вызовет приведения. Наличие в иерархии с единым классом корневого типа `object` упрощает ряд вещей, но сам по себе тип `object` чрезвычайно слаб, и для того, чтобы сделать с `object` что-нибудь по-настоящему полезное, почти всегда приходится выполнять его приведение.

Так что же, приведения плохи? Они *не* плохи, если применяются по принципу "почти никогда не делать этого" (подобно изменяемым структурам и незакрытым полям), однако плохи, когда они считаются "необходимым злом". Приведения являются признаком того, что вы должны были каким-то образом предоставить компилятору дополнительную информацию, и решили предложить ему доверять вам на этапе компиляции, а также сгенерировать проверку, которая будет предпринята во время выполнения, чтобы подтвердить вашу правоту.

Если вы должны были предоставить информацию компилятору, то велики шансы, что эта же информация понадобится и тому, кто будет *читать* написанный вами код.

Конечно, несложно заметить, в каком месте осуществляется приведение, но это не особенно полезно. Идеальным местом для размещения такого рода информации обычно является точка, в которой объявляется переменная или метод. Это становится еще более важным, если вы предлагаете тип или метод для вызова, *не предоставляя доступа к исходному коду.* Обобщения позволяют поставщикам библиотек предотвращать для своих потребителей компиляцию кода, в котором присутствуют обращения к библиотекам с указанием недопустимых аргументов.

В C# 1 приходилось полагаться на вручную написанную документацию, которая вполне могла оказаться неполной или неточной, как часто бывает с дублированной информацией. Указание дополнительной информации в коде как части объявления метода или типа позволяет увеличить продуктивность работы. Компилятор способен делать дополнительные проверки; на основе этой дополнительной информации IDE-среда может предоставлять подсказки IntelliSense (например, предлагая члены типа `string` в качестве следующего шага при доступе к элементу внутри списка строк); в коде вызова методов можно иметь большую уверенность в корректности передаваемых аргументов и возвращаемых значений; а лица, сопровождающие ваш код, могут лучше понять замысел, который вы воплотили в коде.

Снизят ли обобщения количество ошибок?

В каждом описании обобщений, которое мне приходилось читать (включая мое собственное), подчеркивается важность проверки типов на этапе компиляции по сравнению с проверкой типов на этапе выполнения. Раскрою вам один секрет: я не припоминаю случая, когда в выпущенном коде исправлялась ошибка, вызванная непосредственно отсутствием проверки типов. Другими словами, согласно моему опыту, приведения, помещаемые в код C# 1, всегда работают. Эти приведения были своего рода предупреждающими знаками, заставляя нас думать о безопасности типов явно, а не пускать ее на самотек. Но хотя обобщения могут и не привести к радикальному уменьшению количества ошибок, связанных с нарушениями безопасности типов, обеспечиваемое ими улучшение читабельности способствует сокращению числа ошибок по всем категориям. Код, который проще понять, проще и сделать правильным. Подобным же образом, код, который должен быть надежным в отношении неподходящих вызовов, намного проще написать корректно, имея соответствующие гарантии со стороны системы типов.

Всего перечисленного уже вполне достаточно, чтобы считать обобщения полезными, но есть еще и улучшения в плане производительности. Во-первых, компилятор может предпринять больше принудительных действий, оставляя меньшее число проверок на этап выполнения. Во-вторых, компилятор JIT может трактовать типы значений более интеллектуальным способом, который позволяет избегать упаковки и распаковки во многих ситуациях. В ряде случаев это может привести к огромной разнице в производительности в терминах скорости выполнения и потребления памяти.

Многие преимущества обобщений могут выглядеть похожими на преимущества статически типизированных языков перед динамически типизированными языками: лучшая проверка на этапе компиляции, больший объем информации, выраженной прямо в коде, улучшенная поддержка со стороны IDE-среды, более высокая производительность.

Причина довольно проста: используя общий API-интерфейс (вроде `ArrayList`), который не способен различать разные типы, вы на самом деле *находитесь* в динамической ситуации в терминах доступа к этому API-интерфейсу. Кстати, обратное, как правило, не верно — преимущества, обеспечиваемые динамическими языками, редко принимаются в расчет при выборе между обобщенными и необобщенными API-интерфейсами. При наличии разумной *возможности* применения обобщений решение об этом не требует особых размышлений.

Итак, памятуя обо всех прелестях, которые нам сулит C# 2, приступим к реальному использованию обобщений.

3.2. Простые обобщения для повседневного использования

С темой обобщений связано немало темных закоулков, если вы планируете узнать о ней *абсолютно все*. В спецификации языка C# приведено достаточно деталей, раскрывающих практически каждый мыслимый случай. Однако для продуктивной работы разбирать все краевые случаи необязательно. (На самом деле это справедливо и в других областях. Например, вам не нужно знать все точные правила, касающиеся присваивания — если компилятор сообщит об ошибке, вы просто соответствующим образом исправите код.)

В этом разделе будет дана большая часть сведений, необходимых при повседневной работе с обобщениями, которые созданы как вами, так и другими. Если чтение этой главы дается с трудом, но вы хотите ускорить продвижение вперед, я советую сосредоточить внимание на том, что нужно знать для использования обобщенных типов и методов из инфраструктуры и других библиотек. Потребность в написании собственных обобщенных типов и методов возникает гораздо реже, чем необходимость в применении таких типов и методов, определенных внутри инфраструктуры.

Начнем с рассмотрения одного из классов коллекций, появившихся в .NET 2.0 — `Dictionary<TKey,TValue>`.

3.2.1. Обучение на примерах: обобщенный словарь

С использованием обобщенных типов не связаны какие-либо сложности, если только не нарушать присущие им ограничения. Догадаться о том, что будет делать тот или иной фрагмент кода, довольно легко и без знания какой-то терминологии, а через серию проб и ошибок можно опытным путем вывести собственный способ написания работающего кода. (Одно из преимуществ обобщений состоит в том, что больший объем проверок осуществляется на этапе компиляции, поэтому если код успешно компилируется, он с высокой вероятностью окажется работоспособным; такой подход упрощает экспериментирование.) Разумеется, целью этой главы является формирование у вас собственного багажа знаний, чтобы вы не оперировали только догадками, а четко представляли себе, что происходит на том или ином этапе.

А теперь давайте взглянем на код, который достаточно прост, несмотря на незнакомый синтаксис. В листинге 3.1 с применением класса `Dictionary<TKey,TValue>` (грубого обобщенного эквивалента необобщенного класса `Hashtable`) производится подсчет слов, встречающихся в заданном фрагменте текста.

Листинг 3.1. Использование класса `Dictionary<TKey,TValue>` для подсчета слов в тексте

```
static Dictionary<string,int> CountWords(string text)
{
  Dictionary<string,int> frequencies;  <-- ❶ Создание новой карты для слов и частот
  frequencies = new Dictionary<string,int>();
  string[] words = Regex.Split(text, @"\W+"); <-- ❷ Разбиение текста на слова
  foreach (string word in words)
  {
    if (frequencies.ContainsKey(word))
    {
      frequencies[word]++;
    }
    else
    {
      frequencies[word] = 1;
    }
  }
  return frequencies;
}
...
string text = @"Do you like green eggs and ham?
              I do not like them, Sam-I-am.
              I do not like green eggs and ham.";
Dictionary<string,int> frequencies = CountWords(text);
foreach (KeyValuePair<string,int> entry in frequencies)
{
  string word = entry.Key;
  int frequency = entry.Value;
  Console.WriteLine ("{0}: {1}", word, frequency);
}
```

❸ Добавление или обновление карты

❹ Вывод пар "ключ/значение" на основе карты

В методе `CountWords()` сначала создается пустая карта для отображения `string` на `int` ❶. Она позволит вычислять частоту использования каждого слова в заданном тексте. Затем с помощью регулярного выражения ❷ текст разбивается на отдельные слова. Регулярное выражение довольно сырое — из-за наличия точки в конце текста появится пустая строка, к тому же слова *do* и *Do* будут подсчитываться отдельно. Указанные проблемы легко исправляются, но в рассматриваемом примере это не делается, чтобы сохранить его простым.

Для каждого слова производится проверка, присутствует ли оно в карте. Если это так, существующее значение счетчика инкрементируется; иначе слову назначается новый счетчик с начальным значением 1 ❸. Обратите внимание, что в коде инкрементирования приведение к `int` не требуется; нам известно, что извлекаемое значение имеет тип `int`, на этапе компиляции. На шаге увеличения счетчика фактически выполняется метод извлечения на индексаторе для карты, затем собственно инкрементирование и, наконец, метод установки на индексаторе. Вместо этого можно было бы указать явный оператор: `frequencies[word] = frequencies[word] + 1;`.

Последняя часть листинга 3.1 должна быть знакомой: перечисление по последовательности `Hashtable` дает для каждого элемента сходный экземпляр (необобщенного) класса `DictionaryEntry` с соответствующим образом установленными свойствами `Key` и `Value` ❶. Но в C# 1 пришлось бы приводить значения слова и частоты, поскольку ключ и значение возвращались бы в виде экземпляров типа `object`. Это также означало бы упаковку значения частоты. Правда, вы не обязаны помещать значения слова и частоты внутрь переменных — можно было бы предусмотреть единственный вызов метода `Console.WriteLine()` с передачей ему `entry.Key` и `entry.Value` в качестве аргументов. Переменные применялись, просто чтобы подчеркнуть отсутствие необходимости в каком-либо приведении.

А теперь, после ознакомления с примером, давайте посмотрим, что в первую очередь имеется в виду, когда речь идет о `Dictionary<TKey,TValue>`. Что собой представляют `TKey` и `TValue` и почему они в угловых скобках?

3.2.2. Обобщенные типы и параметры типов

В языке C# существуют две формы обобщений: *обобщенные типы* (к которым относятся классы, интерфейсы, делегаты и структуры — но обобщенных перечислений не бывает) и *обобщенные методы*. Обе эти формы по существу являются способами выражения некоторого API-интерфейса (то ли для единственного обобщенного метода, то ли для целого обобщенного типа) таким образом, что в ряде мест, где ожидается встретить обычный тип, вместо него будет находиться *параметр типа*.

Параметр типа — это заполнитель для реального типа. Параметры типов задаются в угловых скобках внутри обобщенного объявления с использованием запятых в качестве разделителей. Таким образом, в `Dictionary<TKey,TValue>` параметрами типов являются `TKey` и `TValue`. Когда обобщенный тип или метод применяется, указываются желаемые *реальные* типы. Они называются *аргументами типов* — например, в листинге 3.1 аргументами типов были `string` (для `TKey`) и `int` (для `TValue`).

Внимание, жаргон!

С темой обобщений связана подробная терминология. Она включена здесь для справочных целей, а также потому, что иногда терминология намного упрощает рассмотрение темы. Это также может быть полезно, когда необходимо обратиться за справкой к спецификации языка, но вряд ли данная терминология понадобится в повседневной жизни. Просто пока смиритесь с этим. Большая часть терминологии определена в разделе 4.4 спецификации C# 5 ("Constructed Types" ("Сконструированные типы")) — туда и обращайтесь за дополнительной информацией.

Форма обобщенного типа, при которой ни одному из параметров типов не были предоставлены аргументы типов, называется *несвязанным обобщенным типом*. Когда аргументы типов указаны, говорят, что тип является *сконструированным типом*. Несвязанные обобщенные типы фактически выступают в качестве шаблонов для сконструированных типов, во многом похоже на то, как типы (обобщенные или нет) могут рассматриваться как шаблоны для объектов. Это своего рода дополнительный уровень абстракции. На рис. 3.1 представлена графическая иллюстрация сказанного.

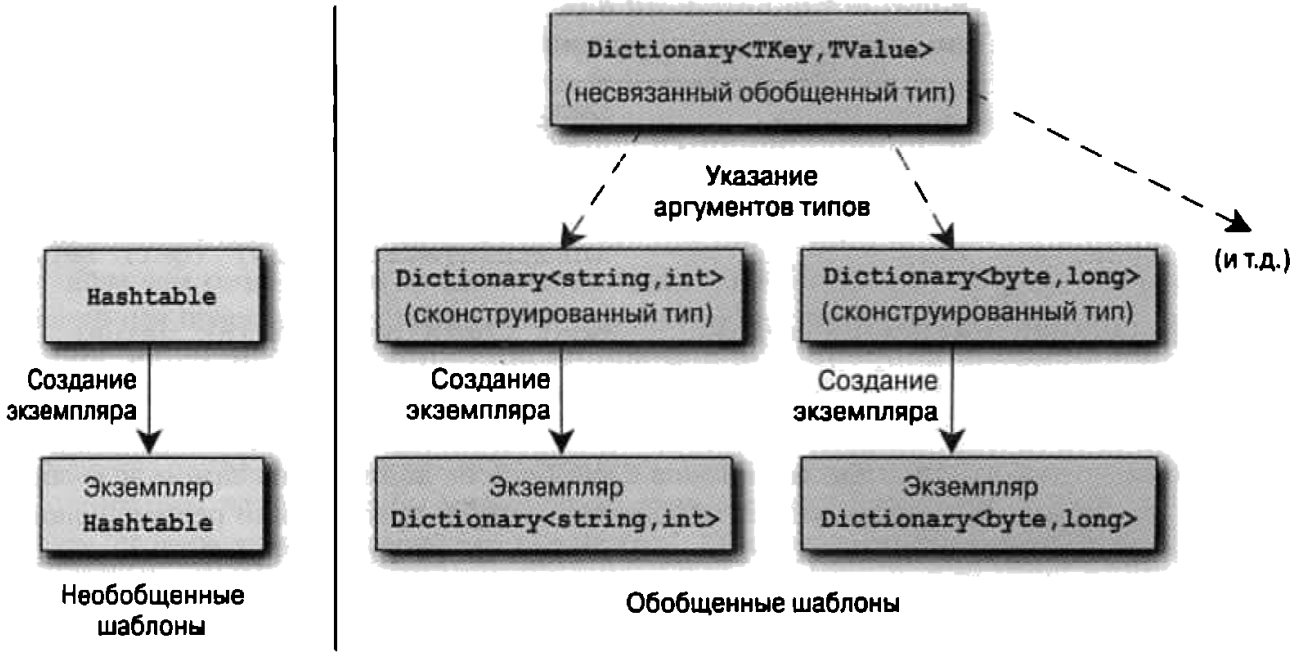

Рис. 3.1. Несвязанные обобщенные типы выступают в качестве шаблонов для сконструированных типов, которые затем служат шаблонами для действительных объектов, в точности как это делают необобщенные типы

В качестве дополнительной сложности, типы могут быть открытыми (open) или закрытыми (closed). *Открытый тип* — это такой тип, который по-прежнему содержит какой-то параметр типа (например, как один из аргументов типов или как тип элементов массива), а *закрытый тип* представляет собой тип, не являющийся открытым, т.е. каждый аспект типа точно известен. В действительности весь код *выполняется* в контексте закрытого сконструированного типа. Единственный раз, когда в коде C# можно встретить несвязанный обобщенный тип (кроме места его объявления) — внутри операции `typeof`, которая рассматривается в разделе 3.4.4.

Смысл параметра типа, "получающего" информацию, и аргумента типа, "предоставляющего" эту информацию (пунктирные линии на рис. 3.1), в точности соответствует смыслу, заложенному в параметры и аргументы метода, хотя аргументы типов должны быть типами, а не просто произвольными значениями. Аргумент типа должен быть известен на этапе компиляции, но он может быть (или содержать в себе) параметр типа из связанного контекста.

Закрытый тип можно представлять себе как тип, имеющий API-интерфейс открытого типа, в котором параметры типов заменены соответствующими им аргументами типов[2]. В табл. 3.1 приведен ряд объявлений открытых методов и свойств из открытого типа `Dictionary<TKey,TValue>`, а также эквивалентные члены из построенного на его основе закрытого типа — `Dictionary<string,int>`.

[2] Это не всегда работает в точности так, как описано — существуют краевые случаи, нарушающие указанное простое правило, — но так проще всего представлять работу обобщений в подавляющем большинстве ситуаций.

Таблица 3.1. Примеры наличия заполнителей в обобщенных типах, которые заменяются в случае указания аргументов типов

Сигнатура метода в обобщенном типе	Сигнатура метода после замены параметров типов
`void Add(TKey key, TValue value)`	`void Add(string key, int value)`
`TValue this[TKey key] { get; set; }`	`int this[string key] {get; set;}`
`bool ContainsValue(TValue value)`	`bool ContainsValue(int value)`
`bool ContainsKey(TKey key)`	`bool ContainsKey(string key)`

Важно отметить, что ни один из методов в табл. 3.1 не является на самом деле обобщенным. Это обычные методы внутри обобщенного типа, просто было решено использовать в них параметры типов, объявленные в виде части типа. Обобщенные методы будут рассматриваться в следующем разделе.

Теперь, когда вы знаете, что означают `TKey` и `TValue`, а также для чего предназначены угловые скобки, можно взглянуть, как объявления из табл. 3.1 будут выглядеть в определении класса. Ниже показано, каким образом может выглядеть код для `Dictionary<TKey,TValue>`, хотя действительные реализации методов опущены, а в реальности может быть определено намного больше членов:

```
namespace System.Collections.Generic
{
  public class Dictionary<TKey,TValue>          ← Объявление обобщенного класса
   : IEnumerable<KeyValuePair<TKey,TValue>>     ← Реализация обобщенного интерфейса
  {
    public Dictionary() { ... }                 ← Объявление метода с применением параметров типов
    public void Add(TKey key, TValue value) { ... }   ← Объявление конструктора без параметров
    public TValue this[TKey key]
    {
      get { ... }
      set { ... }
    }
    public bool ContainsValue(TValue value) { ... }
    public bool ContainsKey(TKey key) { ... }
    [... другие члены ...]
  }
}
```

Обратите внимание на то, что класс `Dictionary<TKey,TValue>` реализует обобщенный интерфейс `IEnumerable<KeyValuePair<TKey,TValue>>` (а в реальности также многие другие интерфейсы). Любые аргументы типов, которые вы указываете для класса, применяются к интерфейсу, в котором используются те же самые параметры типов, поэтому в приведенном примере класс `Dictionary<string,int>` будет реализовывать интерфейс `IEnumerable<KeyValuePair<string,int>>`. В какой-то мере это дважды обобщенный интерфейс — он представляет собой интерфейс `IEnumerable<T>` со структурой `KeyValuePair<string,int>` в качестве аргумента типа. Именно по причине реализации этого интерфейса в коде из листинга 3.1 была возможность выполнять перечисление по ключам и значениям.

Стоит также отметить, что параметры типов в конструкторе не содержат угловые скобки. Параметры типов являются частью *типа*, а не отдельного конструктора, поэтому именно в типе они и объявлены. В членах объявляются только новые параметры типов — и это разрешено делать только методам.

Устное описание обобщений

Если вам когда-нибудь придется рассказывать об обобщениях коллегам, то знайте, что для представления параметров или аргументов типов принято применять предлог "из" — например, `List<T>` вслух можно назвать "списком из элементов типа `T`". В VB англоязычный вариант этого предлога ("of") является частью самого языка: тип может быть записан как `List(Of T)`. При наличии множества параметров типов я считаю разумным разделять их посредством слова, которое соответствует назначению всего типа, поэтому, чтобы акцентировать внимание на отображении, я использую фразу "*словарь*, отображающий `string` на `int`", а не "*кортеж* из типов `string` и `int`".

Обобщенные типы могут быть перегружены на основе количества параметров типов, так что допустимо определять `MyType`, `MyType<T>`, `MyType<T,U>`, `MyType<T,U,V>` и так далее, причем в рамках одного и того же пространства имен. При перегрузке имена параметров типов во внимание не принимаются — важно только их количество. Эти типы не связаны друг с другом за исключением имени — например, стандартное преобразование одного типа в другой не существует. То же самое справедливо и в отношении обобщенных методов: сигнатуры двух методов могут отличаться только по количеству параметров типов. Хотя это может звучать как рецепт для обеспечения катастрофы, прием удобен, когда нужно воспользоваться *выведением обобщенного типа*, при котором компилятор может самостоятельно определить ряд аргументов типов. Мы вернемся к этой теме в разделе 3.3.2.

Соглашения об именовании для параметров типов

Несмотря на *возможность* объявления типа с параметрами типов `T`, `U` и `V`, такие имена мало говорят об их предназначении или способе применения. Сравните это с `Dictionary<TKey,TValue>`, где вполне очевидно, что `TKey` представляет тип ключей, а `TValue` — тип значений. Когда имеется единственный параметр типа с понятным назначением, по соглашению используется имя `T` (хорошим примером служит `List<T>`). Множество параметров типов обычно должны именоваться согласно своему назначению, с применением префикса `T` для указания, что они являются параметрами типов. Время от времени можно встретить тип с несколькими однобуквенными параметрами типов (например, `SynchronizedKeyedCollection<K,T>`), но вы должны стараться сами не создавать такого рода ситуации.

Теперь, когда вы уловили идею, положенную в основу обобщенных типов, давайте взглянем на обобщенные методы.

3.2.3. Обобщенные методы и чтение обобщенных объявлений

Обобщенные методы уже несколько раз упоминались, но пока еще не приводилось ни одного примера. Вы можете счесть общую идею обобщенных методов более запутанной, чем в случае обобщенных типов — они менее естественны для восприятия, — но в

них заложен тот же самый базовый принцип. Мы привыкли, что параметры и возвращаемое значение метода имеют четко указанные типы, и вы видели, что обобщенный тип может использовать свои параметры типов в объявлениях методов. Обобщенные методы делают еще один шаг вперед: даже если точно не известно, с каким сконструированным типом придется иметь дело, отдельный метод может также содержать параметры типов. Не переживайте, если это пока еще не понятно — по мере рассмотрения примеров в какой-то момент концепция станет ясной.

Тип `Dictionary<TKey,TValue>` не содержит обобщенных методов, но его близкий сосед, `List<T>`, содержит. Как и можно было вообразить, `List<T>` — это список элементов любого указанного типа; например, `List<string>` представляет собой список строк. Памятуя о том, что `T` является параметром типа для целого *класса*, давайте разберем на части объявление обобщенного метода. На рис. 3.2 показаны различные части объявления метода `ConvertAll()`.

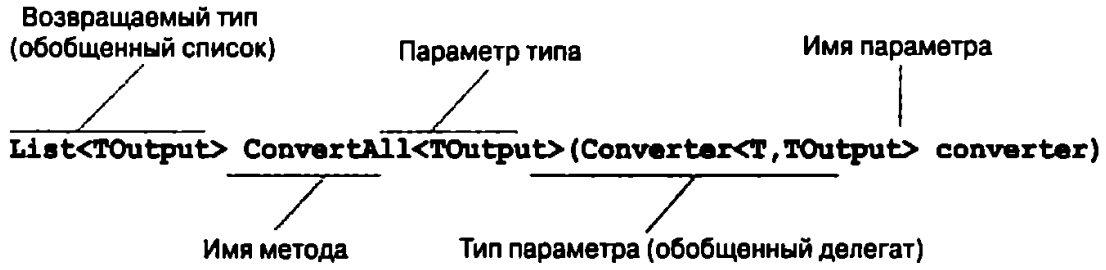

Рис. 3.2. Структура объявления обобщенного метода

Глядя на объявление обобщенного типа или обобщенного метода, может быть непросто понять, что оно означает, особенно если приходится иметь дело с обобщенными типами обобщенных типов, как это было в случае интерфейса, реализованного словарем. Важно не впадать в панику — воспринять все хладнокровно и подобрать пример ситуации. Используйте для каждого параметра типа разные типы и применяйте их последовательно. В данном случае давайте начнем с замены параметра типа в типе, который содержится в методе (часть `<T>` в `List<T>`). Мы будем придерживаться концепции списка строк, поэтому заменим `T` типом `string` везде в объявлении метода:

```
List<TOutput> ConvertAll<TOutput>(Converter<string,TOutput> converter)
```

Это выглядит немного лучше, но по-прежнему приходится иметь дело с `TOutput`. Можно сказать, что `TOutput` является параметром типа метода (прошу прощения за сбивающую с толку терминологию), т.к. он находится в угловых скобках прямо после имени метода, поэтому давайте попробуем указать в качестве аргумента типа для `TOutput` другой известный тип — `Guid`. Мы снова должны заменить везде этот параметр типа упомянутым аргументом типа. Теперь метод можно рассматривать, как если бы он был необобщенным, удалив часть с параметром типа из объявления:

```
List<Guid> ConvertAll(Converter<string,Guid> converter)
```

Итак, все выражено в терминах конкретного типа, что упрощает его восприятие. Несмотря на то что реальный метод является обобщенным, ради лучшего понимания мы будем считать, что это не так. Пройдемся по элементам этого объявления слева направо:

- метод возвращает `List<Guid>`;
- имя метода выглядит как `ConvertAll`;
- метод принимает единственный параметр по имени `converter` с типом `Converter<string,Guid>`.

Осталось только узнать, что собой представляет тип `Converter<string,Guid>`. Неудивительно, что `Converter<string,Guid>` является сконструированным *обобщенным типом делегата* (несвязанный обобщенный тип выглядит как `Converter <TInput,TOutput>`), который используется для преобразования строки в GUID-идентификатор.

Таким образом, имеется метод, который оперирует на списке строк, применяя конвертер для построения списка GUID-идентификаторов. Разобравшись с сигнатурой метода, намного проще понять документацию, которая подтверждает, что указанный метод создает новый список `List<Guid>`, преобразует каждый элемент исходного списка в целевой тип, добавляет его в новый список и затем возвращает итоговый список. Представление сигнатуры с помощью конкретных терминов дает более чистую умозрительную модель и упрощает обдумывание того, что может делать этот метод. Хотя такой прием может показаться чрезмерно простым, я нахожу его удобным в отношении сложных методов даже сейчас. Сигнатуры некоторых методов LINQ, имеющих по четыре параметра типов, выглядят настоящими свирепыми животными, но помещение в рамки конкретных терминов значительно укрощает их.

Просто чтобы доказать, что я не водил вас за нос, давайте посмотрим на метод `ConvertAll()` в действии. В листинге 3.2 демонстрируется преобразование списка целых чисел в список чисел с плавающей точкой, причем каждый элемент во втором списке представляет собой квадратный корень из значения соответствующего элемента в первом списке. После преобразования результаты выводятся на консоль.

Листинг 3.2. Метод `List<T>.ConvertAll<TOutput>()` в действии

```
static double TakeSquareRoot(int x)
{
  return Math.Sqrt(x);
}
...
List<int> integers = new List<int>();
integers.Add(1);
integers.Add(2);
integers.Add(3);
integers.Add(4);
Converter<int,double> converter = TakeSquareRoot;
List<double> doubles;
doubles = integers.ConvertAll<double>(converter);

foreach (double d in doubles)
{
  Console.WriteLine(d);
}
```

❶ Создание и заполнение списка целых чисел

❷ Создание экземпляра делегата

❸ Вызов обобщенного метода для преобразования списка

Создание и заполнение списка ❶ довольно прямолинейно — это всего лишь строго типизированный список целых чисел. В присваивании `converter` ❷ используется возможность, поддерживаемая делегатами (преобразования групп методов), которая появилась в C# 2 и обсуждается более подробно в разделе 5.2. Несмотря на то что мне не нравится применять средство до того, как оно будет полностью описано, отмечу, что использование синтаксиса делегатов C# 1 привело бы к получению слишком длинной строки, не умещающейся в печатную страницу. Тем не менее, средство выполняет вполне ожидаемое действие. Здесь вызывается обобщенный метод ❸ с указанием аргумента

типа таким же образом, как это делалось для обобщенных типов. Это одна из ситуаций, когда можно было бы применить выведение типа, чтобы избежать явного указания аргумента типа, но я предпочитаю совершать по одному шагу за раз. Вывод на консоль возвращенного списка очень прост, и после запуска кода вы увидите вполне ожидаемые значения 1, 1.414..., 1.732... и 2.

В чем смысл всего этого? Разумеется, мы могли бы просто с помощью цикла `foreach` пройтись по списку целых чисел и непосредственно вывести значения их квадратных корней. Однако необходимость в преобразовании списка элементов одного типа в список элементов другого типа, с выполнением в отношении них определенной логики, возникает не так уж редко. Код, используемый для реализации этого вручную, довольно прост, но версия, которая делает это посредством единственного вызова метода, легче в восприятии. Это часто бывает с обобщенными методами — они нередко выполняют действия, которые ранее благополучно делались "длинным способом", но с помощью вызова метода они оказываются проще. До появления обобщений в классе `ArrayList` существовала операция похожая на `ConvertAll()`, которая осуществляла преобразование из `object` в `object`, однако она была значительно менее удачной. Анонимные методы (описанные в разделе 5.4) также здесь помогут — если вводить дополнительный метод нежелательно, преобразование можно указать встроенным образом. Как будет показано в части III книги, LINQ и лямбда-выражения дополнительно развивают этот шаблон.

Обратите внимание, что обобщенные методы могут также быть частью необобщенных типов. В листинге 3.3 обобщенный метод объявляется и применяется внутри обычного необобщенного типа.

Листинг 3.3. Реализация обобщенного метода внутри необобщенного типа

```
static List<T> MakeList<T>(T first, T second)
{
  List<T> list = new List<T>();
  list.Add(first);
  list.Add(second);
  return list;
}
...
List<string> list = MakeList<string>("Line 1", "Line 2");
foreach (string x in list)
{
  Console.WriteLine (x);
}
```

Обобщенному методу `MakeList<T>` необходим только один параметр типа (`T`). Метод строит список, содержащий два параметра. Стоит отметить, что при создании списка `List<T>` в методе можно использовать `T` в качестве аргумента типа. Как и при описанном ранее анализе обобщенных объявлений, реализацию можно рассматривать (грубо говоря) как замену всех вхождений `T` типом `string`. Для вызова метода применяется тот же самый синтаксис, который вы видели ранее при указании аргументов типов.

Пока все в порядке? Теперь вы должны понимать простые обобщения. Боюсь, что дальше будет несколько сложнее, но если вы уловили основную идею обобщений, то тем самым преодолели самое большое препятствие. Не переживайте, если все еще не полностью ясно (особенно когда дело доходит до терминов "открытый/закрытый/несвязанный/сконструированный"); сейчас самое время поэкспериментировать, чтобы взглянуть на обобщения в действии, прежде чем двигаться вперед. Если вы не имели

дело с обобщенными коллекциями ранее, можете просмотреть приложение Б, в котором описаны доступные варианты. Типы коллекций обеспечивают простую отправную точку для работы с обобщениями и широко используются практически в каждой нетривиальной программе .NET.

Во время экспериментов вы можете обнаружить, что с трудом пройдена только часть пути. После превращения одной части API-интерфейса в обобщенную часто возникает необходимость в переписывании другого кода, чтобы сделать его также обобщенным или чтобы добавить приведения, которые требуются вызовами новых более строго типизированных методов. Альтернативой может быть строго типизированная реализация, использующая внутри обобщенные классы, но оставляющая на время слабо типизированный API-интерфейс. Со временем вы обретете большую уверенность в том, когда уместно применять обобщения.

3.3. Дополнительные сведения

Приведенных до этого момента относительно простых случаев использования обобщений может хватить надолго, но есть еще несколько средств, которые помогут в дальнейшем продвижении.

Мы начнем с рассмотрения *ограничений типов*, которые предоставляют больший контроль над тем, какие аргументы типов могут быть указаны. Ограничения типов полезны при создании собственных обобщенных типов и методов, и вы должны также понимать их, чтобы знать доступные варианты при работе с инфраструктурой.

После этого мы перейдем к ознакомлению с *выведением типа* — удобным приемом компилятора, который позволяет во время применения обобщенных методов не указывать явно аргументы типов. Вы не обязаны использовать выведение типа, однако при должном применении оно может существенно упростить чтение кода. Как будет показано в части III книги, компилятору C# постепенно разрешается выводить намного больше информации на основе кода, при этом язык по-прежнему остается безопасным и статически типизированным[3].

В последней части этого раздела пойдет речь о получении стандартного значения параметра типа и о сравнениях, доступных при написании обобщенного кода. В завершение будет приведен пример, который демонстрирует большинство рассмотренных средств и представляет собой удобный класс сам по себе.

Хотя в данном разделе приходится немного глубже погружаться в тему обобщений, с ними не связано ничего *действительно* сложного. Есть много, чего понадобится запомнить, но все эти средства служат одной цели, и когда в них возникнет нужда, вы оцените их по достоинству. Итак, приступим.

3.3.1. Ограничения типов

Все рассмотренные до сих пор параметры типов могут быть применены вообще к любому типу — они являются *неограниченными*. Можно иметь `List<int>`, `Dictionary<object,FileMode>` — что угодно. Это нормально при работе с коллекциями, которые не должны взаимодействовать со своим содержимым, но не все случаи использования обобщений выглядят подобным образом. Часто требуется вызывать методы на экземплярах параметра типа, создавать их новые экземпляры или обеспечивать прием

[3] Во всяком случае, за исключением кода C# 4, в котором явно используется динамическая типизация.

только ссылочных типов (либо только типов значений). Другими словами, необходимо указать правила, которые определяют, какие аргументы типов считаются допустимыми для обобщенного типа или метода. В C# 2 это делается с помощью *ограничений*.

Доступны четыре вида ограничений, имеющих общий синтаксис. Ограничения задаются в конце объявления обобщенного метода или типа с применением контекстного ключевого слова `where`. Как вы увидите позже, они могут объединяться вместе удобными способами. Однако сначала мы по очереди исследуем все виды ограничений.

Ограничения ссылочного типа

Ограничение первого вида позволяет обеспечить, что используемый аргумент типа является ссылочным типом. Оно выражается как `T : class` и должно быть первым ограничением, указанным для данного параметра типа. Аргумент типа может быть любым классом, интерфейсом, массивом, делегатом или другим параметром типа, о котором известно, что он относится к ссылочному типу. Например, взгляните на следующее объявление:

```
struct RefSample<T> where T : class
```

Допустимыми закрытыми типами, применяющими это объявление, являются:

- `RefSample<IDisposable>`
- `RefSample<string>`
- `RefSample<int[]>`

А вот недопустимые закрытые типы:

- `RefSample<Guid>`
- `RefSample<int>`

Я умышленно сделал `RefSample` структурой (т.е. типом значения), чтобы подчеркнуть разницу между ограниченным параметром типа и самим типом. `RefSample<string>` по-прежнему представляет собой тип значения с повсеместной семантикой значения – просто так получилось, что в коде применяется тип `string` везде, где указано `T`.

Когда параметр типа ограничен таким способом, можно сравнивать ссылки (включая `null`) с помощью операций `==` и `!=`, но знайте, что в отсутствие любых ограничений будут сравниваться только ссылки, даже если в рассматриваемом типе указанные операции перегружены (как это делано, например, в типе `string`). Посредством ограничения преобразования типа (описано ниже) можно получить в свое распоряжение *гарантированные компилятором* перегруженные версии операций `==` и `!=`, которые в этом случае и будут использоваться, однако такая ситуация встречается относительно редко.

Ограничения типа значения

Ограничение типа значения, выражаемое как `T : struct`, позволяет обеспечить, что применяемый аргумент типа является типом значения, включая перечисления. Однако сюда не входят типы, допускающие `null` (по причинам, указанным в главе 4). Рассмотрим следующий пример объявления:

```
class ValSample<T> where T : struct
```

Ниже перечислены допустимые закрытые типы:

- `ValSample<int>`
- `ValSample<FileMode>`

К недопустимым закрытым типам относятся:

- `ValSample<object>`
- `ValSample<StringBuilder>`

На этот раз `ValSample` — это ссылочный тип, несмотря на то, что `T` органичен типом значения. Обратите внимание, что `System.Enum` и `System.ValueType` сами по себе являются ссылочными типами, так что они не могут выступать в качестве допустимых аргументов типов для `ValSample`. Когда к параметру типа применено ограничение типа значения, сравнение с использованием операций == и != запрещено.

Сам я редко применяю ограничения типа значения или ссылочного типа, хотя в следующей главе вы увидите, что типы значений, допускающие `null`, полагаются на ограничения типа значения. Оставшиеся два ограничения, скорее всего, окажутся более полезными при написании собственных обобщенных типов.

Ограничения конструктора типа

Ограничение конструктора типа выражается как `T : new()` и должно указываться *последним* для любого отдельного параметра типа. Оно просто проверяет, что используемый аргумент типа имеет конструктор без параметров, который может применяться для создания экземпляра. Это относится к любому типу значений, любому нестатическому неабстрактному классу, не имеющему явно объявленных конструкторов, а также любому неабстрактному классу с явно определенным открытым конструктором, который не принимает параметров.

Сравнение стандартов C# и CLI

Между стандартами C# и CLI существует несоответствие, касающееся типов значений и конструкторов. В спецификации C# утверждается, что все типы значений имеют стандартный конструктор без параметров, а для вызова явно объявленных конструкторов и стандартного конструктора без параметров в языке применяется одинаковый синтаксис, при этом выбор правильной версии возлагается на компилятор. В спецификации CLI не содержится такого требования, но предоставляется специальная инструкция для создания стандартного значения без указания параметров. Увидеть это несоответствие можно во время использования рефлексии для поиска конструкторов типа значения — конструктор без параметров не обнаружится.

Давайте снова обратимся к короткому примеру, на этот раз для метода. Просто чтобы показать, насколько это удобно, приведена также и реализация метода:

```
public T CreateInstance<T>() where T : new()
{
  return new T();
}
```

Этот метод возвращает новый экземпляр любого указанного типа при условии, что в нем имеется конструктор без параметров. Это означает, что вызовы `CreateInstance<int>()` и `CreateInstance<object>()` допустимы, но вызов `CreateInstance<string>()` — нет, т.к. в типе `string` конструктор без параметров отсутствует.

Не существует способа ограничения параметров типов для принудительного применения других сигнатур конструктора. Например, невозможно указать, что должен

присутствовать конструктор, принимающий единственный строковый параметр. К большому разочарованию это так. Мы рассмотрим данную проблему более подробно, когда будем исследовать различные ограничения обобщений .NET в разделе 3.5. Ограничения конструктора типа могут быть удобны, когда нужно использовать фабричные шаблоны, при которых один объект будет создавать другой объект в случае возникновения в нем необходимости. Фабрики часто должны генерировать объекты, совместимые с определенным интерфейсом, и здесь в действие вступает последний вид ограничений.

Ограничения преобразования типа

Последний (и наиболее сложный) вид ограничений позволяет указывать другой тип, в который аргумент типа должен поддерживать неявное преобразование посредством преобразования идентичности, ссылочного преобразования или упаковывающего преобразования. Допустимо также определять, что один аргумент типа должен иметь возможность преобразования в другой аргумент типа — это называется *ограничением параметра типа*. Такие ограничения затрудняют понимание объявления, но временами они оказываются удобными. В табл. 3.2 приведены примеры объявлений обобщенного типа с ограничениями преобразования типа, а также допустимые и недопустимые варианты соответствующих сконструированных типов.

Третье ограничение в табл. 3.2, `T : IComparable<T>`, является лишь одним примером применения обобщенного типа в качестве ограничения. Вполне подойдут и другие вариации, такие как `T : List<U>` (где `U` — другой параметр типа) и `T : IList<string>`.

Таблица 3.2. Примеры ограничений преобразования типа

Объявление	Примеры сконструированных типов
`class Sample<T>` `  where T : Stream`	*Допустимый*: `Sample<Stream>` (преобразование идентичности)
	Недопустимый: `Sample<string>`
`struct Sample<T>` `  where T : IDisposable`	*Допустимый*: `Sample<SqlConnection>` (ссылочное преобразование)
	Недопустимый: `Sample<StringBuilder>`
`class Sample<T>` `  where T : IComparable<T>`	*Допустимый*: `Sample<int>` (упаковывающее преобразование)
	Недопустимый: `Sample<FileInfo>`
`class Sample<T,U>` `  where T : U`	*Допустимый*: `Sample<Stream,IDisposable>` (ссылочное преобразование)
	Недопустимый: `Sample<string,IDisposable>`

Можно указывать несколько интерфейсов, но только один класс. Например, следующее объявление вполне допустимо (хотя удовлетворить его трудно):

```
class Sample<T> where T : Stream,
                      IEnumerable<string>,
                      IComparable<int>
```

Но такое объявление не годится:

```
class Sample<T> where T : Stream,
                      ArrayList,
                      IComparable<int>
```

Поскольку в любом случае тип не может быть унаследован напрямую от более чем одного класса, такое ограничение обычно либо невозможно (подобно представленному выше), либо его часть будет избыточной (например, указание на то, что тип должен быть производным от `Stream` и `MemoryStream`).

Существует еще один набор ограничений: указываемый тип не может быть типом значения, запечатанным классом (таким как `string`) или любым из следующих "специальных" типов:

- `System.Object`
- `System.Enum`
- `System.ValueType`
- `System.Delegate`

Обход отсутствия ограничений с типами перечислений и делегатов

Кажется, что невозможность указания типов перечислений и делегатов в ограничениях преобразования типа связана с ограничением среды CLR, но это не так. Это может быть верным с исторической точки зрения (в какой-то момент, когда обобщения все еще находились на стадии разработки), но если написать соответствующий код на IL, он будет нормально работать. В спецификации CLI даже приводятся ограничения с типами перечислений и делегатов в качестве примеров вместе с объяснениями, что будет допустимым, а что — нет. Это разочаровывает, т.к. есть немало обобщенных методов, которые бы выиграли в случае ограничения типами делегатов или перечислений. Я веду проект с открытым кодом под названием Unconstrained Melody (`http://code.google.com/p/unconstrained-melody/`), в котором с помощью ряда трюков строится библиотека классов, *имеющая* упомянутые ограничения на разнообразных служебных методах. Хотя компилятор C# не позволит объявить такие ограничения, они будут благополучно применены при вызове этих методов из библиотеки. Возможно, этот запрет будет снят в будущей версии языка C#.

Ограничения преобразования типа, пожалуй, можно считать наиболее удобным видом, т.к. они означают возможность использования членов указанного типа на экземплярах параметра типа. Особенно полезным примером может служить ограничение `T : IComparable<T>`, которое позволяет сравнивать два экземпляра типа `T` осмысленным и непосредственным образом. Пример такого сравнения (а также обсуждение других форм сравнений) приводится в разделе 3.3.3.

Объединение ограничений

Ранее уже упоминалось о возможности наличия множества ограничений, и вы видели это в действии для ограничений преобразования типа, однако пока еще не было показано, как объединять вместе ограничения разных видов. Очевидно, что тип не может быть одновременно ссылочным типом и типом значения, поэтому такая комбинация запрещена. Подобным же образом, *каждый* тип значения располагает конструктором без параметров, так что указать ограничение конструктора, когда уже имеется ограничение типа значения, не получится (хотя по-прежнему можно применять ограничение `new T()` внутри методов, если `T` ограничен типом значения). При наличии нескольких ограничений преобразования типа, одним из которых является `class`, это ограничение должно находиться перед интерфейсами — кроме того, один и тот же интерфейс не должен быть

указан более одного раза. Разные параметры типов могут иметь различные ограничения, каждое из которых вводится с помощью отдельного ключевого слова `where`.

Давайте рассмотрим ряд примеров допустимых и недопустимых объявлений.

Допустимые объявления:

```
class Sample<T> where T : class, IDisposable, new()
class Sample<T> where T : struct, IDisposable
class Sample<T,U> where T : class where U : struct, T
class Sample<T,U> where T : Stream where U : IDisposable
```

Недопустимые объявления:

```
class Sample<T> where T : class, struct
class Sample<T> where T : Stream, class
class Sample<T> where T : new(), Stream
class Sample<T> where T : IDisposable, Stream
class Sample<T> where T : XmlReader, IComparable, IComparable
class Sample<T,U> where T : struct where U : class, T
class Sample<T,U> where T : Stream, U : IDisposable
```

Последний пример объявления был включен в каждый список потому, что очень просто получить недопустимую версию объявления вместо допустимой, и сообщение компилятора об ошибке мало чем поможет. Просто запомните, что каждый список ограничений для параметра типа должен указываться посредством собственного ключевого слова `where`. Третий пример допустимого объявления довольно интересен; если `U` — тип значения, то как он может быть производным от `T`, который является ссылочным типом? Ответ: типом `T` может быть `object` либо интерфейс, который `U` реализует. Хотя, конечно, это довольно неуклюжее ограничение.

Терминология, применяемая в спецификации

Спецификация разбивает ограничения на категории несколько по-другому; в ней предусмотрены *первичные* ограничения, *вторичные* ограничения и ограничения *конструктора*. Первичное ограничение — это ограничение ссылочного типа, ограничение типа значения или ограничение преобразования типа, использующее класс. Вторичное ограничение представляет собой ограничение преобразования типа, в котором применяется интерфейс или другой параметр типа. Я не считаю такое разделение особенно удобным, но эти категории упрощают определение грамматики ограничений: первичное ограничение является необязательным, но его допускается иметь только одно; вторичных ограничений может быть сколько угодно; ограничение конструктора также необязательно (если только не используется ограничение типа значения, в случае чего ограничение конструктора запрещено).

Теперь, когда вам известно все, что необходимо для чтения объявлений обобщенных типов, давайте рассмотрим выведение аргументов типов, которое упоминалось ранее. В листинге 3.2 аргументы типов для метода `List<T>.ConvertAll()` были заданы явно, и то же самое делалось в листинге 3.3 в отношении метода `MakeList()`. А сейчас мы предложим компилятору поработать за нас, когда это возможно, упрощая вызов обобщенных методов.

3.3.2. Выведение типов для аргументов типов в обобщенных методах

Указание аргументов типов при вызове обобщенного метода часто выглядит несколько избыточным. Обычно на основе самих аргументов метода можно очевидно сказать, к каким типам они должны относиться. Для упрощения разработки в C# 2 и последующих версиях компилятору разрешено проявлять свои интеллектуальные способности рядом четко определенных способов, что позволяет вызывать метод, не указывая явно аргументы типов. Однако прежде чем двигаться дальше, я должен подчеркнуть, что это справедливо только для обобщенных *методов*, но не для обобщенных *типов*.

Давайте взглянем на соответствующие строки в листинге 3.3 и посмотрим, как их можно упростить. Вот строки с объявлением и вызовом метода:

```
static List<T> MakeList<T>(T first, T second)
...
List<string> list = MakeList<string>("Line 1", "Line 2");
```

Как видите, оба аргумента являются строками. Каждый параметр в методе объявлен с типом `T`. Даже если бы часть `<string>` выражения вызова метода отсутствовала, было бы вполне очевидным намерение вызывать метод с применением `string` в качестве аргумента типа для `T`. Компилятор позволяет не указывать эту часть, оставляя только код:

```
List<string> list = MakeList("Line 1", "Line 2");
```

Немного аккуратнее, не так ли? Во всяком случае, короче. Разумеется, это не всегда означает, что код окажется более читабельным. В некоторых ситуациях читателю кода будет труднее выяснить, какие аргументы типов должны использоваться, хотя компилятор делает это легко. Я рекомендую судить о каждом случае по существу. Лично я предпочитаю, чтобы компилятор выводил аргументы типов в *большинстве* ситуаций, где это срабатывает.

Компилятору точно известно, что мы используем `string` в качестве аргумента типа, поскольку присваивание `list` тоже работает, а в нем *указан* аргумент типа (и это должно быть сделано). Однако присваивание не оказывает влияния на процесс выведения типов аргументов. Это просто означает, что если компилятор примет неправильное решение относительно того, какие аргументы типов нужны, то вы, скорее всего, получите ошибку на этапе компиляции.

Но как компилятор может ошибиться? Предположим, что в действительности вы хотите использовать `object` для аргумента типа. Параметры метода по-прежнему допустимы, но компилятор считает, что вы рассчитываете на тип `string`, т.к. в вызове присутствуют две строки. Явное приведение одного из параметров к `object` нарушит работу выведения типов, поскольку один из аргументов метода ожидает, что типом `T` должен быть `string`, а другой — что типом `T` должен быть `object`. Компилятор *мог* бы с учетом этого решить, что установка `T` в `object` удовлетворит всем требованиям, а установка `T` в `string` — нет, однако в спецификации предусмотрено только ограниченное количество шагов, которым нужно следовать. Эта тема довольно сложна в C# 2 и даже больше усложняется в C# 3. Я не буду пытаться охватить здесь все тонкости правил C# 2, но ниже перечислены базовые шаги.

1. Для каждого аргумента метода (в круглых, а не угловых скобках) попытаться вывести некоторые аргументы типов из обобщенного метода, используя относительно простые приемы.

2. Проверить, что все результаты, полученные на первом шаге, являются согласованными. Другими словами, если для отдельного параметра типа один аргумент подразумевает применение одного аргумент типа, а другой — другого аргумента типа, не совпадающего с первым, то выведение для вызова метода дает отказ.
3. Проверить, что были выведены все параметры типов, необходимые для обобщенного метода. Не допускается разрешать компилятору выводить одни типы, но указывать другие явно — выведение должно осуществляться либо для всех аргументов типов, либо не использоваться вообще.

Чтобы не изучать все правила (к тому же я не рекомендую тратить на это время, если только вы специально не интересуетесь мельчайшими подробностями), то вот вам простая рекомендация, как действовать: пробуйте и смотрите, что в результате происходит. Если вы думаете, что компилятор *может* быть способен вывести все аргументы типов, попытайтесь вызвать метод, не указывая ни одного из них. Если это не удалось, указывайте аргументы типов явно. Вы потеряете не более чем время, отнимаемое однократной компиляцией, но зато не придется держать в голове массу лишней информации.

Для упрощения работы с обобщенными типами прием выведения типов можно комбинировать с идеей перегрузки имен типов на основе количества параметров типов. Пример будет приведен через некоторое время, когда мы начнем собирать все вместе.

3.3.3. Реализация обобщений

Скорее всего, вы будете больше времени уделять использованию обобщенных типов и методов, чем их самостоятельному написанию. Даже когда вы предоставляете реализацию, обычно можно предположить, что `T` (или любое другое имя, назначенное параметру типа) является именем типа, и писать код, как если бы обобщения не применялись вообще. Тем не менее, есть несколько дополнительных моментов, которые вы должны знать.

Выражения для стандартных значений

Когда тип, с которым предстоит работа, известен точно, то известно и его стандартное значение — значение, которое будет иметь неинициализированное поле, например. Однако если этот тип не известен, то указать стандартное значение напрямую невозможно. Использовать значение `null` нельзя, потому что это может быть не ссылочный тип. Не получится также применить значение `0`, поскольку тип может оказаться нечисловым.

Стандартное значение требуется довольно редко, но время от времени это может быть удобно. Хорошим примером служит тип `Dictionary<TKey,TValue>` — он имеет метод `TryGetValue()`, который работает подобно методам `TryParse()` числовых типов: использует выходной параметр для извлекаемого значения и возвращаемое значение булевского типа для указания, успешно ли прошло извлечение. Это означает, что метод должен располагать некоторым значением типа `TValue` для заполнения выходного параметра. (Вспомните, что выходным параметрам необходимо обязательно присвоить значения, прежде чем метод сможет нормально завершиться.)

Как раз для этого в C# 2 предлагается *выражение для стандартного значения*. В спецификации оно не называется операцией, но вы можете считать его похожим на операцию `typeof`, которая просто возвращает другое значение.

Шаблон TryXXX()

Несколько шаблонов в .NET легко идентифицировать по именам связанных с ними методов — скажем, `BeginXXX()` и `EndXXX()` указывают на асинхронную операцию. Шаблон `TryXXX()` является одним из тех, применение которого охватывало версии .NET 1.1 и .NET 2.0. Он предназначен для ситуаций, которые могут обычно рассматриваться как ошибки (в том, что метод не может выполнить свою основную работу), но при которых отказ может также не свидетельствовать о наличии серьезной проблемы и не считаться исключительным. Например, пользователи часто вводят числа некорректно, поэтому возможность *попробовать* разобрать какой-то текст, не приводя к генерации и перехвату исключения, весьма полезна. Она не только улучшает производительность в случае сбоя, но — и это более важно — оставляет механизм исключения для подлинных ошибочных случаев, когда что-то пошло не так в самой *системе* (независимо от того, насколько широко вы хотите интерпретировать это). При должном применении данный шаблон удобен для разработчиков библиотек.

В листинге 3.4 демонстрируется использование выражения для стандартного значения в обобщенном методе, а также приводится пример выведения типов и ограничения преобразования типа в действии.

Листинг 3.4. Сравнение заданного значения со стандартным значением в обобщенном методе

```
static int CompareToDefault<T>(T value)
  where T : IComparable<T>
{
  return value.CompareTo(default(T));
}
...
Console.WriteLine(CompareToDefault("x"));
Console.WriteLine(CompareToDefault(10));
Console.WriteLine(CompareToDefault(0));
Console.WriteLine(CompareToDefault(-10));
Console.WriteLine(CompareToDefault(DateTime.MinValue));
```

В листинге 3.4 показан обобщенный метод, который вызывается с тремя разными типами: `string`, `int` и `DateTime`. Метод `CompareToDefault()` диктует условие о том, что он может применяться только с типами, которые реализуют интерфейс `IComparable<T>`, позволяя вызывать `CompareTo(T)` на переданном значении. Другим значением, используемым при сравнении, является стандартное значение для типа. Поскольку `string` — ссылочный тип, стандартное значение равно `null`, а документация по методу `CompareTo()` гласит, что в случае ссылочных типов любое значение должно быть больше `null`, и первым результатом оказывается `1`. В следующих трех строках представлены сравнения со стандартным значением `int`, демонстрирующие, что это стандартное значение равно `0`. Последняя строка выводит `0`, указывая на то, что стандартным значением типа `DateTime` является `DateTime.MinValue`.

Конечно, метод в листинге 3.4 даст отказ в случае передачи `null` в качестве аргумента — строка с вызовом `CompareTo()` приведет к генерации исключения `NullReferenceException`. Пока что не беспокойтесь об этом — как вскоре вы узнаете, существует альтернатива применению интерфейса `IComparer<T>`.

Прямые сравнения

Хотя в листинге 3.4 было показано, что сравнение возможно, не всегда хочется ограничивать свои типы реализацией интерфейса `IComparable<T>` или родственного ему интерфейса `IEquatable<T>`, который предоставляет строго типизированный метод `Equals(T)` для дополнения метода `Equals(object)`, имеющегося во всех типах. Без дополнительной информации, к которой обеспечивают доступ эти интерфейсы, в отношении сравнений можно делать немногим более чем просто вызов `Equals(object)`, приводящий в результате к упаковке сравниваемого значения, когда оно является типом значения. (Для помощи в некоторых ситуациях предусмотрена пара типов — мы вскоре обратимся к ним.)

Если параметр типа является неограниченным (т.е. никаких ограничений к нему не применено), можно использовать операции == и !=, но *только* для сравнения значения этого типа с `null`; сравнивать два значения типа `T` друг с другом нельзя. Если аргумент типа относится к ссылочному типу, будет применяться нормальное ссылочное сравнение. В случае, когда аргумент типа, предоставленный для `T`, является типом значения, не допускающим `null`, сравнение с `null` будет всегда давать неравенство (поэтому такое сравнение может быть удалено JIT-компилятором). Когда же аргумент типа представляет собой тип значения, допускающий `null`, сравнение будет вести себя естественным образом, обеспечивая сравнение со значением `null` этого типа[4]. (Не беспокойтесь, если последнее утверждение пока что не понятно — это прояснится во время чтения следующей главы. К сожалению, некоторые средства слишком переплетены, чтобы можно было описать одно из них без ссылки на другое.)

Когда параметр типа ограничен типом значения, операции == и != не могут использоваться вообще. Если он ограничен ссылочным типом, то вид выполняемого сравнения зависит от того, каким образом ограничен параметр типа. Если единственное ограничение заключается в том, что параметр типа является ссылочным типом, выполняются простые ссылочные сравнения. Если он дополнительно ограничен, чтобы быть производным от определенного типа, в котором операции == и != перегружены, применяются эти перегруженные версии. Однако будьте осторожны — дополнительные перегруженные версии операций, которые иногда делаются доступными посредством аргумента типа, указанного в вызывающем коде, *не* используются. Сказанное продемонстрировано в листинге 3.5 на примере простого ограничения ссылочного типа и аргумента типа `string`.

Листинг 3.5. Применение операций == и !=, выполняющих ссылочные сравнения

```
static bool AreReferencesEqual<T>(T first, T second)
  where T : class
{
  return first == second;     <-- Сравнение ссылок
}
...
string name = "Jon";
```

[4] На момент написания этой главы (материал проверялся в .NET 4.5 и предшествующих версиях) код, генерируемый JIT-компилятором для сравнения значений неограниченных параметров типов с `null`, исключительно медленно работал в случае типов значений, допускающих `null`. Если ограничить параметр типа `T` как не допускающий `null`, и затем сравнивать значение типа `T?` с `null`, то такое сравнение будет выполняться намного быстрее. Это демонстрирует поле деятельности для будущей JIT-оптимизации.

```
string intro1 = "My name is " + name;
string intro2 = "My name is " + name;
Console.WriteLine(intro1 == intro2);    ←❷ Сравнение с использованием перегруженной версии операции в типе string
Console.WriteLine(AreReferencesEqual(intro1, intro2));
```

Несмотря на то что в типе `string` операция `==` перегружена (что подтверждается выводом `True` в точке ❶), эта перегруженная версия не используется сравнением в точке ❷. В сущности, при компиляции типа `AreReferencesEqual<T>` компилятору ничего не известно о том, какие перегруженные версии будут доступны — как если бы передаваемые параметры имели тип `object`.

Такое поведение не является специфичным для операций. Встретив обобщенный тип, компилятор распознает все перегруженные версии методов при компиляции несвязанного обобщенного типа, не пересматривая каждый возможный вызов метода на предмет наличия более специфичных перегруженных версий во время выполнения. Например, оператор `Console.WriteLine(default(T));` всегда будет давать в результате вызов `Console.WriteLine(object value)`, а не вызов `Console.WriteLine(string value)`, когда для `T` указан тип `string`. Это напоминает обычную ситуацию с выбором перегруженных версий на этапе компиляции, а не во время выполнения, тем не менее, читатели, знакомые с шаблонами в языке C++, могут быть удивлены[5].

При сравнении значений *исключительно* полезны классы `EqualityComparer<T>` и `Comparer<T>`, определенные в пространстве имен `System.Collections.Generic`. Они реализуют, соответственно, интерфейсы `IEqualityComparer<T>` и `IComparer<T>`, и свойство `Default` возвращает реализацию, которая в целом делает то, что нужно для подходящего типа.

Обобщенные интерфейсы для сравнений

Существуют четыре главных обобщенных интерфейса для сравнений. Два из них — `IComparer<T>` и `IComparable<T>` — предназначены для сравнения значений на предмет *порядка* (является ли одно значение меньшим, равным или большим другого значения). Другие два интерфейса — `IEqualityComparer<T>` и `IEquatable<T>` — позволяют сравнивать элементы на предмет *эквивалентности* согласно определенному критерию и вычислять хеш-код элемента (способом, совместимым с тем же самым понятием эквивалентности).

Разделяя эти четыре интерфейса по-другому, следует отметить, что `IComparer<T>` и `IEqualityComparer<T>` реализуются типами, которые обладают способностью сравнения двух разных значений, тогда как экземпляр типа, реализующего интерфейс `IComparable<T>` или `IEquatable<T>`, позволяет сравнивать *себя самого* с другим значением.

За дополнительными деталями обращайтесь в документацию и подумайте об использовании упомянутых выше классов (и подобных типов, таких как `StringComparer`) при выполнении сравнений. Тип `EqualityComparer<T>` применяется в следующем примере.

[5] В главе 14 будет показано, что динамическая типизация предоставляет возможность распознавания перегруженных версий во время выполнения.

Полноценный пример сравнения: представление пары значений

В завершение раздела, посвященного реализации обобщений, рассмотрим полноценный пример. В нем реализуется удобный обобщенный тип `Pair<T1,T2>`, который хранит два значения вместе подобно паре "ключ/значение", но не ожидает наличия между ними отношения.

Платформа .NET 4 и кортежи

Платформа .NET 4 предлагает большой объем такой функциональности в готовом виде — равно как и для множества разных количеств параметров типов. Взгляните на `Tuple<T1>`, `Tuple<T1,T2>` и тому подобные типы в пространстве имен `System`.

Кроме предоставления свойств для доступа к самим значениям будут переопределены методы `Equals()` и `GetHashCode()`, чтобы позволить экземплярам этого типа успешно использоваться в качестве ключей внутри словаря. В листинге 3.6 приведен полный код.

Листинг 3.6. Обобщенный класс, представляющий пару значений

```
using System;
using System.Collections.Generic;
public sealed class Pair<T1, T2> : IEquatable<Pair<T1, T2>>
{
  private static readonly IEqualityComparer<T1> FirstComparer =
    EqualityComparer<T1>.Default;
  private static readonly IEqualityComparer<T2> SecondComparer =
    EqualityComparer<T2>.Default;
  private readonly T1 first;
  private readonly T2 second;
  public Pair(T1 first, T2 second)
  {
    this.first = first;
    this.second = second;
  }
  public T1 First { get { return first; } }
  public T2 Second { get { return second; } }
  public bool Equals(Pair<T1, T2> other)
  {
    return other != null &&
      FirstComparer.Equals(this.First, other.First) &&
      SecondComparer.Equals(this.Second, other.Second);
  }
  public override bool Equals(object o)
  {
    return Equals(o as Pair<T1, T2>);
  }
  public override int GetHashCode()
  {
    return FirstComparer.GetHashCode(first) * 37 +
      SecondComparer.GetHashCode(second);
  }
}
```

Код в листинге 3.6 довольно прямолинеен. Образующие пару значения хранятся в подходяще типизированных переменных-членах, а доступ к ним предоставляется с помощью простых свойств, предназначенных только для чтения. Интерфейс `IEquatable<Pair<T1,T2>>` реализован с целью предоставления строго типизированного API-интерфейса, который позволит избежать нежелательных проверок во время выполнения. Сравнения эквивалентности и вычисления хеш-кодов используют стандартный компаратор эквивалентности для двух параметров типов — они автоматически поддерживают значения `null`, что несколько упрощает код. Статические переменные, хранящие компараторы эквивалентности для `T1` и `T2`, применяются по большей части ради того, чтобы уместить код в печатную страницу, однако они также будут полезны в качестве опорной точки в следующем разделе.

Вычисление хеш-кодов

Формула, используемая для вычисления хеш-кода, основана на двух "частичных" результатах, описанных в книге Джошуа Блоха *Effective Java*, 2-е изд. (Addison-Wesley, 2008 г.). Она определенно не гарантирует хорошее распределение хеш-кодов, но, по моему мнению, эта формула все же лучше, чем применение операции побитового "исключающего ИЛИ". За дополнительными сведениями обращайтесь к указанной книге; там можно обнаружить также и многие другие полезные советы и приемы.

Как теперь создать экземпляр построенного класса `Pair`? В данный момент понадобится записать примерно так:

```
Pair<int,string> pair = new Pair<int,string>(10, "value");
```

Выглядит не особенно хорошо. Было бы неплохо использовать выведение типов, однако оно работает только для обобщенных методов, а их нет. Если поместить обобщенный метод в обобщенный тип, то перед тем, как этот метод можно будет вызвать, по-прежнему придется указывать аргументы типов для обобщенного типа, что нивелирует всю цель. Решение заключается в том, чтобы применить необобщенный вспомогательный класс с обобщенным методом внутри, как показано в листинге 3.7.

Листинг 3.7. Использование необобщенного типа с обобщенным методом для обеспечения выведения типов

```
public static class Pair
{
  public static Pair<T1,T2> Of<T1,T2>(T1 first, T2 second)
  {
    return new Pair<T1,T2>(first, second);
  }
}
```

Если вы читаете эту книгу впервые, проигнорируйте пока тот факт, что класс объявлен статическим — мы вернемся к этому вопросу в главе 7. Важно то, что имеется необобщенный класс с обобщенным методом. Это означает, что предыдущий пример можно превратить в намного более удачную версию:

```
Pair<int,string> pair = Pair.Of(10, "value");
```

В C# 3 удалось бы даже обойтись без явной типизации переменной `pair`, но давайте не забегать вперед. Такое применение необобщенных вспомогательных классов (или *частично* обобщенных вспомогательных классов, если есть два или больше параметров типов и необходимо, чтобы одни из них выводились, а другие указывались явно) является удобным трюком.

На этом рассмотрение промежуточных средств завершено. Я понимаю, что все они поначалу могут показаться сложными, но не пугайтесь: преимущества обобщений намного перевешивают привносимую ими сложность. Со временем обобщения станут вашей второй натурой. Имея класс `Pair` в качестве примера, полезно просмотреть собственный код на предмет наличия в нем каких-нибудь шаблонов, которые приходилось реализовывать исключительно для использования разных типов.

В любой крупной теме всегда есть, что изучать дополнительно. В следующем разделе будут рассмотрены наиболее важные сложные вопросы, связанные с обобщениями. Если вы испытываете трудности в текущий момент, переходите сразу к разделу 3.5, в котором исследуются недостатки обобщений. Разбираться в темах, поднятых в последующем разделе, *в конечном итоге* полезно, но если все описанное до этого момента оказалось для вас в диковинку, то не повредит пока пропустить данный раздел.

3.4. Дополнительные темы, связанные с обобщениями

Возможно, вы ожидаете, что в оставшихся разделах этой главы будут раскрыты все аспекты обобщений, которые не были показаны до сих пор. Однако с обобщениями связано настолько много мелких углов и закоулков, что это попросту невозможно – я определенно не хотел бы читать обо всех деталях, а тем более писать о них. К счастью, усилиями специалистов из Microsoft и ECMA все эти детали были описаны в спецификации языка, так что если вы хотите прояснить непонятную ситуацию, которая здесь не была рассмотрена, обращайтесь к спецификации. Увы, я не могу указать на какую-то одну область внутри спецификации, где были бы раскрыты обобщения: они встречаются повсюду. Можно аргументировано утверждать, что если вы столкнулись при написании кода с настолько сложным краевым случаем, что приходится обращаться к спецификации, то, скорее всего, вы должны, так или иначе, привести его в более очевидную форму. Вряд ли вы хотите, чтобы специалист по сопровождению кода постоянно рылся в спецификации, отыскивая мельчайшие подробности.

Цель, которую я преследую в этом разделе – раскрыть все, что, по всей видимости, вы хотите узнать относительно обобщений. Речь пойдет больше о среде CLR и инфраструктурной стороне обобщений, чем о конкретном синтаксисе языка C# 2, хотя все это важно при разработке на C#. Мы начнем с рассмотрения статических членов обобщенных типов, включая инициализацию типов. После этого вполне естественно задаться вопросом, как все это реализовано "за кулисами", но мы не будем вдаваться в особые детали, сосредоточив внимание на важных последствиях решений при реализации. Мы посмотрим, что происходит при перечислении обобщенной коллекции с применением цикла `foreach` в C# 2, а в завершение раздела взглянем, каким образом рефлексия в .NET Framework воздействует на обобщения.

3.4.1. Статические поля и статические конструкторы

Подобно тому, как поля экземпляра принадлежат экземпляру, статические поля принадлежат типу, в котором они объявлены. Если вы объявили статическое поле x в классе SomeClass, существует в точности одно поле SomeClass.x независимо от того, сколько экземпляров SomeClass будет создано и сколько типов будет унаследовано от класса SomeClass[6]. Это знакомый сценарий из C# 1 — но как он отображается на обобщения?

Ответ в том, что каждый *закрытый* тип имеет собственный набор статических полей. Это можно было видеть в листинге 3.6, когда стандартные компараторы эквивалентности для T1 и T2 сохранялись в статических полях, но давайте проанализируем это более подробно на другом примере. В листинге 3.8 создается обобщенный тип, включающий статическое поле. Затем значение этого поля устанавливается для разных закрытых типов и результирующие значения выводятся на консоль с целью демонстрации, что они разные.

Листинг 3.8. Доказательство того, что разные закрытые типы имеют разные статические поля

```
class TypeWithField<T>
{
  public static string field;
  public static void PrintField()
  {
    Console.WriteLine(field + ": " + typeof(T).Name);
  }
}
...
TypeWithField<int>.field = "First";
TypeWithField<string>.field = "Second";
TypeWithField<DateTime>.field = "Third";

TypeWithField<int>.PrintField();
TypeWithField<string>.PrintField();
TypeWithField<DateTime>.PrintField();
```

Каждое поле устанавливается в отличающееся значение, после чего поля выводятся вместе с именем аргумента типа, использованного для этого закрытого типа. Ниже показан вывод, полученный в результате запуска кода из листинга 3.8:

```
First: Int32
Second: String
Third: DateTime
```

Базовое правило предусматривает наличие одного статического поля на закрытый тип. То же самое правило применяется для статических инициализаторов и статических конструкторов. Однако можно иметь один обобщенный тип, вложенный внутрь другого, и типы с множеством обобщенных параметров. Хотя это *выглядит* намного более сложным, работает оно вполне предсказуемо. Сказанное демонстрируется в действии в листинге 3.9, где с помощью статических конструкторов показано, сколько типов существует.

---

[6] Если придерживаться точности, то одно на домен приложения. В этом разделе мы будем предполагать, что имеем дело только с одним доменом приложения. Концепции для разных доменов приложений работают с обобщенными типами таким же образом, как с необобщенными. Переменные, декорированные атрибутом [ThreadStatic], тоже нарушают это правило.

Листинг 3.9. Статические конструкторы с вложенными обобщенными типами

```
public class Outer<T>
{
  public class Inner<U,V>
  {
    static Inner()
    {
      Console.WriteLine("Outer<{0}>.Inner<{1},{2}>",
                        typeof(T).Name,
                        typeof(U).Name,
                        typeof(V).Name);
    }
    public static void DummyMethod() {}
  }
}
...
Outer<int>.Inner<string,DateTime>.DummyMethod();
Outer<string>.Inner<int,int>.DummyMethod();
Outer<object>.Inner<string,object>.DummyMethod();
Outer<string>.Inner<string,object>.DummyMethod();
Outer<object>.Inner<object,string>.DummyMethod();
Outer<string>.Inner<int,int>.DummyMethod();
```

Первый вызов метода DummyMethod() для любого типа приведет к инициализации типа, во время которой статический конструктор выводит диагностическую информацию. Каждый отличающийся список аргументов типов считается другим закрытым типом, поэтому вывод из кода в листинге 3.9 выглядит следующим образом:

```
Outer<Int32>.Inner<String,DateTime>
Outer<String>.Inner<Int32,Int32>
Outer<Object>.Inner<String,Object>
Outer<String>.Inner<String,Object>
Outer<Object>.Inner<Object,String>
```

Как и в случае необобщенных типов, статический конструктор для любого закрытого типа выполняется только один раз. Именно поэтому последняя строка кода в листинге 3.9 не создает шестой строки вывода — статический конструктор для класса Outer<string>.Inner<int,int> выполнился ранее, создав вторую строку вывода.

Без сомнений, при наличии необобщенного класса PlainInner внутри Outer попрежнему был бы возможен только один тип Outer<T>.PlainInner на каждый закрытый тип Outer, так что тип Outer<int>.PlainInner оказался бы отдельным от Outer<long>.PlainInner, со своим набором статических полей, как было показано ранее.

Теперь, когда вы знаете, из чего образованы различные типы, мы должны проанализировать возможное влияние этого на объем генерируемого машинного кода. Все не так плохо, как вы могли подумать...

3.4.2. Обработка обобщений JIT-компилятором

Учитывая наличие всего этого разнообразия закрытых типов, работа JIT-компилятора заключается в преобразовании кода IL обобщенного типа в машинный код, который мо-

жет быть фактически выполнен. В определенном смысле вас не должно заботить то, как он в точности это делает. Не обращая внимания на объем памяти и время центрального процессора, вы не заметили бы особой разницы, если бы JIT-компилятор принял простейший из возможных подход и сгенерировал машинный код для каждого закрытого типа по отдельности, как будто бы все эти типы были независимыми друг от друга. Однако авторы JIT-компилятора оказались более искусными, поэтому полезно взглянуть на то, что они сделали.

Давайте начнем с простой ситуации с единственным параметром типа — ради удобства будем использовать `List<T>`. Компилятор JIT создает отдельный код для каждого закрытого типа с аргументом типа значения — `int`, `long`, `Guid` и т.д. Но при этом он разделяет машинный код, сгенерированный для всех закрытых типов, в которых в качестве аргумента типа применяется ссылочный тип, такой как `string`, `Stream` и `StringBuilder`. Компилятор JIT может это делать, поскольку все ссылки имеют один и тот же размер (размер варьируется в зависимости от того, какая среда CLR используется — 32-разрядная или 64-разрядная, но в рамках любой версии CLR все ссылки имеют одинаковые размеры). Массив ссылок будет всегда иметь один и тот же размер, какие бы ссылки он не содержал. Пространство в стеке, требуемое для ссылки, будет всегда одинаковым. Компилятор JIT может применять те же самые оптимизации для сохранения ссылок в регистрах независимо от типа — будь это даже `List<Reason>`.

Каждый тип по-прежнему располагает собственными статическими полями, как было описано в разделе 3.4.1, но исполняемый код используется повторно. Разумеется, JIT-компилятор делается все это "ленивым" образом, не генерируя кол для типа `List<int>` до того, как он потребуется, и будет кешировать данный код для всех будущих случаев применения `List<int>`.

Теоретически возможно разделять код, по крайней мере, для *некоторых* типов значений. Компилятор JIT должен также учитывать процесс сборки мусора и иметь возможность быстро идентифицировать области значения структур, которые содержат актуальные ссылки. Однако типы значений, которые имеют один и тот же размер и одинаковый отпечаток в памяти, насколько это касается сборщика мусора, *могли бы* разделять код. На момент написания книги указанная особенность имела довольно низкий приоритет, из-за чего и не была реализована, и вполне вероятно, что такая ситуация не изменится.

Хотя такой уровень детализации представляет, в первую очередь, академический интерес, следует отметить наличие небольшого влияния на производительность в смысле большего объема кода, подвергаемого JIT-компиляции. Но выигрыш в плане производительности, обеспечиваемый обобщениями, может оказаться огромным, и снова это сводится к обеспечению возможности компиляции разного кода для разных типов. Рассмотрим для примера тип `List<byte>`. В .NET 1.1 добавление отдельных байтов в `ArrayList` означало упаковку каждого из них и сохранение ссылки на упакованные значения. Использование `List<byte>` устраняет это — в типе `List<T>` предусмотрен член типа `T[]`, заменяющий `object[]` внутри `ArrayList`, и этот массив имеет подходящий тип, занимающий нужный объем пространства. В `List<byte>` для хранения элементов массива применяется обычный тип `byte[]`. (Это во многом определяет то, что тип `List<byte>` ведет себя подобно типу `MemoryStream`.)

В качестве иллюстрации на рис. 3.3 показаны типы `ArrayList` и `List<byte>`, каждая из которых содержит по шесть значений. В самих массивах присутствует больше шести элементов, чтобы обеспечить возможность разрастания. Оба типа `List<T>` и `ArrayList` имеют буфер, который при необходимости может быть создан и большего размера.

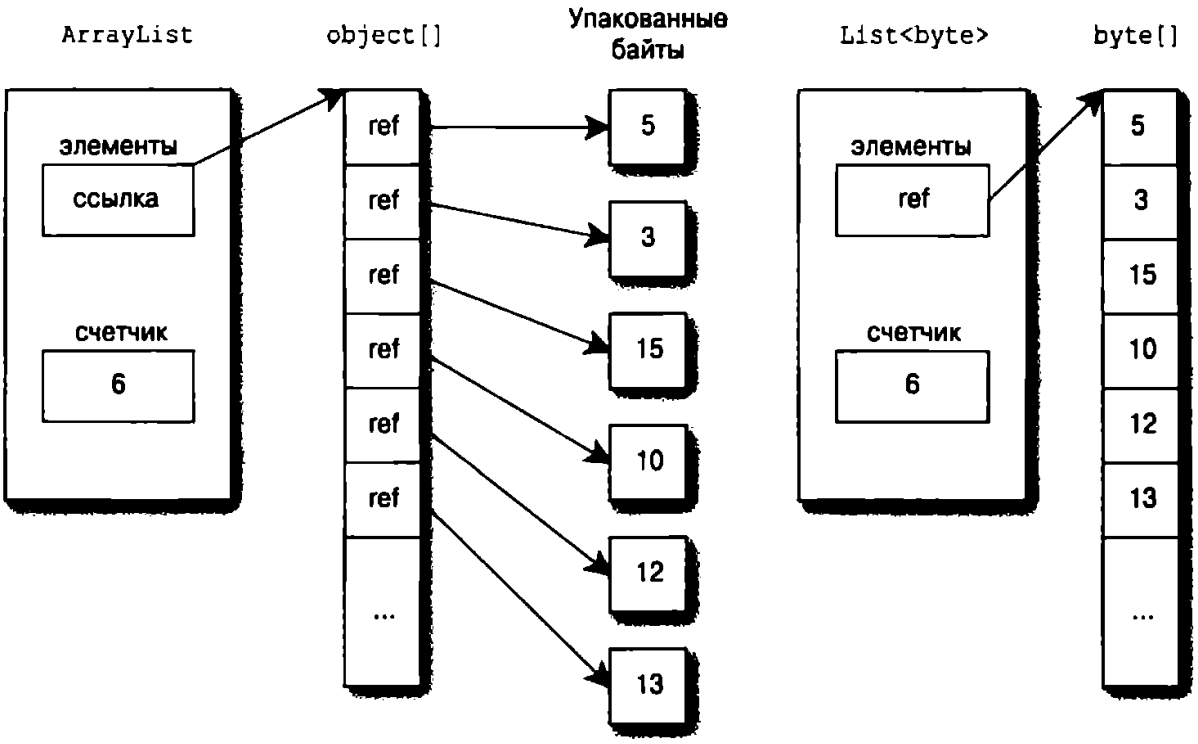

Рис. 3.3. Визуальная демонстрация причины того, что при сохранении типов значений `List<T>` требует намного меньше пространства, чем `ArrayList`

Разница в эффективности здесь невообразима. Давайте сначала рассмотрим тип `ArrayList`, предполагая, что используется 32-разрядная среда CLR[7]. Каждый упакованный байт потребует 8 байтов накладных расходов, связанных с объектом, и еще 4 байта (1 байт, выровненный по границе слова) для самих данных. В довершение всего, сами ссылки занимают по 4 байта каждая. Таким образом, для каждого байта полезных данных приходится расходовать, по меньшей мере, 16 байтов — а есть еще и дополнительное неиспользуемое пространство для ссылок в буфере.

Сравним это с типом `List<byte>`. Каждый байт в списке требует одного байта внутри массива элементов. По-прежнему имеется напрасно расходуемое пространство в буфере, которое ожидает появления новых элементов, но, во всяком случае, на неиспользуемые элементы здесь тратится всего по одному байту.

Сокращается не только объем занимаемого пространства, но также достигается выигрыш в скорости выполнения. Экономится время, необходимое для выделения памяти при упаковке, для проведения проверки типов во время распаковки байтов с целью их извлечения и для обработки упаковок сборщиком мусора, когда на них больше нет ссылок.

Тем не менее, для выяснения происходящего вовсе не обязательно обращаться к уровню среды CLR. С помощью синтаксических сокращений языка C# все значительно упрощается. В следующем разделе будет представлен уже знакомый пример, но с обобщенным трюком: итерация посредством `foreach`.

3.4.3. Обобщенная итерация

Одной из наиболее распространенных операций, которые понадобится выполнять над коллекцией, является итерация по всем ее элементам. Обычно самый простой способ ее реализации предусматривает применение оператора `foreach`. В C# 1 для этого кол-

[7] В случае 64-разрядной среды CLR накладные расходы еще выше.

лекция либо должна реализовывать интерфейс System.Collections.IEnumerable, либо располагать аналогичным методом GetEnumerator(), который возвращает тип с подходящим методом MoveNext() и свойством Current. Свойство Current не должно иметь тип object, вот поэтому определены такие дополнительные правила, выглядящие на первый взгляд несколько странными. Как видите, даже в C# 1 при наличии специального типа для итерации можно избежать упаковки и распаковки.

В C# 2 до некоторой степени это упрощается, т.к. правила для оператора foreach были расширены с целью использования также интерфейса System.Collections.Generic.IEnumerable<T> вместе с его партнером IEnumerator<T>. Они представляют собой просто обобщенные эквиваленты старых интерфейсов для итерации, и применять их предпочтительнее, чем необобщенные версии. Это означает, что при итерации по обобщенной коллекции элементов, имеющих типы значений (List<int>, например), никакой упаковки происходить не будет вообще. Если взамен воспользоваться старым интерфейсом, упаковка не будет выполняться при *сохранении* элементов в списке, но по-прежнему будет присутствовать при их извлечении в цикле foreach.

Все, что было описано выше, происходит "за кулисами" — вам придется лишь работать с оператором foreach нормальным образом, применяя переменную итерации подходящего типа. Тем не менее, история на этом не заканчивается. В относительно редких ситуациях, когда вам нужно реализовать итерацию по собственным типам, обнаружится, что интерфейс IEnumerable<T> расширяет старый интерфейс IEnumerable, а это означает необходимость реализации двух разных методов:

```
IEnumerator<T> GetEnumerator();
IEnumerator GetEnumerator();
```

Смогли уловить проблему? Эти методы отличаются только возвращаемым типом, а правила переопределения языка C# предотвращают написание подобных методов нормальным образом. В разделе 2.2.2 была показана похожая ситуация, и здесь можно воспользоваться тем же самым обходным приемом. Если вы реализуете IEnumerable посредством явной реализации интерфейсов, то сможете реализовать IEnumerable<T> с "нормальным" методом. К счастью, поскольку IEnumerator<T> расширяет IEnumerator, одно и то же возвращаемое значение можно применить для обоих методов и реализовать необобщенный метод, просто вызывая его обобщенную версию. Конечно, осталось еще реализовать IEnumerator<T>, так что вы быстро столкнетесь с аналогичными проблемами, но на этот раз со свойством Current.

В листинге 3.10 приведен полный пример реализации перечислимого класса, который всегда перечисляет целые числа от 0 до 9.

Листинг 3.10. Полноценный обобщенный итератор для чисел от 0 до 9

```
class CountingEnumerable: IEnumerable<int>
{
  public IEnumerator<int> GetEnumerator()          <-- 1 Реализация IEnumerable<T> неявным образом
  {
    return new CountingEnumerator();
  }

  IEnumerator IEnumerable.GetEnumerator()          <-- 2 Реализация IEnumerable явным образом
  {
    return GetEnumerator();
  }
}
```

```
class CountingEnumerator : IEnumerator<int>
{
  int current = -1;

  public bool MoveNext()
  {
    current++;
    return current < 10;
  }

  public int Current { get { return current; } }

  object IEnumerator.Current { get { return Current; } }

  public void Reset()
  {
    current = -1;
  }

  public void Dispose() {}
}
...
CountingEnumerable counter = new CountingEnumerable();

foreach (int x in counter)
{
  Console.WriteLine(x);
}
```

❹ Реализация IEnumerator<T>.Current неявным образом

❺ Реализация IEnumerator.Current явным образом

❻ Доказательство функционирования перечислимых типов

Ясно, что полученный результат не особенно полезен, но код демонстрирует, через какие, пусть и небольшие, испытания придется пройти для реализации обобщенной итерации должным образом — во всяком случае, если делать все собственноручно. (И это еще не учитывая затраты на генерацию исключений, если доступ к свойству `Current` происходит в ненадлежащее время.) Если вы считаете, что в листинге 3.10 содержится слишком много кода, как для простого вывода чисел от 0 до 9, то не могу не согласиться с вами; кода было бы даже больше, если бы понадобилось организовать итерацию по чему-то более полезному. К счастью, как будет показано в главе 6, во многих случаях C# 2 позволяет существенно сократить объем работ по реализации итераторов. В листинге представлена полная версия, чтобы вы смогли оценить мелкие недостатки, связанные с принятием решения о том, что интерфейс `IEnumerable<T>` должен расширять интерфейс `IEnumerable`. Тем не менее, я не утверждаю, что такое решение было ошибочным; это позволяет передавать любой тип `IEnumerable<T>` методу, который написан на C# 1 и принимает параметр типа `IEnumerable`. Хотя важность данного решения снизилась по сравнению с ситуацией 2005 года, оно по-прежнему является удобным способом перехода.

Потребуется только дважды применить трюк с явной реализацией интерфейса — один раз для метода `IEnumerable.GetEnumerator()` ❹ и один раз для свойства `IEnumerator.Current` ❺. В обоих случаях производится обращение к обобщенным эквивалентам (соответственно, ❶ и ❸). Еще одно добавление к `IEnumerator<T>` связано с тем, что он расширяет интерфейс `IDisposable`, поэтому вы должны предоставить метод `Dispose()`. Оператор `foreach` в C# 1 уже вызывает метод `Dispose()` на итераторе, если он реализует интерфейс `IDisposable`, однако в C# 2 никакие проверки

времени выполнения не требуются — если компилятор обнаружит, что был реализован интерфейс `IEnumerable<T>`, он предусмотрит безусловный вызов метода `Dispose()` в конце цикла (в блоке `finally`). Многим итераторам в действительности не нужно что-либо освобождать, но полезно знать, что когда это *требуется*, то самый распространенный способ работы с итератором (оператор `foreach` ❺) поддерживает освобождение автоматически. Чаще всего это используется для освобождения ресурсов после завершения итерации. Например, итератор может читать строки из файла и должен закрыть файловый дескриптор, когда вызывающий код завершит цикл.

А теперь переключимся с эффективности этапа компиляции на гибкость во время выполнения: последней дополнительной темой, которую мы рассмотрим, является рефлексия. Рефлексия могла оказаться запутанной даже в .NET 1.0/1.1, а обобщенные типы и методы привносят еще один уровень сложности. Все необходимое предоставляет инфраструктура (вместе с долей полезного синтаксиса со стороны C# 2 как языка), и хотя дополнительные соображения могут обескураживать, они не так уж страшны, если делать по одному шагу за раз.

3.4.4. Рефлексия и обобщения

Рефлексия применяется для самых разнообразных целей. Она может использоваться для самоанализа объектов во время выполнения при построении простой разновидности привязки данных. С помощью рефлексии можно инспектировать каталог со сборками в поисках нужного интерфейса подключаемого модуля. Можно было бы написать файл для инфраструктуры инверсии управления (`www.martinfowler.com/articles/injection.html`), который позволит загружать и динамически конфигурировать компоненты приложения. Ввиду широкого разнообразия применений рефлексии я не буду концентрироваться на каком-то одном случае, а предоставлю более общее руководство по решению распространенных задач. Начнем с рассмотрения расширений для операции `typeof`.

Использование `typeof` с обобщенными типами

Рефлексия позволяет исследовать объекты и их типы. В сущности, одним из самых важных действий, которые должна быть возможность предпринять, является получение ссылки на отдельно взятый объект `System.Type`, обеспечивающий доступ ко всей информации о данном типе. Чтобы получить такую ссылку для типов, известных на этапе компиляции, в C# применяется операция `typeof`, которая была расширена с целью охвата обобщенных типов.

Существуют два способа использования операции `typeof` с обобщенными типами: первый извлекает *определение обобщенного типа* (другими словами, несвязанный обобщенный тип), а второй — отдельный сконструированный тип. Чтобы получить определение обобщенного типа, т.е. тип без указания аргументов типов, необходимо просто взять имя типа, как он был объявлен, и удалить имена параметров типов, сохранив запятые. Для получения сконструированного типа нужно указать аргументы типов таким же образом, как это делается при объявлении переменной обобщенного типа. В листинге 3.11 приведен пример обоих способов. Здесь используется обобщенный метод, поэтому мы можем снова посмотреть, как применяется операция `typeof` для параметра типа, что уже было показано в листинге 3.8.

Листинг 3.11. Применение операции typeof к параметрам типов

```
static void DemonstrateTypeof<X>()
{
  Console.WriteLine(typeof(X));                    <-- Отображение параметра типа из метода
  Console.WriteLine(typeof(List<>));               <-- Отображение обобщенных типов
  Console.WriteLine(typeof(Dictionary<,>));
  Console.WriteLine(typeof(List<X>));              <-- ❶ Отображение закрытых типов (несмотря
                                                       на использование параметра типа)
  Console.WriteLine(typeof(Dictionary<string,X>));
  Console.WriteLine(typeof(List<long>));           <-- Отображение закрытых типов
  Console.WriteLine(typeof(Dictionary<long,Guid>));
}
...
DemonstrateTypeof<int>();
```

Большая часть кода в листинге 3.11 функционирует так, как и можно было ожидать, но стоит обратить внимание на два аспекта. Во-первых, взгляните на синтаксис, используемый для получения определения обобщенного типа `Dictionary<TKey,TValue>`. Запятая в угловых скобках требуется для того, чтобы сообщить компилятору о необходимости поиска типа с двумя параметрами типов; вспомните, что можно объявить несколько обобщенных типов с одним и тем же именем при условии, что они будут отличаться по количеству своих параметров типов. Аналогично, определение обобщенного типа для `MyClass<T1,T2,T3,T4>` извлекается посредством `typeof(MyClass<,,,>)`. Количество параметров типов указывается в коде IL (и в полных именах типов внутри инфраструктуры) за счет помещения после первой части имени типа символа обратного апострофа и числа. Параметры типов задаются в квадратных, а не в угловых скобках. Например, вторая строка вывода заканчивается на `List'1[T]`, показывая, что имеется один параметр типа, а третья строка включает `Dictionary'2[TKey,TValue]`.

Во-вторых, обратите внимание, что везде, где применяется параметр типа метода (`X`), во время выполнения используется действительное значение аргумента типа. Поэтому строка ❶ выводит `List'1[System. Int32]`, а не `List'1[X]`, как вы, возможно, ожидали[8]. Другими словами, тип, являющийся открытым на этапе компиляции, во время выполнение может быть закрытым. *Все это слишком запутанно. Если ожидаемые результаты не получены, то вы должны знать об этом, но в противном случае ни о чем беспокоиться не придется.* Получение по-настоящему открытого сконструированного типа во время выполнения требует немного большей работы. В документации MSDN по типу `Type.IsGenericType` (`http://msdn.microsoft.com/en-us/library/system.type.isgenerictype.aspx`) приведен довольно запутанный пример.

Для справки ниже показан вывод кода из листинга 3.11:

```
System.Int32
System.Collections.Generic.List'1[T]
System.Collections.Generic.Dictionary'2[TKey,TValue]
System.Collections.Generic.List'1[System.Int32]
System.Collections.Generic.Dictionary'2[System.String,System.Int32]
System.Collections.Generic.List'1[System.Int64]
System.Collections.Generic.Dictionary'2[System.Int64,System.Guid]
```

[8] Я сознательно нарушил соглашение о применении параметра типа под названием `T` именно для того, чтобы можно было указать на разницу между `T` в объявлении `List<T>` и `X` в объявлении метода.

После получения над объектом, представляющим обобщенный тип, можно проделать множество действий. Все доступные ранее действия (нахождение членов типа, создание экземпляра и т.д.) по-прежнему возможны (хотя некоторые из них неприменимы к определениям обобщенных типов), и вдобавок существуют новые действия, которые позволяют исследовать обобщенную природу типа.

Методы и свойства класса `System.Type`

В классе `System.Type` имеется слишком много новых методов и свойств, чтобы всех их можно было рассмотреть здесь подробно, но следующие два метода особенно важны: `GetGenericTypeDefinition()` и `MakeGenericType()`. Они фактически являются противоположностями: первый действует на сконструированном типе, извлекая определение обобщенного типа, а второй — на определении обобщенного типа, возвращая сконструированный тип. Вероятно, было бы яснее, если бы второй метод назывался `ConstructType()`, `MakeConstructedType()` либо носил еще какое-нибудь имя, содержащее в себе слово *Construct* или *Constructed*, но мы должны довольствоваться тем, что имеем.

Как и у нормальных типов, для каждого отдельного типа предусмотрен только один объект `Type`, поэтому двукратный вызов метода `MakeGenericType()` с передачей ему одних и тех же типов в качестве аргументов каждый раз возвратит ту же самую ссылку. Подобным же образом, вызовы метода `GetGenericTypeDefinition()` на двух типах, сконструированных из одного и того же определения обобщенного типа, дадут тот же самый результат в обоих случаях, даже если сконструированные типы отличаются (например, `List<int>` и `List<string>`).

Полезно также исследовать два других метода, на этот раз уже существующих в .NET 1.1 — `Type.GetType(string)` и связанный с ним `Assembly.GetType(string)`; они оба предоставляют динамический эквивалент операции `typeof`. Вы могли ожидать, что каждую строку вывода кода из листинга 3.11 можно было бы передать методу `GetType()`, вызванному на подходящей сборке, однако в реальности, к сожалению, все не так просто. В случае закрытых сконструированных типов это сработает — нужно просто поместить аргументы типов в квадратные скобки. Тем не менее, для определений обобщенных типов понадобится удалить квадратные скобки вообще — иначе метод `GetType()` сочтет, что вы имели в виду тип массива. В листинге 3.12 все эти методы демонстрируются в действии.

Листинг 3.12. Разнообразные способы извлечения обобщенных и сконструированных объектов `Type`

```
string listTypeName = "System.Collections.Generic.List`1";

Type defByName = Type.GetType(listTypeName);

Type closedByName = Type.GetType(listTypeName + "[System.String]");
Type closedByMethod = defByName.MakeGenericType(typeof(string));
Type closedByTypeof = typeof(List<string>);

Console.WriteLine(closedByMethod == closedByName);
Console.WriteLine(closedByName == closedByTypeof);

Type defByTypeof = typeof(List<>);
Type defByMethod = closedByName.GetGenericTypeDefinition();

Console.WriteLine(defByMethod == defByName);
Console.WriteLine(defByName == defByTypeof);
```

Вывод, получаемый в результате выполнения кода из листинга 3.12, содержит четыре значения `True`, подтверждая, что несмотря на получение ссылки на определенный объект типа, существует только один такой объект.

Как упоминалось ранее, в классе `Type` появилось много новых методов и свойств, таких как `GetGenericArguments()`, `IsGenericTypeDefinition` и `IsGenericType`. Опять-таки, документация по свойству `IsGenericType` является, пожалуй, лучшей отправной точкой для дальнейших исследований.

Рефлексия обобщенных методов

Обобщенные методы имеют похожий (хотя и меньшего размера) набор дополнительных свойств и методов. В листинге 3.13 представлена краткая демонстрация этого на примере вызова обобщенного метода с помощью рефлексии.

Листинг 3.13. Извлечение и вызов обобщенного метода посредством рефлексии

```
public static void PrintTypeParameter<T>()
{
  Console.WriteLine(typeof(T));
}
...
Type type = typeof(Snippet);
MethodInfo definition = type.GetMethod("PrintTypeParameter");
MethodInfo constructed = definition.MakeGenericMethod(typeof(string));
constructed.Invoke(null, null);
```

Сначала извлекается определение обобщенного метода, после чего с использованием `MakeGenericMethod()` создается сконструированный обобщенный метод. Что касается типов, при желании можно было бы пойти другим путем, но в отличие от `Type.GetType()`, в вызове `GetMethod()` указать сконструированный метод не удастся. В инфраструктуре также будет возникать проблема, если методы переопределены исключительно по количеству параметров типов — тип `Type` не располагает методами, которые позволили бы указывать количество параметров типов, так что вместо этого понадобится вызвать `Type.GetMethods()` и найти нужный метод, просматривая *все* методы.

После извлечения сконструированный метод вызывается. В этом примере оба аргумента равны `null`, поскольку вызывается статический метод, не имеющий нормальных параметров. Как и можно было ожидать, выводится `System.String`. Обратите внимание, что методы, извлеченные из определений обобщенных типов, не могут быть вызваны напрямую — вместо этого придется получать методы из сконструированного типа. Сказанное применимо и к обобщенным, и к необобщенным методам.

Спасение в лице C# 4

Соглашусь с тем, что все это выглядит как полнейший беспорядок. К счастью, во многих случаях на выручку приходит динамическая типизация C#, беря на себя большую часть работы по рефлексии обобщений. Она не помогает абсолютно во всех ситуациях, поэтому полезно быть в курсе общей идеи приведенного ранее кода, однако там, где динамическая типизация *применима*, она обеспечивает великолепные результаты. Динамическая типизация будет подробно рассматриваться в главе 14.

В классе `MethodInfo` доступно множество других методов и свойств. Хорошей отправной точкой может послужить свойство `IsGenericMethod`, описанное в документации MSDN по адресу `http://msdn.microsoft.com/en-gb/library/system.reflection.methodinfo.isgenericmethod.aspx`. Надеюсь, что в этом разделе было предоставлено достаточно информации для успешного начала работы и указаны дополнительные сложности, которые невозможно было предугадать, впервые приступая к доступу к обобщенным типам и методам посредством рефлексии.

На этом рассмотрение дополнительных возможностей завершено. Просто повторюсь: эта глава никоим образом не претендует на то, чтобы служить полным руководством по обобщениям, но большинству разработчиков вряд ли понадобится знать все мелкие детали. Я надеюсь, что к вам это тоже относится, т.к. чем глубже приходится вникать в спецификации, тем труднее их становится читать. Помните, что если только вы не занимаетесь разработкой самостоятельно и только для себя, то вряд ли будете единственным, кто работает с вашим кодом. Если вам нужны средства, более сложные по сравнению с продемонстрированными здесь, вы должны взять на себя ответственность за то, что любому разработчику, читающему ваш код, потребуется помощь в его понимании. С другой стороны, если обнаружится, что ваши коллеги не разбираются в некоторых темах, раскрытых до сих пор, не стесняйтесь рекомендовать им настоящую книгу...

В последнем основном разделе этой главы описаны недостатки обобщений в C#, а также аналогичные возможности в других языках.

3.5. Недостатки обобщений в C# и сравнение с другими языками

Вне всяких сомнений, обобщения внесли большой вклад в язык C# в плане выразительности, безопасности типов и производительности. Данное средство было тщательно спроектировано для того, чтобы справиться с большинством задач, которые программисты на C++ обычно решают с помощью шаблонов, но без сопутствующих ограничений. Однако это не говорит о том, что недостатки вообще отсутствуют. Существует ряд задач, которые легко решаются посредством шаблонов C++, но не могут быть решены с помощью обобщений C#. Аналогично, хотя обобщения в языке Java в целом менее мощные, чем в C#, некоторые их концепции, выражаемые на Java, не имеют эквивалентов в C#. В этом разделе будут показаны наиболее часто встречающиеся недостатки и приведено краткое сравнение реализации обобщений в C#/.NET с шаблонами C++ и обобщениями Java.

Важно подчеркнуть, что упоминание об этих помехах вовсе не означает, что их вообще следует избегать. В частности, я ни в коей мере не имею в виду, что смог бы выполнить работу лучше! Проектировщикам языков и платформ приходилось находить правильное соотношение между мощностью и сложностью (а также выполнять проектные решения и реализацию в приемлемые сроки). Скорее всего, вы не столкнетесь с проблемами, но если это все же произойдет, то вы будете в состоянии обойти их, воспользовавшись приведенным здесь руководством.

Мы начнем с ответа на вопрос, который возникает рано или поздно: почему нельзя преобразовать `List<string>` в `List<object>`?

3.5.1. Отсутствие обобщенной вариантности

В разделе 2.2.2 обсуждалась *ковариантность* массивов — тот факт, что массив ссылочного типа может быть представлен как массив своего базового типа или массив любого из интерфейсов, которые он реализует. В действительности эта идея воплощена в две формы, *ковариантность* и *контравариантность*, или вместе — *вариантность*. Обобщения вариантность не поддерживают — они *инвариантны*. Как будет показано, это связано с безопасностью типов, но иногда оно может и раздражать.

Для начала следует прояснить один момент: в C# 4 ситуация с обобщенной вариантностью несколько улучшена. Хотя многие из перечисленных здесь ограничений по-прежнему применимы, этот раздел служит полезным введением в идею вариантности. В главе 13 объясняется, каким образом в данном отношении помогает C# 4, однако многие яркие примеры обобщенной вариантности основаны на других новых средствах версии C# 3, в числе которых LINQ. Тема вариантности и сама по себе достаточно сложна, поэтому прежде чем касаться ее, имеет смысл подождать, пока вы хорошо не освоите оставшийся материал по версиям C# 2 и C# 3. Ради удобства чтения в настоящем разделе не будут указываться особенности, которые несколько отличаются в C# 4. Все это прояснится в главе 13.

Почему обобщения не поддерживают ковариантность?

Предположим, что есть два класса, `Turtle` и `Cat`, оба производные от абстрактного класса `Animal`. В приведенном ниже примере код с массивом (слева) является допустимым кодом C# 2, а код с обобщениями (справа) — нет.

Допустимый код (на этапе компиляции)	Недопустимый код
`Animal[] animals = new Cat[5];`	`List<Animal> animals = new List<Cat>();`
`animals[0] = new Turtle();`	`animals.Add(new Turtle());`

В обоих случаях компилятор без проблем воспримет вторые строки, но первая строка кода, показанного справа, вызывает следующую ошибку:

```
error CS0029: Cannot implicitly convert type
     'System.Collections.Generic.List<Cat>' to
     'System.Collections.Generic.List<Animal>'
```

```
ошибка CS0029: Не удается неявно преобразовать тип
      'System.Collections.Generic.List<Cat>' в
      'System.Collections.Generic.List<Animal>'
```

Это был обдуманный выбор со стороны проектировщиков инфраструктуры и языка. Возникает очевидный вопрос о том, *почему* это запрещено; ответ кроется во второй строке.

Вторая строка кода не содержит ничего подозрительного. В конце концов, класс `List<Animal>` на самом деле имеет метод с сигнатурой `void Add(Animal value)` — вы должны иметь возможность поместить экземпляр `Turtle` в любой список типа `Animal`, например. Однако *действительным* объектом, на который ссылается `animals`, является `Cat[]` (в коде слева) или `List<Cat>` (в коде справа), которые оба требуют, чтобы в нем хранились только ссылки на экземпляры класса `Cat` (или его подклассов). Несмотря на то что версия кода с массивом скомпилируется, во время выполнения она откажется

работать. Такая ситуация была расценена проектировщиками обобщений как худшая по сравнению с отказом на этапе компиляции, что вполне разумно — весь смысл статической типизации заключается в поиске ошибок еще до запуска кода на выполнение.

Почему массивы являются ковариантными?

Получив ответ на вопрос о том, почему обобщения инвариантны, следующий очевидный вопрос касается причины ковариантности массивов. Согласно книге Джеймса Миллера и Сьюзен Регсдейл *Common Language Infrastructure Annotated Standard* (Addison-Wesley Professional, 2003 г.), в первой версии .NET проектировщики хотели охватить как можно более широкую аудиторию, включая тех, кому нужна возможность запуска кода, скомпилированного из исходного кода Java. Другими словами, массивы .NET являются ковариантными из-за ковариантности массивов Java — несмотря на то, что это известный недостаток Java.

Итак, вы узнали причины, по которым дела обстоят именно так, как есть, но почему вы должны об этом беспокоиться и как обойти указанное ограничение?

Где была бы полезна ковариантность?

Приведенный ранее пример со списком содержал очевидную проблему. Можно добавлять элементы в список, но в таком случае теряется безопасность типов, и операция `Add()` является примером значения, используемого в качестве *входного* при обращении к API-интерфейсу: это значение предоставляет вызывающий код. Что произойдет, если ограничиться только выводом значений?

Очевидными примерами могут служить интерфейс `IEnumerator<T>` и связанный с ним интерфейс `IEnumerable<T>`. В действительности они представляют собой едва ли не *канонические* примеры обобщенной ковариантности. Вместе эти интерфейсы описывают последовательность значений; все, что известно о значениях — это то, что каждое из них будет совместимым с `T`, следовательно, всегда можно записать так:

```
T currentValue = iterator.Current;
```

Здесь применяется нормальная идея совместимости — было бы неплохо, если бы `IEnumerator<Animal>` выдавал ссылки на экземпляры `Cat` или `Turtle`, например. Поскольку не существует способа помещения значений, которые являются неподходящими для действительного типа последовательности, желательно иметь возможность трактовать `IEnumerator<Cat>` как `IEnumerator<Animal>`. Давайте рассмотрим пример, где это может быть удобно.

Предположим, что взят привычный пример наследования фигур, но с использованием интерфейса (`IShape`). Теперь обратимся к другому интерфейсу, `IDrawing`, который представляет рисунок, образованный из фигур. Будут применяться два конкретных типа рисунков — `MondrianDrawing` (созданный из прямоугольников) и `SeuratDrawing` (созданный из кружочков)[9].

9 О художниках, имена которых присутствуют в именах типов, можно почитать в Википедии (`http://ru.wikipedia.org/wiki/Мондриан,_Пит` и `http://ru.wikipedia.org/wiki/Сера,_Жорж-Пьер`). Они так много значат для меня по нескольким причинам: Mondrian — это также имя инструмента пересмотра кода, который мы используем в Google, а Сера дал свое имя Джорджу из замечательного мюзикла "Воскресенье в парке с Джорджем", написанного композитором Стивеном Сондхаймом.

Иерархия классов показана на рис. 3.4.

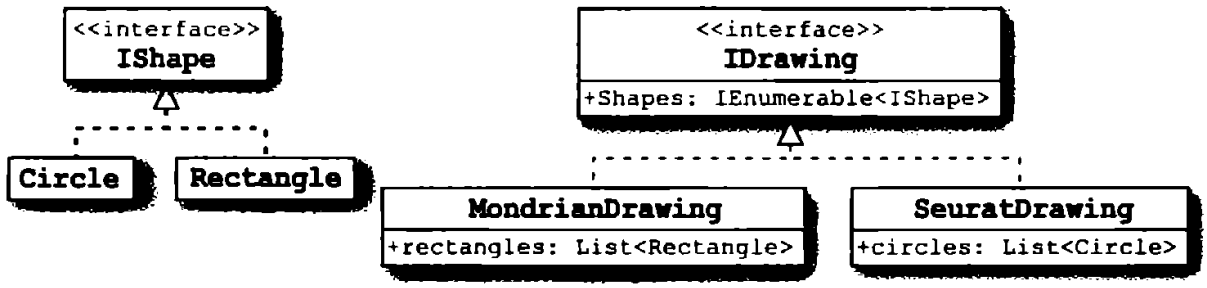

Рис. 3.4. Интерфейсы для фигур и рисунков вместе с двумя их реализациями

Оба типа рисунков должны реализовывать интерфейс `IDrawing`, поэтому им понадобится предоставлять свойство со следующей сигнатурой:

```
IEnumerable<IShape> Shapes { get; }
```

Но для каждого типа чертежа, возможно, будет проще внутренне поддерживать более строго типизированный список. Например, тип `SeuratDrawing` может включать поле типа `List<Circle>`. Это удобнее, чем иметь поле типа `List<IShape>`, т.к. если необходимо манипулировать кружочками специфическим для них образом, то не придется использовать приведение.

При наличии поля `List<IShape>` можно было бы либо возвратить его значение непосредственно, либо поместить его внутрь `ReadOnlyCollection<IShape>`, предотвращая вызывающий код от применения приведения — в любом случае реализация будет простой и недорогой в плане затрат. Однако сделать это не получится, когда типы не совпадают. Преобразование из `IEnumerable<Circle>` в `IEnumerable<IShape>` невозможно. А что *можно* сделать?

Существует несколько вариантов, которые перечислены ниже.

- Изменить тип поля на `List<IShape>` и пользоваться приведениями. Это не особенно удобно и во многом нивелирует смысл применения обобщений.
- Воспользоваться новыми средствами реализации итераторов, предоставляемыми C# 2, как будет показано в главе 6. Это рациональное решение для конкретного рассматриваемого случая, но только для него (когда работа ведется с `IEnumerable<T>`).
- Сделать каждую реализацию свойства `Shapes` создающей новую копию списка, возможно, с применением `List<T>.ConvertAll` для простоты. Так или иначе, создание независимой копии коллекции часто является правильным решением в API-интерфейсе, однако оно приводит к большому объему копирования, которое во многих случаях может оказаться нежелательным.
- Сделать интерфейс `IDrawing` обобщенным с возможностью указания типа фигур внутри рисунка. Таким образом, `MondrianDrawing` будет реализовывать `IDrawing<Rectangle>`, а `SeuratDrawing` — `IDrawing<Circle>`. Это жизнеспособный вариант, только когда вы владеете интерфейсом.
- Создать вспомогательный класс для адаптации одного вида интерфейса `IEnumerable<T>` в другой:

```
class EnumerableWrapper<TOriginal, TWrapper> : IEnumerable<TWrapper>
  where TOriginal : TWrapper
```

Опять-таки, поскольку эта отдельная ситуация (`IEnumerable<T>`) является особой, можно было бы выйти из положения посредством простого служебного метода. На самом деле в .NET 3.5 доступны два удобных метода подобного рода: `Enumerable.Cast<T>()` и `Enumerable.OfType<T>()`. Они представляют собой часть LINQ и рассматриваются в главе 11. Хотя это специальный случай, он является, пожалуй, самой распространенной формой обобщенной ковариантности, которую только можно встретить.

Когда вы столкнетесь с проблемами, связанными с ковариантностью, может понадобиться оценить все описанные выше возможности, а также любые другие, которые вы придумаете самостоятельно. В большой степени это зависит от точной природы ситуации. К сожалению, ковариантность — не единственная проблема, с которой придется иметь дело. Существует также вопрос *контравариантности*, которая похожа на ковариантность, но только наоборот.

Где была бы полезна контравариантность

Контравариантность не так интуитивно понятна, как ковариантность, но в этом есть смысл. С помощью ковариантности мы пытались выполнить преобразование из `SomeType<Circle>` в `SomeType<IShape>` (используя `IEnumerable<T>` для `SomeType` в предыдущем примере). Контравариантность касается преобразования в обратном направлении — из `SomeType<IShape>` в `SomeType<Circle>`. Как такое действие может быть безопасным? Ковариантность безопасна, когда в `SomeType` описаны только операции, которые *возвращают* параметр типа, а контравариантность безопасна в ситуации, когда `SomeType` только операции, *принимающие* параметр типа[10].

Простейшим примером типа, в котором параметр типа применяется только во входной позиции, является `IComparer<T>`, обычно используемый для сортировки коллекций. Давайте расширим интерфейс `IShape` (который до сих пор был пустым), включив свойство `Area`. Теперь легко написать реализацию интерфейса `IComparer<IShape>`, предназначенную для сравнения двух фигур по площади. Затем *хотелось* бы иметь возможность написать следующий код:

```
IComparer<IShape> areaComparer = new AreaComparer();
List<Circle> circles = new List<Circle>();
circles.Add(new Circle(Point.Empty, 20));
circles.Add(new Circle(Point.Empty, 10));
circles.Sort(areaComparer);
```

НЕПРАВИЛЬНО

Тем не менее, этот код работать не будет, т.к. метод `Sort()` в `List<Circle>` на самом деле получает `IComparer<Circle>`. Тот факт, что тип `AreaComparer` может сравнивать *любые* формы, а не просто кружочки, никакого влияния на компилятор не оказывает. Компилятор расценивает типы `IComparer<Circle>` и `IComparer<IShape>` как совершенно разные. С ума сойти, не так ли? Хорошо, если бы вместо этого метод `Sort()` имел такую сигнатуру:

```
void Sort<S>(IComparer<S> comparer) where T : S
```

К сожалению, сигнатура метода `Sort()` не только *другая*, но она и *не может* быть такой — ограничение недопустимо, т.к. оно сделано на `T`, а не на `S`. Необходимо ограничение преобразования типа, но в обратном направлении, указывающее, что тип `S` должен находиться где-то *выше*, а не *ниже* в дереве наследования `T`.

---

[10] В главе 13 вы увидите, что есть несколько больше условий, однако так выглядит общий принцип.

Учитывая невозможность этого, как *можно* поступить? Теперь вариантов меньше. Первый вариант — возврат к идее создания обобщенного вспомогательного класса, как показано в листинге 3.14.

Листинг 3.14. Обход проблемы отсутствия контравариантности с помощью вспомогательного класса

```
class ComparisonHelper<TBase, TDerived> : IComparer<TDerived>
   where TDerived : TBase                    <-- ❶ Соответствующее ограничение параметра типа
{
  private readonly IComparer<TBase> comparer; <-- ❷ Запоминание исходного компаратора
  public ComparisonHelper(IComparer<TBase> comparer)
  {
     this.comparer = comparer;
  }
  public int Compare(TDerived x, TDerived y)
  {
     return comparer.Compare(x, y);          <-- ❸ Использование неявных преобразований для
                                                  обращения к компаратору
  }
}
```

Это пример шаблона проектирования "Адаптер" в действии, хотя в нем вместо подгонки одного интерфейса к совершенно другому интерфейсу осуществляется просто адаптация `IComparer<TBase>` к `IComparer<TDerived>`. Мы запоминаем исходный компаратор ❷, предоставляющий действительную логику для сравнения элементов базового типа, после чего обращаемся к нему, когда требуется сравнение элементов производного типа ❸. Факт отсутствия приведений (даже скрытых) должен вселить определенную уверенность: этот вспомогательный класс полностью безопасен в отношении типов. Возможность обращения к базовому компаратору существует благодаря доступности неявного преобразования из `TDerived` в `TBase`, которое затребовано с помощью ограничения типа ❶.

Второй вариант предусматривает превращение класса для сравнения площадей в обобщенный класс с ограничением преобразования типа, чтобы он позволял сравнивать любые два значения одного и того же типа при условии реализации этим типом интерфейса `IShape`. Для простоты в ситуации, когда такая функциональность не нужна, можно было бы оставить класс необобщенным, унаследовав его от обобщенного класса:

```
class AreaComparer<T> : IComparer<T> where T : IShape
class AreaComparer : AreaComparer<IShape>
```

Конечно, это можно делать только когда есть возможность изменения кода класса для сравнения. Хотя решение достаточно эффективно, выглядит оно по-прежнему неестественно — почему компаратор должен конструироваться для разных типов по-разному, если он не собирается вести себя по-другому? Зачем наследовать от класса, если в действительности поведение не специализируется?

Обратите внимание, что различные возможности для ковариантности и контравариантности предполагают интенсивное использование обобщений и ограничений для выражения интерфейса в более общей манере или для предоставления обобщенных вспомогательных классов. Я понимаю, что добавление ограничения приводит к тому, что интерфейс кажется *менее* общим, однако обобщенность добавляется, прежде всего, за счет

превращения типа или метода в обобщенный. Когда вы столкнетесь с проблемой вроде описанной, добавление куда-либо уровня обобщенности с соответствующим ограничением должно быть первым рассматриваемым вариантом. Здесь часто полезны обобщенные *методы* (в отличие от обобщенных типов), т.к. выведение типов может сделать отсутствие вариантности невидимым невооруженному глазу. Это особенно справедливо в языке C# 3, который обладает более развитыми возможностями выведения типов, чем C# 2.

Такой недостаток является *очень* частой причиной поднятия вопросов на дискуссионных площадках C#. Оставшиеся проблемы либо относительно академичны, либо касаются только небольшой части сообщества разработчиков. Следующая проблема главным образом затрагивает тех, кто при своей работе выполняет множество расчетов (обычно научных или финансовых).

3.5.2. Отсутствие ограничений операций или "числового" ограничения

Когда дело доходит до написания сложного математического кода, язык C# не лишен недостатков. Необходимость в явном применении класса `Math` для всех операций кроме простейших арифметических и отсутствие объявлений `typedef` в стиле C, которые бы позволяли легко изменять представление данных, используемых повсюду в программе, всегда трактовались научной общественностью как преграды для широкого принятия языка C#. Скорее всего, обобщения не решат полностью ни одну из указанных проблем, но есть одна общая проблема, которая не позволяет обобщениям помочь настолько, насколько они могли бы потенциально сделать это.

Рассмотрим следующий (недопустимый) обобщенный метод:

```
public T FindMean<T>(IEnumerable<T> data)
{
  T sum = default(T);
  int count = 0;
  foreach (T datum in data)
  {
    sum += datum;
    count++;
  }
  return sum / count;
}
```

НЕПРАВИЛЬНО

Очевидно, что этот код никогда не заработал бы для *всех* типов данных — например, что могло бы означать сложение двух экземпляров класса `Exception`? Ясно, что существует определенное ограничение на то, каким образом можно выражать необходимые действия: суммировать экземпляры `T` и делить `T` на целое число. Если бы можно было записывать код, подобный показанному выше, пусть даже только для встроенных типов, появилась бы возможность реализовывать обобщенные алгоритмы, в которых не играет роли, с какими типами данных они работают — `int`, `long`, `double`, `decimal` и т.д.

Ограничение только встроенными типами, конечно, не радовало бы, но это все же лучше, чем ничего. Идеальное решение также позволяло бы пользовательским типам выступать в качестве числовых, чтобы можно было определить тип `Complex` для поддержки комплексных чисел, например[11].

[11] Конечно, здесь предполагается, что вы не работаете с .NET 4 или последующей версией, потому что тогда можно было бы воспользоваться структурой `System.Numerics.Complex`.

Затем тип комплексного числа мог бы хранить все свои компоненты также и обобщенным путем, позволяя иметь `Complex<float>`, `Complex<double>` и т.д.

Представляются возможными два связанных (но гипотетических) решения. Одно из них могло бы разрешить применение ограничений на операциях, чтобы можно было устанавливать набор ограничений, подобный приведенному ниже (в данный момент неправильному):

```
where T : T operator+ (T, T), T operator/ (T, int)
```

Это требует, чтобы тип `T` имел операции, используемые в показанном ранее коде. Другое решение предполагало бы определение нескольких операций и, возможно, преобразований, которые должны поддерживаться типом для удовлетворения дополнительного ограничения — его можно было бы сделать "числовым ограничением", записав `where T : numeric`.

С обоими решениями связана одна проблема — они не могут быть представлены как нормальные интерфейсы, потому что перегрузка операций реализуется с помощью *статических* членов, которые нельзя применять для реализации интерфейсов. Я нахожу привлекательной идею *статических интерфейсов*, т.е. интерфейсов, в которых объявлены только статические члены, в том числе методы, операции и конструкторы. Статические интерфейсы подобного рода были бы удобны только в рамках ограничений типов, однако они предоставляют безопасный к типам обобщенный способ доступа к статическим членам. Тем не менее, все это витание в облаках (дополнительные сведения по данной теме можно найти в моем блоге: `http://mng.bz/3Rk3`). Мне ничего не известно о планах по включению статических интерфейсов в будущую версию C#.

Двумя самыми ясными способами обхода этой проблемы на сегодняшний день требуют более поздних версий .NET. Первый способ разработан Марком Гревеллом (`http://yoda.arachsys.com/csharp/genericoperators.html`) и предполагает использование деревьев выражений (которые будут представлены в главе 9) для построения динамических методов; второй способ предусматривает применение средств C# 4. Пример использования второго способа будет приведен в главе 14. Но, как можно понять по описаниям, оба способа являются динамическими, т.е. чтобы выяснить, работает ли код с отдельно взятым типом, придется подождать до времени выполнения. Существует несколько обходных путей, в которых по-прежнему применяется статическая типизация, но им присущи другие недостатки (как ни странно, иногда они могут быть медленнее, чем динамический код).

Два недостатка, рассмотренные до сих пор, были достаточно реальными — они отражали проблемы, которые могли возникать во время действительной разработки. Если вам интересно, можете задаться также вопросом о других недостатках, которые не обязательно замедляют разработку, но просто любопытны сами по себе. В частности, почему обобщения охватывают только типы и методы?

3.5.3. Отсутствие обобщенных свойств, индексаторов и других членов типа

Ранее нам приходилось видеть обобщенные типы (классы, структуры, делегаты и интерфейсы) и обобщенные методы. Есть много других членов, которые *могли бы* быть параметризованными, однако не существует никаких обобщенных свойств, индексаторов, операций, конструкторов, финализаторов или событий.

Прежде всего, проясним, что здесь имеется в виду: несомненно, индексатор может иметь возвращаемый тип, являющийся параметром типа; очевидный пример — `List<T>`. Аналогичным примером для свойств может служить тип `KeyValuePair<TKey,TValue>`.

Тем не менее, *нельзя* определить индексатор либо свойство (или любой другой член в указанном выше списке) с *дополнительными* параметрами типов.

Отложив на время в сторону возможный синтаксис для объявления, давайте взглянем на то, как к таким членам можно было бы обратиться:

```
SomeClass<string> instance = new SomeClass<string><Guid>("x");
int x = instance.SomeProperty<int>;
byte y = instance.SomeIndexer<byte>["key"];
instance.Click<byte> += ByteHandler;
instance = instance +<int> instance;
```

Надеюсь, вы согласитесь с тем, что все это выглядит слегка нелепо. Финализаторы даже нельзя вызывать явно в коде C#, поэтому строка для них отсутствует. Насколько я вижу, невозможность сделать ничего из показанного выше кода не создает значительных проблем — просто полезно знать об этом как об академическом недостатке.

Пожалуй, больше всего это ограничение раздражает применительно к конструктору. Однако хорошим обходным способом решения данной проблемы является статический обобщенный метод в классе, а приведенный ранее пример синтаксиса обобщенного конструктора с двумя списками аргументов типов выглядит ужасно.

Это никоим образом не единственные недостатки обобщений C#, но я уверен, что именно с ними вы, скорее всего, столкнетесь в повседневной работе, при обсуждениях в сообществе или при неспешном исследовании средства как единого целого. В следующих двух разделах будет показано, что определенные аспекты из числа рассмотренных ранее не являются проблемами в двух других языках, функциональные возможности которых чаще всего сравнивают с обобщениями C#: C++ (с шаблонами) и Java (с обобщениями в версии Java 5). Начнем с языка C++.

3.5.4. Сравнение с шаблонами C++

Шаблоны C++ немного напоминают макросы, возведенные в n-ную степень. Они невероятно мощные, но за это приходится платить разбуханием кода и сложностью его восприятия.

Когда используется шаблон C++, код компилируется для конкретного набора аргументов шаблона, как если бы данные аргументы шаблона находились в исходном коде. Это означает, что потребность в ограничениях не так велика, поскольку компилятор будет проверять, какие действия разрешены для типа, во время компиляции кода для указанного набора аргументов шаблона. Но все же в комитете по стандартам C++ признали, что ограничения по-прежнему остаются удобными. Ограничения были включены и затем изъяты из C++11 (последняя версия C++), но они еще могут увидеть свет под названием *концепций*.

Компилятор C++ достаточно интеллектуален, чтобы осуществлять компиляцию кода только один раз для любого заданного набора аргументов шаблона, однако он не способен разделять код тем способом, как это делает среда CLR для ссылочных типов. Тем не менее, в отсутствии такого разделения есть и свои преимущества — становится возможными специфические для типа оптимизации, такие как встраивание вызовов методов для одних параметров типов, но не для других, в рамках того же самого шаблона. Это также означает, что распознавание перегруженных версий может выполняться отдельно для каждого набора параметров типов, а не только один раз исключительно на основе ограниченных знаний компилятора C# вследствие любых присутствующих ограничений.

Не забывайте, что в языке C++ предусмотрена только одна компиляция, а не модель "компиляция в код IL и затем JIT-компиляция в машинный код", принятая в .NET.

Программа на C++, в которой стандартный шаблон используется десятью разными способами, будет содержать в себе десять копий кода. Аналогичная программа на C#, в которой применяется обобщенный тип из инфраструктуры десятью разными путями, не будет включать код для обобщенного типа вообще — на него будет присутствовать только ссылка, а во время выполнения JIT-компилятор скомпилирует столько разных версий, сколько потребуется (как описано в разделе 3.4.2).

Одно из значительных преимуществ шаблонов C++ по сравнению с обобщениями C# состоит в том, что аргументы шаблона не обязательно должны быть именами типов. Можно также применять имена переменных, имена функций и константные выражения. Распространенным примером является тип буфера под названием `buffer`, который в качестве одного из аргументов шаблона получает размер буфера. Скажем, `buffer<int,20>` будет всегда представлять буфер из 20 целых чисел, а `buffer<double,35>` — буфер из 35 значений типа `double`. Эта возможность критически важна для шаблонного метапрограммирования (`http://en.wikipedia.org/wiki/Template_metaprogramming`), представляющего собой расширенный прием программирования на C++, сама идея которого меня отпугивает, но в руках экспертов такой прием может оказаться довольно мощным инструментом.

Шаблоны C++ обладают большей гибкостью также и в других отношениях. Они не страдают из-за отсутствия ограничений операций, описанных в разделе 3.5.2, вдобавок в C++ не существует несколько других ограничений: класс можно наследовать от одного из его параметров типов, и шаблон можно специализировать для отдельного набора аргументов типов. Последняя возможность позволяет автору шаблона предусматривать общий код для использования в ситуациях, когда дополнительные сведения не доступны, и специальный (часто высокооптимизированный) код для конкретных типов.

Те же самые проблемы с вариантностью, которые присущи обобщениям .NET, присутствуют также и в шаблонах C++. Пример, предоставленный Бьярне Страуструпом (изобретателем языка C++), свидетельствует о том, что по похожим причинам не существует неявных преобразований между `vector<shape*>` и `vector<circle*>` — в данном случае это было бы совершенно неуместным.

За дополнительными деталями о шаблонах C++ рекомендую обратиться к книге Страуструпа *The C++ Programming Language*, 3-е изд. (Addison-Wesley Professional, 1997 г.). Читать эту книгу не всегда легко, но глава, посвященная шаблонам, написана довольно понятно (как только вы освоитесь с терминологией и синтаксисом C++).

Больше случаев сравнения с обобщениями .NET можно найти в статье от команды создателей Visual C++ по адресу `http://blogs.msdn.com/b/branbray/archive/2003/11/19/51023.aspx`.

Другим очевидным языком для сравнения с C# в смысле обобщений является Java, в котором это средство появилось в выпуске 1.5[12] через несколько лет после того, как другие проекты привели к созданию Java-подобных языков, поддерживающих обобщения.

3.5.5. Сравнение с обобщениями Java

В то время как компилятор C++ помещает в генерируемый код *больше* копий шаблона, чем C#, компилятор Java включает их *меньше*. В действительности исполняющей среде Java вообще ничего не известно об обобщениях. Байт-код Java (грубый эквивалент кода IL) для обобщенного типа имеет дополнительные метаданные, указывающие на то,

[12] Или в выпуске 5.0 в зависимости от используемой системы нумерации. Только не провоцируйте меня критиковать это.

что это обобщение, но после компиляции вызывающий код в принципе не содержит ничего такого, что говорило бы о применении обобщений, и экземпляру обобщенного типа известна только его необобщенная сторона. Например, экземпляр HashSet<E> не знает о том, каким образом он создавался — как HashSet<String> или как HashSet<Object>. Компилятор добавляет приведения там, где необходимо, и выполняет большее количество проверок. Ниже приведен пример — сначала обобщенный код Java:

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("hello");
String entry = strings.get(0);
strings.add(new Object());
```

А вот эквивалентный ему необобщенный код:

```
ArrayList strings = new ArrayList();
strings.add("hello");
String entry = (String) strings.get(0);
strings.add(new Object());
```

Оба фрагмента генерируют одинаковый байт-код Java кроме последней строки, которая является допустимой в необобщенном случае, но вызовет ошибку на этапе компиляции в обобщенной версии кода. Обобщенный тип можно использовать как низкоуровневый, что подобно применению java.lang.Object для каждого аргумента типа. Такое переписывание — и утеря информации — называется *стиранием типов*. В Java отсутствует понятие типов значений, определяемых пользователем, и в качестве аргументов типов нельзя указывать даже встроенные типы значений. Вместо этого придется применять упакованные версии — к примеру, ArrayList<Integer> для списка целых чисел.

Все это может показаться неутешительным по сравнению с обобщениями C#, но обобщения Java обладают также и рядом интересных особенностей.

- Виртуальной машине ничего не известно об обобщениях. Это означает возможность использования скомпилированного кода, работающего с обобщениями, в старой версии виртуальной машины, при условии, что в коде не применяются классы или методы, которые в старой версии отсутствуют. Контроль версий в .NET в целом намного строже — для каждой сборки, на которую производится ссылка, можно указать о необходимости точного совпадения номера версии. Вдобавок код, скомпилированный для запуска в среде CLR 2.0, не сможет выполняться под управлением .NET 1.1.
- Для использования обобщений Java изучать какой-то новый набор классов не понадобится — там, где при отказе от обобщений применялся бы тип ArrayList, в случае обобщений просто используется тип ArrayList<E>. Существующие классы довольно легко могут быть модернизированы обобщенными версиями.
- Предыдущая возможность эффективно эксплуатируется системой рефлексии — java.lang.Class (эквивалент System.Type) является обобщенным, что позволяет распространить безопасность типов этапа компиляции на многие ситуации, в которых применяется рефлексия. Однако в ряде других ситуаций это по-прежнему затруднительно.
- В Java имеется поддержка обобщенной вариантности с использованием групповых символов. Например, ArrayList<? extends Base> можно интерпретировать как "список ArrayList какого-то типа, производного от Base, который в точности не известен". Во время обсуждения поддержки версией C# 4 обобщенной вариантности в главе 13 будет представлен небольшой пример.

По моему мнению, обобщения .NET более совершенны почти во всех отношениях, хотя когда я сталкиваюсь с проблемами ковариантности/контравариантности, то часто жалею, что у меня нет групповых символов. Положение дел в какой-то мере улучшает ограниченная обобщенная вариантность версии C# 4, однако по-прежнему существуют ситуации, при которых модель вариантности Java работает лучше. Хотя язык Java с обобщениями намного лучше языка Java без обобщений, никакого выигрыша в плане производительности обобщения не дают, а безопасность типов поддерживается только на этапе компиляции.

3.6. Резюме

Уф! Обобщения гораздо легче использовать в реальности, чем описать их. Хотя обобщения *могут* стать сложными, они рассматриваются как самое важное дополнение C# 2 и невероятно удобны. Более того, если когда-либо после написания кода с обобщениями вам придется возвратиться снова к C# 1, вам будет крайне не хватать их. (К счастью, это становится все менее вероятным.)

В этой главе я не пытался раскрыть каждую деталь о том, что при работе с обобщениями доступно, а что нет — это забота спецификации языка, которая предназначена для сухого изложения фактов. Вместо этого я избрал практический подход, предоставляя информацию, которая будет необходима для повседневного использования, с редкими вкраплениями теории ради чисто академического интереса.

Были показаны три основных преимущества обобщений: безопасность типов периода компиляции, производительность и выразительность кода. Возможность обеспечить раннюю проверку кода с помощью IDE-среды и компилятора определенно удобна, но более чем спорно считать, что можно получить большую выгоду от инструментов, которые интеллектуальным образом предлагают варианты на основе применяемых типов, нежели от действительного аспекта безопасности.

Производительность увеличивается наиболее радикально, когда дело доходит до типов значений, которые больше не должны упаковываться и распаковываться, когда они используются в строго типизированных обобщенных API-интерфейсах. Особенно это касается обобщенных типов коллекций в .NET 2.0. Производительность при работе со ссылочными типами обычно улучшается, но незначительно.

С применением обобщений код способен выражать свои намерения более ясно — вместо комментария или длинного имени переменной, требуемого для точного описания участвующих типов, эту работу могут сделать подробности самого типа. Комментарии и имена переменных со временем могут стать неточными из-за того, что в них забыли внести изменения при модификации кода, а информация о типе корректна по определению.

Обобщения не позволяют делать *абсолютно все*, что заблагорассудится, и в этой главе были раскрыты некоторые их недостатки, но если вы действительно изберете C# 2 и обобщенные типы внутри .NET 2.0 Framework, то найдете невероятное множество случаев их использования в своем коде.

Обобщения то и дело будут упоминаться в последующих главах, поскольку на них основаны другие новые средства. И действительно, не будь обобщений, тема следующей главы была бы совершенно иной — мы рассмотрим типы, допускающие `null`, как это реализовано с помощью `Nullable<T>`.

Глава 4

Типы, допускающие значения null

В этой главе...

- Причины использования значений null
- Поддержка типов, допускающих null, в инфраструктуре и исполняющей среде
- Поддержка типов, допускающих null, в языке C# 2
- Шаблоны применения типов, допускающих null

Концепция значений null была темой обсуждений на протяжении многих лет. Является ли ссылка null значением или же она указывает на отсутствие значения? Считать ли, что "ничего" — это "что-нибудь"? Должны ли языки вообще поддерживать концепцию значений null или она должна быть представлена с помощью других моделей?

В этой главе я постараюсь придерживаться практической стороны, а не философской. Сначала мы посмотрим, почему данная проблема возникла в принципе — почему в C# 1 нельзя установить в null переменную типа значения, и каковы были традиционные альтернативы. В конце концов, я представлю вам нашего рыцаря в сияющих доспехах — тип System.Nullable<T> — и затем мы ознакомимся с тем, как версия C# 2 обеспечивает простоту и лаконичность кода при работе с типами, допускающими null. Подобно обобщениям, допускающие null типы иногда используются в ситуациях, которые выходят за рамки ожидаемых, и в конце главы мы рассмотрим несколько примеров таких случаев.

Итак, когда значение не является значением? Давайте узнаем.

4.1. Что делать, когда значение просто отсутствует?

Проектировщики C# и .NET не добавляют новые средства только потехи ради. Должна существовать реальная и значительная проблема, требующая немедленного решения, чтобы они приступили к изменению C# как языка или .NET на уровне платформы. В этом случае проблема лучше всего была сформулирована в виде одного из наиболее часто задаваемых вопросов на форумах, посвященных C# и .NET.

Мне нужно установить переменную типа `DateTime` в `null`, но компилятор не позволяет этого. Что я должен делать?[1]

Этот вопрос возникает вполне естественно — примером может служить приложение электронной коммерции, в котором пользователи могут просматривать хронологию движения средств на своих счетах. Если заказ был размещен, однако еще не доставлен, может присутствовать дата покупки, но отсутствовать дата отправки. Каким же образом выразить это в типе, предназначенном для представления детальных сведений о заказе?

До появления C# 2 ответ на поставленный выше вопрос обычно состоял из двух частей: прежде всего, давалось объяснение невозможности применения `null`, а затем приводился список доступных вариантов. В настоящее время ответ обычно содержит только объяснение типов, допускающих `null`, однако полезно взглянуть на варианты в C# 1, чтобы лучше понять, откуда происходит проблема.

4.1.1. Почему переменные типов значений не могут быть установлены в `null`

Как было показано в главе 2, значением переменной ссылочного типа является ссылка, а значением переменной типа значения — сами реальные данные. Ссылка, отличная от `null`, представляет собой способ получения объекта, но `null` действует в качестве специального значения, которое указывает, что ссылка на какой-либо объект отсутствует.

Если хотите думать о ссылках как об URL, то `null` является (*очень* грубо говоря) ссылочным эквивалентом `about:blank`. Ссылка `null` представлена в памяти всеми нулями (именно по этой причине она является стандартным значением для всех ссылочных типов — обнуление целого блока памяти не требует больших затрат, так что это способ инициализации объектов), но по-прежнему в основном хранится тем же образом, что и другие ссылки. Никаких дополнительных битов для переменных ссылочных типов не предусмотрено. Другими словами, использовать значение "все нули" для реальной ссылки невозможно, но это и хорошо, поскольку память исчерпалась бы задолго до того, как появилось настолько много активных объектов, что дело дошло, наконец, до указанного значения. Это является ключевым аспектом для понимания, почему `null` не может быть допустимым значением для типа *значения*.

Давайте рассмотрим хорошо знакомый и простой тип `byte`. Значение переменной типа `byte` хранится в одиночном байте — он может быть дополнен с целью выравнивания по границе слова, но само значение концептуально образовано из одного байта. Мы *должны иметь* возможность хранить в этой переменной значения 0–255; в противном

---

[1] Вопрос почти всегда касается типа `DateTime`, а не какого-то другого типа значения. Причины этого не совсем ясны; скорее всего, разработчики по существу понимают, почему байт не должен иметь значение `null`, но чувствуют, что даты по своей природе могут допускать `null`.

случае она бесполезна при чтении произвольных двоичных данных. С учетом 256 нормальных значений и одного значения null, придется иметь дело суммарно с 257 значениями, а уместить так много значений в одиночный байт не удастся. Проектировщики могли бы решить, что для каждого типа значения должен быть где-то предусмотрен дополнительный бит флага, который определяет, равно ли значение null или же содержит реальные данные. Однако последствия от такого использования памяти были бы ужасными, не говоря уже о том, что пришлось бы проверять этот флаг каждый раз, когда нужно применять значение. Выражаясь кратко, в случае типов значений часто необходимо обеспечивать наличие полного диапазона возможных битовых комбинаций, доступных в качестве реальных значений, тогда как для ссылочных типов вполне нормально пожертвовать одним потенциальным значением, чтобы получить преимущества от доступности ссылки null.

Это обычная ситуация. Но почему в принципе возникает *желание* иметь возможность представления null типом значения? Самая распространенная причина связана с тем, что в базах данных, как правило, поддерживается значение NULL для каждого типа (если только поле специально не сделано не допускающим NULL), поэтому допускать значение null могут символьные данные, целые числа, булевские значения — словом, все, что угодно. При выборке данных из базы обычно нежелательно терять информацию, так что важно иметь возможность каким-то образом представлять значения null в читаемых данных.

Тем временем, вопрос переходит на следующий уровень. Почему в базах данных разрешены значения null для дат, целых чисел и тому подобного? Как правило, значения null используются для неизвестных или отсутствующих значений, таких как дата отправки в ранее упомянутом примере приложения электронной коммерции. С помощью null представляется отсутствие точной информации, что может оказаться важным во многих ситуациях. На самом деле типы значений, допускающие null, удобны не только при взаимодействии с базами данных; просто это сценарий, в котором разработчики обычно впервые сталкиваются с проблемой. Это подводит нас к ознакомлению с вариантами представления значений null в C# 1.

4.1.2. Шаблоны для представления значений null в C# 1

В C# 1 распространены три базовых шаблона, позволяющие обойти отсутствие типов значений, допускающих null. Каждый из них обладает своими достоинствами и недостатками — главным образом, недостатками — и все они в той или иной степени неудовлетворительны. Тем не менее, эти шаблоны полезно знать, отчасти для более полной оценки преимуществ интегрированного решения в версии C# 2.

Шаблон 1: "магическое" значение

Первый шаблон предусматривает выделение одного значения специально для представления null. Как правило, это применяется в качестве решения для типа DateTime; некоторые разработчики предполагают, что их базы данных *действительно* содержат даты в AD 1, поэтому DateTime.MinValue может послужить удобным "магическим" значением без утери каких-либо полезных данных. Другими словами, это идет вразрез с приведенной ранее цепочкой рассуждений, где было указано, что каждое возможное значение должно быть доступным для нормальных целей. *Семантический* смысл такого значения null будет варьироваться от приложения к приложению — оно может говорить, например, о том, что пользователь пока еще не ввел значение в форму, или о том, что оно для данной записи не обязательно.

Важно отметить, что использование "магического" значения не приводит к напрасному расходу памяти и не требует каких-то новых типов. Но в этом случае придется выбрать подходящее значение, которое никогда не будет применяться для реальных данных. Кроме того, в своей основе шаблон не элегантен. Он не выглядит правильным. Если вы когда-либо решите двигаться этим путем, то должны хотя бы использовать константу (или статическое значение только для чтения в случае типов, которые не могут быть выражены в виде констант) для представления "магического" значения – например, сравнение повсюду с `DateTime.MinValue` не выражает смысла "магического" значения. Вдобавок довольно легко неумышленно применить "магическое" значение, как если бы оно было нормальным значением — ни компилятор, ни исполняющая среда не помогут обнаружить ошибку. В противоположность этому, большинство представленных здесь других решений (в том числе решение на C# 2) приведут либо к ошибке компиляции, либо к исключению времени выполнения — в зависимости от точной ситуации.

Шаблон с "магическим" значением имеет глубокие корни в вычислениях, представляя двоичные типы с плавающей точкой в формате IEEE-754, такие как `float` и `double`. Они продвигаются дальше идеи одиночного значения, представляющего ситуацию "на самом деле это не число" — существует *много* битовых комбинаций, которые классифицируются как "не числа" (not-a-number — NaN), а также как положительная и отрицательная бесконечность. Я подозреваю, что некоторые программисты (включая меня) достаточно осторожно относятся к этим значениям, как и должно быть, и это еще одно свидетельство изъянов данного шаблона.

В инфраструктуре ADO.NET имеется разновидность этого шаблона, при котором одно и то же "магическое" значение `DBNull.Value` используется для *всех* значений `null`, независимо от типа. В таком случае вводилось дополнительное значение, и даже дополнительный тип, для указания на ситуацию, когда база данных возвращает `null`. Однако это применимо только в случаях, когда безопасность типов на этапе компиляции не важна (другими словами, когда вполне устраивает применение типа `object` и выполнение приведения после проверки на предмет `null`), и снова такой подход не выглядит правильным. В действительности он представляет собой смесь шаблона с "магическим" значением и шаблона с оболочкой ссылочного типа, который рассматривается следующим.

Шаблон 2: оболочка ссылочного типа

Второе решение может принимать две формы. Простая форма предусматривает использование `object` для типа переменной, а также упаковку и распаковку значений по мере необходимости. Более сложная (и более привлекательная) форма заключается в создании для каждого типа значения, который нуждается в поддержке `null`, ссылочного типа, содержащего единственную переменную экземпляра этого типа значения и операции неявного преобразования в тип значения и из него. Все это *можно* было бы сделать в одном обобщенном типе, но если уж применяется C# 2, то взамен можно было бы воспользоваться типами, допускающими `null`, которые описаны в настоящей главе. Если же вы придерживаетесь C# 1, то должны написать дополнительный исходный код для каждого типа, которому необходима оболочка. Прием несложно воспроизвести в виде шаблона для автоматического генерации кода, но все равно это порождает накладные расходы, которых по возможности лучше избегать.

Обеим формам второго шаблона присуща проблема: хотя они позволяют применять значение `null` напрямую, обе формы требуют создания объектов в куче, что может при-

вести к повышенной нагрузке на сборщик мусора, если данный подход нужно использовать часто, и увеличению расхода памяти на создаваемые объекты. В более сложном решении можно было бы сделать ссылочный тип изменяемым, что позволило бы сократить количество экземпляров, подлежащих созданию, но также привело бы к получению менее интуитивно понятного кода.

Шаблон 3: дополнительный булевский флаг

Последний рассматриваемый шаблон предусматривает применение обычного значения, относящегося к типу значения, и еще одного значения — булевского флага, который указывает, является ли значение "реальным" или же им следует пренебречь. Опять-таки, существуют два способа реализации этого решения. Можно или поддерживать две независимых переменных в коде, в котором используется значение, или инкапсулировать значение вместе с флагом внутри другого типа значения.

Второй способ довольно похож на описанную ранее идею более сложного ссылочного типа, но только здесь устраняется проблема с повышенной сборкой мусора за счет использования типа значения и указания на `null` внутри инкапсулированного значения, а не с помощью ссылки `null`. Тем не менее, недостаток, связанный с необходимостью создания новых ссылочных типов для всех поддерживаемых типов значений остается тем же. Кроме того, даже если значение по какой-то причине упаковывается, это делается обычным способом независимо от того, является оно `null` или нет.

Вариант шаблона с инкапсуляцией по существу демонстрирует работу типов, допускающих `null`, в C# 2, хотя комбинация новых средств инфраструктуры, CLR и языка позволяет предоставить решение, которое будет значительно яснее, чем это было возможно в C# 1. В следующем разделе обсуждается поддержка, предоставляемая инфраструктурой и CLR в .NET 2: даже если бы в C# 2 поддерживались *только* обобщения, большая часть раздела 4.2 по-прежнему была бы актуальной и эта возможность все равно работала бы и была удобной. Однако в C# 2 предлагается дополнительный синтаксический сахар, улучшающий положение дел — это тема раздела 4.3.

4.2. Типы `System.Nullable<T>` и `System.Nullable`

Основной структурой, которая лежит в основе типов, допускающих `null`, является `System.Nullable<T>`. Кроме того, статический класс `System.Nullable` предоставляет служебные методы, которые иногда упрощают работу с типами, допускающими `null`. (Для простоты в дальнейшем пространство имен указываться не будет.) Мы рассмотрим оба упомянутых типа, но в этом разделе не будут описаны дополнительные языковые средства. Это позволит лучше понять, что происходит на уровне кода IL, когда будет обсуждаться сокращение, предлагаемое в C# 2.

4.2.1. Введение в `Nullable<T>`

Как должно быть понятно по имени, тип `Nullable<T>` является обобщенным. Параметр типа `T` имеет ограничение типа значения, поэтому использовать, например, `Nullable<Stream>`, нельзя. В разделе 3.3.1 уже упоминалось, что это также означает невозможность применения в качестве аргумента другого типа, допускающего `null`, т.е. тип `Nullable<Nullable<int>>` запрещен, хотя во всех остальных отношениях `Nullable<T>` представляет собой тип значения. Тип `T` для любого отдельно взятого

типа, допускающего null, называется *базовым типом* для данного типа, допускающего null. Например, базовым типом Nullable<int> является int.

Наиболее важными частями типа Nullable<T> считаются его свойства, HasValue и Value. Функциональность их очевидна: Value представляет значение, не допускающее null (*реальное* значение, если угодно), когда оно имеется, и генерирует исключение InvalidOperationException в случае (концептуального) отсутствия реального значения. Свойство HasValue булевского типа указывает на то, присутствует ли реальное значение либо же экземпляр должен трактоваться как null. Пока что для обозначения экземпляра, свойство HasValue которого возвращает true или false, будут применяться понятия "экземпляр со значением" или "экземпляр без значения", соответственно.

Эти свойства поддерживаются простыми полями очевидным образом. На рис. 4.1 показаны экземпляры типа Nullable<int>, представляющие (слева направо) отсутствие значения, 0 и 5. Вспомните, что Nullable<T> — по-прежнему тип значения, поэтому при наличии переменной типа Nullable<int> значение этой переменной будет непосредственно содержать значения bool и int — это не будет ссылкой на отдельный объект.

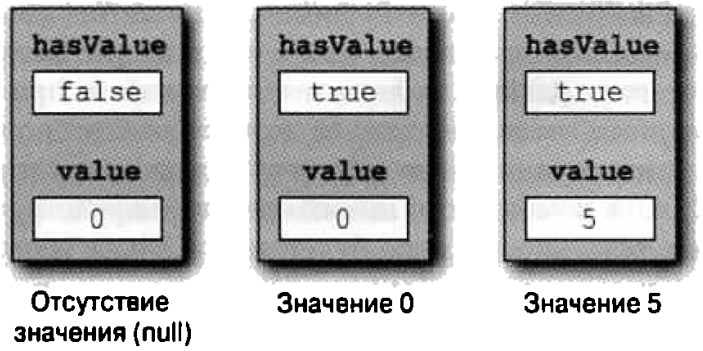

Рис. 4.1. Примеры значений типа Nullable<int>

Теперь, когда вы знаете необходимые свойства, давайте посмотрим, как можно создать экземпляр этого типа. Тип Nullable<T> имеет два конструктора: стандартный (создающий экземпляр без значения) и конструктор, получающий в качестве значения экземпляр T. После конструирования экземпляр является неизменяемым.

Типы значений и изменяемость

Говорят, что тип *неизменяемый*, если он спроектирован так, что его экземпляр не может быть изменен после конструирования. Неизменяемые типы часто приводят к получению более ясного проектного решения, когда необходимо отслеживать то, что *могло бы* изменять разделяемые значения — особенно в многопоточном приложении.

Неизменяемость в особенности важна для типов значений; они должны быть неизменяемыми почти всегда. Большинство типов значений в инфраструктуре являются неизменяемыми, но существует ряд распространенных исключений — в частности, структуры Point в Windows Forms и Windows Presentation Foundation спроектированы как изменяемые.

Если вам необходим какой-то способ базирования одного значения на другом, последуйте примеру типов DateTime и TimeSpan — предоставьте методы и операции, которые возвращают новое значение, а не модифицируют существующее. Это позволит избежать тонких ошибок всех видов, включая ситуации, при которых кажется, что производится изменение чего-либо, но в действительности изменяется копия. Просто скажите "нет" изменяемым типам значений.

В типе Nullable<T> появился единственный новый метод GetValueOrDefault(), который имеет две перегруженных версии. Обе версии возвращают значение экземпляра, если оно есть, или стандартное значение в противном случае. Одна перегруженная версия не принимает каких-либо параметров (в этом случае используется стандартное значение базового типа), а другая версия позволяет указать стандартное значение для его возвращения при необходимости.

Все остальные методы, реализованные в типе Nullable<T>, переопределяют существующие методы: GetHashCode(), ToString() и Equals(). Метод GetHashCode() возвращает 0, если экземпляр не имеет значения, или результат вызова GetHashCode() на значении, если оно присутствует. Метод ToString() возвращает пустую строку, когда нет значения, или результат вызова ToString() на значении, когда оно есть. Метод Equals() несколько сложнее — мы вернемся к нему после обсуждения упаковки.

Наконец, инфраструктура предоставляет два преобразования. Прежде всего, существует *неявное* преобразование из T в Nullable<T>. Оно всегда дает в результате экземпляр, свойство HasValue которого возвращает true. Аналогично, имеется явное преобразование из Nullable<T> в T, которое ведет себя в точности как свойство Value, в том числе генерируя исключение при отсутствии реального значения для возврата.

Упаковка и распаковка

В спецификации C# процесс преобразования экземпляра типа T в экземпляр типа Nullable<T> называется упаковкой, а очевидный обратный процесс — *распаковкой*. Эти термины определены в спецификации со ссылкой, соответственно, на конструктор, принимающий параметр, и свойство Value. На самом деле эти вызовы генерируются кодом C#, даже если они *выглядят* так, как если бы применялись преобразования, предоставляемые инфраструктурой. Однако в любом случае результат будет тем же. В оставшихся разделах главы никаких отличий между доступными двумя реализациями проводиться не будет.

Прежде чем двигаться дальше, давайте взглянем на все это в действии. В листинге 4.1 демонстрируется все то, что можно делать с помощью типа Nullable<T> напрямую, оставив пока в стороне метод Equals().

Листинг 4.1. Использование различных членов типа Nullable<T>

```
static void Display(Nullable<int> x)      ←❶ Отображение диагностической информации
{
  Console.WriteLine("HasValue: {0}", x.HasValue);
  if (x.HasValue)
  {
    Console.WriteLine("Value: {0}", x.Value);
    Console.WriteLine("Explicit conversion: {0}", (int)x);
  }
  Console.WriteLine("GetValueOrDefault(): {0}",
                    x.GetValueOrDefault());
  Console.WriteLine("GetValueOrDefault(10): {0}",
                    x.GetValueOrDefault(10));
  Console.WriteLine("ToString(): \"{0}\"", x.ToString());
  Console.WriteLine("GetHashCode(): {0}", x.GetHashCode());
  Console.WriteLine();
}
```

```
...
Nullable<int> x = 5;
x = new Nullable<int>(5);                    ❷ Создание оболочки для значения 5
Console.WriteLine("Instance with value:");
Display(x);

x = new Nullable<int>();                     ❸ Конструирование экземпляра без значения
Console.WriteLine("Instance without value:");
Display(x);
```

В листинге 4.1 продемонстрированы два разных способа (в терминах исходного кода C#) помещения в оболочку значения базового типа ❷ и применение разнообразных членов экземпляра `Nullable<int>` ❶. Затем создается экземпляр, *не имеющий* значения ❸, для которого используются те же самые члены в том же порядке, но без свойства `Value` и явного преобразования в `int`, т.к. они привели бы к генерации исключения.

Ниже показан вывод кода в листинге 4.1:

```
Instance with value:
HasValue: True
Value: 5
Explicit conversion: 5
GetValueOrDefault(): 5
GetValueOrDefault(10): 5
ToString(): "5"
GetHashCode(): 5

Instance without value:
HasValue: False
GetValueOrDefault(): 0
GetValueOrDefault(10): 10
ToString(): ""
GetHashCode(): 0
```

До сих пор вы, скорее всего, могли предугадать все результаты, просто взглянув на методы, которые предоставляет тип `Nullable<T>`. Однако когда дело доходит до упаковки и распаковки, можно обеспечить *желаемое* поведение типов, допускающих `null`, а не поведение, которое было бы в случае следования обычным правилам упаковки.

4.2.2. Упаковка и распаковка типа `Nullable<T>`

Важно не забывать, что `Nullable<T>` является структурой, т.е. типом значения. Это означает, что для ее преобразования в ссылочный тип (наиболее очевидным примером можно считать `object`) понадобится упаковка. Специальное поведение в отношении типов, допускающих `null`, в среде CLR касается только упаковки и распаковки — оно отличается от такого поведения для стандартных обобщений, преобразований, вызовов методов и т.д. На самом деле это поведение было изменено незадолго до выпуска .NET 2.0 в результате запросов со стороны сообщества разработчиков. В выпусках, предназначенных для предварительного просмотра, упаковка типов, допускающих `null`, производилась подобно другим типам значений.

Экземпляр `Nullable<T>` упаковывается либо в ссылку `null` (если он не имеет значения), либо в упакованное значение типа `T` (при наличии значения), как показано на рис. 4.2. Он никогда не упаковывается в "упакованное значение типа `int`, допускающий `null`" — такого типа не существует.

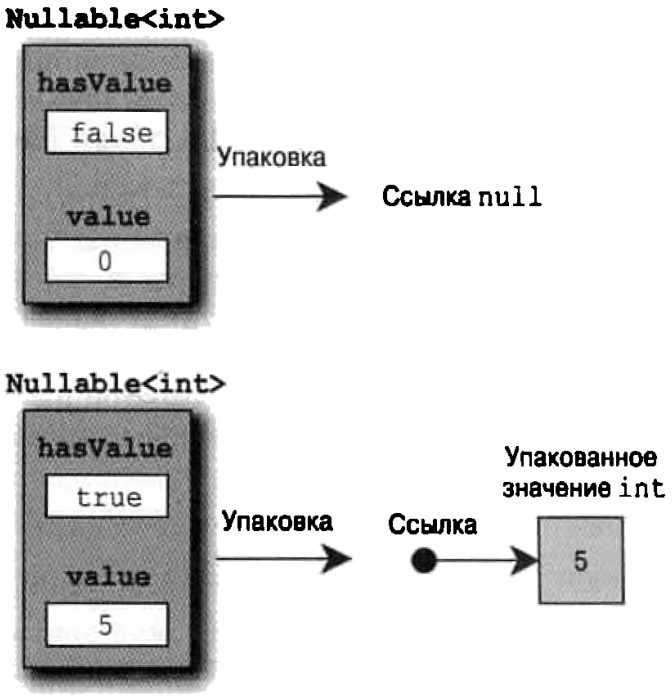

Рис. 4.2. Результаты упаковки экземпляра без значения (вверху) и со значением (внизу)

Распаковать упакованное значение можно либо в нормальный тип, либо в соответствующий тип, допускающий `null`. Распаковка ссылки `null` в нормальный тип вызовет генерацию исключения `NullReferenceException`, но распаковка в соответствующий тип, допускающий `null`, позволит получить экземпляр без значения. Это поведение демонстрируется в листинге 4.2.

Листинг 4.2. Поведение упаковки и распаковки для типов, допускающих `null`

```
Nullable<int> nullable = 5;

object boxed = nullable;          <-- Упаковка экземпляра типа, допускающего null, со значением
Console.WriteLine(boxed.GetType());

int normal = (int)boxed;          <-- Распаковка в переменную типа, не допускающего null
Console.WriteLine(normal);

nullable = (Nullable<int>)boxed;  <-- Распаковка в переменную типа, допускающего null
Console.WriteLine(nullable);

nullable = new Nullable<int>();
boxed = nullable;                 <-- Упаковка экземпляра типа, допускающего null, без значения
Console.WriteLine(boxed == null);

nullable = (Nullable<int>)boxed;  <-- Распаковка в переменную типа, допускающего null
Console.WriteLine(nullable.HasValue);
```

В выводе кода из листинга 4.2 указано, что типом упакованного значения является `System.Int32` (не `System.Nullable<System.Int32>`). Это подтверждает возможность извлечения значения за счет распаковки либо в `int`, либо в `Nullable<int>`. Наконец, вывод показывает, что можно упаковать экземпляр типа, допускающего `null`, без значения в ссылку `null` и затем успешно распаковать его в другой экземпляр типа, допускающего `null`, без значения. Если попытаться распаковать последнее значение переменной `boxed` в тип `int`, не допускающий `null`, возникло бы исключение `NullReferenceException`.

Разобравшись с поведением упаковки и распаковки, можно приступать к анализу поведения метода `Nullable<T>.Equals()`.

4.2.3. Равенство экземпляров типа `Nullable<T>`

В типе `Nullable<T>` переопределен метод `object.Equals(object)`, но не введены какие-либо операции проверки на равенство или метод `Equals(Nullable<T>)`. Поскольку инфраструктура предоставляет элементарные строительные блоки, языки могут поверх них добавлять дополнительную функциональность, включая обеспечение работы существующих операций так, как ожидается от типа. Более подробные сведения будут предоставлены в разделе 4.3.3, но базовое равенство, как определено простым методом `Equals()`, следует перечисленным ниже правилам при вызове `first.Equals(second)`:

- если `first` не содержит значения и `second` имеет значение `null`, то они равны;
- если `first` не содержит значения и `second` имеет значение, отличное от `null`, то они не равны;
- если `first` содержит значение и `second` имеет значение `null`, то они не равны;
- в противном случае они равны, если значение `first` равно `second`.

Обратите внимание, что вы не обязаны рассматривать случай, когда переменная `second` относится к другому типу `Nullable<T>`, т.к. правила упаковки запрещают эту ситуацию. Типом `second` является `object`, поэтому для превращения его в `Nullable<T>` понадобится упаковка, а как только что было показано, упаковка экземпляра типа, допускающего `null`, создает упаковку типа, не допускающего `null`, или возвращает ссылку `null`. Поначалу первое правило может выглядеть как нарушение контракта для `object.Equals(object)`, который требует, чтобы вызов `x.Equals(null)` возвращал `false` — но только тогда, когда `x` представляет собой ссылку, отличную от `null`. Опять-таки, из-за поведения упаковки в отношении реализации `Nullable<T>` никогда не возникнет обращения через ссылку.

По большей части эти правила совместимы с правилами эквивалентности, применяемыми в .NET повсюду, поэтому экземпляры типов, допускающих `null`, можно использовать в качестве ключей в словарях и любых других ситуациях, когда необходима поддержка эквивалентности. Но только не ожидайте, что эквивалентность будет проводить различие между экземпляром типа, не допускающего `null`, и экземпляром типа, допускающего `null`, со значением — поддержка эквивалентности настроена так, что эти два случая трактуются как одинаковые.

Приведенный выше материал охватывает саму структуру `Nullable<T>`, но у нее имеется не очень ясный партнер: класс `Nullable`.

4.2.4. Поддержка необобщенного класса `Nullable`

Структура `System.Nullable<T>` делает практически все, что от нее требуется, но в этом ей помогает класс `System.Nullable`. Это статический класс — он содержит только статические методы, и создавать его экземпляры нельзя [2].

На самом деле то, что он делает, можно было бы с таким же успехом реализовать посредством других типов, и будь в Microsoft более дальновидными, класс `Nullable` вообще мог бы не появиться — это сократило бы путаницу, связанную с тем, для чего

[2] Статические классы более подробно описаны в главе 7.

предназначены упомянутые два типа. Но это историческое недоразумение обладает тремя методами, которые по-прежнему полезны.

Первые два метода — это методы сравнения:

```
public static int Compare<T>(Nullable<T> n1, Nullable<T> n2)
public static bool Equals<T>(Nullable<T> n1, Nullable<T> n2)
```

Метод `Compare()` использует свойство `Comparer<T>.Default` для сравнения двух базовых значений (если они существуют), а метод `Equals()` применяет свойство `EqualityComparer<T>.Default`. В случае экземпляров без значений результаты, возвращаемые обоими методами, соответствуют соглашениям .NET о том, что при сравнении значений `null` друг с другом они считаются равными, но меньшими в сравнении с любыми другими значениями.

Оба метода вполне благополучно могли быть частью типа `Nullable<T>` как статические, но необобщенные методы. Небольшое достоинство их существования как обобщенных методов в необобщенном типе связано с возможностью применения выведения обобщенных типов, поэтому редко когда необходимо явно указывать параметр типа.

Последний метод в типе `System.Nullable` не является обобщенным — он и не мог бы им быть. Сигнатура выглядит следующим образом:

```
public static Type GetUnderlyingType(Type nullableType)
```

Если параметр относится к типу, допускающему `null`, метод возвращает его базовый тип; в противном случае возвращается `null`. Причина того, что метод не определен как обобщенный, заключается в том, что если бы базовый тип был известен с самого начала, то вызывать бы этот метод не пришлось.

Теперь вы знаете, как инфраструктура и среда CLR предоставляют поддержку типов, допускающих `null`, тем не менее, в C# 2 были добавлены языковые средства, позволяющие удобно работать с ними.

4.3. Синтаксический сахар C# 2 для работы с типами, допускающими `null`

В приведенных до сих пор примерах типы, допускающие `null`, выполняли необходимую работу, но код выглядел не особо изящно. Общеизвестно, что обеспечение возможности установки значений `null` требует помещения имени интересующего типа внутрь конструкции `Nullable<>`, однако это отвлекает внимание от собственно базового типа, а потому обычно не может считаться хорошей идеей.

Вдобавок само название "допускающий `null`" предполагает, что должна существовать возможность присваивания значения `null` переменной такого типа, но это пока не демонстрировалось, поскольку ранее всегда применялся стандартный конструктор типа. В этом разделе мы рассмотрим, как эти и другие задачи решаются в C# 2.

Перед тем, как переходить к детальному исследованию того, что предлагает C# 2 как язык, необходимо ввести еще одно определение. Значение `null` типа значения, допускающего `null` — это значение, для которого свойство `HasValue` возвращает `false`, или "экземпляр без значения", как он назывался в разделе 4.2. Ранее этот термин не использовался, т.к. он характерен для языка C#. В спецификации CLI он не упоминается, равно как отсутствует и в документации по самому типу `Nullable<T>`. Я отложил представление этого термина до обсуждения непосредственно версии C# 2. Этот термин применим также к ссылочным типам: значение `null` ссылочного типа является просто ссылкой `null`, хорошо известной вам из версии C# 1.

Сравнение типа, допускающего null, и типа значения, допускающего null

В спецификации языка C# понятие *тип, допускающий* `null`, применяется в отношении любого типа со значением `null` — т.е. ссылочного типа или любой структуры `Nullable<T>`. Вы могли заметить, что этот термин используется, как если бы он был синонимом понятия *тип значения, допускающий* `null` (которое очевидно не охватывает ссылочные типы). Хотя обычно я большой педант, когда дело доходит до терминологии, но применение варианта "тип значения, допускающий `null`" повсюду в этой главе значительно затруднило бы ее чтение. С другой стороны, следует также ожидать двусмысленного использования термина "тип, допускающий `null`" в реальном мире: он очень часто применяется при описании структуры `Nullable<T>` вместо того, чтобы придерживаться определения этой структуры, которое приведено в спецификации.

Имея все это в виду, давайте посмотрим, какие средства предлагает версия C# 2, начиная с сокращения беспорядка в коде.

4.3.1. Модификатор ?

Существует ряд элементов синтаксиса, которые на первых порах могут оказаться незнакомыми, но относительно которых имеется *интуитивное понимание* того, как они работают. Одним из таких элементов для меня является условная операция (`a ? b : c`) — она ставит вопрос и затем предлагает два соответствующих ответа. Аналогичным образом можно воспринимать модификатор `?` применительно к типам, допускающим `null`.

Модификатор `?` представляет собой сокращенный способ указания типа, допускающего `null`, поэтому повсеместно в коде вместо `Nullable<byte>` можно использовать тип `byte?`. Обе формы взаимозаменяемы и компилируются в один и тот же код IL, так что при желании их можно применять одновременно — но от имени тех, кто будет читать ваш код впоследствии, настоятельно рекомендую придерживаться какой-то одной формы. Код в листинге 4.3 полностью эквивалентен коду в листинге 4.2, но в нем используется модификатор `?`, что выделено полужирным.

Листинг 4.3. Код, эквивалентный коду в листинге 4.2, но в котором применяется модификатор ?

```
int? nullable = 5;

object boxed = nullable;
Console.WriteLine(boxed.GetType());

int normal = (int)boxed;
Console.WriteLine(normal);

nullable = (int?)boxed;
Console.WriteLine(nullable);

nullable = new int?();
boxed = nullable;
Console.WriteLine(boxed == null);

nullable = (int?)boxed;
Console.WriteLine(nullable.HasValue);
```

Не имеет смысла объяснять, что и как делает этот код, поскольку результат его выполнения будет в точности таким же, как у кода из листинга 4.2. Код из обоих листингов компилируется в тот же самый код IL — в этих листингах просто используется разный синтаксис, почти так же, как тип `int` взаимозаменяем с типом `System.Int32`. Сокращенную версию можно применять повсюду, включая сигнатуры методов, выражения `typeof`, приведения и тому подобное.

Причина, по которой я считаю модификатор `?` удачным выбором, связана с тем, что он придает природе переменной атмосферу неопределенности. Имеет ли переменная `nullable` в листинге 4.3 целочисленное значение? Действительно, в любой момент времени она может иметь такое значение, но может быть и `null`.

Начиная с этого места, модификатор `?` будет использоваться во всех примерах — это более лаконично и, пожалуй, является идиоматическим способом работы с типами, допускающими `null`, в C#. Но если вам кажется, что этот модификатор очень легко пропустить при чтении кода, то нет никаких препятствий тому, чтобы применять более длинный синтаксис. Просто сравните листинги в этом и предыдущем разделе и определитесь, какой вариант для вас выглядит более ясным.

Учитывая, что в спецификации C# 2 определено значение `null`, было бы странно, если бы не было возможности использовать для его представления литерал `null`, уже присутствующий в языке. К счастью, это можно делать...

4.3.2. Присваивание и сравнение с `null`

Немногословный автор мог бы раскрыть тему этого раздела с помощью одного предложения: "Компилятор C# позволяет применять литерал `null` для представления значения `null` типа, допускающего `null`, в операциях сравнения и присваивания". Я предпочитаю показать, что это означает, на примере реального кода и объяснить, почему в язык была введена эта возможность.

Возможно, вы испытывали неудобства каждый раз, когда использовали стандартный конструктор типа `Nullable<T>`. Это позволяет достичь желаемого поведения, но не выражает *причину*, по которой нужно так поступить — он не оставляет правильного впечатления у читателя кода. В идеальном случае должно быть такое же впечатление, как и во время применения `null` со ссылочными типами.

Если вам кажется странным, что в этом и предыдущем разделах речь идет о впечатлениях, то просто подумайте о тех, кто пишет код и кто его читает. Несомненно, компилятор поймет код, не заботясь о нюансах стиля кода, но в производственных системах лишь очень немногие фрагменты кода пишутся только для того, чтобы никогда не пересматриваться в будущем. При первоначальном написании кода хороши любые действия, с помощью которых можно заставить читателя размышлять, и использование известного литерала `null` помогает достичь этого.

Имея все это в виду, мы перейдем с примера, который просто демонстрирует синтаксис и поведение, на пример, который позволит получить впечатление о том, как могут применяться типы, допускающие `null`. Мы будем моделировать класс `Person`, в рамках которого необходимо знать имя человека, дату его рождения и дату смерти. Будут отслеживаться люди, которые определенно родились, и часть из них все еще живы — в этом случае дата смерти будет представлена посредством `null`. В листинге 4.4 показан возможный код. Реальный класс имел бы больше доступных операций — в этом примере мы просто взглянем на вычисление возраста.

Листинг 4.4. Часть класса Person, включающая вычисление возраста

```
class Person
{
  DateTime birth;
  DateTime? death;
  string name;

  public TimeSpan Age
  {
    get
    {
      if (death == null)                    <-- ❶ Проверка свойства HasValue
      {
        return DateTime.Now - birth;
      }
      else
      {
        return death.Value - birth;         <-- ❷ Распаковка в целях вычисления
      }
    }
  }

  public Person(string name,
                DateTime birth,
                DateTime? death)
  {
    this.birth = birth;
    this.death = death;
    this.name = name;
  }
}
...
Person turing = new Person("Alan Turing ",
                           new DateTime(1912, 6, 23),
                           new DateTime(1954, 6, 7));  <-- ❸ Упаковка DateTime как типа, допускающего null

Person knuth = new Person("Donald Knuth ",
                          new DateTime(1938, 1, 10),
                          null);                       <-- ❹ Указание значение null для даты смерти
```

Код в листинге 4.4 не производит какой-либо вывод, но тот факт, что он успешно компилируется, мог бы удивить вас, если бы вы еще не приступили к чтению этой главы. Помимо того, что использование модификатора ? вызывает путаницу, может показаться странным и наличие возможности сравнения экземпляра DateTime? с null или передачи null в качестве аргумента для параметра типа DateTime?.

К счастью, к этому времени смысл должен быть понятен — когда производится сравнение переменной death с null, выясняется ответ на вопрос, равно ли значение указанной переменной значению null. Подобным же образом, когда значение null выступает как экземпляр типа DateTime?, на самом деле создается значение null для этого типа за счет вызова стандартного конструктора. И действительно, в коде IL, сгенерированном для листинга 4.4, можно заметить просто обращение к свойству death.HasValue ❶ и создание нового экземпляра типа DateTime? ❹ с применением стандартного конс-

труктора (что представлено в коде IL с помощью инструкции `initobj`). Дата смерти Алана Тьюринга ❸ создается путем вызова обычного конструктора типа `DateTime` и последующей передачи результата конструктору `Nullable<DateTime>`, принимающему параметр.

Как уже упоминалось, просмотр кода IL может оказаться удобным способом выяснения, что в действительности делает написанный код, особенно если что-то скомпилировалось, в то время как вы ожидали, что это не должно было произойти. Для этого можно воспользоваться инструментом `ildasm`, входящим в состав .NET SDK, либо одним из многих доступных декомпиляторов, таких как .NET Reflector, ILSpy, dotPeek или JustDecompile. (Когда я ссылаюсь в данной книге на Reflector, то это лишь потому, что пользуюсь указанным инструментом по привычке. Я уверен, что другие инструменты в равной степени хороши.)

Вы уже видели, что в C# предоставляется сокращенный синтаксис для концепции значения `null`, который позволяет сделать код более выразительным при условии в первую очередь понимания типов, допускающих `null`. Однако одна часть листинга 4.4 выполняет несколько бо́льшую работу, чем можно было ожидать — вычитание в строке ❷. По какой причине пришлось распаковывать значение? Почему бы просто не возвратить результат `death - birth` напрямую? Что бы означало это выражение в случае, если переменная `death` равна `null` (хотя такая ситуация в коде исключается предварительной проверкой `death` на предмет `null`)? На все эти, а также многие другие вопросы будут даны ответы в следующем разделе.

4.3.3. Преобразования и операции над типами, допускающими `null`

Вы уже видели, что экземпляры типов, допускающих `null`, можно сравнивать с `null`, но доступны и другие сравнения, которые могут быть реализованы, и другие операции, применяемые в ряде случаев. Аналогично, вы видели упаковку и распаковку, однако с некоторыми типами могут использоваться и другие преобразования. В настоящем разделе объясняется, что именно доступно. Боюсь, что сделать данную тему по-настоящему интересной практически нереально, но тщательно спроектированные средства подобного рода являются именно тем, что в конечном итоге обеспечивает успех языку C#. Не беспокойтесь, если поначалу не все окажется понятным: просто не забывайте, что приводимые здесь сведения пригодятся во время написания кода.

Основной вывод заключается в том, что если какая-то операция или преобразование доступны для типа значения, не допускающего `null`, и эта операция или преобразование затрагивает только другие типы значений, то не допускающий `null` тип значения также имеет ту же самую операцию или преобразование. Это преобразование обычно преобразует типы значений, не допускающие `null`, в их разрешающие `null` эквиваленты. Рассмотрим более конкретный пример. Существует неявное преобразование из `int` в `long`, а это означает, что также имеется неявное преобразование из `int?` в `long?` с очевидным поведением.

К сожалению, хотя такое широкое описание дает правильное общее представление, точные правила несколько сложнее. Каждое отдельно взятое правило формулируется просто, но самих правил довольно много. Эти правила полезно знать, т.к. в противном случае есть шанс столкнуться с непонятной ошибкой или предупреждением на этапе компиляции, связанным с попыткой преобразования, применять которое, как вам кажется, вы совершенно не собирались. Мы начнем с преобразований, а затем перейдем к операциям.

Преобразования, которые затрагивают типы, допускающие null

Для полноты давайте сначала рассмотрим преобразования, которые вы уже знаете:

- неявное преобразование из литерала `null` в `T?`;
- неявное преобразование из `T` в `T?`;
- явное преобразование из `T?` в `T`.

Теперь обратимся к предварительно определенным и пользовательским преобразованиям, которые доступны для типов. Например, существует предварительно определенное преобразование из `int` в `long`. Для любого преобразования подобного рода, из одного типа значения, не допускающего `null` (`S`), в другой такой тип (`T`), имеются также и следующие преобразования:

- из `S?` в `T?` (явное или неявное, в зависимости от исходного преобразования);
- из `S` в `T?` (явное или неявное, в зависимости от исходного преобразования);
- из `S?` в `T` (всегда явное).

Продолжим развитие примера. Перечисленные выше преобразования означают возможность выполнения неявных преобразований из `int?` в `long?` и из `int` в `long?`, а также неявного преобразования из `int?` в `long`. Преобразования ведут себя естественным образом, т.е. значения `null` типа `S?` преобразуются в значения `null` типа `T?`, а для значений, отличных от `null`, используются исходные преобразования. Как и ранее, явное преобразование из `S?` в `T` приведет к генерации исключения `InvalidOperationException`, когда значением типа `S?` является `null`. Для преобразований, определенных пользователем, такие дополнительные преобразования, которые затрагивают типы, допускающие `null`, известны как *поднятые преобразования*.

До сих пор все относительно просто. А теперь перейдем к рассмотрению операций, дела с которым обстоят несколько сложнее.

Операции, которые затрагивают типы, допускающие null

В языке C# разрешено перегружать следующие операции[3]:

- унарные: `+ ++ - -- ! ~ true false`
- бинарные: `+ - * / % & | ^ << >>`
- эквивалентности: `== !=`
- отношения: `< > <= >=`

Когда эти операции перегружаются для типа значения `T`, не допускающего `null`, то тип `T?`, допускающий `null`, получает те же самые операции со слегка отличающимися типами операндов и результатов. Они называются *поднятыми операциями* независимо от того, являются ли предварительно определенными, такими как сложение для числовых типов, или определенными пользователем, вроде сложения `TimeSpan` и `DateTime`. Во время применения этих операций имеется несколько ограничений, которые перечислены ниже.

- Операции `true` и `false` никогда не поднимаются. Хотя с учетом их крайне редкого использования, потеря невелика.

---

[3] Операции эквивалентности и отношения — это также бинарные операции, но их поведение немного отличается от поведения других операций, из-за чего в списке они указаны по отдельности.

- Поднимаются только операции с операндами, которые имеют типы значений, не допускающие null.
- Для унарных и бинарных операций (отличных от операций эквивалентности и отношения) возвращаемый тип должен быть типом значения, не допускающим null.
- Для операций эквивалентности и отношения возвращаемый тип должен быть типом bool.
- Операции & и | для типа bool? поддерживают отдельно определенное поведение, как будет показано в листинге 4.3.4.

Для всех операций типы операндов становятся их эквиваленты допускающие null. Для унарных и бинарных операций возвращаемый тип также становится допускающим null, и если любой из операндов равен null, то возвращается значение null. Операции эквивалентности и отношения сохраняют свои булевские возвращаемые типы, не допускающие null. При определении эквивалентности два значения null считаются равными, а значение null и любое значение, отличное от null — разными, что согласуется с поведением, описанным в разделе 4.2.3. Операции отношения всегда возвращают false, если один из операндов имеет значение null. Когда операнды не равны null, очевидным образом выполняется операция типа, не допускающего null.

Все эти правила выглядят сложнее, чем есть на самом деле — по большей части, все работает вполне предсказуемо. Проще всего увидеть, что происходит, рассмотрев несколько примеров, и поскольку тип int имеет настолько много предварительно определенных операций (к тому же целочисленные значения очень легко выражать), он представляется естественным кандидатом для демонстрации. В табл. 4.1 приведены примеры выражений, сигнатуры поднятых операций и результаты. При этом предполагается, что определены переменные four, five и nullInt, каждая из которых относится к типу int? и имеет очевидное значение.

Таблица 4.1. Примеры применения поднятых операций к значениям целочисленного типа, допускающего null

Выражение	Поднятая операция	Результат
`-nullInt`	`int? -(int? x)`	`null`
`-five`	`int? -(int? x)`	`-5`
`five + nullInt`	`int? +(int? x, int? y)`	`null`
`five + five`	`int? +(int? x, int? y)`	`10`
`nullInt == nullInt`	`bool ==(int? x, int? y)`	`true`
`five == five`	`bool ==(int? x, int? y)`	`true`
`five == nullInt`	`bool ==(int? x, int? y)`	`false`
`five == four`	`bool ==(int? x, int? y)`	`false`
`four < five`	`bool <(int? x, int? y)`	`true`
`nullInt < five`	`bool <(int? x, int? y)`	`false`
`five < nullInt`	`bool <(int? x, int? y)`	`false`
`nullInt < nullInt`	`bool <(int? x, int? y)`	`false`
`nullInt <= nullInt`	`bool <=(int? x, int? y)`	`false`

Пожалуй, наибольшее удивление вызывает последняя строка таблицы — значение null не считается меньшим или равным другому значению null, несмотря на то, что они трактуются как эквивалентные друг другу (согласно пятой строке таблицы)! Очень странная ситуация, но в реальности она не приводит к проблемам, насколько я могу судить по собственному опыту.

Одним из аспектов поднятых операций и преобразований типов, допускающих null, которые вызывают определенную путаницу, являются неожиданные сравнения с null, когда используется тип значения, не допускающий null. Следующий код допустим, но бесполезен:

```
int i = 5;
if (i == null)
{
  Console.WriteLine ("Never going to happen");  // Это никогда не произойдет
}
```

При обработке этого кода компилятор C# выдает предупреждения, но вас может удивить, что такой код вообще разрешен. Здесь происходит вот что: компилятор видит выражение int слева от операции ==, значение null справа от нее, и знает о существовании неявного преобразования в тип int? для каждой части выражения. Поскольку сравнение двух значений int? полностью допустимо, код не приводит к генерации ошибки, а только предупреждения. Дополнительное осложнение связано с тем, что это *не* разрешено делать в случае, когда вместо int применяется обобщенный параметр типа, ограниченный типом значения — правила обобщений запрещают сравнение с null в такой ситуации.

Так или иначе, возникнет ошибка либо предупреждение, и если вы внимательно следите за предупреждениями, то не должны в конечном итоге получить дефектный код из-за указанной индивидуальной особенности, и я надеюсь, что мои пояснения помогут вам лучше понять, что происходит.

Теперь можно ответить на вопрос, поставленный в конце предыдущего раздела: почему в листинге 4.4 используется выражение death.Value - birth, а не просто death - birth? Применяя предыдущие правила, *можно было бы* указать второе выражение, но результат имел бы тип TimeSpan? вместо TimeSpan. Это вызвало бы необходимость либо привести результат к типу TimeSpan, используя свойство Value, либо изменить свойство Age с целью возвращения типа TimeSpan?, который всего лишь перемещает проблему в вызывающий код. Это по-прежнему выглядит несколько неуклюже, но в разделе 4.3.6 будет показана более удачная реализация свойства Age.

В списке ограничений, касающихся поднятия операций, упоминалось, что по сравнению с другими типами тип bool? работает по-другому. В следующем разделе будут даны соответствующие пояснения и представлена более широкая картина, иллюстрирующая причины, по которым все эти операции действуют именно так, а не иначе.

4.3.4. Булевская логика, допускающая значение null

Я часто вспоминаю свои школьные уроки по электронике. Мне кажется, они всегда сводились либо к вычислению напряжения в различных частях электронной схемы с использованием формулы V=I×R, либо к применению таблиц истинности — эталонных диаграмм, объясняющих отличия между логическими вентилями "И-НЕ" и "ИЛИ-НЕ" и т.д. Идея проста: таблица истинности собирает все возможные комбинации на входах внутри интересующего фрагмента логической схемы и сообщает, что будет на выходе.

Таблицы истинности, которые мы рисовали для простых логических вентилей с двумя входами, всегда состояли из четырех строк — каждый вход имел два возможных

значения, что приводило к четырем возможным комбинациям. Булевская логика столь же проста, но что произойдет при наличии логического типа с тремя состояниями? Таким типом является `bool?` — его значением может быть `true`, `false` или `null`. Из этого следует, что теперь таблицы истинности должны содержать по девять строк для бинарных операций, т.к. всего получается девять комбинаций. В спецификации отражены только операции логического "И" и включающего "ИЛИ" (соответственно `&` и `|`), поскольку остальные операции — унарное логическое отрицание (`!`) и исключающее "ИЛИ" (`^`) — следуют тем же самым правилам, как и другие поднятые операции. Условные логические операции (сокращенно вычисляемые операции `&&` и `||`) для типа `bool?` не определены, что весьма упрощает ситуацию.

Ради полноты в табл. 4.2 представлена таблица истинности для всех четырех допустимых логических операций, работающих с типом `bool?`.

Таблица 4.2. Таблица истинности для логических операций "И", включающего "ИЛИ", исключающего "ИЛИ" и логического отрицания применительно к типу `bool?`

x	y	x & y	x | y	x ^ y	!x
true	true	true	true	false	false
true	false	false	true	true	false
true	null	null	true	null	false
false	true	false	true	true	true
false	false	false	false	false	true
false	null	false	null	null	true
null	true	null	true	null	null
null	false	false	null	null	null
null	null	null	null	null	null

Если вам проще понять обоснование правил, чем искать значения в таблицах, то идея заключается в том, что значение `null` типа `bool?` в некотором смысле можно считать как "неопределенность". Представим, что каждая запись `null` на входной стороне таблицы является не значением, а переменной. Тогда на выходной стороне таблицы будет всегда получаться значение `null`, если результат зависит от значения этой переменной. Например, рассмотрим третью строку таблицы истинности. Выражение `true & y` будет равно `true`, если `y` равно `true`, но выражение `true | y` будет равно `true` всегда, каким бы ни было значение `y`, поэтому результатами, допускающими `null`, являются `null` и `true`, соответственно.

При обдумывании поднятых операций и особенно функционирования логики, допускающей `null`, проектировщикам языка пришлось иметь дело с двумя несовместимыми друг с другом наборами поведенческих аспектов — ссылками `null` в языке C# 1 и значениями `NULL` в языке SQL. Во многих случаях они никак не конфликтуют — поскольку в C# 1 отсутствует концепция применения логических операций к ссылкам `null`, проблемы с использованием полученных ранее SQL-подобных результатов не возникали. Тем не менее, показанные ранее определения могут вызвать удивление у некоторых разработчиков на языке SQL, когда дело дойдет до сравнений. В стандартном SQL результат сравнения двух значений (на предмет эквивалентности или выяснения больше/меньше чем) всегда неизвестен, если одним из значений является `NULL`. В C# 2 результат *никогда* не равен `null`, в частности, два значения `null` считаются равными друг другу.

Напоминание: это специфично для языка C#!

Важно помнить, что поднятые операции и преобразования наряду с логикой типа `bool?`, описанной в этом разделе, предоставляются компилятором C#, а *не* средой CLR или самой инфраструктурой. В результате анализа с помощью `ildasm` кода, в котором применяется любая из операций, допускающих `null`, вы обнаружите, что компилятор сгенерировал соответствующий код IL для проверки на предмет значений `null` и обработки их подходящим образом. Это означает, что разные языки в таких вопросах могут вести себя по-разному — на данный аспект следует в первую очередь обращать внимание при переносе кода между различными языками, основанными на .NET. Например, в VB поднятые операции трактуются гораздо ближе к тому, как они реализованы в SQL, поэтому результатом `x < y` будет `Nothing`, если `x` или `y` имеет значение `Nothing`.

Для типов, допускающих `null`, теперь доступна еще одна известная операция с поведением, которое легко предсказать, если обратиться к существующим знаниям ссылок `null` и просто подкорректировать их с учетом понятия *значений* `null`.

4.3.5. Использование операции as с типами, допускающими null

До выхода версии C# 2 операция `as` была доступна только для ссылочных типов. В C# 2 ее можно применять также и к типам значений, допускающим `null`. Результатом этой операции является значение этого типа, допускающего `null` — либо значение `null`, если исходная ссылка имела некорректный тип или была `null`, либо осмысленное значение в противном случае. Ниже приведен короткий пример:

```
static void PrintValueAsInt32(object o)
{
  int? nullable = o as int?;
  Console.WriteLine(nullable.HasValue ?
                    nullable.Value.ToString() : "null");
}
...
PrintValueAsInt32(5);                    <— Выдает 5
PrintValueAsInt32("some string");        <— Выдает null
```

Это позволяет выполнять безопасное преобразование из произвольной ссылки в значение за один шаг — хотя обычно предпринималась бы проверка. В C# 1 пришлось бы использовать операцию `is` и вслед за ней приведение, что выглядит не особенно элегантным: в сущности, среде CLR предлагалось выполнить одну и ту же проверку типа дважды.

Неожиданный провал производительности

Я всегда предполагал, что одна проверка должна выполняться быстрее двух проверок, но похоже здесь ситуация иная — во всяком случае, с версиями .NET, на которых я проводил тестирование (включая .NET 4.5). В написанном простом эталонном тесте производительности, который суммирует все целые числа внутри массива типа `object[]`, где только треть значений были упакованными целыми, применение `is` с последующим приведением оказалось *в 20 раз быстрее*, чем использование операции `as`. Подробный анализ этого выходит за рамки настоящей книги. Как обычно, перед избранием наилучшего направления действий в конкретной ситуации вы должны проверять производительность написанного кода, но об упомянутой выше проблеме полезно знать.

Теперь вы располагаете достаточными знаниями для использования типов, допускающих null, и прогнозирования их поведения, но в отношении синтаксических улучшений версия C# 2 предлагает своего рода "бонус-трек": операцию объединения с null.

4.3.6. Операция объединения с null

За исключением модификатора ?, все остальные описанные ранее трюки компилятора C#, которые имели отношение к типам, допускающим null, работали с существующим синтаксисом. Однако в C# 2 появилась новая операция, которая иногда может сделать код короче или яснее. Она называется *операцией объединения с null* и обозначается в коде с помощью символов ?? между двумя операндами. Она похожа на условную операцию, но специально настроена на значения null.

Эта бинарная операция вычисляет значение выражения first ?? second в соответствие со следующими шагами (грубо говоря).

1. Выполнение оценки first.
2. Если результат отличен от null, он является результатом целого выражения.
3. В противном случае выполнение оценки second; после этого результат становится результатом целого выражения.

Формулировка "грубо говоря" применена из-за того, что официальные правила в спецификации должны иметь дело с ситуациями, предусматривающими преобразования между типами операндов first и second. Как обычно, они не важны в большинстве случаев применения операции ??, поэтому здесь они не рассматриваются; если вас интересуют подробности, почитайте раздел 7.13 ("The Null Coalescing Operator" ("Операция объединения с null")) спецификации.

Важно отметить, что если тип операнда second является базовым типом операнда first (и, таким образом, не допускающим null), общий результат имеет этот базовый тип. Например, показанный ниже код совершенно допустим:

```
int? a = 5;
int b = 10;
int c = a ?? b;
```

Обратите внимание на присваивание напрямую переменной c, несмотря на то, что ее тип — int, не допускающий null. Это можно делать только потому, что тип переменной b не допускает null, следовательно, известно, что в конечном итоге будет получен результат типа, не допускающего null.

Очевидно, что данный пример сильно упрощен; давайте найдем более практичное применение для этой операции, вернувшись к свойству Age из листинга 4.4. В качестве напоминания, вот как оно было реализовано ранее вместе со связанными объявлениями переменных:

```
DateTime birth;
DateTime? death;
public TimeSpan Age
{
  get
  {
    if (death == null)
    {
      return DateTime.Now - birth;
    }
```

```
        else
        {
            return death.Value - birth;
        }
    }
}
```

Обратите внимание, что в обеих ветвях оператора `if` производится вычитание значения переменной `birth` из некоторого значения типа `DateTime`, отличного от `null`. Нас интересует актуальный день в жизни человека — время его смерти, если это уже произошло, или текущее время в противном случае. Чтобы добиться постепенного прогресса, попробуем для начала воспользоваться нормальной условной операцией:

```
DateTime lastAlive = (death == null ? DateTime.Now : death.Value);
return lastAlive - birth;
```

Это своего рода прогресс, но условная операция скорее затруднила чтение кода, чем упростила его, несмотря на то, что новый код стал короче. С условной операцией часто так происходит — частота ее применения зависит от личных предпочтений, хотя прежде чем интенсивно ею пользоваться, имеет смысл проконсультироваться с остальными членами команды разработчиков. Давайте посмотрим, как с помощью операции объединения с `null` улучшить положение дел. Значение переменной `death` должно применяться, если оно отлично от `null`, иначе необходимо использовать `DateTime.Now`. Реализацию можно изменить следующим образом:

```
DateTime lastAlive = death ?? DateTime.Now;
return lastAlive - birth;
```

Обратите внимание на то, что типом результата является `DateTime`, а не `DateTime?`, поскольку в качестве второго операнда применяется свойство `DateTime.Now`. Реализацию *можно было бы* сократить до одного выражения:

```
return (death ?? DateTime.Now) - birth;
```

Однако при этом теряется ясность — в частности, имя переменной `lastAlive` в двухстрочной версии помогает понять, по какой причине используется операция объединения с `null`. Полагаю, вы согласитесь с тем, что двухстрочная версия проще и читабельнее, чем первоначальная версия с оператором `if` или версия, в которой применяется обычная условная операция из C# 1. Разумеется, при этом предполагается понимание того, что делает операция объединения с `null`. Согласно моему опыту, это один из наименее известных аспектов C# 2, но он достаточно ценен для того, чтобы проинформировать о нем своих коллег, а не всячески избегать его применения.

Существуют еще два аспекта, которые увеличивают пользу этой операции. Прежде всего, она применима не только к типам значений, допускающих `null` — она работает также и ссылочными типами; просто в первом операнде нельзя использовать тип значения, допускающий `null`, т.к. это бессмысленно. Кроме того, эта операция является *правоассоциативной*, т.е. выражение в форме `first ?? second ?? third` оценивается как `first ?? (second ?? third)` — и так продолжается для большего числа операндов. Можно иметь любое количество выражений, и они будут оцениваться по порядку с остановом на первом результате, отличном от `null`. Если все выражения оценены как `null`, результатом также будет `null`.

Рассмотрим конкретный пример. Предположим, что имеется онлайновая система заказов с концепциями адресов для платежа, контакта и доставки. В бизнес-правилах

заявлено, что любой пользователь *должен* иметь адрес для платежа, но не обязательно адрес для контакта. Адрес для доставки в отдельном заказе также не является обязательным и по умолчанию совпадает с адресом для платежа. В коде эти необязательные адреса легко представляются как ссылки null. Чтобы определить, с кем нужно связаться в случае проблемы с доставкой, можно написать следующий код C# 1:

```
Address contact = user.ContactAddress;
if (contact == null)
{
  contact = order.ShippingAddress;
  if (contact == null)
  {
    contact = user.BillingAddress;
  }
}
```

Применение условной операции в данной ситуации дает еще более запутанный код. Однако использование операции объединения с null существенно упрощает код:

```
Address contact = user.ContactAddress ??
                  order.ShippingAddress ??
                  user.BillingAddress;
```

Если бизнес-правила изменятся, чтобы установить применение по умолчанию адреса для доставки вместо адреса для контакта, то изменение здесь совершенно очевидно. Оно не будет *чрезмерно* трудным по сравнению с версией if/else, но мне придется дважды подумать и мысленно проверить код. Я также полагаюсь на модульное тестирование, так что шансы ошибиться невелики, но я предпочитаю не думать о подобного рода вещах, если только в этом не возникает абсолютная необходимость.

Все в меру

На тот случай, если вы думаете, что мой код захламлен операциями объединения с null — на самом деле это не так. Я стремлюсь обдумывать ее использование, когда сталкиваюсь с механизмами установки стандартных значений, в которых задействованы значения null и возможно условная операция, но это происходит нечасто. Тем не менее, когда случай применения операции объединения с null является естественным, она может быть мощным инструментом в достижении лучшей читабельности кода.

Вы уже видели, как использовать типы, допускающие null, для обычных свойств объектов — случаи, при которых вполне естественно может отсутствовать значение для какого-то аспекта, по-прежнему лучше всего выражаемого с помощью типа значения. Таковы наиболее очевидные применения типов, допускающих null, и в действительности они же являются самыми распространенными. Несколько других шаблонов менее очевидны, но вполне могут оказаться мощными. Два таких шаблона будут рассматриваться в следующем разделе. Материал предлагается больше ради интереса, чем в плане изучения поведенческих аспектов самих типов, допускающих null, поскольку вы уже располагаете всеми инструментами, необходимыми для работы с такими типами в своем коде. Однако если вас интересуют нестандартные идеи и что-то совершенно новое, смело читайте следующий раздел.

4.4. Новаторское использование типов, допускающих null

До того как типы, допускающие null, стали реальностью, многие разработчики нуждались в них, обычно в связи с доступом к базам данных. Тем не менее, это не единственная ситуация, в которой они могут применяться. Приведенные в этом разделе шаблоны являются нетрадиционными, но они упрощают код. Если вы придерживаетесь только нормальных конструкций языка C#, то все в порядке — возможно, этот раздел не для вас, и я искренне уважаю такую точку зрения. Я обычно отдаю предпочтение простоте кода, а не его искусности, однако если преимущества сулит целый *шаблон*, то временами небесполезно его изучить. Использовать ли такие приемы, зависит исключительно от вас — вы можете обнаружить, что они предлагают другие идеи, которые пригодны для применения где-нибудь в разрабатываемом коде.

Не мудрствуя лукаво, давайте начнем с рассмотрения альтернативы шаблону TryXXX(), который упоминался в разделе 3.3.3.

4.4.1. Проба выполнения операции без использования выходных параметров

Шаблон, в котором возвращаемое значение применяется для сообщения о работоспособности операции, а выходной параметр — для возвращения действительного результата, становится все более распространенным в .NET Framework. Относительно целей нет никаких проблем — идея того, что некоторые методы, возможно, откажутся выполнять свое основное предназначение в неисключительных обстоятельствах, вполне обоснована. Моя проблема в том, что я не особый приверженец выходных параметров. Есть что-то неуклюжее в этом синтаксисе, когда в одной строке объявляется переменная, которая затем немедленно используется в качестве выходного параметра.

В методах, возвращающих ссылочные типы, часто применяется шаблон, который предусматривает возврат значения null в случае отказа и отличного от null значения при успешном выполнении, но он не особенно хорошо работает в ситуации, когда в случае успеха null является допустимым возвращаемым значением. Примером этих двух утверждений может быть тип Hashtable, хотя и слегка противоречивым образом. Теоретически null — допустимое значение в Hashtable, но согласно моему опыту, в подавляющем большинстве ситуаций в Hashtable значения null не используются, поэтому вполне приемлемо предполагать в коде, что значение null соответствует отсутствующему ключу.

Распространенный сценарий заключается в том, что каждое значение в Hashtable определяется в виде списка: при первом добавлении элемента для отдельного ключа создается новый список и к нему добавляется элемент. После этого добавление еще одного элемента для того же самого ключа приводит к добавлению элемента в существующий список. Ниже показан код на C# 1:

```
ArrayList list = hash[key];
if (list == null)
{
   list = new ArrayList();
   hash[key] = list;
}
list.Add(newItem);
```

Скорее всего, имена переменных у вас будут больше соответствовать конкретной ситуации, но я уверен, что вы уловили идею и, возможно, воспользуетесь этим шаблоном в своем коде[4]. Благодаря типам, допускающим null, указанный шаблон может охватить также типы значений. В случае типов значений он даже *безопаснее*, поскольку если нормальным возвращаемым типом является тип значения, то значение null могло бы возвращаться *только* в результате отказа. Типы, допускающие null, добавляют такую дополнительную порцию булевского типа в общем виде при поддержке языка, так почему бы ни прибегнуть к ним?

Для демонстрации этого шаблона на практике и в контексте, отличающемся от поиска в словаре, предлагается рассмотреть классический пример применения шаблона TryXXX() — синтаксический разбор целого числа. Реализация метода TryParse() в листинге 4.5 отражает версию этого шаблона, использующую выходной параметр, но в главной части внизу присутствует также и версия, в которой применяется тип, допускающий null.

Листинг 4.5. Альтернативная реализация шаблона TryXXX()

```
static int? TryParse(string text)
{
  int ret;
  if (int.TryParse(text, out ret))
  {
    return ret;
  }
  else
  {
    return null;
  }
}
...
int? parsed = TryParse("Not valid");
if (parsed != null)
{
  Console.WriteLine ("Parsed to {0}", parsed.Value);
}
else
{
  Console.WriteLine ("Couldn't parse"); // Не удается провести разбор
}
```

Может показаться, что показанные две версии мало чем отличаются — в конце концов, даже количество строк у них одинаковое. Но я считаю, что есть разница в акцентах. Версия с типом, допускающим null, инкапсулирует естественное возвращаемое значение и признак успешности или отказа внутри одной переменной. По моему мнению, он также отделяет *действие* от *проверки*, что смещает акцент в правильную сторону. Обычно если какой-то метод вызывается в части условия оператора if, то его главной целью является возврат булевского значения. Но в определенном смысле возвращаемое

[4] Разве не было бы великолепно, если бы типы Hashtable и Dictionary<TKey, TValue> имели возможность принимать делегат, который должен вызываться всякий раз, когда требуется новое значение из-за обращения к несуществующему ключу? Это позволило бы намного упростить ситуации вроде описываемой.

значение здесь обладает меньшей важностью, чем выходной параметр. В итоге при чтении кода выходной параметр внутри вызова метода довольно легко упустить из виду, а потом удивляться, откуда волшебным образом был получен результат. Версия с типом, допускающим `null`, позволяет выразить намерение более ясно — результат выполнения метода содержит всю интересующую вас информацию. Я пользовался таким приемом во многих местах (часто с большим числом параметров метода, когда обнаружить выходные параметры было еще труднее), и уверен, что он улучшает общий настрой кода. Конечно, прием работает только с типами значений.

Другое преимущество этого шаблона связано с тем, что его можно применять в сочетании с операцией объединения с `null` — попробуйте реализовать распознавание последовательности входных значений, останавливаясь на первом допустимом значении. Обычный шаблон `TryXXX()` позволяет делать это с помощью сокращенно вычисляемых операций, но понять оператор, в котором одна и та же переменная используется для двух выходных параметров, уже не так легко.

Использование кортежа в качестве альтернативы...

Еще одной альтернативой использованию для представления результата типа, допускающего `null`, является применение возвращаемого типа с двумя очень четко разделенными членами, один из которых отвечает за указание на успех или отказ, в другой — за предоставление значения в случае успеха. Удобным для этого типом является `Nullable<T>`, т.к. в нем определено булевское свойство и свойство типа `T`, но *смысл* возвращаемого значения можно было бы сделать более ясным. В состав .NET 4 входит семейство типов `Tuple`: так может быть тип `Tuple<int, bool>` здесь окажется яснее, нежели `int?`. Даже более ясным может быть специальный тип, предназначенный для представления результата операции синтаксического разбора: `ParseResult<T>`, например. В этом случае значение можно было бы передать другому коду, не опасаясь, что его смысл будет завуалирован, и добавить дополнительную информацию, такую как причина возникновения отказа при разборе.

Следующий шаблон решает специфичную проблему — неудобство и малозначащий код, который может сопровождать написание многоуровневых сравнений.

4.4.2. Безболезненные сравнения с использованием операции объединения с `null`

Подозреваю, что вам, как и мне, совершенно не нравится писать один и тот же код снова и снова. Рефакторинг часто может помочь избавиться от дублирования, но в некоторых ситуациях он сталкивается с удивительно стойким сопротивлением. Зачастую в эту категорию попадает код для методов `Equals()` и `Compare()`.

Предположим, что вы занимаетесь построением сайта электронной коммерции и имеете дело со списком товаров. Может возникнуть необходимость в сортировке товаров по популярности (по убыванию), затем по цене и, наконец, по имени, чтобы самые популярные товары (с пятизвездочным рейтингом) находились в списке первыми, начиная с менее дорогих и заканчивая более дорогими. Если есть товары с одинаковыми ценами, то товары с названиями, начинающимися с буквы *A*, располагаются в списке перед товарами, названия которых начинаются с буквы *B*. Эта задача не является специфичной только для сайтов электронной коммерции; сортировка данных по множеству критериев — довольно распространенное требование при организации вычислений.

Предполагая наличие подходящего типа Product, реализовать сравнение можно было бы с помощью следующего кода C# 1:

```
public int Compare(Product first, Product second)
{
  // Обратное сравнение популярности для сортировки по убыванию
  int ret = second.Popularity.CompareTo(first.Popularity);
  if (ret != 0)
  {
    return ret;
  }
  ret = first.Price.CompareTo(second.Price);
  if (ret != 0)
  {
    return ret;
  }
  return first.Name.CompareTo(second.Name);
}
```

В коде принято допущение, что сравнение ссылок null запрашиваться не будет, а все свойства будут возвращать ссылки, не равные null. Для обработки таких случаев можно было бы предусмотреть упреждающие сравнения с null и свойство Comparer<T>.Default, но это привело бы к дополнительному росту объема кода и решению новых проблем. Код можно сократить (избежав возврата из середины метода), слегка переупорядочив его, однако по-прежнему должен присутствовать фундаментальный шаблон "сравнение, проверка, сравнение, проверка", и завершение работы после получения ненулевого ответа не будет настолько очевидным.

Последнее предложение напоминает кое-что другое: операцию объединения с null. Как было показано в разделе 4.3, при наличии множества выражений, разделенных посредством ??, эта операция будет многократно применяться, пока не столкнется с выражением, не равным null. Теперь осталось лишь выработать способ возврата из метода сравнения null вместо нуля. Это легко сделать в отдельном методе, в котором можно также инкапсулировать использование стандартного компаратора. При желании можно даже иметь перегруженную версию для применения специфичного компаратора. Можно также учесть случай, когда любая из передаваемых ссылок на Product является null.

Для начала рассмотрим класс, реализующий вспомогательные методы, как показано в листинге 4.6.

Листинг 4.6. Вспомогательный класс, предназначенный для обеспечения частичных сравнений

```
public static class PartialComparer
{
  public static int? Compare<T>(T first, T second)
  {
    return Compare(Comparer<T>.Default, first, second);
  }

  public static int? Compare<T>(IComparer<T> comparer,
                                T first, T second)
  {
    int ret = comparer.Compare(first, second);
    return ret == 0 ? new int?() : ret;
  }
```

```
    public static int? ReferenceCompare<T>(T first, T second)
      where T : class
    {
       return first == second ? 0
            : first == null ? -1
            : second == null ? 1
            : new int?();
    }
}
```

Методы `Compare()` в листинге 4.6 трогательно просты — когда компаратор не указан, применяется стандартный компаратор для типа, и нулевое возвращаемое значение сравнения просто транслируется в значение `null`.

Значения null и условная операция

Вас может удивить использование конструкции `new int?()`, а не `null`, для возврата значения `null` во втором методе `Compare()`. Условная операция требует, чтобы ее второй и третий операнды либо имели один и тот же тип, либо существовало неявное преобразование из типа одного операнда в тип другого, а в случае `null` это не так, поскольку компилятору не известно, к какому типу должно было относиться значение. При исследовании подвыражений правила языка не учитывают общую цель оператора (возвращающего из метода значение типа `int?`). Другие варианты включают или явное приведение операнда к типу `int?`, или применение для значения `null` конструкции `default(int?)`. В принципе, важно убедиться, что один из операндов в точности является значением типа `int?`.

В методе `ReferenceCompare()` используется еще одна условная операция — на самом деле их три. Вы можете счесть такой код менее читабельным, чем (более длинный) эквивалентный код, в котором применяются блоки `if/else` — все зависит от того, насколько вам удобно пользоваться условной операцией. Мне нравится такой подход тем, что он делает очевидным порядок проведения сравнений. Кроме того, это можно было бы легко организовать в виде необобщенного метода с двумя параметрами типа `object`, но форма с условными операциями предотвращает непреднамеренное применение метода для сравнения значений через упаковку. В действительности метод пригоден только для ссылочных типов, что отражено ограничением параметра типа.

Несмотря на простоту, этот класс удивительно полезен. Теперь предыдущее сравнение товаров можно заменить более лаконичной реализацией:

```
public int Compare(Product first, Product second)
{
  return PC.ReferenceCompare(first, second) ??
         // Обратное сравнение популярности для сортировки по убыванию
         PC.Compare(second.Popularity, first.Popularity) ??
         PC.Compare(first.Price, second.Price) ??
         PC.Compare(first.Name, second.Name) ??
         0;
}
```

Наверняка вы обратили внимание, что в коде используется `PC`, а не `PartialComparer`. Это сделано для того, чтобы не выйти за пределы печатной строки в книге. В реальном

коде было бы указано полное имя типа и по-прежнему по одному сравнению в строке. Разумеется, если по какой-то причине вы хотите сократить длину самих строк, можете с помощью директивы `using` определить PC как псевдоним для `PartialComparer` — но я не рекомендую поступать так.

Финальная константа `0` означает, что если все предшествующие сравнения прошли успешно, то два экземпляра `Product` являются эквивалентными. В качестве завершающего сравнения *можно было бы* просто указать `Comparer<string>.Default.Compare(first.Name, second.Name)`, но это нарушило бы гармонию метода.

Реализованное сравнение эффективно работает со значениями `null`, легко модифицируется, формирует простой шаблон для применения с другими сравнениями и выполняет сравнение только в случае необходимости — если цены отличаются, названия не сравниваются.

Может возникнуть вопрос о том, применим ли тот же самый прием к проверкам на предмет эквивалентности, которые часто имеют похожие шаблоны. В случае эквивалентности сложностей намного меньше, поскольку после выполнения проверок на `null` и ссылочную эквивалентность можно просто использовать операцию `&&` для обеспечения функциональности сокращенных вычислений с булевскими значениями. Тем не менее, для получения начального результата на основе ссылок вида *определенно равно*, *определенно не равно* или *неизвестно* может применяться метод, возвращающий `bool?`. Полный код для `PartialComparer`, доступный для загрузки, содержит подходящий служебный метод и примеры его использования.

4.5. Резюме

Столкнувшись с проблемой, разработчики склонны выбирать самое простое краткосрочное решение, даже если оно выглядит не особенно элегантно. Часто это решение и является правильным — в конце концов, вам же не нужны обвинения в выполнении излишней работы. Всегда приятно, когда *хорошее* решение также оказывается *простейшим*.

Типы, допускающие `null`, решают конкретную проблему, которая единственная имела неуклюжие решения до появления C# 2. Предоставленные возможности позволяют получить лучше поддерживаемую версию решения, которая была осуществима в C# 1, но требовала больших временных затрат. Сочетание обобщений (позволяющих избежать дублирования кода), поддержки со стороны среды CLR (для обеспечения подходящего поведения упаковки и распаковки) и языковой поддержки (для предоставления согласованного синтаксиса наряду с удобными преобразованиями и операциями) делает решение намного более мощным и убедительным, чем было ранее.

Кроме того, в результате появления типов, допускающих `null`, проектировщики языка C# и самой инфраструктуры сделали доступными несколько других шаблонов, которые ранее не стоили усилий по их реализации. В главе были описаны некоторые из них, и я не удивлюсь, если со временем будут появляться другие шаблоны подобного рода.

До сих пор обобщения и типы, допускающие `null`, касались областей, где временами в версии C# 1 приходилось сталкиваться с признаками плохого кода. Этот шаблон будет продолжен в следующей главе, в которой мы обсудим делегаты. Они формируют важную часть тонкого изменения направленности языка C# и инфраструктуры .NET Framework в более функциональную сторону. В версии C# 3 этот акцент сделан даже еще более явным, и хотя мы пока не рассматривали данные средства *вообще*, можно утверждать, что усовершенствования делегатов в C# 2 действуют в качестве моста между знанием C# 1 и идиоматическим стилем C# 3, который часто может радикально отличаться от предшествующих версий.

Глава 5

Оперативно о делегатах

В этой главе...

- Многословный синтаксис C# 1
- Упрощенное конструирование делегатов
- Ковариантность и контравариантность
- Анонимные методы
- Захваченные переменные

Тема делегатов в C# и .NET относится к числу самых интересных и демонстрирует поразительную предусмотрительность (или действительное везение) определенной части команды проектировщиков. Соглашения, предложенные для обработчиков событий в .NET 1.0/1.1, не имели особого смысла вплоть до выхода версии C# 2. Аналогичным образом усилия, вложенные в делегаты для C# 2, выглядят в некоторых отношениях непропорциональными широте их использования — до тех пор, пока вы не увидите, насколько глубоко они проникают в идиоматический код на C# 3. Другими словами, создается впечатление, что у проектировщиков языка и платформы было представление о том, каким должно быть, по меньшей мере, примерное направление движения, за годы до того, как прояснилась сама цель.

Конечно, сама по себе версия C# 3 не является конечным пунктом назначения. Обобщенные делегаты обрели еще более высокую гибкость в C# 4, в версии C# 5 упростилось написание асинхронных делегатов, а в будущем мы можем увидеть даже больше улучшений, однако отличия между версиями C# 1 и C# 3 в этой области являются наиболее удивительными. (Основное изменение в поддержке делегатов C# 3 касается лямбда-выражений, которые будут рассматриваться в главе 9.)

В отношении делегатов версия C# 2 представляет собой своего рода стартовую площадку. Ее новые средства устилают путь к резким изменениям в версии C# 3, сохраняя *разумное* удобство для разработчиков и в то же время предоставляя полезные преиму-

щества. Из достоверных источников мне известно, что проектировщики языка были осведомлены, что комбинированный набор возможностей версии C# 2 открывал новые пути к восприятию кода, но они не обязательно знали, куда приведут эти пути. Пока что их инстинкты оказались удивительно благотворными в области делегатов.

Делегаты играют более заметную роль в .NET 2.0, чем в предшествующих версиях, хотя и не такую значительную, как в .NET 3.5. В главе 3 было показано, как их можно применять для преобразования из одного типа списка в другой, а в главе 1 осуществлялась сортировка списка товаров с использованием делегата `Comparison` вместо интерфейса `IComparer`. Хотя между инфраструктурой и языком C# сохраняется почтительное расстояние, где только возможно, я уверен, что язык и платформа в этом случае влияли друг на друга: добавление поддержки больше ориентированных на делегаты обращений к API-интерфейсам улучшает синтаксис, доступный в C# 2, и наоборот.

В этой главе мы рассмотрим два небольших изменения в C# 2, которые позволяют упростить создание экземпляров делегатов из обычных методов, и затем взглянем на крупнейшее изменение — анонимные методы, которые позволяют указывать действие экземпляра делегата прямо в точке его создания. Самый большой по объему раздел главы выделен для освещения наиболее сложной части в рамках темы анонимных методов — захваченным переменным, которые предоставляют экземплярам делегатов среду с увеличенными возможностями. Ввиду важности и сложности эта тема будет раскрыта максимально подробно. После того как вы разберетесь с анонимными методами, понять лямбда-выражения не составит особого труда.

Однако, прежде всего, давайте вспомним недостатки делегатов в версии C# 1.

5.1. Прощание с неуклюжим синтаксисом для делегатов

Синтаксис для делегатов в версии C# 1 не *выглядит* слишком плохим — язык уже располагает синтаксическим сахаром для `Delegate.Combine()`, `Delegate.Remove()`, и обращения к экземплярам делегатов. В результате тип делегата имеет смысл указывать при создании экземпляра делегата; в конце концов, это тот же самый синтаксис, который применяется при создании экземпляров других типов.

Все это верно, но по ряду причин также и печально. Трудно точно сказать, почему выражения для создания делегатов в C# 1 вызывают отвращение, но это так — во всяком случае, у меня. Привязывая набор обработчиков событий, я считаю неуклюжей необходимость повсеместного написания конструкций `new EventHandler` (или всего, что требуется), в то время как само событие указывает тип делегата, который оно будет использовать. Конечно, на вкус и цвет товарищей нет, и вы могли бы привести аргументы в пользу того, что при чтении кода привязки обработчиков событий в стиле C# 1 приходится меньше строить догадок, однако дополнительный текст только отвлекает внимание от важной части кода — метода, который должен обрабатывать событие.

Все становится еще более сложным, когда подумать о ковариантности и контравариантности применительно к делегатам. Предположим, что имеется метод обработки события, который сохраняет текущий документ, либо фиксирует в журнале факт своего вызова, либо выполняет любое количество других действий, которым могут быть не нужны детальные сведения о событии. Само событие не должно обращать внимание на то, что метод способен работать только с информацией, предоставляемой сигнатурой `EventHandler`, даже если событие объявлено для передачи деталей о событии мыши.

К сожалению, в C# 1 для каждой отличающейся сигнатуры обработчика событий необходимо иметь свой метод.

Точно так же вызывает сомнения необходимость в написании методов, реализация которых оказывается короче их сигнатуры, исключительно из-за того, что делегаты должны иметь предназначенное для выполнения действие в форме метода. Это добавляет дополнительный уровень косвенности между кодом, *создающим* экземпляр делегата, и кодом, который должен быть выполнен при его вызове. Такие дополнительные уровни косвенности почти всегда только приветствуются, и эта возможность не была устранена из C# 2, но в то же время они часто приводят к усложнению восприятия кода и засорению кода класса множеством методов, которые используются только для делегатов.

Неудивительно, что все эти аспекты в C# 2 были существенно улучшены. Синтаксис по-прежнему может оказаться более многословным, чем того хотелось (до появления лямбда-выражений в C# 3), но разница значительна. Чтобы проиллюстрировать проблему, мы начнем с определенного кода на C# 1, который будем улучшать в следующих двух разделах. В листинге 5.1 строится очень простая форма с кнопкой и производится подписка на три события этой кнопки.

Листинг 5.1. Подписка на три события кнопки

```
static void LogPlainEvent(object sender, EventArgs e)
{
  Console.WriteLine("LogPlain");
}

static void LogKeyEvent(object sender, KeyPressEventArgs e)
{
  Console.WriteLine("LogKey");
}

static void LogMouseEvent(object sender, MouseEventArgs e)
{
  Console.WriteLine("LogMouse");
}
...
Button button = new Button();
button.Text = "Click me";
button.Click      += new EventHandler(LogPlainEvent);
button.KeyPress   += new KeyPressEventHandler(LogKeyEvent);
button.MouseClick += new MouseEventHandler(LogMouseEvent);

Form form = new Form();
form.AutoSize = true;
form.Controls.Add(button);
Application.Run(form);
```

Вывод внутри методов обработки событий предназначен для доказательства работоспособности кода: нажатие клавиши пробела при выделенной кнопке приводит к возникновению событий `Click` и `KeyPress`. Нажатие клавиши <Enter> вызывает событие `Click`, а щелчок на кнопке — события `Click` и `MouseClick`. В следующих разделах мы усовершенствуем этот код с применением ряда средств C# 2.

Давайте начнем с того, что предложим компилятору сделать довольно очевидный вывод — какой тип делегата необходимо использовать при подписке на событие.

5.2. Преобразования групп методов

В C# 1 для создания экземпляра делегата должны быть указаны тип делегата и действие. В главе 2 *действие* определялось как метод, предназначенный для вызова, а *цель* (для методов экземпляра) — как объект, на котором этот метод должен вызываться.

Например, в листинге 5.1 для создания экземпляра делегата `KeyPressEventHandler` применялось следующее выражение:

```
new KeyPressEventHandler(LogKeyEvent)
```

Как отдельное выражение, оно не выглядит слишком плохо. Даже при простой подписке на событие это вполне сносно. Однако конструкция становится неуклюжей, когда она является частью более длинного выражения. Распространенным примером может служить запуск нового потока:

```
Thread t = new Thread(new ThreadStart(MyMethod));
```

Все, что здесь требуется сделать — запустить новый поток, который будет выполнять метод `MyMethod()`. Как обычно, желательно выразить все максимально просто, и версия C# 2 позволяет реализовать это посредством неявного преобразования *группы методов* в совместимый тип делегата. Группа методов — это просто имя метода с необязательной целью, т.е. точно такой же вид выражения, который использовался в C# 1 для создания экземпляров делегатов. (На самом деле тогда само выражение называлось группой методов — из-за того, что такое преобразование просто не было доступно.) Если метод является обобщенным, то группа методов может также указывать аргументы типов, хотя, согласно моему опыту, это встречается редко. Новое неявное преобразование позволяет реализовать подписку на событие следующим образом:

```
button.KeyPress += LogKeyEvent;
```

Аналогично, становится более простым также и код создания потока:

```
Thread t = new Thread(MyMethod);
```

Разница в читабельности исходной и упрощенной версий не так велика для одной строки, но в контексте значительного объема кода они могут существенно сократить беспорядок. Чтобы это не выглядело какой-то магией, рассмотрим, что делает указанное преобразование.

Для начала взглянем на выражения `LogKeyEvent()` и `MyMethod()` из приведенных примеров. Причина их классификации как *групп* методов в том, что из-за перегрузки может быть доступно более одного метода. Доступное неявное преобразование будет преобразовывать группу методов в любой тип делегата с совместимой сигнатурой. Пусть имеются две сигнатуры методов, показанные ниже:

```
void MyMethod()
void MyMethod(object sender, EventArgs e)
```

Тогда `MyMethod()` можно применить как группу методов в присваивании либо `ThreadStart`, либо `EventHandler`:

```
ThreadStart x = MyMethod;
EventHandler y = MyMethod;
```

Тем не менее, его *нельзя* использовать в качестве параметра метода, который сам был перегружен для приема экземпляра `ThreadStart` или `EventHandler` — компилятор сообщит о том, что такой вызов неоднозначен. Подобным же образом, к сожалению,

преобразование группы методов невозможно применять для преобразования в простой тип System.Delegate, поскольку компилятору не известен конкретный тип делегата, экземпляр которого должен быть создан. Возникает сложность, но, во всяком случае, можно еще немного сократить код по сравнению с кодом на C# 1, сделав преобразование явным. Ниже приведен пример:

```
Delegate invalid = SomeMethod;
Delegate valid = (ThreadStart)SomeMethod;
```

Для локальных переменных это обычно не проблема, но становится таковой при использовании API-интерфейса, который имеет параметр типа Delegate, такой как Control.Invoke(). Здесь существует несколько решений: применение вспомогательного метода, приведение или использование промежуточной переменной. Далее представлен пример применения типа делегата MethodInvoker, который не принимает параметров и не имеет возвращаемого типа:

```
static void SimpleInvoke(Control control, MethodInvoker invoker)
{
  control.Invoke(invoker);
}
...
SimpleInvoke(form, UpdateUI);                    <-- Вызов вспомогательного метода
form.Invoke((MethodInvoker)UpdateUI);            <-- Вызов приведения
MethodInvoker invoker = UpdateUI;
form.Invoke(invoker);                            <-- Вызов с помощью локальной переменной
```

Разные ситуации способствуют построению разных решений; ни одно из них не является особо привлекательным, однако их нельзя считать и ужасными[1].

Как и в случае обобщений, точные правила для определения допустимости преобразования довольно сложны, поэтому хорошо работает подход с апробированием; если компилятор уведомит о том, что ему не хватает информации, нужно лишь сообщить ему, какое преобразование использовать, и все должно пройти нормально. За подробными сведениями обращайтесь в раздел 6.6 ("Method group conversions" ("Преобразования групп методов")) спецификации языка. Возможных преобразований может быть больше, чем кажется, в чем вы сможете убедиться в следующем разделе.

5.3. Ковариантность и контравариантность

Мы уже немало говорили о концепциях ковариантности и контравариантности в различных контекстах, обычно сожалея об их отсутствии, но конструирование делегатов является одной из областей, в которых ковариантность и контравариантность были доступны до выхода версии C# 4. Эти термины относительно подробно были описаны в разделе 2.2.2. Суть темы, касающейся делегатов, выражается так: если вызов метода и применение его возвращаемого значения допустимо (в смысле статической типизации) во всех местах, где взамен можно было бы обратиться к экземпляру конкретного типа делегата и использовать *его* возвращаемое значение, то такой метод может применяться для создания экземпляра этого типа делегата. Прояснить это лучше всего на примерах.

[1] Расширяющие методы (обсуждаемые в главе 10) делают подход со вспомогательным методом несколько более привлекательным при использовании версии C# 3.

Различные типы вариантности в разных версиях

Возможно, вам уже известно, что в версии C# 4 предлагаются *обобщенные* ковариантность и контравариантность для делегатов и интерфейсов. Это полностью отличается от вариантности, показанной до сих пор — в данный момент мы имеем дело с созданием *новых* экземпляров делегатов. Обобщенная вариантность в C# 4 использует *ссылочные преобразования*, которые не создают новые объекты — они просто представляют существующий объект как относящийся к другому типу.

Сначала мы рассмотрим контравариантность, а затем ковариантность.

5.3.1. Контравариантность для параметров делегата

Давайте возвратимся к обработчикам событий из небольшого приложения Windows Forms, которое было показано в листинге 5.1. Сигнатуры трех типов делегатов выглядят следующим образом[2]:

```
void EventHandler(object sender, EventArgs e)
void KeyPressEventHandler(object sender, KeyPressEventArgs e)
void MouseEventHandler(object sender, MouseEventArgs e)
```

Представим себе, что типы `KeyPressEventArgs` и `MouseEventArgs` являются производными от `EventArgs` (как и множество других типов — в MSDN упоминается о 403 типах, напрямую унаследованных от `EventArgs` в .NET 4). Если есть метод с параметром `EventArgs`, то его всегда можно вызвать с аргументом `KeyPressEventArgs`. Следовательно, появляется смысл в наличии возможности использовать метод с такой же сигнатурой, как у `EventHandler()`, для создания экземпляра `KeyPressEventHandler()`, и именно так сделано в C# 2. Это пример контравариантности типов параметров.

Для демонстрации этого в действии снова возвратимся к листингу 5.1 и предположим, что знание того, какое событие произошло, не требуется — нужно лишь зафиксировать факт возникновения события. За счет применения преобразований групп методов и контравариантности код намного упрощается, как показано в листинге 5.2.

Листинг 5.2. Демонстрация преобразований групп методов и контравариантности делегатов

```
static void LogPlainEvent(object sender, EventArgs e) ←❶ Обработка всех событий
{
  Console.WriteLine("An event occurred"); // Произошло какое-то событие
}
...
Button button = new Button();
button.Text = "Click me";
button.Click += LogPlainEvent;          ←❷ Использование преобразования группы методов
button.KeyPress += LogPlainEvent;       ┐
button.MouseClick += LogPlainEvent;     ┘←❸ Использование преобразования и контравариантности

Form form = new Form();
form.AutoSize = true;
form.Controls.Add(button);
Application.Run(form);
```

[2] Часть `public delegate` была удалена ради краткости.

Два метода обработчиков, которые имели дело с событиями клавиатуры и мыши, были устранены и теперь для обработки всех событий используется один метод ❶. Разумеется, это не особенно полезно, если нужно проводить отличия между разными типами событий, но иногда все, что требуется знать — это сам факт возникновения события и, возможно, его источник. При подписке на событие `Click` ❷ применяется только неявное преобразование, которое обсуждалось в предыдущем разделе, т.к. оно имеет простой параметр `EventArgs`, но другие подписки на события ❸ предусматривают использование преобразования и контравариантности из-за разных типов параметров.

Ранее упоминалось, что соглашение относительно обработчиков событий в .NET 1.0/1.1 не имело большого смысла, когда оно было впервые введено. Данный пример наглядно демонстрирует, почему руководящие принципы более полезны в версии C# 2. Соглашение предписывает, что обработчики событий должны иметь сигнатуру с двумя параметрами, первый из которых относится к типу `object` и представляет источник события, а второй хранит дополнительную информацию о событии в экземпляре типа, производного от `EventArgs`. До того как стала доступной контравариантность, это было бесполезно — определение параметра для дополнительной информации с типом, унаследованным от `EventArgs`, не давало никаких преимуществ, а иногда не было особого смысла в работе с источником события. Часто было более разумно передавать необходимую информацию непосредственно в виде обычных параметров подходящих типов, как это делается в случае любого другого метода. Теперь можно применять метод с сигнатурой `EventHandler()` как действия для *любого* типа делегата, который следует соглашению.

До сих пор мы имели дело с входными значениями метода или делегата, а что можно сказать о выходном значении?

5.3.2. Ковариантность возвращаемых типов делегатов

Продемонстрировать ковариантность труднее, т.к. лишь относительно немногие делегаты, доступные в .NET 2.0, объявлены с возвращаемым типом, отличным от `void`, и они, как правило, возвращают типы значений. Некоторые делегаты все же доступны, но проще объявить собственный тип делегата, использующий `Stream` в качестве возвращаемого типа. Для еще большей простоты он не будет принимать параметры[3]:

```
delegate Stream StreamFactory();
```

Теперь этот тип можно применять с методом, который объявлен как возвращающий специальный тип потока (листинг 5.3). Объявленный метод всегда возвращает экземпляр `MemoryStream` с последовательностью данных (байты 0, 1, 2 и так далее до 15). Этот метод используется в качестве действия для экземпляра делегата `StreamFactory`.

Листинг 5.3. Демонстрация ковариантности возвращаемых типов для делегатов

```
delegate Stream StreamFactory(); <—❶ Объявление типа делегата, возвращающего тип Stream
static MemoryStream GenerateSampleData()   <—❷ Объявление метода, возвращающего тип MemoryStream
{
  byte[] buffer = new byte[16];
  for (int i = 0; i < buffer.Length; i++)
```

[3] Ковариантность возвращаемых типов и ковариантность типов параметров могут применяться одновременно, но вряд ли вы столкнетесь с ситуациями, в которых это бы пригодилось.

```
    {
        buffer[i] = (byte) i;
    }
    return new MemoryStream(buffer);
}
...
StreamFactory factory = GenerateSampleData;   <-- Преобразование группы методов с помощью ковариантности

using (Stream stream = factory())   <-- Обращение к делегату для получения экземпляра потока
{
    int data;
    while ((data = stream.ReadByte()) != -1)
    {
        Console.WriteLine(data);
    }
}
```

Генерация и отображение данных в листинге 5.3 нужны просто для того, чтобы код делал что-нибудь полезное. Важными являются аннотированные строки. В объявленном типе делегата применяется возвращаемый тип `Stream` ❶, но метод `GenerateSampleData()` имеет возвращаемый тип `MemoryStream` ❷. Строка кода, в которой создается экземпляр делегата ❸, выполняет упомянутое ранее преобразование и использует ковариантность возвращаемых типов, чтобы позволить методу `GenerateSampleData()` выступать в качестве действия для делегата `StreamFactory`. До момента вызова экземпляра делегата ❹ компилятору ничего не известно о том, что будет возвращен экземпляр типа `MemoryStream` — если изменить тип переменной `stream` на `MemoryStream`, возникнет ошибка при компиляции.

Ковариантность и контравариантность могут также применяться для конструирования одного экземпляра делегата на основе другого. Например, рассмотрим следующие две строки кода (в которых предполагается наличие соответствующего метода `HandleEvent()`):

```
EventHandler general = new EventHandler(HandleEvent);
KeyPressEventHandler key = new KeyPressEventHandler(general);
```

Первая строка представляет собой допустимый код C# 1, но вторая им не является — чтобы сконструировать в C# 1 один экземпляр делегата из другого, сигнатуры этих двух типов делегатов должны совпадать. Например, экземпляр `MethodInvoker()` можно было бы создать из `ThreadStart()`, но нельзя создать экземпляр `KeyPressEventHandler` из `EventHandler`, как показано во второй строке. Контравариантность используется для создания нового экземпляра делегата на основе существующего экземпляра с *совместимой* сигнатурой типа делегата, причем совместимость в C# 2 определяется в менее ограничивающей манере, чем в версии C# 1.

Все это хорошо за исключением одной небольшой ложки дегтя в бочке меда.

5.3.3. Небольшой риск несовместимости

Появившаяся гибкость C# 2 приводит к возникновению нескольких ситуаций, при которых допустимый код C# 1 может выдавать разные результаты в случае компиляции с помощью компилятора C# 2. Предположим, что в производном классе перегружен метод, определенный в его базовом классе, и предпринимается попытка создания экземпляра делегата с применением преобразования группы методов. Преобразование,

которое ранее соответствовало только методу базового класса, теперь может соответствовать методу производного класса из-за ковариантности или контравариантности в C# 2, в случае чего этот метод производного класса будет выбран компилятором. В листинге 5.4 приведен пример.

Листинг 5.4. Демонстрация нарушающего изменения между C# 1 и C# 2

```
delegate void SampleDelegate(string x);

public void CandidateAction(string x)
{
  Console.WriteLine("Snippet.CandidateAction");
}
public class Derived : Snippet
{
  public void CandidateAction(object o)
  {
    Console.WriteLine("Derived.CandidateAction");
  }
}
...
Derived x = new Derived();
SampleDelegate factory = new SampleDelegate(x.CandidateAction);
factory("test");
```

Вспомните, что инструмент Snippy[4] будет генерировать весь этот код внутри класса по имени `Snippet`, от которого наследуется вложенный тип. В случае C# 1 код из листинга 5.4 выведет на консоль строку `Snippet.CandidateAction`, поскольку метод, принимающий параметр `object`, не был совместимым с `SampleDelegate()`. В случае C# 2 метод *является* совместимым, так что он будет выбран по причине объявления в более производном типе, поэтому на консоль выводится строка `Derived.CandidateAction`.

К счастью, компилятору C# 2 известно, что это нарушающее изменение, и он выдает соответствующее предупреждение. Данный раздел включен потому, что вы должны быть осведомлены о самой возможности такой проблемы, тем не менее, я уверен, что вы редко с ней столкнетесь в реальности.

Потенциальное нарушение достаточно отпугивает. Однако мы пока еще не обратились к самому важному новому средству, относящемуся к делегатам — анонимным методам. Они несколько сложнее, чем рассмотренные до сих пор темы, но они также отличаются *большой* мощностью и являются крупным шагом в сторону версии C# 3.

5.4. Встраивание действий делегатов с помощью анонимных методов

В C# 1 приходилось реализовывать делегат с определенной сигнатурой, даже если уже существовал метод с точно совпадающим поведением, но слегка отличающимся набором параметров. Аналогично, часто делегат нужен для выполнения одной крохотной

[4] На случай, если вы пропустили первую главу: Snippy — это инструмент, построенный мною для поддержки кратких, но вместе с тем полноценных примеров кода. За дополнительными сведениями обращайтесь в раздел 1.8.1.

функции — но для этого необходимо создать целый дополнительный метод. Новый метод представлял бы поведение, которое было подходящим только в рамках исходного метода, но он был бы виден всему классу, добавляя излишние сведения в IntelliSense и в целом создавая нежелательные препятствия.

Все это крайне разочаровывало. Средства ковариантности и контравариантности, о которых шла речь выше, *иногда* могут помочь справиться с первой проблемой, но чаще всего им это не удается. А вот *анонимные методы*, которые также являются нововведением версии C# 2, могут почти *всегда* помочь в решении указанных ранее проблем.

Неформально анонимные методы позволяют указывать действие для экземпляра делегата встроенным образом как часть выражения, создающего этот экземпляр делегата. Они также предоставляют гораздо более мощное поведение в форме *замыканий*, но они будут рассматриваться в разделе 5.5. Пока что давайте придерживаться относительно простой функциональности.

Сначала мы рассмотрим примеры анонимных методов, которые принимают параметры, но не возвращают значений, а затем исследуем синтаксис, используемый для возвращения значений, и также сокращение, доступное на случай, когда переданные значения параметров не нужны.

5.4.1. Начинаем с простого: действие над одним параметром

В .NET 2.0 появился обобщенный тип делегата по имени `Action<T>`, который и будет применяться в последующих примерах. Его сигнатура проста (помимо того факта, что он является обобщенным):

```
public delegate void Action<T>(T obj)
```

Другими словами, делегат `Action<T>` делает что-то со значением типа `T`; например, `Action<string>` мог бы изменять порядок следования символов в строке на противоположный и выводить результат на консоль, `Action<int>` — выводить значение квадратного корня для переданного числа, а `Action<IList<double>>` — находить среднее для всех указанных чисел и выводить его на консоль. В листинге 5.5 все эти примеры реализованы с использованием анонимных методов.

Листинг 5.5. Анонимные методы, работающие с типом делегата `Action<T>`

```
Action<string> printReverse = delegate(string text)
{
  char[] chars = text.ToCharArray();
  Array.Reverse(chars);
  Console.WriteLine(new string(chars));
};
Action<int> printRoot = delegate(int number)
{
  Console.WriteLine(Math.Sqrt(number));
};
Action<IList<double>> printMean = delegate(IList<double> numbers)
{
  double total = 0;
  foreach (double value in numbers)
  {
    total += value;
  }
```

❶ Использование анонимного метода для создания делегата `Action<string>`

● Использование цикла в анонимном методе

```
    Console.WriteLine(total / numbers.Count);
};
printReverse("Hello world");                    <-- ❹ Вызов делегата обычным образом
printRoot(2);
printMean(new double[] { 1.5, 2.5, 3, 4.5 });
```

В листинге 5.5 продемонстрировано несколько возможностей анонимных методов. Первым делом, показан синтаксис анонимных методов: применение ключевого слова `delegate`, за которым следуют параметры (если они предусмотрены), после чего идет код для действия экземпляра делегата в виде блока. Код обращения строки ❶ иллюстрирует, что блок может содержать объявления локальных переменных, а код вычисления среднего значения для списка ❷ показывает организацию цикла внутри блока. По существу в анонимном методе можно делать (почти) все, что допустимо внутри тела обычного метода. Аналогично, результат анонимного метода — это экземпляр делегата, который можно использовать подобно любому другому экземпляру такого рода ❸. Однако имейте в виду, что к анонимным методам контравариантность не применяется; вы должны указывать типы параметров, которые в точности совпадают с типом делегата.

Пара ограничений...

Одна небольшая особенность связана с тем, что если анонимный метод пишется в типе значения, то внутри него нельзя ссылаться на `this`. В ссылочном типе такое ограничение отсутствует. Кроме того, в предлагаемых Microsoft реализациях компиляторов C# 2 и C# 3 доступ к базовому члену внутри анонимного метода через ключевое слово `base` приводило к выдаче предупреждения о том, что результирующий код является не поддающимся проверке. В компиляторе C# 4 данная проблема была устранена.

В терминах реализации для каждого анонимного метода в исходном коде по-прежнему создается метод в коде IL. Компилятор сгенерирует метод внутри существующего класса, а затем будет использовать его в качестве действия при создании экземпляра делегата, как если бы это был обычный метод[5]. Среда CLR не знает и не заботится о том, что применялся анонимный метод. Просмотреть эти дополнительные методы в скомпилированном коде можно с помощью инструмента `ildasm` или Reflector. (Инструменту Reflector известно, как интерпретировать код IL для отображения анонимных методов в методе, который их использует, и дополнительные методы по-прежнему видимы.) Такие методы имеют *непроизносимые имена* — имена, которые допустимы в IL, но недопустимы в C#. Это препятствует попыткам ссылаться на данные методы напрямую в коде C# и устраняет возможность возникновения конфликтов имен. Многие средства C# 2 и последующих версий реализованы похожим образом; проще всего их обнаружить по наличию угловых скобок. Например, анонимный метод внутри метода `Main()` *может* привести к созданию метода по имени `<Main>b__0()`. Тем не менее, это всецело зависит от реализации. Например, в будущей версии компилятора Microsoft могут быть изменены собственные соглашения. Это не должно что-либо нарушить, поскольку ничего не должно полагаться на такие имена.

[5] В разделе 5.5.4 вы увидите, что хотя всегда существует новый метод, он не всегда создается там, где этого можно было ожидать.

На этом этапе полезно отметить, что код в листинге 5.5 совершенно не похож на то, как анонимные методы обычно выглядят в реальном коде. Вы часто будете сталкиваться с их применением в качестве аргументов другого метода (вместо присваивания переменной типа делегата) и они будут разнесены на несколько строк — в конце концов, компактность является одной из причин их использования. Чтобы продемонстрировать это, мы воспользуемся методом `List<T>.ForEach()`, который принимает `Action<T>` как параметр и выполняет это действие над каждым элементом. В листинге 5.6 показан экстремальный пример, в котором применяется то же самое действия извлечения квадратного корня, что и в листинге 5.5, но в компактной форме.

Листинг 5.6. Экстремальный пример компактности кода. Внимание: нечитабельный код!

```
List<int> x = new List<int>();
x.Add(5);
x.Add(10);
x.Add(15);
x.Add(20);
x.Add(25);
x.ForEach(delegate(int n){Console.WriteLine(Math.Sqrt(n));});
```

Код выглядит довольно-таки устрашающе — особенно, с учетом того, что последние шесть символов производят впечатление расположенных наугад. Конечно, существует и золотая середина. В отношении анонимных методов я предпочитаю нарушать свое обычное правило "фигурные скобки в собственной строке" (которое применяю к простым свойствам), но по-прежнему допускаю порядочное количество пробельных символов. Последнюю строку кода из листинга 5.6 я мог бы также написать в следующих двух формах:

```
x.ForEach(delegate(int n)
  { Console.WriteLine(Math.Sqrt(n)); }
);
x.ForEach(delegate(int n) {
  Console.WriteLine(Math.Sqrt(n));
});
```

Даже простое добавление пробелов в код из листинга 5.6 способствует его лучшему пониманию. В каждом из этих форматов круглые и фигурные скобки теперь меньше запутывают, а часть, отвечающая за полезную работу, соответствующим образом выделена. Конечно, разбивка кода всецело зависит от ваших предпочтений, но я рекомендую хорошо подумать о соблюдении определенного баланса и обсудить с членами команды вопросы по достижению некоторой согласованности. Тем не менее, согласованность не *всегда* приводит к получению наиболее читабельного кода — иногда представление всей функциональности в одной строке является самым простым форматом.

До сих пор взаимодействие с вызывающим кодом осуществлялось с помощью параметров. А как насчет возвращаемых значений?

5.4.2. Возвращение значений из анонимных методов

Делегат `Action<T>` имеет возвращаемый тип `void`, поэтому пока еще нечего было возвращать из анонимных методов. Для демонстрации того, как это можно сделать в случае необходимости, мы воспользуемся типом делегата `Predicate<T>` из .NET 2.0, который имеет следующую сигнатуру:

```
public delegate bool Predicate<T>(T obj)
```

В листинге 5.7 приведен код анонимного метода, который создает экземпляр делегата `Predicate<T>` для возвращения признака четности или нечетности передаваемого ему аргумента. Предикаты обычно применяются при фильтрации и сопоставлении – к примеру, код из листинга 5.7 можно было бы использовать для фильтрации списка с целью получения только четных элементов.

Листинг 5.7. Возвращение значения из анонимного метода

```
Predicate<int> isEven = delegate(int x) { return x % 2 == 0; };
Console.WriteLine(isEven(1));
Console.WriteLine(isEven(4));
```

Новый синтаксис почти наверняка выглядит вполне предсказуемо — соответствующее значение возвращается, как если бы анонимный метод был обычным методом. Возможно, вы ожидали увидеть объявление возвращаемого типа поблизости к ключевому слову `delegate`, но в этом нет необходимости. Компилятор проверяет совместимость всех возможных возвращаемых значений с возвращаемым типом, объявленным в типе делегата, при попытке преобразования в него анонимного метода.

Что именно вы собрались возвращать?

Возвращение значения из анонимного метода — это всего лишь возвращение из анонимного метода, а не возвращение из метода, создающего экземпляр делегата. Будьте внимательны, т.к. увидев ключевое слово `return` в некотором коде, легко посчитать его точкой выхода из текущего метода.

Как упоминалось ранее, в .NET 2.0 лишь относительно немногие делегаты возвращают значения, хотя, как будет показано в части 3 этой книги, в .NET 3.5 идея возврата значений применяется гораздо чаще, особенно в случае LINQ. Тем не менее, в .NET 2.0 имеется еще один довольно популярный тип делегата: `Comparison<T>`, который может использоваться при сортировке коллекций. Это эквивалент интерфейса `IComparer<T>` в форме делегата. Часто возникает ситуация, когда необходим только определенный порядок сортировки, в связи с чем нужна возможность указывать требуемый порядок встроенным образом, а не открывать его в виде метода для остальной части класса. Сказанное демонстрируется в листинге 5.8, код в котором выводит на консоль список файлов в каталоге `C:\`, упорядочивая сначала по имени, а затем (отдельно) по размеру.

Листинг 5.8. Использование анонимных методов для простой сортировки имен файлов

```
static void SortAndShowFiles(string title, Comparison<FileInfo> sortOrder)
{
  FileInfo[] files = newDirectoryInfo(@"C:\").GetFiles();
  Array.Sort(files, sortOrder);
  Console.WriteLine(title);
  foreach (FileInfo file in files)
  {
    Console.WriteLine (" {0} ({1} bytes)", file.Name, file.Length);
  }
}
...
```

```
SortAndShowFiles("Sorted by name:", delegate(FileInfo f1, FileInfo f2)
  { return f1.Name.CompareTo(f2.Name); }
);
SortAndShowFiles("Sorted by length:", delegate(FileInfo f1, FileInfo f2)
  { return f1.Length.CompareTo(f2.Length); }
);
```

Если не использовать анонимные методы, то пришлось бы предусматривать отдельный метод для каждого порядка сортировки. Вместо этого в листинге 5.8 сделано очевидным, как будет в каждом случае выполняться сортировка, прямо при вызове метода `SortAndShowFiles()`. (Иногда метод `Sort()` будет вызываться непосредственно в точке, где вызывается анонимный метод. В листинге 5.8 одна и та же последовательность извлечение/сортировка/отображение выполняется дважды, просто с разными порядками сортировки, поэтому указанные шаги инкапсулированы в отдельный метод.)

Временами применимо одно специфичное синтаксическое сокращение. Если вас не заботят параметры делегата, то вы не обязаны даже их объявлять. Давайте посмотрим, как это работает.

5.4.3. Игнорирование параметров делегата

Иногда необходимо реализовать делегат, который не зависит от значений своих параметров. Может понадобиться написать обработчик событий, поведение которого подходит только для одного события и не зависит от аргументов этого события — например, оно сохраняет результаты работы, выполненной пользователем. Обработчики событий из листинга 5.1 идеально соответствуют этим характеристикам. В таком случае можно вообще не включать список параметров, а просто указать ключевое слово `delegate` и затем блок кода в качестве действия для метода. В листинге 5.9 приведен код, эквивалентный коду в листинге 5.1, но использующий более короткий синтаксис.

Листинг 5.9. Подписка на события с применением анонимных методов, которые игнорируют параметры

```
Button button = new Button();
button.Text = "Click me";
button.Click      += delegate { Console.WriteLine("LogPlain"); };
button.KeyPress   += delegate { Console.WriteLine("LogKey"); };
button.MouseClick += delegate { Console.WriteLine("LogMouse"); };
Form form = new Form();
form.AutoSize = true;
form.Controls.Add(button);
Application.Run(form);
```

Обычно каждую подписку пришлось бы представлять в следующем виде:

```
button.Click += delegate(object sender, EventArgs e) { ... };
```

Здесь понапрасну тратится много места без веских на то оснований — значения параметров не нужны, поэтому компилятор разрешает не указывать их вообще.

Я нахожу это сокращение наиболее удобным, когда дело доходит до реализации собственных событий. Например, мне категорически не нравится необходимость в выполнении проверки на предмет `null` перед генерацией события.

Один из способов обойти это предполагает обеспечение того, что событие запускается с помощью обработчика, который затем никогда не удаляется. Учитывая, что этот обработчик ничего не делает, теряется лишь небольшая часть производительности. До появления C# 2 нужно было явно создавать метод с правильной сигнатурой, что не приносило никакой пользы, но теперь можно писать такой код:

```
public event EventHandler Click = delegate {};
```

Начиная с этого момента, метод `Click()` можно просто вызывать безо всяких проверок на предмет `null`.

Вы должны знать об одной ловушке, связанной с этой возможностью подстановки параметров — если анонимный метод может быть преобразован в несколько типов делегатов (например, для вызова различных перегруженных версий метода), то компилятору понадобится дополнительная помощь. Чтобы прояснить, о чем идет речь, возвратимся к тому же самому трудному примеру, который рассматривался для преобразований групп методов: запуск нового потока. В .NET 2.0 доступны четыре конструктора потоков:

```
public Thread(ParameterizedThreadStart start)
public Thread(ThreadStart start)
public Thread(ParameterizedThreadStart start, int maxStackSize)
public Thread(ThreadStart start, int maxStackSize)
```

При этом применяются два типа делегатов:

```
public delegate void ThreadStart()
public delegate void ParameterizedThreadStart(object obj)
```

А теперь взглянем на следующие три попытки создания нового потока:

```
new Thread(delegate()        { Console.WriteLine("t1"); } );
new Thread(delegate(object o) { Console.WriteLine("t2"); } );
new Thread(delegate          { Console.WriteLine("t3"); } );
```

Первая и вторая строки содержат списки параметров — компилятору известно, что анонимный метод из первой строки не может быть преобразован в `ParameterizedThreadStart()`, а анонимный метод из второй строки — в `ThreadStart()`. Эти строки компилируются, поскольку в каждом случае существует только одна подходящая перегруженная версия конструктора. Однако третья строка неоднозначна — анонимный метод может быть преобразован в любой из двух типов делегатов, поэтому применимы обе перегруженных версии конструктора с одним параметром. В такой ситуации компилятор выдает сообщение об ошибке. Решить проблему можно либо явным указанием списка параметров, либо приведением анонимного метода к правильному типу делегата.

Надеюсь, что анонимные методы, которые вы видели до сих пор, поспособствовали определенным размышлениям о вашем коде и возможным применениям этих приемов для достижения хороших результатов. И действительно, даже если бы анонимные методы могли делать *только* то, что было показано ранее, они уже были бы очень удобны. Но анонимные методы позволяют не только избежать определения дополнительных методов в коде. Анонимные методы представляют собой реализацию версией C# 2 средства, известного в других местах как *замыкания* через *захваченные переменные*. В следующем разделе объясняются оба эти термина и показано, что анонимные методы могут оказаться исключительно полезными — но также и запутанными в случае неосторожного применения.

5.5. Захватывание переменных в анонимных методах

Хоть мне и не нравится делать предупреждения, но я думаю, что одно предупреждение вполне уместно: если эта тема нова для вас, не приступайте к изучению этого раздела без достаточного понимания и готовности уделить ему должное время. Не хочу вас попусту тревожить, к тому же вы должны быть уверены в том, что нет настолько сложного материала, который невозможно понять, приложив лишь небольшие усилия. Но вот только захваченные переменные могут поначалу выглядеть запутанными, отчасти из-за того, что противоречат существующим знаниям и интуиции.

Тем не менее, придерживайтесь их! Взамен вы получите невероятную простоту и читабельность кода. Эта тема также будет критически важной при рассмотрении лямбда-выражений и LINQ в C# 3, так что ей стоит уделить внимание.

Итак, начнем с нескольких определений.

5.5.1. Определение замыканий и различных типов переменных

Концепция *замыканий* далеко не нова; она впервые была реализована в языке Scheme, но в последние годы она обретает все большую известность по мере того, как ее берут на вооружение ведущие языки программирования. Базовая идея заключается в том, что некоторая функция[6] способна взаимодействовать со средой за пределами предоставляемых ей параметров. В абстрактных терминах больше сказать нечего, но для понимания, как это применимо к C# 2, понадобится пара дополнительных терминов.

- *Внешняя переменная* — это локальная переменная или параметр (кроме параметров `ref` и `out`), область действия которой включает анонимный метод. Ссылка `this` также считается внешней переменной любого анонимного метода внутри члена экземпляра класса.
- *Захваченная внешняя переменная* (обычно для краткости называемая *захваченной переменной*) — это внешняя переменная, которая используется внутри анонимного метода. Возвращаясь к замыканиям, функциональная часть является анонимным методом, а среда, с которой этот метод может взаимодействовать — набором захваченных им переменных.

Возможно, это крайне сухое изложение трудно понять, но основной смысл в том, что в анонимном методе могут применяться локальные переменные, определенные внутри метода, в котором объявлен этот анонимный метод. Сказанное может не выглядеть особенно важным, но во многих ситуациях это чрезвычайно удобно — можно использовать имеющуюся в распоряжении контекстную информацию вместо того, чтобы настраивать дополнительные типы лишь для хранения данных, которые уже известны. Вскоре мы рассмотрим полезные конкретные примеры, но до этого имеет смысл взглянуть на код, проясняющий приведенные выше определения.

В листинге 5.10 представлен пример с несколькими локальными переменными и единственным методом, поэтому код не может быть запущен сам по себе. В данный момент объяснения, как этот код работает, приводиться не будут, а речь пойдет о классификации разных переменных. Тип делегата `MethodInvoker` применяется в целях простоты.

---

[6] Это общая терминология, принятая в вычислительной технике, а не терминология языка C#.

Листинг 5.10. Примеры видов переменных в отношении анонимных методов

```
void EnclosingMethod()
{
  int outerVariable = 5;                                  ← ❶ Внешняя переменная (незахваченная)
  string capturedVariable = "captured";                   ← ❷ Внешняя переменная, захваченная
                                                              анонимным методом
  if (DateTime.Now.Hour == 23)
  {
    int normalLocalVariable = DateTime.Now.Minute;        ← ❸ Локальная переменная
                                                              обычного метода
    Console.WriteLine(normalLocalVariable);
  }
  MethodInvoker x = delegate()
  {
    string anonLocal = "local to anonymous method";       ← ❹ Локальная переменная
                                                              анонимного метода
    Console.WriteLine(capturedVariable + anonLocal);      ← ❺ Захват внешней переменной
  };
  x();
}
```

Давайте пройдемся по всем переменным, начиная с простейшей и заканчивая самой сложной.

- `normalLocalVariable` ❸ не является внешней переменной, т.к. в рамках ее области действия анонимные методы отсутствуют. Ее поведение в точности совпадает с обычным поведением локальных переменных.
- `anonLocal` ❹ также не является внешней переменной, но представляет собой локальную переменную в анонимном методе, а не в методе `EnclosingMethod()`. Она существует (с точки зрения присутствия в стековом фрейме во время выполнения), только когда производится обращение к экземпляру делегата.
- `outerVariable` ❶ является внешней переменной, потому что в области ее действия объявлен анонимный метод. Однако в анонимном методе отсутствуют ссылки на нее, поэтому данная переменная не захвачена.
- `capturedVariable` ❷ является внешней переменной, поскольку в области ее действия объявлен анонимный метод и в силу того, что эта переменная используется внутри анонимного метода, она становится *захваченной* ❺.

Теперь вы знаете терминологию, но пока еще не сильно приблизились к пониманию того, что делают захваченные переменные. Я подозреваю, что вы смогли бы предугадать вывод, получаемый в результате выполнения метода из листинга 5.10, тем не менее, имеется ряд других случаев, которые, возможно, вызовут удивление. Мы начнем с простого примера и построим на его основе более сложные примеры.

5.5.2. Исследование поведения захваченных переменных

На самом деле при захвате переменной анонимным методом захватывается сама *переменная*, а не значение, которое она имела при создании экземпляра делегата. Позже вы увидите, что это имеет далеко идущие последствия, но сначала необходимо понять смысл данного утверждения в сравнительно простой ситуации.

В листинге 5.11 имеется захваченная переменная и анонимный метод, который выводит на консоль значение этой переменной и затем изменяет его. Вы заметите, что

изменение значения переменной за пределами анонимного метода видно внутри анонимного метода и наоборот.

Листинг 5.11. Доступ к переменной внутри и снаружи анонимного метода

```
string captured = "before x is created"; // перед созданием x
MethodInvoker x = delegate
{
  Console.WriteLine(captured);
  captured = "changed by x";              // изменено внутри x
};
captured = "directly before x is invoked";// непосредственно перед вызовом x
x();
Console.WriteLine(captured);
captured = "before second invocation";   // после второго вызова
x();
```

Вывод кода из листинга 5.11 выглядит следующим образом:

```
directly before x is invoked
changed by x
before second invocation
```

Давайте посмотрим, почему так происходит. Прежде всего, объявляется переменная captured, значение которой устанавливается с помощью обычного строкового литерала. До сих пор с поведением переменной не связано ничего особенного. Затем объявляется делегат x и его значение устанавливается с применением анонимного метода, который захватывает переменную captured. Экземпляр делегата будет всегда выводить на консоль текущее значение captured, после чего устанавливать его в "changed by x". Не забывайте, что создание экземпляра делегата *не приводит к его выполнению.*

Чтобы максимально прояснить, что одно лишь создание экземпляра делегата не приводит к чтению переменной и сохранению ее значения где-либо, значение captured изменяется на "directly before x is invoked". Далее экземпляра делегата x вызывается в первый раз. Он читает значение переменной captured и выводит его на консоль — первая строка вывода. Код внутри делегата устанавливает значение captured в "changed by x" и завершается. После возвращения управления из экземпляра делегата обычный метод продолжает выполнение традиционным путем. Он выводит текущее значение captured, давая вторую строку вывода.

Затем обычный метод снова изменяет значение переменной captured (в этом случае на "before second invocation") и вызывает экземпляр делегата x во второй раз. Текущее значение captured выводится, порождая последнюю строку вывода. Экземпляр делегата изменяет значение captured на "changed by x" и возвращается, после чего обычный метод завершает свою работу.

Описание функционирования такого короткого кода потребовало довольно большого количества деталей, но только одна идея является ключевой: *захваченная переменная — это та же самая переменная, которая используется в остальном коде метода.* Для одних людей этот факт трудно уловить, для других он воспринимается вполне естественно. Не переживайте, если поначалу испытываете трудности в понимании — с течением времени все станет проще.

Даже если вы понимаете абсолютно весь материал, приведенный до сих пор, может возникнуть вопрос: для чего все это нужно делать? Наступило время рассмотреть действительно полезный пример.

5.5.3. Смысл захваченных переменных

Попросту говоря, захваченные переменные устраняют необходимость в написании дополнительных классов, предназначенных только для хранения информации, с которой должен взаимодействовать делегат, кроме данных, передаваемых через параметры. До появления типа делегата `ParameterizedThreadStart()` для запуска нового потока (не из пула потоков) и предоставления ему нужной информации — например, URL извлекаемой страницы — требовалось создавать дополнительный тип для удержания URL и помещать в этот тип действие экземпляра делегата `ThreadStart()`. Но даже с учетом `ParameterizedThreadStart()` метод должен принимать параметр типа `object` и приводить его к типу, с которым необходимо работать. Это был довольно неуклюжий способ достижения цели, которая в принципе является простой.

В качестве еще одного примера предположим, что имеется список людей и нужно написать метод, который бы возвращал второй список, содержащий людей моложе заданного возраста. В типе `List<T>` есть метод по имени `FindAll()`, который возвращает другой список с элементами, соответствующими указанному предикату. До введения анонимных методов и захваченных переменных особого смысла в существовании метода `List<T>.FindAll()` не было из-за всех тех барьеров, которые требовалось преодолевать, чтобы создать правильный тип делегата. В то время итерацию и копирование было проще делать вручную. Однако версия C# 2 позволяет реализовать это очень легко:

```
List<Person> FindAllYoungerThan(List<Person> people, int limit)
{
   return people.FindAll(delegate (Person person)
      { return person.Age < limit; }
   );
}
```

Здесь внутри экземпляра делегата захватывается параметр `limit` — если бы имелись только анонимные методы, но не захваченные переменные, пришлось бы выполнять проверку с жестко закодированным пределом, а не с тем, который передается методу в виде параметра. Надеюсь, вы согласитесь с тем, что такой подход аккуратнее: он точно выражает то, *что* необходимо делать, с гораздо меньшей неразберихой, возникающей при точном выражении того, *каким образом* это должно случиться, что можно было наблюдать в версии кода на C# 1. (Надо сказать, что в версии C# 3 все даже еще более аккуратно...[7]) Ситуация, когда требуется *записывать* в захваченную переменную, возникает относительно редко, но такое использование вполне допустимо.

Вы все еще здесь? Тогда продолжим. До сих пор экземпляр делегата применялся только внутри метода, в котором он был создан. При этом не возникали многие вопросы относительно времени жизни захваченных переменных — но что произойдет, если экземпляр делегата выйдет за узкие рамки своего метода? Как он будет существовать после того, как создавший его метод завершится?

5.5.4. Продленное время жизни захваченных переменных

Простейший способ обсудить эту тему заключается в оглашении правила, предоставлении примера и затем осмыслении того, что случится, если такого правила не будет. Вот это правило.

---

[7] На тот случай, если вы интересуетесь прямо сейчас: `return people.Where(person => person.Age < limit);`

Захваченная переменная существует до тех пор, пока на нее ссылается, по крайней мере, один экземпляр делегата.

Не переживайте, если это правило выглядит не особенно понятным — оно станет яснее при рассмотрении примера. В листинге 5.12 показан метод, который *возвращает* экземпляр делегата. Этот экземпляр делегата создан с использованием анонимного метода, который захватывает внешнюю переменную. Так что же произойдет, когда этот делегат будет вызван после возврата управления из метода?

Листинг 5.12. Демонстрация продленного времени жизни захваченной переменной

```
static MethodInvoker CreateDelegateInstance()
{
  int counter = 5;
  MethodInvoker ret = delegate
  {
    Console.WriteLine(counter);
    counter++;
  };
  ret();
  return ret;
}
...
MethodInvoker x = CreateDelegateInstance();
x();
x();
```

Вывод кода из листинга 5.12 содержит числа 5, 6 и 7 в отдельных строках. Первая строка вывода поступает из вызова экземпляра делегата внутри метода `CreateDelegateInstance()`, поэтому имеет смысл утверждать, что значение `counter` доступно в данной точке. Но что можно сказать о моменте, когда уже произошел возврат из метода? Обычно предполагается, что переменная `counter` находится в стеке, а когда стековый фрейм для метода `CreateDelegateInstance()` уничтожен, можно считать, что переменная `counter` по существу должна была бы исчезнуть..., но кажется, что последующие вызовы возвращенного экземпляра делегата продолжают использовать ее.

Секрет кроется в некорректности предположения о том, что переменная `counter` находится в стеке. Это не так. Для хранения переменной компилятор в действительности создает дополнительный класс. Метод `CreateDelegateInstance()` имеет ссылку на экземпляр этого класса, так что он может использовать `counter`, а делегат имеет ссылку на тот же самый экземпляр, который в обычных условиях находится в куче. Этот экземпляр не может быть обработан сборщиком мусора до тех пор, пока к сборке мусора не будет готов сам делегат.

Определенные аспекты анонимных методов сильно зависят от компилятора (разные компиляторы могут обеспечивать одну и ту же семантику по-разному), но трудно увидеть, каким образом указанное поведение могло быть получено *без* применения дополнительного класса, предназначенного для хранения захваченной переменной. Обратите внимание, что если захватить только `this`, никакие дополнительные типы не требуются — компилятор просто создает метод экземпляра, выступающий в качестве действия делегата. Как упоминалось ранее, не следует слишком сильно беспокоиться о деталях, связанных со стеком и кучей, но полезно знать, какие функции компилятор способен выполнять, не путаясь в вопросе, возможно ли в принципе то или иное поведение.

Итак, локальные переменные могут существовать даже после возврата управления из метода. Вам может быть интересно, какой сюрприз я преподнесу следующим. Как насчет захватывания несколькими делегатами разных экземпляров одной и той же переменной? Звучит нелепо, но именно этого теперь придется ожидать.

5.5.5. Создание экземпляров локальных переменных

В удачные дни захваченные переменные действуют сразу именно так, как я от них ожидаю. В неудачные дни я продолжаю удивляться своей неосмотрительности. Чаще всего проблемы возникают из-за моей забывчивости относительно того, сколько "экземпляров" локальных переменных в действительности было создано. Говорят, что *экземпляр* локальной переменной *создается* каждый раз, когда поток выполнения входит в область действия, в которой эта переменная объявлена.

Ниже приведен простой пример, сравнивающий два похожих фрагмента кода.

```
int single;
for (int i = 0; i < 10; i++)
{
  single = 5;
  Console.WriteLine(single + i);
}
```

```
for (int i = 0; i < 10; i++)
{
  int multiple = 5;
  Console.WriteLine(multiple + i);
}
```

В старые добрые времена можно было вполне обоснованно утверждать, что фрагменты кода, подобные показанным выше, семантически идентичны. Действительно, обычно они компилировались в один и тот же код IL — и это по-прежнему будет так, если не задействованы анонимные методы. Все пространство для локальных переменных выделяется в стеке в начале метода, поэтому никаких накладных расходов при повторном объявлении переменной для каждой итерации цикла не возникает[8]. В нашей новой терминологии экземпляр переменной `single` будет создан только один раз, но экземпляры переменной `multiple` будут создаваться 10 раз — как если бы существовало 10 локальных переменных, каждая по имени `multiple`, которые были созданы друг за другом.

Уверен, что вы заметили, к чему я веду — при захвате переменной захватывается соответствующий "экземпляр" этой переменной. Когда захватывается переменная `multiple` внутри цикла, то переменная, захваченная на первой итерации, будет отличаться от переменной, захваченной на второй итерации, и т.д. В листинге 5.13 представлена демонстрация этого эффекта.

Листинг 5.13. Захватывание нескольких экземпляров переменной с помощью множества делегатов

```
List<MethodInvoker> list = new List<MethodInvoker>();

for (int index = 0; index < 5; index++)
{
  int counter = index * 10;          <-- 1 Создание экземпляра counter
  list.Add(delegate
  {
    Console.WriteLine(counter);      <-- 2 Вывод на консоль и инкрементирование захваченной переменной
    counter++;
  });
}
```

[8] На мой взгляд, повторное объявление переменной, если только не требуется поддержка ее значения между итерациями, также обеспечивает более ясный код.

```
foreach (MethodInvoker t in list)
{
   t();          <-- ❹ Выполнение всех пяти экземпляров делегата
}
list[0]();       <-- ❶ Трехкратное выполнение первого экземпляра делегата
list[0]();
list[0]();

list[1]();       <-- ❷ Однократное выполнение второго экземпляра делегата
```

В листинге 5.13 создаются пять разных экземпляров делегатов ❸ — по одному на каждой итерации цикла. Обращение к делегату приводит к выводу на консоль значения `counter` и затем к его инкрементированию. Поскольку переменная `counter` объявлена *внутри* цикла, ее экземпляр создается для каждой итерации ❶, и каждый делегат захватывает свою переменную. Во время вызова делегатов поочередно ❹ вы увидите, что переменной `counter` изначально присваиваются разные значения: 0, 10, 20, 30, 40. И чтобы вбить последний гвоздь: когда первый экземпляр делегата выполняется три дополнительных раза ❶, он продолжает с того места, где был оставлен экземпляр переменной `counter`: 1, 2, 3. И, наконец, выполняется второй экземпляр делегата ❷, который продолжает с места, где был оставлен *его* экземпляр переменной `counter`: 11.

Как видите, каждый экземпляр делегата захватил собственную переменную. Прежде чем завершить этот пример, я должен обратить внимание на то, что произошло бы, если бы вместо `counter` осуществлялось захватывание `index` — переменной, которая объявлена самим циклом `for`. В таком случае все делегаты разделяли бы одну и ту же переменную. Вывод выглядел бы как последовательность чисел от 5 до 13; первым было бы число 5, т.к. при завершении цикла последним присвоенным переменной `index` значением являлось 5, и та же самая переменная инкрементировалась бы независимо от того, какой делегат был задействован. Аналогичное поведение демонстрирует и цикл `foreach` (в версиях C# 2 — C# 4): экземпляр переменной, объявленной в начальной части цикла, создается только один раз. Здесь очень легко допустить ошибку! Если необходимо захватить значение переменной цикла для конкретной итерации цикла, объявите внутри цикла другую переменную, скопируйте в нее значение переменной цикла и захватите новую переменную — именно это делалось в листинге 5.13 с переменной `counter`.

В C# 5 это изменилось...

Хотя описанное выше поведение в цикле `for` вполне обоснованно — в конце концов, переменная выглядит как объявленная только раз — в случае цикла `foreach` оно неожиданно. На самом деле, *почти всегда* неправильно захватывать итерационную переменную `foreach` в анонимном методе, который должен существовать за рамками непосредственной итерации. (Если экземпляр делегата используется только внутри данной итерации, то все в порядке.) Это создавало проблемы у настолько большого числа разработчиков, что в команде проектировщиков языка C# в версии C# 5 решили изменить семантику цикла `foreach`, чтобы он действовал более естественным образом — как если бы для каждой итерации была предусмотрена собственная отдельная переменная. Дополнительные сведения ищите в разделе 16.1.

В последнем примере мы рассмотрим кое-что действительно неприятное — разделение одних захваченных переменных, но не других.

5.5.6. Смесь разделяемых и отдельных переменных

Прежде чем показать следующий пример, я хочу отметить, что это *не* тот код, который я бы рекомендовал писать. Вообще говоря, данный пример предназначен для демонстрации того, что если пытаться использовать захваченные переменные излишне сложными путями, то код очень быстро станет запутанным. В листинге 5.14 создаются два экземпляра делегата, каждый из которых захватывает "те же самые" две переменных. Но сюжет станет более захватывающим, когда мы начнем выяснять, что на самом деле захватывается.

Листинг 5.14. Захватывание переменных в разных областях действия. Внимание: рискованный код!

```
MethodInvoker[] delegates = new MethodInvoker[2];
int outside = 0;                    <-- ❶ Создание экземпляра переменной один раз
for (int i = 0; i < 2; i++)
{
   int inside = 0;                  <-- ❷ Создание экземпляров переменной много раз
   delegates[i] = delegate          <-- ❸ Захватывание переменных с помощью анонимного метода
   {
      Console.WriteLine ("({0},{1})", outside, inside);
      outside++;
      inside++;
   };
}
MethodInvoker first = delegates[0];
MethodInvoker second = delegates[1];
first();
first();
first();
second();
second();
```

Сколько времени вам понадобится на обдумывание вывода кода из листинга 5.14 (даже с учетом аннотаций)? Честно говоря, у меня это заняло некоторое время — больше, чем я предпочитаю тратить на понимание кода. Тем не менее, в качестве упражнения давайте посмотрим, что тут происходит.

Прежде всего, взгляните на переменную `outside` ❶. Управление заходит в область действия, в которой она объявлена, только однажды — по существу есть лишь один ее экземпляр. С переменной `inside` ❷ дела обстоят по-другому — на каждой итерации цикла создается новый ее экземпляр. Это означает, что при создании экземпляра делегата ❸ переменная `outside` разделяется между двумя экземплярами делегата, но каждый из них обладает собственной переменной `inside`.

После завершения цикла трижды вызывается первый созданный экземпляр делегата. Поскольку каждый раз он инкрементирует значения обеих захваченных переменных, и обе они начинаются с 0, вы видите вывод (0,0), затем (1,1) и, наконец, (2,2). Разница между этими двумя переменными в терминах области действия становится заметной при выполнении второго экземпляра делегата. Он имеет другую переменную `inside`, которая по-прежнему содержит значение 0, однако переменная `outside` является той,

что уже была инкрементирована три раза. Вывод, получаемый в результате двукратного вызова второго экземпляра делегата, выглядит следующим образом: (3,0) и (4,1).

Просто ради интереса подумаем о том, как это реализовано — по крайней мере, в компиляторе C# 2 от Microsoft. На самом деле генерируется один дополнительный класс для хранения переменной `outside` и еще один — для удержания переменной `inside` *и ссылки на первый дополнительный класс*. В сущности, каждая область действия, которая содержит захваченную переменную, получает собственный тип, со ссылкой на следующую область действия, содержащую захваченную переменную. В этом примере существуют два экземпляра типа, хранящего `inside`, и они оба ссылаются на один и тот же экземпляр типа, который хранит `outside`. Другие реализации могут варьироваться, но выше был описан наиболее очевидный способ. На рис. 5.1 показаны значения после выполнения кода из листинга 5.14. (Имена, указанные на рисунке, не точно совпадают с теми, которые сгенерирует компилятор, но они довольно близки к ним. Обратите внимание, что в реальности экземпляры делегата будут также иметь и другие члены, однако здесь нас интересует только `target`.)

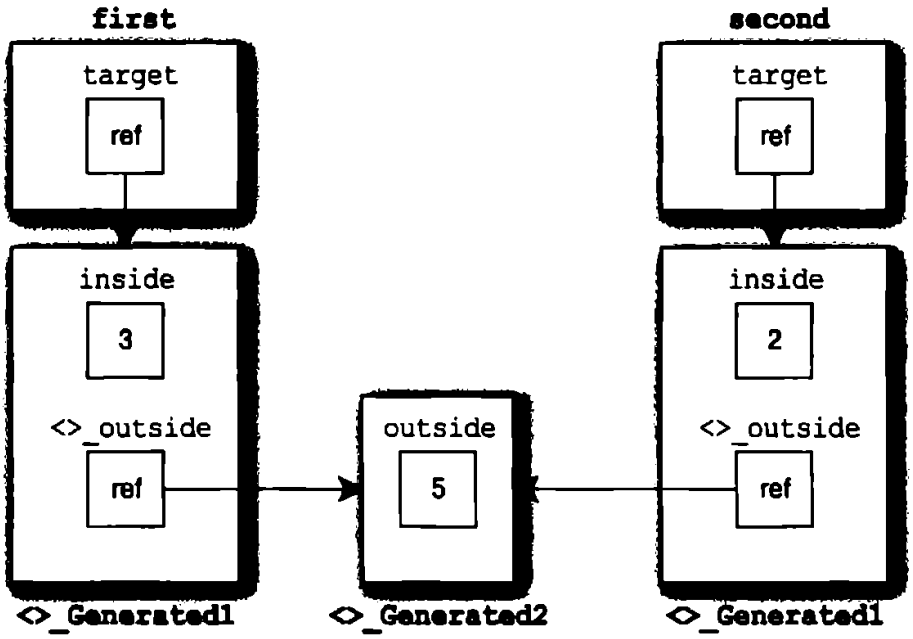

Рис. 5.1. Представление нескольких областей действия захваченных переменных в памяти

Даже после полного понимания этого кода он по-прежнему продолжает быть хорошим шаблоном для проведения экспериментов с другими элементами захваченных переменных. Как упоминалось ранее, определенные элементы захватывания переменных являются специфичными для реализации, и для выяснения того, что они гарантируют, часто полезно обращаться к спецификации. Но не менее важно *экспериментировать* с кодом и наблюдать за тем, что происходит.

Возможно, и есть ситуации, при которых код вроде приведенного в листинге 5.14 может оказаться самым простым и ясным способом выражения желаемого поведения, но мне трудно в это поверить, и даже если так, то код определенно должен быть снабжен комментариями, объясняющими, что будет происходить. Итак, когда уместно применять захваченные переменные и на что при этом обращать внимание?

5.5.1. Руководящие принципы и резюме по захваченным переменным

Надеюсь, что этот раздел склонит вас к соблюдению *крайней* осторожности в отношении захваченных переменных. Они имеют глубокий логический смысл (и практически любое изменение в попытке упрощения, скорее всего, сделало бы их либо менее полезными, либо менее логичными), но также легко приводят к получению чрезвычайно сложного кода.

Тем не менее, не позволяйте, чтобы это препятствовало их разумному использованию. Захваченные переменные могут предотвратить написание массы утомительного кода, а в случае их применения должным образом они могут обеспечить наиболее читабельный способ выполнения работы. Но что считать *разумным*?

Ниже перечислены некоторые соображения относительно использования захваченных переменных.

- Если код, в котором захваченные переменные не применяются, так же прост, как и код, в котором они используются, не применяйте их.
- Перед захватыванием переменной, объявленной оператором `for` или `foreach`, удостоверьтесь, планируется ли существование делегата за пределами итерации цикла, и нужны ли последующие значения этой переменной. Если нет, создайте другую переменную внутри цикла и скопируйте в нее *необходимое* значение. (В C# 5 переживать по поводу операторов `foreach` не придется, но по-прежнему нужно заботиться об операторах `for`.)
- Если вы создаете множество экземпляров делегата (в цикле или явно), которые захватывают переменные, подумайте о том, нужно ли, чтобы они захватывали одну и ту же переменную.
- Если вы захватываете переменную, которая в действительности не изменяется (либо в анонимном методе, либо в теле включающего его метода), то не должны особенно переживать.
- Если создаваемые экземпляры делегатов никогда не покидают метод — другими словами, они никогда не сохраняются где-либо, не возвращаются и не используются для запуска потоков, — то все значительно упрощается.
- Учтите продленное время жизни любых захваченных переменных в контексте сборки мусора. Обычно это не проблема, но в случае захвата объекта со значительными расходами памяти, такой вопрос может оказаться важным.

Первое соображение в этом списке является золотым правилом. Простота — это всегда хорошо. Поэтому каждый раз, когда применение захваченной переменной делает код проще с учетом дополнительной свойственной ему сложности, связанной с обеспечением понимания предназначения этой захваченной переменной со стороны сопровождающих код разработчиков, не отказывайтесь от него. Эту дополнительную сложность необходимо принимать во внимание, а не только стремиться сводить к минимуму количество строк.

В этом разделе было раскрыто много основ, и я осознаю, что могут возникнуть трудности с их освоением. Ниже приведен список наиболее важных моментов, который поможет впоследствии освежить память, не перечитывая всю главу.

- Захватывается сама *переменная*, а не ее значение в точке создания экземпляра делегата.

- Время жизни захваченной переменной продлевается, как минимум, до времени существования захватившего ее делегата.
- Несколько делегатов могут захватывать ту же самую переменную,...
- ... но внутри циклов объявление одной и той же переменной, в сущности, может ссылаться на разные "экземпляры" переменной.
- Объявления в цикле `for` создают переменные, которые существуют на протяжении цикла — их экземпляры не создаются при каждой итерации. То же самое справедливо для оператора `foreach` в версиях языка, предшествующих C# 5.
- При необходимости для хранения захваченных переменных создаются дополнительные типы.
- Соблюдайте осторожность! Простой код почти всегда лучше слишком заумного кода.

Когда мы будем рассматривать версию C# 3 и доступные в ней лямбда-выражения, вы увидите еще немало случаев захватывания переменных, но пока что можете вздохнуть с облегчением, т.к. мы завершили краткое изложение новых возможностей делегатов в C# 2.

5.6. Резюме

В версии C# 2 радикально изменились способы создания делегатов, что обеспечило для инфраструктуры более функциональный стиль программирования. В .NET 2.0 доступно больше методов, принимающих делегаты в качестве параметров, чем в .NET 1.0/1.1, и такая тенденция продолжается в .NET 3.5. Лучшим примером может служить тип `List<T>`, который также является качественным испытательным стендом для проверки уровня мастерства при работе с анонимными методами и захваченными переменными. Программирование в таком стиле требует несколько иного образа мыслей. Вы должны быть в состоянии сделать шаг назад и подумать, в чем заключается конечная цель, и как ее лучше выразить — в традиционной манере C# или с помощью функционального подхода.

Все изменения в обработке делегатов полезны, однако они привносят сложность в язык, особенно когда дело доходит до захваченных переменных. Замыкания всегда запутанны в смысле точного определения, каким образом разделяется доступная среда, и язык C# в этом отношении ничем не отличается. Тем не менее, причина, по которой концепция продержалась так долго, связана с тем, что она может сделать код проще для понимания и более непосредственным. Соблюдать баланс между сложностью и простотой всегда нелегко, и осторожность никогда не будет лишней. Но со временем вы лучше станете понимать поведение захваченных переменных и работу с ними. Язык LINQ еще более способствует их использованию, и в современном идиоматическом коде C# замыкания применяются довольно часто.

Анонимные методы — не единственное изменение в версии C# 2, которое предусматривает скрытое создание компилятором дополнительных типов и выполнение обходных действий в отношении переменных, которые кажутся локальными. Намного больше информации будет предоставлено в следующей главе, где вы узнаете, что компилятор может создавать целый конечный автомат, упрощая тем самым реализацию итераторов.

Глава 6

Простой способ реализации итераторов

В этой главе...

- Реализация итераторов в C# 1
- Итераторные блоки в C# 2
- Пример использования итератора
- Итераторы как сопрограммы

Шаблон итератора представляет собой пример поведенческого шаблона — шаблона, который упрощает взаимодействие между объектами. Он является одним из наиболее легких для понимания шаблонов, и его предельно просто использовать. В сущности, он позволяет получать доступ ко всем элементам в последовательности, не заботясь о том, какой вид имеет эта последовательность — массив, список, связный список или ничего из перечисленного. Это может быть эффективным при построении *конвейера данных*, в котором элемент данных вводится в конвейер и по пути к приемнику подвергается воздействию нескольких трансформаций или фильтров. Как будет показано в части III книги, на самом деле это один из основных шаблонов LINQ.

Внутри .NET шаблон итератора инкапсулирован в интерфейсы `IEnumerator` и `IEnumerable`, а также в их обобщенные эквиваленты. (Имена интерфейсов выбраны неудачно — шаблон обычно называют *итерацией*, чтобы избежать путаницы со смыслом слова *перечисление*. В настоящей главе я повсеместно применяю понятия *итератор* и *итерируемый*.) Если тип реализует `IEnumerable`, это означает, что по нему можно проходить с помощью итерации; вызов метода `GetEnumerator()` возвратит реализацию интерфейса `IEnumerator`, которая представляет собой сам итератор. Об итераторе удобно думать как о курсоре базы данных, т.е. о позиции внутри последовательности.

Итератор может перемещаться внутри последовательности только вперед, а с одной последовательностью одновременно могут работать несколько итераторов.

Как язык, C# 1 обладает встроенной поддержкой для использования итераторов через оператор `foreach`. Он делает итерацию по коллекциям простой (проще, чем применение прямолинейного цикла `for`) и довольно выразительной. Оператор `foreach` компилируется в вызовы методов `GetEnumerator()` и `MoveNext()` и обращение к свойству `Current` с поддержкой последующего освобождения итератора, если им был реализован интерфейс `IDisposable`. Это небольшая, но удобная порция синтаксического сахара.

Однако *реализация* итератора в C# 1 является относительно трудной задачей. В версии C# 2 решение данной задачи намного упростилось. Иногда это приводит к тому, что шаблон итератора реализуется в ситуациях, когда ранее он не сохранял усилия, а наоборот — требовал больше работы.

В этой главе мы посмотрим, что необходимо для реализации итератора, а также на поддержку, предлагаемую C# 2. После детального ознакомления с синтаксисом мы исследуем несколько примеров из реального мира, включая увлекательное (если только не из ряда вон выходящее) использование синтаксиса итераций в библиотеке параллельного выполнения от Microsoft. Рассмотрение примеров откладывается до конца описания, т.к. материалов для изучения *не очень* много, а примеры станут намного яснее, когда вы будете понимать, что делает код. Но если вы действительно хотите сначала просмотреть примеры, то ищите их в разделах 6.3 и 6.4.

Как и в других главах, давайте начнем с того, что взглянем, каким образом обстояли дела до появления версии C# 2. Мы реализуем итератор трудным путем.

6.1. C# 1: сложность написанных вручную итераторов

Вы уже видели один пример реализации итератора в разделе 3.4.3, когда выяснялось, что происходит при итерации по обобщенной коллекции. В некотором смысле код оказался сложнее, чем должна была быть действительная реализация итератора в C# 1, поскольку в нем дополнительно реализовывались обобщенные интерфейсы — но в других отношениях код также был и проще, потому что в нем не выполнялась итерация по чему-либо полезному.

Чтобы поместить средства C# 2 в контекст, мы сначала реализуем итератор, сохраняющий приемлемый предел простоты, который пока еще позволяет ему предоставлять действительную функциональность. Предположим, что необходим новый тип коллекции, основанный на кольцевом буфере. В этом примере мы реализуем интерфейс `IEnumerable`, чтобы пользователи нового класса могли легко проходить по всем значениям в коллекции. Мы здесь проигнорируем внутреннее содержимое коллекции и сосредоточимся только на самой итерации. Коллекция будет хранить свои значения в массиве (`object[]` — никаких обобщений) и обладать интересной особенностью, которая позволит устанавливать логическую начальную точку. Так, если бы массив содержал пять элементов, начальную точку можно было бы установить в 2 и ожидать возвращения элементов 2, 3, 4, 0 и затем 1. В главе не приводится полный код кольцевого буфера, но вы можете найти его в загружаемом коде.

Чтобы сделать класс проще для демонстрации, в конструкторе можно будет указывать значения и начальную точку, поэтому для итерации по коллекции появится возможность писать код вроде показанного в листинге 6.1.

Листинг 6.1. Код, в котором используется новый тип коллекции (пока еще не реализованный)

```
object[] values = {"a", "b", "c", "d", "e"};
IterationSample collection = new IterationSample(values, 3);
foreach (object x in collection)
{
  Console.WriteLine (x);
}
```

Запуск кода из листинга 6.1 должен (в конечном счете) сгенерировать вывод d, e, a, b и c, т.к. в качестве начальной точки было указано значение 3. Теперь, когда вы знаете, чего нужно достигнуть, давайте рассмотрим каркас класса, который показан в листинге 6.2.

Листинг 6.2. Каркас нового типа коллекции без реализации итератора

```
using System;
using System.Collections;

public class IterationSample : IEnumerable
{
  object[] values;
  int startingPoint;

  public IterationSample(object[] values, int startingPoint)
  {
    this.values = values;
    this.startingPoint = startingPoint;
  }

  public IEnumerator GetEnumerator()
  {
    throw new NotImplementedException();
  }
}
```

Хотя метод `GetEnumerator()` пока не реализован, остальной код готов. А как приступить к написанию кода `GetEnumerator()`? Прежде всего, следует понять, что нам необходимо где-то хранить определенное *состояние*. Важный аспект шаблона итератора заключается в том, что все данные не возвращаются за один шаг — клиент запрашивает по одному элементу за раз. Это означает, что нужно отслеживать, насколько далеко зашло продвижение по массиву. Поддерживающая состояние природа итераторов будет важна, когда мы начнем рассматривать, что делает компилятор C# 2, поэтому внимательно следите за состоянием, требуемым в данном примере.

Где должно находиться это состояние? Предположим, вы пытаетесь поместить его в класс `IterationSample`, обеспечив реализацию этим классом интерфейсов `IEnumerator` и `IEnumerable`. На первый взгляд, это выглядит удачным планом — в конце концов, *данные* располагаются в правильном месте, включая начальную точку. Метод `GetEnumerator()` мог бы просто возвращать `this`. Но с таким подходом связана крупная проблема: если `GetEnumerator()` вызывается несколько раз, то должны возвращаться независимые итераторы. Например, для получения всех возможных пар значений требуется возможность использования двух операторов `foreach`, один

внутри другого. Эти два итератора должны быть независимыми, что означает необходимость создания нового объекта при каждом вызове метода `GetEnumerator()`. Такую функциональность все еще *можно было бы* реализовать непосредственно внутри `IterationSample`, но тогда получился бы класс, не имеющий единственной четкой ответственности, что привело бы к путанице.

Взамен давайте создадим еще один класс с целью реализации самого итератора. Можно воспользоваться тем фактом, что в языке C# вложенный тип имеет доступ к закрытым членам включающего его типа, т.е. допускается сохранить ссылку на родительский экземпляр `IterationSample` наряду с состоянием, отражающим количество выполненных до сих пор итераций. Это показано в листинге 6.3.

Листинг 6.3. Вложенный класс, реализующий итератор по коллекции

```
class IterationSampleIterator : IEnumerator
{
  IterationSample parent;                    ←❶ Коллекция, по которой осуществляется итерация
  int position;                              ←❷ Количество выполненных итераций

  internal IterationSampleIterator(IterationSample parent)
  {
    this.parent = parent;
    position = -1;                           ←❸ Начало перед первым элементом
  }

  public bool MoveNext()
  {
    if (position != parent.values.Length)    ←❹ Увеличение позиции,
                                                если это все еще возможно
    {
      position++;
    }
    return position < parent.values.Length;
  }

  public object Current
  {
    get
    {
      if (position == -1 ||                  ←❺ Предотвращение доступа перед первым
                                                или после последнего элемента
          position == parent.values.Length)
      {
        throw new InvalidOperationException();
      }
      int index = position + parent.startingPoint;
      index = index % parent.values.Length;  ←❻ Реализация
      return parent.values[index];              циклического
    }                                           возврата
  }

  public void Reset()
  {
    position = -1;                           ←❼ Переход к позиции перед первым элементом
  }
}
```

Что-то слишком много кода понадобилось для решения такой простой задачи! Первым делом объявляются исходная коллекция значений, по которой производится итерация ❶, и текущая позиция в простом массиве с индексацией, начинающейся с нуля ❷. Чтобы возвратить элемент, индекс смещается по начальной точке ❸. В соответствие с интерфейсом мы предполагаем, что итератор должен логически стартовать перед первым элементом ❹, поэтому клиент должен будет вызвать метод `MoveNext()` перед обращением к свойству `Current` в первый раз. Условное инкрементирование в операторе ❺ делает проверку в строке ❹ простой и корректной, даже если метод `MoveNext()` вызывается снова после того, как он уже сообщил, что доступных данных больше нет. Для сброса итератора логическая позиция устанавливается опять перед первым элементом ❹.

Большая часть применяемой логики довольно прямолинейна, хотя при этом есть широкий простор для допущения ошибок диапазона; моя первая реализация не прошла модульные тесты именно по этой причине. Хорошие новости в том, что код работает, и для завершения примера в классе `IterationSample` необходимо лишь реализовать интерфейс `IEnumerable`:

```
public IEnumerator GetEnumerator()
{
   return new IterationSampleIterator(this);
}
```

Я не буду воспроизводить здесь полный код, но он доступен в загружаемых примерах, в том числе и код, приведенный в листинге 6.1, который теперь генерирует ожидаемый вывод.

Полезно иметь в виду, что это относительно простой пример — в нем нет большого объема состояния, которое должно отслеживаться, и отсутствуют попытки проверки, не изменилась ли коллекция между итерациями. При таких высоких накладных расходах, сопряженных с реализацией простого итератора, не должно вызывать удивления то, насколько редко встречалась реализация этого шаблона в C# 1. Разработчики обычно благополучно использовали оператор `foreach` для коллекций, предоставляемых инфраструктурой, но более прямой (и специфичный для коллекций) доступ, когда приходилось работать с собственными коллекциями.

Реализация итератора в C# 1 потребовала около 40 строк кода. Давайте посмотрим, может ли C# 2 улучшить ситуацию.

6.2. C# 2: простые итераторы с операторами `yield`

Я всегда относился к тому типу людей, которые предпочитают дождаться полуночи сочельника, чтобы можно было открыть подарок по приходу Рождества. В то же время я считаю едва ли невозможным ждать хоть сколько-нибудь ощутимое время до демонстрации того, насколько лаконичным является решение в C# 2.

6.2.1. Появление итераторных блоков и оператора `yield return`

Эта глава не была бы написана, если бы в версии C# 2 не появилось мощное средство, сокращающее объем кода, который должен быть написан для реализации итераторов. В ряде других тем объем кода уменьшался лишь незначительно либо изменения просто делали что-то более элегантным. Однако в этом случае требуемый объем кода сокращается *существенно*. В листинге 6.4 показана полная реализация метода `GetEnumerator()` в C# 2.

Листинг 6.4. Итерация по коллекции в C# 2 с применением `yield return`

```
public IEnumerator GetEnumerator()
{
   for (int index = 0; index < values.Length; index++)
   {
      yield return values[(index + startingPoint) % values.Length];
   }
}
```

Итого реализация содержит четыре строки кода, две из которых являются просто фигурными скобками.

Точнее говоря, это заменяет целый класс `IterationSampleIterator` — полностью. По крайней мере, в исходном коде... Позже вы увидите, что именно компилятор сделал "за кулисами", а также индивидуальные особенности предоставленной им реализации, но пока давайте уделять внимание исходному коду.

Метод выглядит совершенно обычным, если не считать использования оператора `yield return`. Данный оператор сообщает компилятору C# о том, что это не обычный метод, а метод, реализующий *итераторный блок*. Метод объявлен как возвращающий тип `IEnumerator`. Итераторные блоки можно применять только для реализации методов [11], которые имеют возвращаемый тип `IEnumerable`, `IEnumerator` или один из их обобщенных эквивалентов. *Типом выдачи* итераторного блока будет `object`, если объявленным возвращаемым типом является необобщенный интерфейс, или аргумент типа, если в качестве возвращаемого типа указан обобщенный интерфейс. Например, метод, объявленный как возвращающий `IEnumerable<string>`, будет иметь тип выдачи `string`.

Внутри итераторных блоков не разрешены обычные операторы `return` — допускаются только `yield return`. Все операторы `yield return` в блоке должны пытаться возвращать значение, совместимое с типом выдачи блока. В предыдущем примере нельзя было применять оператор `yield return 1;`, т.к. метод объявлен как возвращающий `IEnumerable<string>`.

Ограничения оператора `yield return`

С операторами `yield` связано еще несколько ограничений. Оператор `yield return` не разрешено использовать внутри блока `try` при наличии любых блоков `catch`, и не допускается применять оператор `yield return` *или* `yield break` (который мы вскоре рассмотрим) в блоке `finally`. Это не означает невозможности использования блоков `try/catch` или `try/finally` внутри итераторов, но просто ограничивает то, что можно делать внутри них. О причинах существования таких ограничений, а также о проектных решениях, положенных в основу итераторов, можно почитать в статьях Эрика Липперта по адресу `http://blogs.msdn.com/b/ericlippert/archive/tags/iterators/`.

Когда речь идет об итераторных блоках, важно понимать, что хотя пишется метод, который выглядит выполняющимся последовательно, в действительности у компилятора запрашивается создание *конечного автомата*. Это необходимо по той же самой при-

[1] Или свойств, как вы увидите позже. Тем не менее, итераторный блок нельзя использовать в анонимном методе.

чине, по которой приходилось прикладывать настолько много усилий при реализации итератора в C# 1 — поскольку вызывающему коду требуется только один элемент за раз, нужно отслеживать то, что было сделано во время последнего возврата значения.

Когда компилятор встречает итераторный блок, он создает вложенный тип, предназначенный для конечного автомата. Этот тип запоминает точное местонахождение внутри блока и значения локальных переменных (в том числе параметров). Сгенерированный класс *отчасти* похож на написанную ранее длинную реализацию тем, что он сохраняет все необходимое состояние в виде переменных экземпляра. Давайте подумаем о том, что этот конечный автомат должен делать в плане реализации итератора.

- Он должен иметь некоторое начальное состояние.
- Всякий раз, когда вызывается метод `MoveNext()`, он должен выполнять код из метода `GetEnumerator()` до тех пор, пока не будет готово к предоставлению следующее значение (другими словами, пока не будет достигнут оператор `yield return`).
- Когда используется свойство `Current`, он должен возвратить последнее выданное значение.
- Он должен знать, когда выдача значений завершена, чтобы метод `MoveNext()` мог возвратить `false`.

Второй пункт в этом списке является сложным, т.к. конечный автомат всегда нуждается в перезапуске кода с точки, которая была достигнута ранее. Отслеживание локальных переменных (поскольку они присутствуют в методе) реализуется не очень сложно — они представлены переменными экземпляра в конечном автомате. Аспект перезапуска еще сложнее, но важно осознавать, что в случае, если вы не занимаетесь написанием компилятора C#, то заботиться о том, как это реализовано, не придется: в соответствие с принципом "черного ящика" все просто работает должным образом. Внутрь итераторного блока можно помещать совершенно нормальный код, и компилятор отвечает за то, что поток выполнения будет точно таким же, как в любом другом методе. Разница в том, что оператор `yield return` осуществляет только временный выход из метода — на самом деле о нем можно думать как о средстве реализации паузы.

Теперь мы приступим к детальным исследованиям потока выполнения в более наглядном виде.

6.2.2. Визуализация рабочего потока итератора

Работу итератора удобно рассматривать с помощью диаграммы последовательности действий. Вместо рисования такой диаграммы вручную в листинге 6.5 приведен код, который ее выводит на консоль. Итератор предоставляет последовательность чисел (0, 1, 2, –1) и затем завершается. Однако более интересны не сами числа, а *поток* выполнения кода.

Листинг 6.5. Отображение последовательности обращений между итератором и вызывающим кодом

```
static readonly string Padding = new string(' ', 30);

static IEnumerable<int> CreateEnumerable()
{
  Console.WriteLine("{0}Start of CreateEnumerable()", Padding);
                    // Начало CreateEnumerable()
```

```
    for (int i=0; i < 3; i++)
    {
      Console.WriteLine("{0}About to yield {1}", Padding, i);
                        // Выдача значения
      yield return i;
      Console.WriteLine("{0}After yield", Padding);
                        // После выдачи значения
    }
    Console.WriteLine("{0}Yielding final value", Padding);
                        // Выдача финального значения
    yield return -1;
    Console.WriteLine("{0}End of CreateEnumerable()", Padding);
                        // Конец CreateEnumerable()
}
...
IEnumerable<int> iterable = CreateEnumerable();
IEnumerator<int> iterator = iterable.GetEnumerator();
Console.WriteLine("Starting to iterate");
                        // Начало итерации
while (true)
{
    Console.WriteLine("Calling MoveNext()...");
                        // Вызов MoveNext()
    bool result = iterator.MoveNext();
    Console.WriteLine("... MoveNext result={0}", result);
                        // Результат выполнения MoveNext()
    if (!result)
    {
      break;
    }
    Console.WriteLine("Fetching Current...");
                        // Извлечение Current
    Console.WriteLine("... Current result={0}", iterator.Current);
                        // Результат Current
}
```

Код в листинге 6.5 не выглядит идеально, особенно с точки зрения итерации. При нормальном ходе событий можно было бы просто воспользоваться циклом `foreach`, но чтобы четко показать, *что* происходит и *когда*, я разбил использование итератора на несколько частей. По большому счету этот код делает то, что делает цикл `foreach`, хотя `foreach` также вызывает в конце метод `Dispose()`. Как вскоре будет показано, для итераторных блоков это важно. Как видите, никаких отличий в синтаксисе внутри метода итератора нет, несмотря на то, что в этот раз вместо `IEnumerator<int>` возвращается `IEnumerable<int>`. Обычно с целью реализации `IEnumerable<T>` будет возвращаться только `IEnumerator<T>`; если из метода необходимо выдать последовательность, должен возвращаться тип `IEnumerable<T>`.

Ниже представлен вывод кода из листинга 6.5:

```
Starting to iterate
Calling MoveNext()...
                                    Start of CreateEnumerable()
                                    About to yield 0
... MoveNext result=True
```

```
Fetching Current...
... Current result=0
Calling MoveNext()...
                                    After yield
                                    About to yield 1
... MoveNext result=True
Fetching Current...
... Current result=1
Calling MoveNext()...
                                    After yield
                                    About to yield 2
... MoveNext result=True
Fetching Current...
... Current result=2
Calling MoveNext()...
                                    After yield
                                    Yielding final value
... MoveNext result=True
Fetching Current...
... Current result=-1
Calling MoveNext()...
                                    End of CreateEnumerable()
... MoveNext result=False
```

В этом выводе следует отметить несколько важных моментов.

- Обратите внимание, что код в методе `CreateEnumerable()` выполняется до первого обращения к `MoveNext()`.
- Вся реальная работа делается при вызове метода `MoveNext()`. Во время извлечения свойства `Current` никакой ваш код не выполняется.
- Код останавливает выполнение на операторе `yield return` и возобновляет его снова при следующем вызове `MoveNext()`.
- Можно иметь множество операторов `yield return` в разных местах метода.
- Код не заканчивается на последнем операторе `yield return`. Вместо этого вызов метода `MoveNext()`, приводящий к концу метода, возвращает `false`.

Первый момент особенно важен, поскольку он означает, что итераторный блок нельзя использовать для кода, который должен быть выполнен немедленно, когда вызывается метод, к примеру, кода проверки достоверности аргументов. Если поместить обычный код проверки внутрь метода, реализованного с помощью итераторного блока, то его поведение будет некорректным. В какой-то момент вы непременно столкнетесь с подобной ситуацией — это исключительно распространенная ошибка, которую трудно понять, если не думать о том, что именно делает итераторный блок. Решение данной проблемы будет показано в разделе 6.3.3.

Есть два вопроса, которые пока еще не рассматривались — альтернативный способ прекращения итерации и работа блоков `finally` в этой несколько необычной форме выполнения. Давайте взглянем на них сейчас.

6.2.3. Расширенный поток выполнения итератора

В обычных методах оператор `return` выполняет два действия. Во-первых, он поставляет значение, которое вызывающий код трактует как возвращаемое. Во-вторых, он пре-

кращает выполнение метода, при выходе выполняя любые подходящие блоки `finally`. Вы видели, что оператор `yield return` временно завершает метод, но только до следующего вызова `MoveNext()`, и мы совсем не рассматривали поведение блоков `finally`. Каким образом *на самом деле* остановить метод и что происходит со всеми блоками `finally`? Мы начнем с довольно простой конструкции — оператора `yield break`.

Завершение итератора с помощью оператора `yield break`

Всегда можно найти способ организации в методе единственной точки выхода, и многие упорно работают для достижения этой цели[2]. Те же самые приемы применимы и к итераторным блокам. Если вам нужно организовать ранний выход, следует обратиться к услугам оператора `yield break`. Он фактически завершает итератор, обеспечивая возврат значения `false` текущим вызовом `MoveNext()`.

В листинге 6.6 демонстрируется использование `yield break` на примере подсчета до 100 с остановом в случае нехватки времени. Этот код также иллюстрирует применение параметра метода в итераторном блоке и доказывает, что имя метода несущественно[3].

Листинг 6.6. Демонстрация использования оператора `yield break`

```
static IEnumerable<int> CountWithTimeLimit(DateTime limit)
{
  for (int i = 1; i <= 100; i++)
  {
    if (DateTime.Now >= limit)
    {
      yield break;                    <-- Останов в случае, если время истекло
    }
    yield return i;
  }
}
...
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
  Console.WriteLine("Received {0}", i);
  Thread.Sleep(300);
}
```

Обычно в результате запуска кода из листинга 6.6 будет получен вывод, состоящий из примерно семи строк. Цикл `foreach` завершается совершенно нормально — как и было задумано, у итератора просто заканчиваются элементы для перебора. Оператор `yield break` ведет себя подобно оператору `return` в нормальном методе.

До сих пор все было просто. Осталось выяснить последний аспект, связанный с потоком выполнения: как и когда выполняются блоки `finally`?

---

[2] Я считаю, что препятствия, которые при этом приходится преодолевать, часто делают код намного более трудным для чтения, чем наличие множества точек возврата, особенно в случае применения блоков `try/finally` для очистки и с учетом возможности возникновения исключений. Идея состоит в том, что это может быть сделано.

[3] Обратите внимание, что методы, принимающие параметры `ref` или `out`, не могут быть реализованы в виде итераторных блоков.

Выполнение блоков finally

Мы привыкли, что блоки finally выполняются каждый раз, когда покидается определенная область действия. Однако итераторные блоки ведут себя совсем не так, как обычные методы. Как уже было показано, оператор yield return фактически приостанавливает выполнение метода, а не осуществляет выход из него. Следуя такой логике, вы не должны ожидать выполнения в этот момент любых блоков finally, и это действительно так. Тем не менее, соответствующие блоки finally *выполняются*, когда достигнут оператор yield break, в точности как можно было ожидать при возвращении из обычного метода[4].

Наиболее распространенным использованием finally в итераторном блоке является освобождение ресурсов, обычно с помощью удобного оператора using. Реалистичный пример будет приведен в разделе 6.3.2, а пока мы просто попытаемся посмотреть, как и когда блоки finally выполняются. В листинге 6.7 это демонстрируется в действии – он содержит тот же самый код, что и листинг 6.6, но с блоком finally. Изменения выделены полужирным.

Листинг 6.7. Демонстрация работы yield break с блоком try/finally

```
static IEnumerable<int> CountWithTimeLimit(DateTime limit)
{
  try
  {
  for (int i = 1; i <= 100; i++)
  {
    if (DateTime.Now >= limit)
    {
      yield break;
    }
    yield return i;
  }
  }
  finally
  {
    Console.WriteLine("Stopping!");      <-- Выполняется, несмотря на конец цикла
                      // Останов
  }
}
...
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
  Console.WriteLine("Received {0}", i);  // Вывод полученного значения
  Thread.Sleep(300);
}
```

[4] Блоки finally также работают ожидаемым образом, когда поток выполнения покидает соответствующую область действия, не достигая оператора yield return или yield break. Я сосредоточил здесь внимание на поведении этих операторов yield только потому, что с ними связана новизна и отличия потока выполнения.

Блок finally в листинге 6.7 выполняется либо когда итераторный блок завершает свою работу, досчитав до 100, либо из-за достижения установленного лимита времени. (Он также бы выполнялся в случае генерации исключения в коде.) Но есть и другие способы, посредством которых можно попытаться избежать обращения к блоку finally... Попробуем проявить коварство.

Вы уже видели, что код в итераторном блоке выполняется только при вызове метода MoveNext(). А что произойдет, если никогда не вызывать MoveNext()? Или если вызвать его несколько раз и затем прекратить это делать? Давайте изменим вызывающую часть кода в листинге 6.7 следующим образом:

```
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
  Console.WriteLine ("Received {0}", i);
  if (i > 3)
  {
    Console.WriteLine("Returning");
    return;
  }
  Thread.Sleep(300);
}
```

Здесь производится не ранний останов в коде итератора, а ранний останов в коде, *использующем* этот итератор. Вывод, возможно, удивит:

```
Received 1
Received 2
Received 3
Received 4
Returning
Stopping!
```

Как видите, выполнение кода продолжается даже после достижения оператора return в цикле foreach. Это не должно происходить, если только не был задействован блок finally — и в данном случае их два! Вы уже знаете о блоке finally внутри метода итератора, но вопрос в том, что стало причиной его выполнения.

Ранее я уже давал подсказку — цикл foreach вызывает метод Dispose() на итераторе, возвращаемом GetEnumerator() в собственном блоке finally (подобно оператору using). Когда метод Dispose() вызывается на итераторе, созданном с помощью итераторного блока, до завершения итерации, конечный автомат выполняет любые блоки finally, которые находятся в области действия во время текущей "приостановки" кода. Это сложное и довольно подробное объяснение, но результат выразить проще: до тех пор, пока вызывающий код использует цикл foreach, блок finally работает внутри итераторных блоков так, как требуется.

Можно легко доказать, что причиной такого поведения является вызов метода Dispose(), применив итератор вручную:

```
DateTime stop = DateTime.Now.AddSeconds(2);
IEnumerable<int> iterable = CountWithTimeLimit(stop);
IEnumerator<int> iterator = iterable.GetEnumerator();
iterator.MoveNext();
Console.WriteLine("Received {0}", iterator.Current);
iterator.MoveNext();
Console.WriteLine("Received {0}", iterator.Current);
```

На этот раз строка `Stopping!` на консоль не выводится. Если явно добавить вызов `Dispose()`, то вы снова увидите в выводе дополнительную строку `Stopping!`. Необходимость в прекращении итератора до его фактического завершения будет возникать относительно редко, и столь же редко итерация будет выполняться вручную, а не посредством цикла `foreach`. Однако когда это все же *делается*, нужно не забывать о помещении итератора внутрь оператора `using`.

Большая часть поведения итераторных блоков уже рассмотрена, но в завершение данного раздела полезно ознакомиться с несколькими странностями, возникающими при работе с текущей реализацией Microsoft.

6.2.4. Индивидуальные особенности реализации

Если скомпилировать итераторные блоки с помощью компилятора Microsoft C# 2 и заглянуть в результирующий код IL посредством `ildasm` или Reflector, можно увидеть, что компилятор сгенерировал "за кулисами" вложенный тип. В моем случае компиляция кода из листинга 6.4 дает тип по имени `IterationSample.<GetEnumerator>d__0` (угловые скобки делают имя непроизносимым и никакого отношения к обобщениям не имеют). Я не собираюсь здесь подробно объяснять, что было сгенерировано, но имеет смысл просмотреть это в Reflector, чтобы получить представление о происходящем, предпочтительно с открытым разделом 10.14 ("Iterators" ("Итераторы")) спецификации языка. В спецификации определены различные состояния, в которых может находиться тип, и это описание упростит разбор сгенерированного кода. Основная часть работы выполняется в методе `MoveNext()`, который в целом представляет собой большой оператор `switch`.

К счастью, разработчикам нет необходимости особо переживать по поводу того, с какими препятствиями приходится сталкиваться компилятору. Тем не менее, полезно знать о нескольких странностях, касающихся реализации.

- До первого вызова метода `MoveNext()` свойство `Current` будет всегда возвращать стандартное значение для типа выдачи итератора.
- После того как метод `MoveNext()` возвращает `false`, свойство `Current` будет всегда возвращать значение, выданное последним.
- Метод `Reset()` всегда генерирует исключение вместо того, чтобы выполнять сброс подобно тому, как это делалось в показанной ранее ручной реализации. Это обязательное поведение, изложенное в спецификации.
- Вложенный класс всегда реализует и обобщенную, и необобщенную форму интерфейса `IEnumerator` (а также обобщенную и необобщенную разновидности `IEnumerable`, где это необходимо).

Отказ от реализации метода `Reset()` вполне обоснован — компилятор не в состоянии выяснить, что нужно делать для сброса итератора, или даже осуществимо ли это. Возможно, метода `Reset()` не должно было быть в интерфейсе `IEnumerator` с самого начала, и я затрудняюсь вспомнить, когда вызывал его в последний раз. Многие коллекции его не поддерживают, поэтому вызывающий код в любом случае не может на него полагаться.

Реализация дополнительных интерфейсов не меняет радикально картину. Интересно заметить, что если метод возвращает тип `IEnumerable`, в результате получается один класс, реализующий пять интерфейсов (включая `IDisposable`). Это детально объяс-

няется в спецификации языка, но разработчиков особо не касается. Тот факт, что тип реализует как `IEnumerable`, так и `IEnumerator` несколько необычен. Компилятор прикладывает определенные усилия, чтобы поддержать корректное поведение, но также обеспечивает создание единственного экземпляра вложенного типа в общем случае, когда требуется просто итерация по коллекции, в том же самом потоке, где она создавалась.

Поведение свойства `Current` выглядит странным — в частности, хранение последнего элемента после предположительного перехода далее может воспрепятствовать его обработке сборщиком мусора. Вполне возможно, что ситуация будет исправлена в последующих выпусках компилятора C#, но это маловероятно, т.к. оно приведет к нарушению работы существующего кода (компиляторы Microsoft C#, поставляемые в составе .NET 3.5, .NET 4 и .NET 4.5, ведут себя таким же образом). Строго говоря, это корректно с точки зрения спецификации языка C# 2 — в данной ситуации поведение свойства `Current` не определено. Хотя было бы лучше, если бы указанное свойство было реализовано в соответствие с предложениями, высказанными в документации по инфраструктуре, генерируя исключения в подходящие моменты.

Показанные несколько мелких недостатков были связаны с использованием автоматически сгенерированного кода, но в продуманном вызывающем коде никаких проблем возникать не должно — и давайте посмотрим правде в глаза: благодаря этой реализации вам не приходится писать *большой* объем кода. Это означает, что имеет смысл применять итераторы более широко, чем это могло делаться в C# 1. В следующем разделе приводятся примеры кода, которые позволят проверить свое понимание итераторных блоков и оценить, насколько они могут быть удобны в реальных, а не теоретических сценариях.

6.3. Реальные примеры использования итераторов

Приходилось ли вам когда-либо писать код, который не только действительно прост сам по себе, но также и делает весь проект *более* лаконичным? Со мной это случается довольно часто и обычно приносит гораздо больше удовольствия, чем вероятно должно — хотя вполне достаточно для того, чтобы ловить на себе странные взгляды коллег. Такой своего рода детский восторг особенно силен, когда дело касается использования нового средства языка способом, который очевидно лучше, и это делается не только ради того, чтобы поиграть с "новыми игрушками".

Даже сейчас, после многолетней работы с итераторами, я по-прежнему сталкиваюсь с ситуациями, при которых решение с применением итераторных блоков оказывается удачным, а результирующий код — более кратким, ясным и простым в понимании. В этом разделе будут представлены три таких сценария.

6.3.1. Итерация по датам в расписании

Работая над проектом, в котором используются расписания, я сталкивался с несколькими циклами, начинающимися следующим образом:

```
for (DateTime day = timetable.StartDate;
     day <= timetable.EndDate;
     day = day.AddDays(1))
```

Я много трудился над этой областью кода и всегда испытывал неприязнь к такому циклу, но это было только во время чтения кода вслух другому разработчику как псев-

докода, и как-то раз я понял, что упускал из виду один трюк. Я говорил примерно так: "для каждого дня в рамках расписания...". Оглядываясь назад, вполне очевидно, что на самом деле я хотел иметь дело с циклом `foreach`. (Для вас это может быть очевидным с самого начала, так что прошу прощения, если это так. Хорошо, что я не могу видеть ваше ухмыляющееся лицо.) Цикл выглядит намного лучше, если его переписать так:

```
foreach (DateTime day in timetable.DateRange)
```

В C# 1 я мог считать это заветной мечтой, но не беспокоиться о ее реализации; вы уже видели, насколько запутанной являлась реализация итератора вручную, а конечный результат в этом случае оказывался лишь чуть короче нескольких циклов `for`. Однако в C# 2 все стало легче. Внутрь класса, представляющего расписание, я просто добавил свойство:

```
public IEnumerable<DateTime> DateRange
{
  get
  {
    for (DateTime day = StartDate;
         day <= EndDate;
         day = day.AddDays(1))
    {
      yield return day;
    }
  }
}
```

В итоге исходный цикл перемещается в класс расписания, но это нормально. Для него намного лучше быть инкапсулированным именно здесь, в свойстве, которое просто проходит в цикле по дням, каждый раз выдавая по одному дню, чем находиться в бизнес-коде, имеющем дело с этими днями. Если бы требовалось сделать свойство более сложным (к примеру, пропуская выходные и праздничные дни), то это можно было осуществлять в одном месте и получить преимущества везде.

Одно такое небольшое изменение привело к серьезному улучшению в плане читабельности кода. Поскольку это произошло, я остановил рефакторинг коммерческого кода. Я *обдумывал* введение типа `Range<T>` для представления универсального диапазона, но это было нужно лишь в данной одной ситуации, потому я не считал целесообразным тратить дополнительные усилия на решение задачи. Оказалось, что это был мудрый ход. В первом издании этой книги я создал именно такой тип, но в нем присутствовали определенные недостатки, которые было трудно устранить в дружественном для книги стиле. Я существенно его перепроектировал для своей служебной библиотеки, но все еще испытываю некоторые опасения. Типы вроде него часто выглядят проще, чем есть на самом деле, и довольно скоро приходится постоянно иметь дело с каким-то краевым случаем. Подробные сведения о сложностях, с которыми я сталкивался, выходят за рамки материала этой книги, т.к. они больше касаются общего проектирования, чем языка C#, однако они интересны сами по себе, поэтому я описал их в статье на веб-сайте, посвященном книге (`http://csharpindepth.com/Articles/Chapter6/Ranges.aspx`).

Следующий пример является одним из моих любимых — он демонстрирует все, что мне нравится в итераторных блоках.

6.3.2. Итерация по строкам в файле

Насколько часто вы реализуете чтение текстового файла строка за строкой? Это невероятно распространенная задача. В .NET 4, наконец, в инфраструктуре появился метод, упрощающий решение этой задачи — `File.ReadLines()`, но в предшествующих версиях было довольно легко написать аналогичный метод самостоятельно, что и будет показано в настоящем разделе.

Даже страшно подумать, сколько раз мне приходилось писать код следующего вида:

```
using (TextReader reader = File.OpenText(filename))
{
  string line;
  while ((line = reader.ReadLine()) != null)
  {
    // Обработка строки.
  }
}
```

Здесь присутствуют четыре отдельных концепции.

- Способ получения экземпляра `TextReader`.
- Управление жизненным циклом `TextReader`.
- Итерация по строкам, которые возвращает метод `TextReader.ReadLine()`.
- Обработка каждой такой строки.

Специфичными для данной ситуации являются только первая и последняя концепции — управление жизненным циклом и механизм итерации представляют собой стандартный код. (Во всяком случае, управление жизненным циклом реализуется в языке C# довольно просто, и все это благодаря операторам `using`.) Есть два способа улучшить положение дел. Можно было бы воспользоваться делегатом — написать служебный метод, принимающий в качестве параметров средство чтения и делегат, вызывать этот делегат для каждой строки файла и закрыть средство чтения в конце метода. Такой подход часто служит примером применения замыканий и делегатов, но существует альтернатива, которую я нахожу более элегантной и намного лучше вписывающейся в концепции LINQ. Вместо передачи нужной логики методу в виде делегата можно использовать итератор для возвращения из файла по одной строке за раз, что позволит организовать обычный цикл `foreach`.

Достичь этого можно с помощью полноценного типа, реализующего интерфейс `IEnumerable<string>` (для такой цели в моей библиотеке MiscUtil предусмотрен класс `LineReader`), но автономный метод в другом классе также будет нормально работать. Он действительно прост, что подтверждает код в листинге 6.8.

Листинг 6.8. Проход в цикле по строкам файла с применением итераторного блока

```
static IEnumerable<string> ReadLines(string filename)
{
  using (TextReader reader = File.OpenText(filename))
  {
    string line;
    while ((line = reader.ReadLine()) != null)
    {
```

```
            yield return line;
        }
    }
}
...
foreach (string line in ReadLines("test.txt"))
{
    Console.WriteLine(line);
}
```

Тело метода `ReadLines()` в значительной степени совпадает с тем, что было ранее, за исключением выдачи строки вызывающему коду во время выполнения итерации по коллекции. Как и до этого, файл открывается, из него читается по одной строке за раз и затем средство чтения по завершении закрывается, хотя в этом случае концепция "по завершении" более интересна, чем использование оператора `using` в нормальном методе, где управление потоком более очевидно.

Вот почему настолько важно, что цикл `foreach` освобождает итератор — это гарантирует очистку средства чтения. Оператор `using` в методе итератора действует в качестве блока `try/finally`; этот блок `finally` будет выполняться *либо* по достижении конца файла, *либо* при вызове метода `Dispose()` экземпляра `IEnumerator<string>` где-то на середине пути. Вызывающий код вполне может неправильно употребить экземпляр `IEnumerator<string>`, возвращаемый методом `ReadLines(...).GetEnumerator()`, и в конечном итоге привести к утечке ресурсов, но это обычная ситуация с интерфейсом `IDisposable` — если не вызвать метод `Dispose()`, может возникнуть утечка ресурсов. Однако это редко является проблемой, т.к. цикл `foreach` обеспечивает правильное поведение. Важно помнить об этом потенциальном неправильном применении — если вы полагаетесь внутри итератора на какой-то блок `try/finally` для выдачи определенного разрешения, которое позже отбирается, то в таком случае действительно *может* возникнуть брешь в безопасности.

Этот метод инкапсулирует первые три из перечисленных ранее четырех концепций, но он несколько ограничен. Аспекты управления временем жизни и итерации разумно объединить вместе, но что, если необходимо прочитать текст не из файла, а из сетевого потока? Или нужно использовать кодировку, отличную от UTF-8? Первую часть придется поместить обратно под контроль вызывающего кода, и наиболее очевидный подход заключался бы в изменении сигнатуры метода для приема экземпляра `TextReader`, например:

```
static IEnumerable<string> ReadLines(TextReader reader)
```

Тем не менее, это неудачная идея. Необходимо иметь права на владение средством чтения, чтобы можно было очищать его удобным для вызывающего кода способом, но сам факт возложения ответственности за очистку означает и *обязанность* делать это при условии, что вызывающий код разумно использует результат. Проблема в том, что если что-то произойдет до первого вызова метода `MoveNext()`, то шансов выполнить очистку не будет: никакой ваш код не запустится. Тип `IEnumerable<string>` сам по себе не является освобождаемым, однако он хранил бы эту порцию состояния, которая требует освобождения. Еще одна проблема может возникнуть в случае, если метод `GetEnumerator()` был вызван дважды: это должно сгенерировать два независимых итератора, но они будут работать с одним и тем же средством чтения. Снизить риск возникновения такой проблемы можно было бы, изменив возвращаемый тип на `IEnumerator<string>`, но это означало бы, что результат не удастся применить в

цикле foreach, и по-прежнему не получилось бы выполнить любой код очистки при отсутствии даже первого обращения к методу MoveNext(). К счастью, существует обходной путь.

Точно так же как код не должен запускаться немедленно, не требуется немедленное предоставление и средства чтения. Вместо этого нужен способ получения средства чтения, когда оно необходимо. Можно было бы воспользоваться интерфейсом для представления идеи "я могу предоставить экземпляр TextReader, когда он нужен", но эта идея интерфейса с единственным методом должна обычно приводить к созданию делегата. Взамен я собираюсь немного схитрить, обратившись к делегату, который является частью .NET 3.5. Он имеет несколько перегруженных версий с разным количеством параметров типов, но нам необходима только одна из них:

```
public delegate TResult Func<TResult>()
```

Как видите, этот делегат не принимает параметров, но возвращает результат того же типа, что и параметр типа. Он имеет сигнатуру классического поставщика или фабрики. В данном случае нужно получить экземпляр TextReader, поэтому можно применять Func<TextReader>. Изменения, внесенные в метод, просты (они выделены полужирным):

```
static IEnumerable<string> ReadLines(Func<TextReader> provider)
{
  using (TextReader reader = provider())
  {
    string line;
    while ((line = reader.ReadLine()) != null)
    {
      yield return line;
    }
  }
}
```

Теперь ресурс запрашивается непосредственно, когда он необходим, и к этому моменту мы находимся в контексте интерфейса IDisposable, так что ресурс можно освободить в подходящее время. Более того, если метод GetEnumerator() вызывается несколько раз на возвращаемом значении, то каждый вызов будет приводить к созданию независимого экземпляра TextReader.

С помощью анонимных методов легко добавить перегруженные версии для открытия файлов с возможностью дополнительного указания кодировки:

```
static IEnumerable<string> ReadLines(string filename)
{
  return ReadLines(filename, Encoding.UTF8);
}
static IEnumerable<string> ReadLines(string filename, Encoding encoding)
{
  return ReadLines(delegate {
    return File.OpenText(filename, encoding);
  });
}
```

В этом примере применяются обобщения, анонимный метод (который захватывает параметры) и итераторный блок. Для получения полного набора основных средств C# 2 не хватает только типов значений, допускающих null. Я применял этот код в ряде слу-

чаев, и он всегда был намного аккуратнее, чем громоздкий код, с которого мы начинали. Как упоминалось ранее, в случае работы с последней версией .NET все это доступно в методе `File.ReadLines()`, но это по-прежнему лаконичный пример того, насколько полезными могут быть итераторные блоки.

В качестве последнего примера давайте получим первое впечатление от LINQ – несмотря на то, что будет применяться только версия C# 2.

6.3.3. Ленивая фильтрация элементов с использованием итераторного блока и предиката

Даже с учетом того, что мы еще не начинали рассматривать язык LINQ, как следует, я уверен, что у вас имеется определенное представление о нем: он позволяет запрашивать данные простым и мощным способом из множества источников, таких как коллекции, находящиеся в памяти, и базы данных. В версии C# 2 отсутствует какая-либо языковая интеграция для выражений запросов, а также для лямбда-выражений и расширяющих методов, которые могут сделать их лаконичными, но все равно можно получить ряд аналогичных результатов.

Одной из ключевых возможностей LINQ является фильтрация с помощью метода `Where()`. Данному методу предоставляется коллекция и предикат, а в результате получается оцениваемый ленивым образом запрос, выдающий только те элементы коллекции, которые соответствуют предикату. Это немного похоже на метод `List<T>.FindAll()`, но выполняется ленивым образом и работает с любым типом `IEnumerable<T>`. Среди множества привлекательных характеристик LINQ[5] следует выделить мастерское проектное решение, положенное в его основу. Как мы сейчас докажем, реализовать функциональность LINQ to Objects довольно просто, по крайней мере, в случае метода `Where()`. По иронии судьбы, хотя большинство языковых средств, которые делают LINQ настолько блестящим, являются частью C# 3, почти все они связаны с тем, как можно *получать доступ* к методам вроде `Where()`, а не с тем, каким образом они реализованы.

В листинге 6.9 приведен полный пример, включающий простую проверку достоверности аргументов. В нем используется фильтр для отображения всех директив `using` в исходном файле, который содержит код самого примера.

Листинг 6.9. Реализация метода `Where()` языка LINQ с применением итераторных блоков

```
public static IEnumerable<T> Where<T>(IEnumerable<T> source,
                                       Predicate<T> predicate)
{
  if (source == null || predicate == null)      <-- 1 Энергичная проверка
  {
    throw new ArgumentNullException();
  }
  return WhereImpl(source, predicate);          <-- 2 Ленивая обработка данных
}
private static IEnumerable<T> WhereImpl<T>(IEnumerable<T> source,
                                           Predicate<T> predicate)
{
  foreach (T item in source)
  {
```

[5] Или, точнее, LINQ to Objects. Поставщики LINQ для баз данных и им подобные намного сложнее.

```
      if (predicate(item)) ◄—● Проверка текущего элемента на предмет соответствия предикату
      {
        yield return item;
      }
    }
  }
  ...
  IEnumerable<string> lines = LineReader.ReadLines("../../FakeLinq.cs");
  Predicate<string> predicate = delegate(string line)
    { return line.StartsWith("using"); };
  foreach (string line in Where(lines, predicate))
  {
    Console.WriteLine(line);
  }
```

В этом примере реализация разбита на две части: проверка достоверности аргументов и действительная бизнес-логика фильтрации. Это немного неуклюже, но совершенно необходимо для разумной обработки ошибок. Предположим, что вы поместили все в один метод. Что произойдет при вызове `Where<string>(null, null)`? Ответ: *ничего...* или, во всяком случае, желаемое исключение не будет сгенерировано. Причина кроется в семантике ленивого выполнения итераторных блоков: код в теле метода не запускается до тех пор, пока метод `MoveNext()` не будет вызван первый раз, как объяснялось в разделе 6.2.2. Обычно проверять предусловия необходимо *энергичным образом* — нет никакого смысла в откладывании генерации исключения, т.к. это только усложнит отладку.

Стандартный обходной путь предусматривает разделение метода на две части, как было показано в листинге 6.9. Прежде всего, в нормальном методе проверяются аргументы ●, после чего производится вызов метода, реализованного с использованием итераторного блока, для ленивой обработки данных тогда, когда они запрашиваются ●.

Сам итераторный блок предельно прямолинеен: для каждого элемента в исходной коллекции осуществляется проверка на предмет соответствия предикату ● и выдача значения, если оно является подходящим. Если совпадения нет, проверяется следующий элемент и так до тех пор, пока не будет найден такой, который *совпадает*, или элементы не закончатся. Хотя это просто, но реализацию на C# 1 было бы намного труднее разбирать (и, разумеется, она не была бы обобщенной).

В заключительной порции кода, демонстрирующей метод в действии, для предоставления данных применяется последний пример — в этом случае в качестве данных выступает исходный код самой реализации. Предикат просто проверяет строку на предмет того, начинается ли она с `using` — конечно, с тем же успехом он мог бы содержать намного более сложную логику. Отдельные переменные для данных и предикат были созданы всего лишь для более ясного форматирования, но их можно было бы записать и внутристрочно. Важно отметить главное отличие между данным примером и его эквивалентом, который можно было бы построить с помощью методов `File.ReadAllLines()` и `Array.FindAll<string>()`. Эта реализация полностью соответствует характеристикам ленивого и потокового выполнения. В памяти постоянно должна находиться только одна строка исходного файла. Разумеется, это не играет роли в данном случае, когда файл имеет небольшой размер, но преимущества такого подхода легко заменить в случае, скажем, многогигабайтного журнального файла.

Надеюсь, что приведенные выше примеры дали некоторое представление о важности итераторных блоков, а также, возможно, пробудили желание побыстрее узнать дополнительные сведения о LINQ. Однако перед этим мне хочется немного запудрить вам мозги и представить совершенно странный (но действительно компактный) случай использования итераторов.

6.4. Написание псевдосинхронного кода с помощью библиотеки Concurrency and Coordination Runtime

Библиотека CCR (*Concurrency and Coordination Runtime* — исполняющая среда параллелизма и координации) разработана Microsoft для предоставления альтернативного способа написания асинхронного кода, поддающегося сложной координации. На момент написания этой книги она была доступна только в виде части среды Microsoft Robotics Developer Studio (`http://www.microsoft.com/robotics`). В Microsoft вложили немало средств в реализацию параллелизма в рамках различных проектов, наиболее значимым из которых являются библиотека параллельных задач (Task Parallel Library), появившаяся в .NET 4, и асинхронные возможности языка в C# 5 (поддерживаемые множеством асинхронных API-интерфейсов). Но я планирую применить библиотеку CCR для того, чтобы показать, как итераторные блоки могут изменить всю модель выполнения. На самом деле это раннее вторжение в альтернативный подход к реализации параллелизма использует итераторные блоки для изменения модели выполнения далеко не случайно; сходства между конечными автоматами, генерируемыми для итераторных блоков, и конечными автоматами, которые применяются для асинхронных функций в C# 5, просто поразительны.

Рассматриваемый далее пример кода, который работает с фиктивными службами, важен тем, что он демонстрирует сами идеи, а не детали реализации.

Предположим, что строится сервер, который должен обрабатывать большой объем запросов. В качестве части обработки запросов необходимо сначала обратиться к веб-службе для получения маркера аутентификации, а затем с помощью этого маркера получить данные из двух независимых источников (скажем, из базы данных и еще одной веб-службы). Полученные данные обрабатываются, и результат обработки возвращается. Каждая выборка может занять некоторое время — возможно, несколько секунд. Обычно для реализации применяется простой синхронный или типовой асинхронный подход. Синхронная версия может выглядеть следующим образом:

```
HoldingsValue ComputeTotalStockValue(string user, string password)
{
  Token token = AuthService.Check(user, password);
  Holdings stocks = DbService.GetStockHoldings(token);
  StockRates rates = StockService.GetRates(token);
  return ProcessStocks(stocks, rates);
}
```

Понять этот код очень легко. Но необходимо учитывать, что если каждый запрос занимает 2 секунды, то целая операция потребует 6 секунд и заблокирует поток на все время своего выполнения. Если нужно масштабировать код для параллельной обработки сотен тысяч запросов, возникает проблема.

А теперь давайте рассмотрим довольно простую асинхронную версию, в которой поток не блокируется, когда ничего не происходит[6], и везде, где возможно, применяются параллельные вызовы:

```
void StartComputingTotalStockValue(string user, string password)
{
  AuthService.BeginCheck(user, password, AfterAuthCheck, null);
}
void AfterAuthCheck(IAsyncResult result)
{
  Token token = AuthService.EndCheck(result);
  IAsyncResult holdingsAsync = DbService.BeginGetStockHoldings
    (token, null, null);
  StockService.BeginGetRates(token, AfterGetRates, holdingsAsync);
}
void AfterGetRates(IAsyncResult result)
{
  IAsyncResult holdingsAsync = (IAsyncResult)result.AsyncState;
  StockRates rates = StockService.EndGetRates(result);
  Holdings stocks = DbService.EndGetStockHoldings(holdingsAsync);
  OnRequestComplete(ProcessStocks(stocks, rates));
}
```

Приведенный код намного труднее для чтения и понимания — и это лишь простая версия. Координацию двух параллельных вызовов удалось достичь так легко только потому, что отсутствовала необходимость в передаче любого другого состояния, но даже здесь она не идеальна. Если обращение к службе биржевых котировок (StockService) выполнится быстро, поток из пула все равно будет блокироваться из-за ожидания, пока завершится запрос к базе данных. А еще важнее то, что при этом трудно понять происходящее, поскольку код перескакивает с метода на метод.

К этому моменту может возникнуть вопрос о том, как в общую картину могут вписаться итераторы. Итераторные блоки, предоставляемые C# 2, фактически позволяют приостанавливать текущее выполнение в определенных точках потока через блок и затем возвращаться в те же самые места с тем же состоянием. Талантливые проектировщики библиотеки CCR осознали, что им нужен стиль кодирования, предусматривающий *передачу признака продолжения*. Системе необходимо сообщить о том, что есть набор операций, которые требуется выполнить, включая асинхронный запуск других операций, но затем можно благополучно ожидать завершения асинхронных операций, прежде чем продолжать дальше. Это делается за счет предоставления библиотеке CCR реализации интерфейса IEnumerator<ITask> (где ITask — интерфейс, определенный в библиотеке CCR). Ниже показан код, в котором для достижения тех же целей, что и ранее, применяется описанный стиль:

```
static IEnumerator<ITask> ComputeTotalStockVal.(str.user,str.pass)
{
  string token = null;
  yield return Arbiter.Receive(false, AuthService.CcrCheck(user, pass),
    delegate(string t) { token = t; });
  IEnumerable<Holding> stocks = null;
  IDictionary<string,decimal> rates = null;
```

[6] Ладно, по большей части. Как вскоре будет показано, эта версия по-прежнему может быть неэффективной.

```
        yield return Arbiter.JoinedReceive(false,
          DbService.CcrGetStockHoldings(token),
          StockService.CcrGetRates(token),
          delegate(IEnumerable<Holding> s, IDictionary<string,decimal> r)
            { stocks = s; rates = r; });
        OnRequestComplete(ComputeTotal(stocks, rates));
    }
```

Запутались? Когда я впервые увидел этот код, то определенно запутался, но теперь я восторгаюсь его лаконичностью. Библиотека CCR приводит в действие ваш код (посредством вызова метода `MoveNext()` на итераторе), и выполнение происходит вплоть до первого оператора `yield return`, включая его. Метод `CcrCheck()` внутри типа `AuthService` запускает асинхронный запрос, а библиотека CCR ожидает (не используя отдельный поток) его завершения, вызывая предоставленный делегат для обработки результата. Затем `MoveNext()` вызывается снова, и ваш метод продолжает выполнение. На этот раз запускаются два запроса параллельно, а библиотеке CCR предлагается вызвать другой делегат с передачей ему результатов выполнения обеих операций, когда они *обе* завершатся. Наконец, метод `MoveNext()` вызывается последний раз, и дело доходит до завершения обработки запросов.

Хотя асинхронная версия заметно сложнее синхронной, она по-прежнему содержит один метод, выполняется в том порядке, в каком написан код, и сам метод может хранить состояние (в локальных переменных, которые становятся состоянием внутри дополнительного типа, сгенерированного компилятором). Все выполняется полностью асинхронно с использованием минимально необходимого количества потоков. В коде не была показана обработка ошибок, но она также доступна, причем в стиле, который вынуждает обдумывать возможности возникновения проблем в подходящих местах кода.

Я умышленно не погружаюсь здесь в детали класса `Arbiter`, интерфейса `ITask` и тому подобного. Вдобавок в этом разделе я не пытаюсь рекламировать библиотеку CCR, хотя читать и экспериментировать с ней исключительно интересно; я полагаю, что асинхронные функции в C# 5 окажут намного большее влияние на подавляющее большинство разработчиков. Моей целью было показать, что итераторы могут применяться в радикально отличающихся контекстах, которые имеют мало общего с традиционными коллекциями. В основе этого использования лежит идея конечного автомата: двумя запутанными аспектами асинхронной разработки являются поддержка состояния и действительная приостановка до тех пор, пока не произойдет что-то интересное. Итераторные блоки вполне естественно приспособлены для устранения обеих этих проблем, хотя в главе 15 вы увидите, что целевая языковая поддержка позволяет получать намного более ясные решения.

6.5. Резюме

В C# косвенным образом поддерживается много шаблонов, в плане осуществимости их реализации средствами языка. Однако лишь относительно небольшое число шаблонов поддерживаются *напрямую* в терминах языковых возможностей, специально ориентированных на конкретные шаблоны. В C# 1 шаблон итератора напрямую поддерживался с точки зрения вызывающего кода, но не коллекции, по которой производится итерация. Написание корректных реализаций интерфейса `IEnumerable` было затратным по времени и подверженным ошибкам, к тому же неинтересным. В C# 2 компилятор делает всю рутинную работу за вас, строя конечный автомат, чтобы стравиться с природой обратных вызовов, которая присуща итераторам.

Следует отметить, что итераторные блоки имеют один общий аспект с анонимными методами, рассмотренными в главе 5, невзирая на то, что их действительные возможности сильно отличаются. В обоих случаях могут быть сгенерированы дополнительные типы, а исходный код подвергается потенциально сложной трансформации. Сравните это с C# 1, где большинство трансформаций для синтаксического сахара (самыми очевидными примерами являются `lock`, `using` и `foreach`) были прямолинейными. Такая тенденция более интеллектуальной компиляции будет продолжена в почти каждом аспекте версии C# 3.

В этой главе была показана одна порция функциональности, связанной с LINQ: фильтрация коллекции. Интерфейс `IEnumerable<T>` является одним из наиболее важных типов в LINQ, но даже если нужно записать собственные операции LINQ на основе LINQ to Objects[7], вы будете неизменно благодарны команде разработчиков C# за включение в язык итераторных блоков.

В дополнение к нескольким реалистичным примерам использования итераторов было показано, как одна конкретная библиотека применяла их довольно радикальным способом, который имеет мало общего с тем, что приходит на ум, когда речь идет об итерации по коллекции. Полезно помнить о том, что многие языки сталкивались с подобного рода проблемой ранее — в вычислительной технике для концепций такого вида применяется термин *сопрограмма*. На эти концепции ссылаются в наборе инструментов для разработки трехмерных игр под названием Unity, где они используются для обеспечения асинхронности. Исторически разные языки предлагали поддержку указанных концепций в большей или меньшей степени, иногда с подходящими уловками для их моделирования. Например, у Саймона Тэтхема есть великолепная статья о том, как реализовывать сопрограммы даже на языке C, если вы готовы кое в чем отступить от стандартов кодирования (его статья "Coroutines in C" ("Сопрограммы в C") доступна по адресу `http://mng.bz/H8YX`). Вы увидите, что C# 2 упрощает написание и работу с сопрограммами.

Теперь, когда вы ознакомились с рядом крупных и иногда трудно понимаемых изменений в языке, сосредоточенных вокруг новых основных возможностей, в следующей главе вводятся новые правила. В ней будет описано несколько небольших изменений, которые делают работу с C# 2 более удобной по сравнению с предшествующей версией. Проектировщики извлекли урок из прошлых критических замечаний и построили язык, который имеет меньше острых углов, больше возможностей разрешения ситуаций с затруднениями при обеспечении обратной совместимости и лучшее восприятие генерируемого кода. Каждая возможность относительно проста, но их довольно много.

[7] Это выглядит не таким уж обескураживающим, как может показаться на первый взгляд. Несколько руководящих принципов по этой теме будут представлены в главе 12.

Глава 7

Заключительные штрихи C# 2: финальные возможности

В этой главе...

- Частичные типы
- Статические классы
- Отдельные модификаторы доступа для средств получения/установки свойств
- Псевдонимы пространств имен
- Директивы `pragma`
- Буферы фиксированного размера
- Дружественные сборки

До сих пор вы видели четыре крупнейших новых средства в C# 2: обобщения, типы, допускающие `null`, усовершенствованные делегаты и итераторные блоки. Каждое из них ориентировано на удовлетворение довольно сложной потребности, поэтому они рассматривались относительно подробно. Оставшиеся новые возможности C# 2 срезают несколько острых углов C# 1. Они представляют собой мелочи, которые проектировщики языка решили подправить — либо области, где требовались усовершенствования языка ради себя самого, либо там, где особенности работы с генерацией кода и машинным кодом могли быть улучшены.

На протяжении длительного времени в Microsoft получали множество откликов сообщества программистов на C# (и, конечно же, собственных разработчиков) относительно областей, в которых язык C# не был настолько блестящим, насколько мог

быть. Ситуацию облегчили небольшие изменения, внесенные в C# 2 наряду с крупными изменениями.

Ни одно из рассматриваемых в этой главе средств не является особенно трудным, так что мы пройдем их довольно быстро. Однако их важность нельзя недооценивать. Только тот факт, что тема может быть раскрыта на нескольких страницах, не означает ее бесполезность. Скорее всего, вы будете пользоваться этими средствами на регулярной основе. Ниже приведен краткий обзор средств, описанных в данной главе, вместе с областями их применения.

- **Частичные типы.** Возможность написания кода для типа в нескольких файлах исходного кода. Это особенно удобно для типов, часть кода которых генерируется автоматически, а другая часть пишется вручную.
- **Статические классы.** Предназначены для упорядочения вспомогательных классов, что позволяет компилятору распознавать попытки их некорректного использования и делать намерения более ясными.
- **Отдельные модификаторы доступа для средств получения/установки свойств.** Наконец-то появилась возможность иметь открытое средство получения и закрытое средство установки для свойства! (Это не единственная доступная, но самая распространенная комбинация.)
- **Псевдонимы пространств имен.** Пути выхода из сложных ситуаций, когда имена типов не являются уникальными.
- **Директивы `pragma`.** Специфичные для компилятора инструкции, позволяющие выполнять такие действия, как подавление выдачи определенных предупреждений в отдельной части кода.
- **Буферы фиксированного размера.** Больший контроль над тем, как структуры обрабатывают массивы в небезопасном коде.
- **Класс `InternalsVisibleToAttribute`** (дружественные сборки). Средство, охватывающее язык, инфраструктуру и исполняющую среду, которое при необходимости разрешает более широкий доступ к выбранным сборкам.

Возможно, сейчас вы испытываете непреодолимое желание добраться до привлекательных материалов версии C# 3, и я вас не порицаю за это. Ничто в этой главе не является сенсационным, тем не менее, каждое из рассмотренных средств в ряде случаев может упростить жизнь или закрыть какую-то брешь. Первое средство пользуется достаточно высокой популярностью.

7.1. Частичные типы

Первое изменение, которого мы коснемся, появилось в ответ на сложности, сопровождающие использование генераторов кода в версии C# 1. В случае Windows Forms визуальному конструктору среды Visual Studio требовались собственные области кода, которые не могли затрагиваться разработчиками — и они находились внутри того же самого файла, который разработчики *должны* были редактировать с целью формирования функциональности пользовательского интерфейса. Понятно, что ситуация была хрупкой.

В других случаях генераторы кода создают исходный код, который компилируется наряду с кодом, написанным вручную. В C# 1 добавление дополнительной функциональности предполагает создание новых классов, унаследованных от автоматически сгенерированных, что совершенно неуклюже. Существует множество других сценариев,

при которых наличие излишнего звена в цепочке наследования может вызвать проблемы или ухудшить степень инкапсуляции. Например, если две разных части кода должны вызывать друг друга, в родительском типе понадобятся виртуальные методы для обращения к дочернему типу и защищенные методы для противоположной ситуации, тогда как обычно применялись бы два закрытых невиртуальных метода.

В C# 2 разрешено образовывать тип с помощью более чем одного файла, и среды IDE могут расширять это понятие тем, что код, используемый для типа, может даже не быть видимым как исходный код на C#. Типы, построенные из нескольких файлов исходного кода, называются *частичными*.

В этом разделе мы также обсудим *частичные методы*, которые уместны только в частичных типах и позволяют развитый и эффективный способ добавления вручную написанных привязок в автоматически генерируемый код. В действительности данное средство относится к версии C# 3 (на этот раз оно основано на отзывах о C# 2), но его логичнее обсуждать при исследовании частичных типов, а не ждать следующей части книги.

7.1.1. Создание типа с помощью нескольких файлов

Создание частичного типа — дело легкое; нужно просто добавить контекстное ключевое слово `partial` к объявлению типа в каждом файле, где оно встречается. Частичный тип может быть объявлен внутри любого желаемого количества файлов, хотя во всех примерах в этом разделе применяются два файла.

Компилятор, в сущности, объединяет все файлы исходного кода вместе перед тем, как приступить к собственно компиляции. Это означает, что код в одном файле может обращаться к коду в другом и наоборот (рис. 7.1) — опережающие ссылки и прочие трюки не нужны.

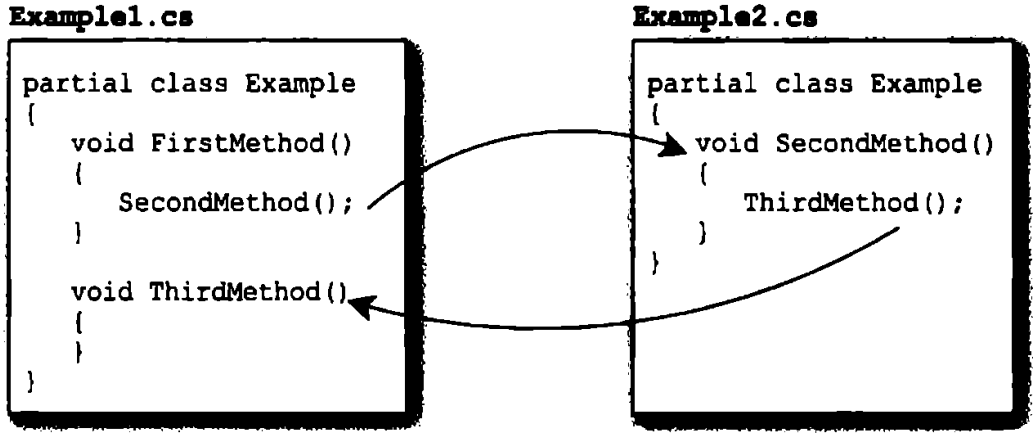

Рис. 7.1. Код в частичных типах имеет доступ ко всем членам типа независимо от того, в каком из файлов они находятся

Не допускается записывать одну часть кода члена в одном файле, а другую во втором — каждый отдельный член должен быть реализован полностью внутри собственного файла. Например, невозможно начать метод в одном файле и завершить его в другом[1]. С объявлениями типа связано несколько очевидных ограничений — объявления должны быть совместимыми. В любом файле можно указывать интерфейсы, предназначенные для реализации (и они не обязаны быть реализованными в этом файле), в любом файле

[1] Здесь имеется одно исключение: частичные типы могут содержать вложенные частичные типы, распространяющиеся по тому же набору файлов.

может быть задан базовый тип, а также могут быть указаны ограничения на параметре типа. Но если в нескольких файлах указаны базовые типы, то они должны быть одинаковыми, а если установлены ограничения для типов, то эти ограничения должны быть идентичными. В листинге 7.1 демонстрируется пример предлагаемой гибкости (хотя код в нем не делает ничего такого, что было бы даже отдаленно полезным).

Листинг 7.1. Демонстрация смешанных объявлений частичного типа

```
// Example1.cs
using System;

partial class Example<TFirst, TSecond>
   : IEquatable<string>                    <-- ❶ Указание интерфейса и ограничения параметра типа
   where TFirst : class
{
   public bool Equals(string other)        <-- ❷ Реализация IEquatable<string>
   {
      return false;
   }
}

// Example2.cs
using System;

partial class Example<TFirst, TSecond>
    : EventArgs, IDisposable               <-- ❸ Указание базового класса и интерфейса
{
   public void Dispose()                   <-- ❹ Реализация IDisposable
   {
   }
}
```

Я подчеркиваю, что код в листинге 7.1 предназначен единственно для целей обсуждения того, что является допустимым в объявлении — используемые в нем типы выбраны только из-за удобства и известности. Как видите, оба объявления (❶ и ❸) вносят свой вклад в список интерфейсов, которые должны быть реализованы. В данном примере внутри каждого файла реализованы интерфейсы, которые в нем же и объявлены, и это распространенный сценарий. Тем не менее, было бы вполне законно перенести реализацию `IDisposable` ❹ в файл `Example1.cs`, а реализацию `IEquatable<string>` ❷ — в файл `Example2.cs`. Я воспользовался возможностью указания интерфейсов отдельно от реализации, инкапсулируя методы с одной и той же сигнатурой, генерируемые множеством разных типов, внутри интерфейса. Генератору кода ничего не известно об интерфейсе, поэтому он также не знает, как объявить о том, что тип его реализует.

Ограничения типа указаны только в первом объявлении ❶, а базовый класс — только во втором ❸. Если бы в первом объявлении ❶ был задан базовый класс, им должен был бы стать `EventArgs`, а если во втором объявлении были указаны ограничения типа, то они должны были бы в точности совпадать с теми, что определялись в первом объявлении. В частности, во втором объявлении нельзя указывать ограничение типа для `TSecond`, даже притом, что оно не упоминалось первым. Оба типа должны иметь одинаковый модификатор доступа, если он есть — к примеру, невозможно делать одно объявление `internal`, а другое `public`. По существу правила объединения файлов в большинстве случаев допускают гибкость, приветствуя согласованность.

Для типов, определенных в единственном файле, инициализация переменных-членов и статических переменных гарантированно происходит в порядке их нахождения в файле, но в случае нескольких файлов какой-либо порядок не обеспечивается. Прежде всего, не следует полагаться на порядок объявления внутри файла, т.к. это спровоцирует появление в вашем коде трудноуловимых ошибок, если разработчик решит "безвредно" изменить положение к лучшему — поэтому по возможности стоит избегать такой ситуации. *Особенно* это касается частичных типов.

Располагая сведениями о том, что можно делать, а что нельзя, давайте внимательнее посмотрим, *почему* это может понадобиться.

7.1.2. Использование частичных типов

Как упоминалось ранее, частичные типы главным образом удобны в сочетании с визуальными конструкторами и другими генераторами кода. Если генератор кода должен модифицировать файл, которым владеет разработчик, всегда есть риск, что что-то пойдет не так. Благодаря модели частичных типов, генератор кода может иметь собственный рабочий файл и полностью переписывать его каждый раз, когда в этом возникает необходимость.

Некоторые генераторы кода могут отложить формирование файла со своим кодом C# до этапа компиляции проекта. Например, в инструменте Snippy предусмотрены файлы на языке XAML (Extensible Application Markup Language — расширяемый язык разметки приложений), содержащие описание пользовательского интерфейса. Во время построения проекта каждый файл XAML преобразуется в файл C# внутри каталога `obj` (имя файла имеет расширение `.g.cs`, которое указывает на то, что он был сгенерирован автоматически). Затем этот файл компилируется наряду с частичным классом, предоставляющим дополнительный код для данного типа (обычно код обработчиков событий и добавочных конструкторов). Это предотвращает вмешательство в сгенерированный код со стороны разработчиков, за исключением разве что прямой корректировки окончательного файла сборки.

Я старался аккуратно использовать формулировку *генератор кода* вместо *визуальный конструктор*, поскольку существует множество генераторов кода, никак не связанных с визуальными конструкторами. Например, в Visual Studio прокси для веб-служб генерируются в виде частичных классов, вдобавок вы можете располагать собственными инструментами, которые генерируют код на основе других источников данных. Одним довольно распространенным примером этого является объектно-реляционное отображение (object-relational mapping — ORM); некоторые инструменты ORM для описания сущностей из базы данных применяют конфигурационный файл (или получают их прямо из базы данных) и генерируют частичные классы, представляющие эти сущности. Подобным же образом моя версия инфраструктуры сериализации языка Google Protocol Buffers для .NET генерирует частичные классы — возможность, которая оказалась полезной даже в рамках самой реализации.

Это упрощает реализацию поведения в типе — переопределение виртуальных методов базового класса, добавление новых членов с бизнес-логикой и т.д. В результате получается отличный подход к совместной работе инструмента и разработчика, не выясняя постоянно, кто тут главный.

Иногда оказывается удобным следующий сценарий: один автоматически генерируемый файл содержит множество частичных типов, в то время как некоторые из этих типов расширяются в других файлах, с выделением одного вручную создаваемого файла на каждый тип. Возвращаясь к примеру с ORM, инструмент мог бы генерировать оди-

ночный файл с определениями всех сущностей, а ряд этих сущностей могли бы иметь дополнительный код, который предоставляется разработчиком в файлах, выделенных для каждой сущности. Такой прием позволяет сохранять количество автоматически генерируемых файлов небольшим, одновременно обеспечивая хорошую поддержку для кода, который пишется вручную.

На рис. 7.2 показано, что применение частичных типов для XAML и сущностей похожи, но слегка отличается в плане координации, когда дело доходит до создания автоматически генерируемого кода C#.

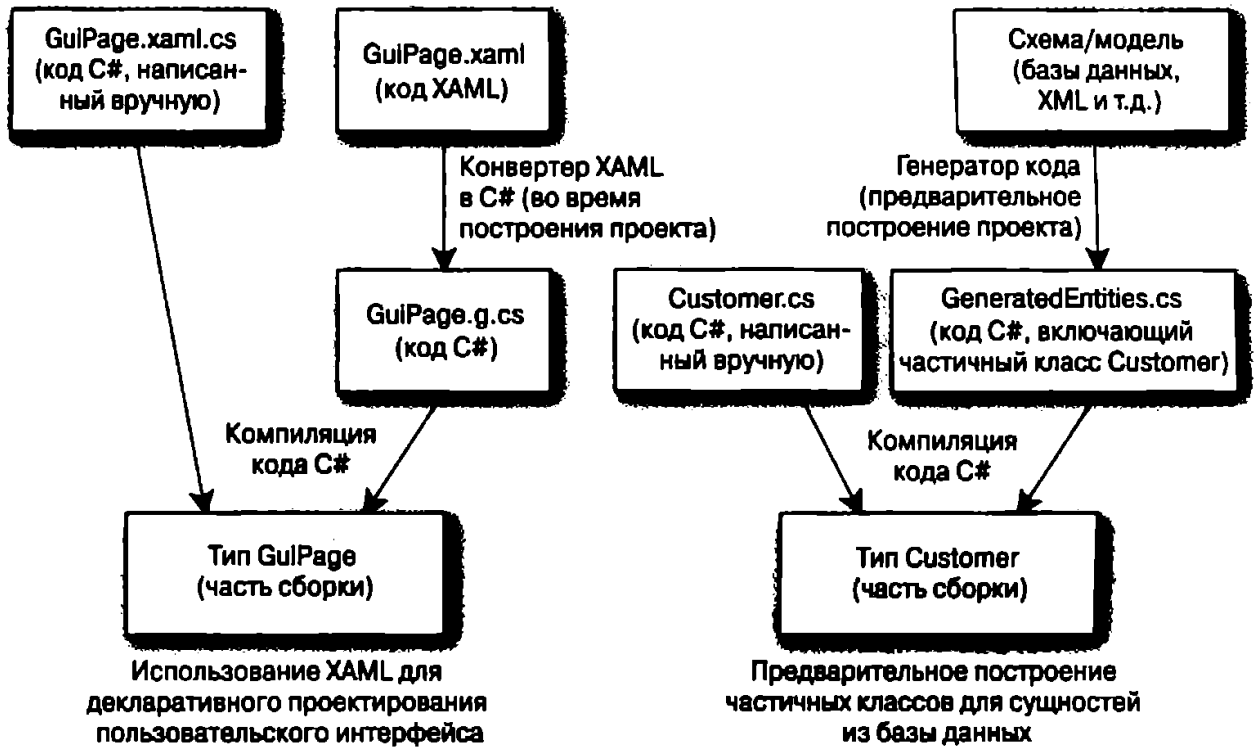

Рис. 7.2. Сравнение предварительной компиляции XAML и автоматически генерируемых классов для сущностей

Существует еще один, несколько отличающийся случай использования частичных типов — содействие рефакторингу. Временами тип становится слишком крупным и предполагает чересчур большое количество ответственностей. Первый шаг разбиения такого раздутого типа на небольшие, более ясные типы предусматривает превращение его в частичный тип, разнесенный по двум и более файлам. Это может быть сделано безо всякого риска и в экспериментальной манере, с перемещением методов между файлами до тех пор, пока каждый файл не получит единственную ответственность. Несмотря на то что следующий шаг разбиения типа по-прежнему далек от автоматического, увидеть конечную цель должно быть намного легче.

Стоит упомянуть о последнем случае применения частичных типов — модульном тестировании. Набор модульных тестов для класса часто становится намного больше самой реализации. Один из способов разделения этих тестов на осмысленные порции заключается в использовании частичных типов. Можно по-прежнему запускать все тесты для типа за один присест (т.к. все еще имеется единственный тестовый класс), но тесты для разных областей функциональности воспринимаются проще, когда они находятся в разных файлах. За счет редактирования вручную файла проекта можно даже обеспечить такое же поведение с раскрытием родительских/дочерних областей кода в окне Solution

Explorer (Проводник решения), которое доступно в случае применения частичных типов в сгенерированном коде Visual Studio. Конечно, это дело вкуса, но лично я нахожу такой подход к управлению тестами весьма удобным.

Когда частичные типы впервые появились в C# 2, поначалу никто в точности не знал, как ими пользоваться. Почти немедленно была затребована возможность предоставления дополнительного кода для вызываемых сгенерированных методов. Это было решено в версии C# 3 введением частичных методов.

7.1.3. Частичные методы (только C# 3)

Повторяю ранее данное объяснение: остальной материал данной главы посвящен средствам C# 2, но частичные методы не вписываются в какие-либо другие возможности C# 3, а хорошо *согласуются* с темой частичных типов. Приношу свои извинения за путаницу, которую это может вызвать.

Вернемся к средству частичных методов: иногда нужно иметь возможность определить поведение внутри вручную созданного файла и обращаться к этому поведению из автоматически сгенерированного файла. Например, в классе, который располагает множеством автоматически сгенерированных свойств, для определенных свойств может понадобиться средство указания кода, отвечающего за проверку достоверности новых значений. Еще один распространенный сценарий касается включения конструкторов для генератора кода. Здесь может потребоваться привязка написанного вручную кода к конструированию объектов для установки стандартных значений, фиксации некоторых сведений в журнале и тому подобных действий.

В C# 2 указанные требования могли быть удовлетворены только либо с помощью событий, на которые подписывался вручную написанный код, либо установки в автоматически сгенерированном коде *предположений* о том, что вручную написанный код будет содержать методы с определенными именами. Если такие методы не были предоставлены, то код не мог быть скомпилирован. В качестве альтернативы сгенерированный код может предоставить базовый класс с виртуальными методами, которые по умолчанию ничего не делают. Затем в коде, написанном вручную, можно создать класс, производный от этого базового класса, и переопределять некоторые или все его методы.

Все описанные решения кое в чем запутаны. Частичные методы C# 3 по существу предлагают *необязательные* привязки, которые не влекут за собой никаких затрат, когда они не реализованы — любые вызовы нереализованных частичных методов удаляются компилятором. Это позволяет инструментам проявлять большую щедрость в смысле количества предоставляемых привязок. В скомпилированном коде затраты будут связаны только с теми привязками, которые используются.

Понять сказанное легче всего на примере. В листинге 7.2 показан частичный тип, определенный в двух файлах. Конструктор этого типа находится в автоматически сгенерированном коде и вызывает два частичных метода, один из которых реализован в коде, написанном вручную.

Листинг 7.2. Частичный метод, вызываемый в конструкторе

```
// Generated.cs
using System;
partial class PartialMethodDemo
{
  public PartialMethodDemo()
  {
```

```
    OnConstructorStart();
    Console.WriteLine("Generated constructor");
    OnConstructorEnd();
  }
  partial void OnConstructorStart();
  partial void OnConstructorEnd();
}
// Handwritten.cs
using System;
partial class PartialMethodDemo
{
  partial void OnConstructorEnd()
  {
    Console.WriteLine("Manual code");
  }
}
```

В листинге 7.2 легко заметить, что частичные методы объявляются почти как абстрактные методы: за счет предоставления сигнатуры без реализации, но только на этот раз применяется модификатор `partial`. Аналогично, действительные реализации просто имеют модификатор `partial`, но в остальном они выглядят как нормальные методы.

Вызов конструктора `PartialMethodDemo` без параметров в результате приведет к выводу на консоль строки `Generated constructor` и затем сроки `Manual code`. Если вы просмотрите код IL для конструктора, то не увидите вызова метода `OnConstructorStart()`, т.к. он больше не существует — в скомпилированном типе вы не обнаружите никаких его следов.

Поскольку метод может не существовать, частичные методы должны иметь возвращаемый тип `void` и не могут принимать параметры `out`. Они должны быть закрытыми, но могут быть статическими и/или обобщенными. Если метод не реализован в одном из файлов, вызывающий его оператор полностью удаляется, *включая любые оценки аргументов*.

Если с оценкой любого аргумента связан побочный эффект, который должен произойти независимо от того, реализован ли частичный метод, то такая оценка должна выполняться отдельно. Например, предположим, что имеется следующий код:

```
LogEntity(LoadAndCache(id));
```

Здесь `LogEntity()` является частичным методом, а метод `LoadAndCache()` загружает сущность из базы данных и помещает ее в кеш. Вместо этого кода может возникнуть желание записать так:

```
MyEntity entity = LoadAndCache(id);
LogEntity(entity);
```

Таким образом, сущность загружается и кешируется независимо от того, предоставлена ли реализация для метода `LogEntity()`. Разумеется, если сущность может быть загружена с эквивалентными затратами позже или это может даже не потребоваться, необходимо оставить первую форму оператора, избегая нежелательной загрузки в определенных случаях.

По правде говоря, если только вы не занимаетесь написанием собственных генераторов кода, то, скорее всего, вы будете чаще реализовывать частичные методы, чем объявлять и вызывать их. В случае только реализации заботиться о проблемах оценки аргументов не придется.

Подводя итог вышесказанному, частичные методы в C# 3 позволяют сгенерированному коду широко взаимодействовать с кодом, написанным вручную, без потери производительности в ситуациях, когда такое взаимодействие не требуется. Это естественное продолжение средства частичных типов C# 2, которое поддерживает намного более продуктивные отношения между генераторами кода и разработчиками.

Возможность, рассматриваемая следующей, является совершенно другой. Она представляет собой способ сообщения компилятору о предполагаемой природе типа, чтобы он мог выполнять больший объем проверок самого типа и любого кода, использующего тип.

7.2. Статические классы

Второе новое средство в некотором смысле совершенно необязательно — оно просто позволяет писать более аккуратный и элегантный код при построении *вспомогательных классов*.

Каждый разработчик имеет свои вспомогательные классы. Мне не приходилось видеть хоть сколько-нибудь значительный проект на Java или C#, в котором бы не присутствовал хотя бы один класс, состоящий исключительно из статических методов. Классическим примером может служить тип со вспомогательными методами для обработки строк, которые выполняют отмену символов, обращение порядка, интеллектуальную замену — словом все, что угодно. В рамках инфраструктуры таким примером является класс `System.Math`.

Ниже перечислены ключевые особенности вспомогательного класса.

- Все члены являются статическими (кроме закрытого конструктора).
- Класс унаследован напрямую от `object`.
- Обычно состояние не поддерживается, если только не задействовано кеширование или шаблон с единственным экземпляром.
- Видимые конструкторы отсутствуют.
- Класс является запечатанным, если разработчик не забыл сделать это.

Последние два пункта необязательны, и при отсутствии видимых извне конструкторов (в том числе защищенных) класс *в любом случае* становится запечатанным. Тем не менее, оба эти пункта позволяют дополнительно прояснить предназначение класса.

В листинге 7.3 приведен пример вспомогательного класса в версии C# 1. После этого мы посмотрим, какие улучшения предлагает версия C# 2.

Листинг 7.3. Типичный вспомогательный класс C# 1

```
public sealed class NonStaticStringHelper   <-- 1 Запечатывание класса во избежание наследования от него
{
  private NonStaticStringHelper()   <-- 2 Предотвращение создания экземпляров в другом коде
  {
  }
  public static string Reverse(string input)   <-- 3 Все методы являются статическими
  {
     char[] chars = input.ToCharArray();
     Array.Reverse(chars);
     return new string(chars);
  }
}
```

Класс запечатан ❶, поэтому наследовать другие классы от него нельзя. Наследование предполагает специализацию, но специализировать здесь нечего, т.к. все члены являются статическими ❷ кроме закрытого конструктора ❸. На первый взгляд этот конструктор выглядит странным — зачем он вообще нужен, если он закрытый и никогда не будет применяться? Причина в том, что если не предоставить конструктор для класса, то компилятор C# 1 всегда создаст *стандартный конструктор*, который будет открытым и не принимающим параметров. В данном случае любые видимые извне конструкторы не должны существовать, поэтому приходится объявлять такой закрытый конструктор.

Описанный шаблон работает достаточно хорошо, но версия C# 2 делает его явным и активно предотвращает его некорректное использование. Прежде всего, мы рассмотрим изменения, которые понадобится внести в код листинга 7.3 для его превращения в надлежащий статический класс, как определено в C# 2. В листинге 7.4 видно, что для этого требуется совсем немного.

Листинг 7.4. Вспомогательный класс из листинга 7.3, преобразованный в статический класс C# 2

```
using System;
public static class StringHelper
{
  public static string Reverse(string input)
  {
    char[] chars = input.ToCharArray();
    Array.Reverse(chars);
    return new string(chars);
  }
}
```

На этот раз в объявлении класса применяется модификатор `static` вместо `sealed`, а конструктор вообще не включен — это единственное отличие в коде. Компилятору C# 2 известно, что статический класс не должен иметь какие-либо конструкторы, поэтому он не предоставляет стандартный конструктор.

На самом деле компилятор принудительно применяет к определению класса несколько ограничений, которые описаны ниже.

- Он не может быть объявлен как абстрактный (`abstract`) или запечатанный (`sealed`), хотя неявно является тем и другим.
- Невозможно указывать любые интерфейсы для реализации.
- Нельзя указывать базовый тип.
- Невозможно включать любые нестатические члены, в том числе конструкторы.
- Нельзя определять любые операции.
- Невозможно включать любые члены `protected` или `protected internal`.

Полезно отметить, что хотя все члены должны быть статическими, вы обязаны сделать их таковыми *явно*. Несмотря на то что вложенные типы неявно являются статическими членами включающего класса, сам вложенный тип при необходимости может быть нестатическим.

Компилятор не просто помещает ограничения в определение статических классов — он также защищает от некорректного пользования этими классами. Поскольку компилятору известно, что экземпляров статического класса быть не должно, он предотвраща-

ет любой случай применения, при котором они могли бы понадобиться. Например, все следующие операторы недопустимы, когда `StringHelper` — статический класс:

```
StringHelper variable = null;
StringHelper[] array = null;
public void Method1(StringHelper x) {}
public StringHelper Method1() { return null; }
List<StringHelper> x = new List<StringHelper>();
```

Ни один из показанных операторов не удалось бы предотвратить, если бы класс следовал шаблону C# 1, но все они по существу бесполезны. Выражаясь кратко, статические классы в C# 2 не позволяют делать что-то, чего невозможно было осуществлять ранее, а запрещают предпринимать действия, которые *не должны* были делаться в любом случае. Они также явно выражают ваши намерения. Делая класс статическим, вы утверждаете, что однозначно *не хотите*, чтобы создавались любые его экземпляры. Это не просто индивидуальная особенность реализации; это проектное решение.

Следующее средство в списке производит более позитивное впечатление. Оно ориентировано на специфичную, хотя широко встречающуюся ситуацию и позволяет строить решение, которое не выглядит неуклюжим и не нарушает инкапсуляцию, как это было при использовании C# 1.

7.3. Отдельные модификаторы доступа для средств получения/установки свойств

Я должен признаться, что был ошеломлен, когда впервые увидел, что версия C# 1 не позволяет иметь открытое средство получения и закрытое средство установки для свойств. Это не единственная комбинация модификаторов доступа, которая запрещена в C# 1, но она является наиболее востребованной. На самом деле в C# 1 средства получения и установки должны иметь один и тот же уровень доступа — он определяется как часть объявления свойства, а не часть средства получения или установки.

Существуют довольно веские причины хотеть разный уровень доступа для средств получения и установки. При изменении значения переменной, поддерживающей свойство, часто требуется выполнение какой-нибудь проверки достоверности, фиксации действия в журнале, блокировки или другого кода, но делать свойство записываемым в коде, внешнем по отношению к классу, нежелательно. В C# 1 альтернативные решения предусматривали либо нарушение инкапсуляции за счет открытия возможности записи в свойство вопреки здравому смыслу, либо создание в классе метода `SetXXX()` для установки значения свойства, что выглядело откровенно неуклюже применительно к реальным свойствам. В C# 2 проблема решается тем, что средство получения или средство установки может явно иметь более ограниченный уровень доступа, чем объявленный для самого свойства. Легче всего показать это на примере:

```
string name;
public string Name
{
  get { return name; }
  private set
  {
    // Проверка достоверности, фиксация в журнале и т.д.
    name = value;
  }
}
```

В этом случае свойство Name поддерживает только чтение для всех остальных типов[2], но внутри типа, в котором оно определено, можно использовать знакомый синтаксис для установки указанного свойства. Аналогичный синтаксис предусмотрен также для индексаторов. Средство установки *можно было бы* сделать более открытым, чем средство получения (например, защищенное средство получения и открытое средство установки), но это достаточно редкая ситуация, подобно тому, как свойства, допускающие только запись, встречаются намного реже, чем свойства, предназначенные только для чтения.

Мелочи: единственное место, где модификатор `private` обязателен

Во всех других местах кода C# в качестве стандартного модификатора доступа в любой конкретной ситуации выбирается самый закрытый модификатор из числа возможных. Например, если что-то может быть объявлено как закрытое, оно таким и будет по умолчанию, если не указаны какие-либо модификаторы доступа. Это хороший элемент проектного решения, заложенного в язык, т.к. его трудно непредумышленно нарушить; если вы хотите, чтобы что-то было более открытым, чем есть на самом деле, то заметите это во время попытки его применения. Но если вы непреднамеренно сделаете что-то слишком открытым, то компилятор не сможет помочь в выявлении этой проблемы. Указание модификатора доступа для средства получения или установки является исключением из данного правила — если ничего не объявлено, средство получения или установки получит тот же уровень доступа, как у самого свойства, к которому оно относится.

Обратите внимание, что объявить само свойство закрытым и делать открытым его средство получения нельзя — отдельное средство получения или установки может быть только *более* закрытым, чем свойство. Кроме того, модификатор доступа не допускается указывать одновременно для средства получения и средства установки — это выглядело бы нелепым, поскольку может оказаться так, что объявленное свойство будет более открытым, чем оба модификатора в средствах доступа.

Эта помощь в отношении инкапсуляции крайне приветствуется. К сожалению, в C# 2 по-прежнему нет ничего такого, что бы препятствовало обходу свойства остальным кодом класса и работе непосредственно с поддерживающим полем. Как будет показано в следующей главе, в версии C# 3 эта проблема исправлена в одном конкретном случае, но не в общем.

А теперь мы оставим средство, которое может применяться регулярно, и обратимся к средству, которого вы будете стараться избегать в большинстве ситуаций. Оно позволяет коду быть абсолютно явным в отношении ссылок на используемые в нем типы, но за счет значительного снижения читабельности.

7.4. Псевдонимы пространств имен

Пространства имен в первую очередь предназначены служить инструментом организации типов в удобную иерархию. Они *также* позволяют сохранить полностью уточненные имена типов отдельными, даже если имена, которые указаны не полностью, могут совпадать. Это не следует рассматривать в качестве призыва применять не полностью уточненные имена типов без уважительных причин, но временами это единственное, что можно предпринять.

2 Кроме вложенных типов, которые всегда имеют доступ к закрытым членам включающих их типов.

Примером может выступить не полностью уточненное имя Button. В .NET 2.0 Framework есть два класса с таким именем: System.Windows.Forms.Button и System.Web.UI.WebControls.Button. Несмотря на то что оба класса называются Button, их легко отличить друг от друга по пространствам имен. Это близко отражает повседневную жизнь — вы можете быть знакомы с несколькими парнями по имени Джон, но вряд ли знаете еще одного человека, кого зовут Джон Скит. Обсуждая с друзьями определенные вопросы, вы вполне можете использовать просто имя Джон, не уточняя, кого конкретно имеете в виду, однако в других обстоятельствах, вполне вероятно, придется приводить уточняющую информацию.

Директива using в версии C# 1 (не путайте ее с оператором using, автоматически вызывающим метод Dispose()) была доступна в двух формах: одна создает *псевдоним* для пространства имен или типа (например, using Out = System.Console;), а другая вносит пространство имен в список контекстов, в которых компилятор будет искать типы (например, using System.IO;). В общем и целом этого вполне достаточно, но были ситуации, с которыми язык не мог справиться. В ряде других случаев автоматически сгенерированный код должен был обеспечивать применение во всех случаях правильных пространств имен.

В C# 2 эти проблемы решены, и язык получил дополнительную выразительность. Теперь можно записывать код, который гарантированно выражает желаемые намерения независимо от появления других типов, сборок и пространств имен. Эти крайние меры редко требуются вне автоматически сгенерированного кода, но полезно знать, что они существуют.

Версия C# 2 поддерживает три типа псевдонимов: псевдонимы пространств имен C# 1, псевдоним *глобального* пространства имен и *внешние* псевдонимы. Мы начнем с рассмотрения одного вида псевдонима, присутствующего в C# 1, но введем новый способ использования псевдонимов, который позволит компилятору трактовать их как псевдонимы, не проверяя, являются ли они именами других пространств имен или типов.

7.4.1. Уточнение псевдонимов пространств имен

Даже в C# 1 было полезно избегать применения псевдонимов пространств имен, где только возможно. Время от времени вы можете обнаружить, что возникает конфликт имен нескольких типов, как в приведенном ранее примере с Button. В такой ситуации необходимо либо каждый раз указывать полное имя, включающее пространство имен, либо вводить псевдоним для различения имен типов, который в какой-то мере будет действовать подобно сокращенной форме записи пространства имени. В листинге 7.5 показан пример, в котором используются два типа Button, уточненные с помощью псевдонима.

Листинг 7.5. Применение псевдонимов для различения разных типов Button

```
using System;
using WinForms = System.Windows.Forms;
using WebForms = System.Web.UI.WebControls;
class Test
{
   static void Main()
   {
      Console.WriteLine(typeof(WinForms.Button));
      Console.WriteLine(typeof(WebForms.Button));
   }
}
```

Код из листинга 7.5 компилируется без ошибок или предупреждений, хотя все еще не выглядит настолько симпатичным, как мог бы в случае работы только с одним типом Button. Тем не менее, существует проблема — что если кто-то определит тип или пространство имен под названием WinForms или WebForms? Компилятору не известно предназначение типа WinForms.Button и чему отдать предпочтение — типу или пространству имен либо псевдониму. Необходимо иметь возможность сообщить компилятору, что он должен трактовать WinForms как псевдоним, несмотря на то, что это имя доступно где-то в другом месте.

Для этой цели в C# 2 был введен *синтаксис уточнителя пространства имен* `::`, использование которого демонстрируется в листинге 7.6.

Листинг 7.6. Применение `::` для сообщения компилятору о том, что он должен использовать псевдонимы

```
using System;
using WinForms = System.Windows.Forms;
using WebForms = System.Web.UI.WebControls;

class WinForms {}

class Test
{
  static void Main()
  {
    Console.WriteLine(typeof(WinForms::Button));
    Console.WriteLine(typeof(WebForms::Button));
  }
}
```

Вместо WinForms.Button в листинге 7.6 используется WinForms::Button, что совершенно устраивает компилятор. Если изменить :: обратно на ., будет получена ошибка компиляции.

Итак, если применять :: везде, где используется псевдоним, то все будет нормально? Не совсем...

7.4.2. Псевдоним глобального пространства имен

Существует одна часть иерархии пространств имен, для которой невозможно определить собственный псевдоним — это корень иерархии, или *глобальное* пространство имен. Предположим, что имеется два класса, имеющие одинаковые имена Configuration — один класс находится внутри пространства имен MyCompany, а для другого пространство имен вообще не указано. Как сослаться на корневой класс Configuration изнутри пространства имен MyCompany? Обычный псевдоним использовать не удастся, но если просто указать Configuration, то компилятор выберет MyCompany.Configuration.

В C# 1 не было никаких способов обойти это. И снова на помощь приходит версия C# 2, позволяя применять конструкцию global::Configuration, чтобы сообщить компилятору о желаемом поведении. В листинге 7.7 демонстрируется проблема и ее решение.

Листинг 7.7. Использование псевдонима глобального пространства имен для точного указания желательного типа

```
using System;

class Configuration {}

namespace Chapter7
{
   class Configuration {}

   class Test
   {
      static void Main()
      {
         Console.WriteLine(typeof(Configuration));
         Console.WriteLine(typeof(global::Configuration));
         Console.WriteLine(typeof(global::Chapter7.Test));
      }
   }
}
```

Большая часть кода в листинге 7.7 просто воспроизводит ситуацию, а особый интерес представляют три строки в методе `Main()`. Первая строка выводит на консоль `Chapter7.Configuration`, поскольку компилятор распознает `Configuration` как этот тип, прежде чем перейти к корню иерархии пространств имен. Вторая строка отражает тот факт, что тип должен находиться в глобальном пространстве имен, поэтому она просто выводит на консоль `Configuration`. Третья строка была включена для демонстрации того, что с применением псевдонима глобального пространства имен можно по-прежнему ссылаться на типы внутри пространств имен, но вы должны указывать полностью уточненное имя.

В данный момент можно достичь любого уникально именованного типа, при необходимости используя глобальное пространство имен. Если вам когда-либо придется строить генератор, порождающий код, который не обязательно должен быть читабельным, можете обильно пользоваться этим средством, чтобы обеспечить ссылку на корректный тип вне зависимости от существования во время компиляции любых других типов. Но что делать, если имя типа не является уникальным даже после указания его пространства имен? Дело принимает другой оборот...

7.4.3. Внешние псевдонимы

В начале этого раздела в качестве примеров пространств имен и контекстов я применял имена людей, при этом, в частности, подчеркивая, что вряд ли вы знаете более одного человека, которого зовут Джон Скит. Но мне известно, что мое имя и фамилию ношу не только я, и вполне есть шанс, что вы знаете двоих и более таких людей. В этом случае для того, чтобы указать, кто конкретно имеется в виду, нужно предоставить дополнительные сведения помимо имени и фамилии — причина, по которой вы знаете конкретного человека, страну его проживания или еще какую-то отличительную особенность.

Версия C# 2 позволяет указывать такую дополнительную информацию в форме *внешнего псевдонима* — имени, которое существует не только в исходном коде, но также и в параметрах, передаваемых компилятору. В случае компилятора Microsoft C# это оз-

начает указание сборки, которая содержит интересующие типы. Предположим, что две сборки — `First.dll` и `Second.dll` — содержат тип по имени `Demo.Example`. Для их различения использовать полностью уточненные имена не получится, т.к. имена типов совпадают. Вместо этого для определения намерений можно применять внешние псевдонимы. В листинге 7.8 приведен пример соответствующего кода C# наряду с командной строкой, необходимой для его компиляции.

Листинг 7.8. Работа с разными типами с одним и тем же именем, находящимися в разных сборках

```
// Командная строка для компиляции выглядит следующим образом:
// csc Test.cs /r:FirstAlias=First.dll /r:SecondAlias=Second.dll
extern alias FirstAlias;                                    ← ❶ Определение двух внешних псевдонимов
extern alias SecondAlias;

using System;
using FD = FirstAlias::Demo;                                ← ❷ Ссылка на внешний псевдоним с помощью псевдонима пространства имен

class Test
{
  static void Main()
  {
    Console.WriteLine(typeof(FD.Example));                  ← ❸ Использование псевдонима пространства имен

    Console.WriteLine(typeof(SecondAlias::Demo.Example));   ← ❹ Использование внешнего псевдонима напрямую


  }
}
```

Код в листинге 7.8 довольно прямолинеен. Сначала вводятся два внешних псевдонима ❶. После этого их можно использовать либо через псевдонимы пространств имен (❷ и ❸), либо напрямую ❹. На самом деле обычная директива `using` без псевдонима (такая как `using FirstAlias::Demo;`) позволила бы применять имя `Example`, вообще никак не уточняя его. Один внешний псевдоним способен покрывать несколько сборок, а множество внешних псевдонимов могут ссылаться на одну и ту же сборку, хотя я хорошо бы подумал, прежде чем воспользоваться любой из этих возможностей, а в особенности их комбинацией.

Чтобы указать внешний псевдоним в среде Visual Studio, просто выберите ссылку на сборку в окне **Solution Explorer** и измените значение в поле **Aliases** (Псевдонимы) окна **Properties** (Свойства), как показано на рис. 7.3.

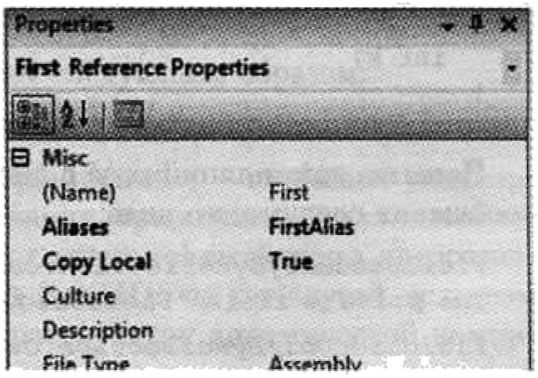

Рис. 7.3. Часть окна Properties среды Visual Studio 2010, отображающая внешний псевдоним `FirstAlias` для ссылки на сборку `First.dll`

Надеюсь, мне не придется убеждать вас в том, что ситуаций подобного рода следует избегать всегда, когда только возможно. Это может понадобиться при взаимодействии со сборками от независимых разработчиков, в которых случайно применяются одинаковые полностью уточненные

имена типов, когда иной способ их использования отсутствует. Однако при наличии большего контроля над именованием удостоверьтесь, что выбираемые имена никогда не приведут к возникновению описанных выше ситуаций.

Возможность, рассматриваемая следующей, по большому счету является метасредством. Точная предлагаемая им функциональность зависит от применяемого компилятора, поскольку его назначение — управлять возможностями, специфичными для компилятора. Мы будет уделять внимание компилятору от Microsoft.

7.5. Директивы pragma

Описать *директивы* `pragma` в целом исключительно легко: директива `pragma` – это директива препроцессора, представляемая строкой, которая начинается с `#pragma`. Остальная часть строки может содержать любой текст. Результат директивы `pragma` не может изменить поведение программы так, что оно будет идти вразрез со спецификацией языка C#, но позволяет делать что-нибудь за пределами области действия спецификации. Если компилятор не может распознать отдельную директиву `pragma`, он выдает сообщение с предупреждением, а не ошибкой.

Вот в принципе и все, что спецификация указывает по данной теме. Компилятор Microsoft C# воспринимает две директивы `pragma`: `#pragma warning` и `#pragma checksum`.

7.5.1. Директивы #pragma warning

Иногда компилятор C# выдает предупреждения, которые вполне оправданы, но надоедливы. Правильная реакция на предупреждение компилятора *почти всегда* предполагает исправление кода — после устранения причины предупреждения код редко становится хуже, а обычно только улучшается.

Однако временами существует веская причина, чтобы игнорировать то или иное предупреждение, и для таких ситуаций предусмотрены директивы `#pragma warning`. В качестве примера будет создано закрытое поле, которое никогда не читается и не устанавливается. Оно практически всегда оказывается бесполезным, если только не учитывать, что оно будет использоваться рефлексией. В листинге 7.9 показан полный класс, демонстрирующий сказанное.

Листинг 7.9. Класс, содержащий неиспользуемое поле

```
public class FieldUsedOnlyByReflection
{
  int x;
}
```

Попытка компиляции кода в листинге 7.9 приводит к получению предупреждающего сообщения следующего вида:

```
FieldUsedOnlyByReflection.cs(3,9): warning CS0169:
The private field 'FieldUsedOnlyByReflection.x' is never used
```

`FieldUsedOnlyByReflection.cs(3,9): предупреждение CS0169:`
`Закрытое поле FieldUsedOnlyByReflection.x никогда не используется`

Так выглядит вывод компилятора командной строки. В окне Error List (Список ошибок) среды Visual Studio можно видеть ту же самую информацию (плюс название проекта, в котором находится класс) *за исключением* номера предупреждения (CS0169).

Чтобы найти этот номер, понадобится либо выбрать предупреждение и просмотреть справку по нему, либо заглянуть в окно Output (Вывод), где отображается полный текст. Номер предупреждения необходим для того, чтобы обеспечить компиляцию кода без выдачи такого предупреждения, что и сделано в листинге 7.10.

Листинг 7.10. Отключение (и восстановление) выдачи предупреждения CS0169

```
public class FieldUsedOnlyByReflection
{
#pragma warning disable 0169
  int x;
#pragma warning restore 0169
}
```

Код в листинге 7.10 особых пояснений не требует — первая директива `pragma` отключает выдачу указанного предупреждения, а вторая восстанавливает ее. Установившаяся практика предполагает отключение выдачи предупреждений на как можно более короткий период, чтобы не пропустить те из них, которые действительно должны быть учтены путем исправления кода. Если нужно отключить или восстановить выдачу нескольких предупреждений в одной строке, укажите желаемые номера предупреждений в виде списка с разделителями-запятыми. Если номера предупреждений вообще не указаны, в результате отключается или восстанавливается выдача *всех* предупреждений, но в почти любом вообразимом сценарии это совершенно неприемлемо.

7.5.2. Директивы #pragma checksum

Вторая форма директивы `pragma`, распознаваемая компилятором Microsoft, вряд ли понадобится. Она поддерживает отладчик, позволяя ему проверять, правильный ли файл исходного кода был найден. Обычно, когда компилируется файл с кодом C#, компилятор генерирует контрольную сумму для этого файла и включает ее в состав отладочной информации. Когда отладчику необходимо отыскать исходный файл и найти потенциальные соответствия, он самостоятельно генерирует контрольную сумму для каждого файла-кандидата и проверяет, какая из них является корректной.

После преобразования страницы ASP.NET в код C# сгенерированный файл — это то, что видит компилятор C#. Генератор вычисляет контрольную сумму для страницы `.aspx` и применяет директиву `#pragma checksum`, чтобы сообщить компилятору C# о том, что он должен использовать *данную* контрольную сумму вместо вычисления новой на основе сгенерированной страницы.

Синтаксис директиву `#pragma checksum` выглядит следующим образом:

```
#pragma checksum "имя файла" "{идентификатор GUID}" "байты контрольной суммы"
```

С помощью идентификатора GUID (Globally Unique Identifier — глобально уникальный идентификатор) указывается алгоритм хеширования, применяемый для вычисления контрольной суммы. В документации по классу `CodeChecksumPragma` предоставляются идентификаторы GUID для алгоритмов SHA-1 и MD5 на тот случай, если вам когда-либо захочется реализовать собственную инфраструктуру динамической компиляции с поддержкой отладчика.

Вполне вероятно, что будущие версии компилятора C# будут включать большее число директив `pragma`, а другие компиляторы (наподобие Mono, `mcs`) получат собственную поддержку для различных средств. За актуальными сведениями обращайтесь в документацию по компилятору.

Следующее средство также является одним из тех, которые редко приходится применять, но если дело до этого все же доходит, оно позволяет кое в чем упростить жизнь.

7.6. Буферы фиксированного размера в небезопасном коде

При обращении к машинному коду с помощью P/Invoke не так уж необычно иметь дело со структурой, внутри которой определен буфер заданной длины. До появления C# 2 такие структуры было трудно обрабатывать напрямую, даже с помощью небезопасного кода. Теперь можно объявлять буфер необходимого размера для встраивания прямо в оставшиеся данные внутри структуры.

Указанная возможность доступна не только для вызова машинного кода, хотя это ее основное использование. Ее можно было бы применять, например, для непосредственного наполнения структуры данных в соответствие с форматом файла. Синтаксис прост и в очередной раз будет продемонстрирован на примере. Для создания поля, которое содержит массив из 20 байтов, внутри заключающей структуры используется такой код:

```
fixed byte data[20];
```

Это позволило бы работать с полем `data`, как если бы оно имело тип `byte*` (указатель на байтовые данные), хотя реализация, используемая компилятором C#, предусматривает создание в рамках объявляющего типа нового вложенного типа и применение к самой переменной нового атрибута `FixedBuffer`. За соответствующее выделение памяти позаботится среда CLR.

Недостаток описанной возможности состоит в том, что она доступна только в небезопасном коде: включающая структура должна быть объявлена в небезопасном контексте, и член, который представляет буфер фиксированного размера, должен использоваться в небезопасном контексте. Это ограничивает набор ситуаций, в которых возможность оказывается удобной, но ее все же стоит иметь в своем арсенале. Кроме того, буферы фиксированного размера применимы только к элементарным типам и не могут быть членами классов (а лишь структур).

На удивление данная возможность полезна только для очень небольшого количества API-интерфейсов Windows. Многочисленные ситуации вызывают необходимость в использовании фиксированного массива символов — например, структура `TIME_ZONE_INFORMATION`, — но, к сожалению, буферы символов фиксированного размера плохо работают с P/Invoke, препятствуя маршализации.

Тем не менее, в листинге 7.11 приведен один пример — консольное приложение, которое отображает цвета, доступные в текущем окне консоли. В нем используется функция API-интерфейса `GetConsoleScreenBufferEx()`, введенная в Windows Vista и Windows Server 2008, которая извлекает расширенную информацию о консоли. Код в листинге 7.11 отображает все 16 цветов в шестнадцатеричном формате (`bbggrr`).

Листинг 7.11. Демонстрация применения буферов фиксированного размера для получения информации о цветах консоли

```
using System;
using System.Runtime.InteropServices;
struct COORD
{
  public short X, Y;
}
```

```
struct SMALL_RECT
{
  public short Left, Top, Right, Bottom;
}
unsafe struct CONSOLE_SCREEN_BUFFER_INFOEX
{
  public int StructureSize;
  public COORD ConsoleSize, CursorPosition;
  public short Attributes;
  public SMALL_RECT DisplayWindow;
  public COORD MaximumWindowSize;
  public short PopupAttributes;
  public int FullScreenSupported;
  public fixed int ColorTable[16];
}
static class FixedSizeBufferDemo
{
  const int StdOutputHandle = -11;

  [DllImport("kernel32.dll")]
  static extern IntPtr GetStdHandle(int nStdHandle);

  [DllImport("kernel32.dll")]
  static extern bool GetConsoleScreenBufferInfoEx
    (IntPtr handle, ref CONSOLE_SCREEN_BUFFER_INFOEX info);

  unsafe static void Main()
  {
    IntPtr handle = GetStdHandle(StdOutputHandle);
    CONSOLE_SCREEN_BUFFER_INFOEX info;
    info = new CONSOLE_SCREEN_BUFFER_INFOEX();
    info.StructureSize = sizeof(CONSOLE_SCREEN_BUFFER_INFOEX);
    GetConsoleScreenBufferInfoEx(handle, ref info);

    for (int i=0; i < 16; i++)
    {
      Console.WriteLine ("{0:x6}", info.ColorTable[i]);
    }
  }
}
```

Буферы фиксированного размера используются в листинге 7.11 для представления таблицы цветов. До появления буферов фиксированного размера функцию API-интерфейса можно было либо применять к каждой записи в таблице цветов, либо за счет маршализации нормального массива как типа `UnmanagedType.ByValArray`. Но это привело бы к созданию отдельного массива в куче вместо сохранения информации полностью внутри структуры. В рассматриваемом примере это не проблема, но в ряде ситуаций, когда требуется высокая производительность, было бы лучше иметь возможность удерживать порции данных вместе. С другой стороны, касающейся производительности, если буфер является частью структуры данных в управляемой куче, перед доступом он должен быть закреплен. Частое закрепление может оказать значительное воздействие на сборщик мусора. Разумеется, структуры, хранящиеся в стеке, лишены этой проблемы.

Я не утверждаю, что буферы фиксированного размера являются чрезвычайно важным средством в C# 2 — во всяком случае, не для большинства разработчиков. Здесь они включены для полноты картины, и вне всяких сомнений в некоторых случаях они окажутся просто бесценными.

Последнюю рассматриваемую возможность можно назвать полностью *языковым* средством C# 2, но это *просто* ради учета.

7.7. Открытие внутренних членов для избранных сборок

Некоторые средства являются явно языковыми — например, итераторные блоки. Другие средства очевидным образом принадлежат исполняющей среде, скажем, набор оптимизаций, выполняемых компилятором JIT. Есть средства, которые явно относятся к указанным двум сторонам, такие как обобщения. Возможность, которая будет описана в данном разделе, относится к обеим сторонам, поэтому довольно странно выглядит тот факт, что она не заслужила упоминания хотя бы в *одной из* спецификаций. Вдобавок здесь используется термин, который имеет другой смысл в языках C++ и VB.NET, привнося третье значение в эту смесь. Справедливости ради стоит отметить, что все термины применяются в контексте прав доступа, но приводят к разным результатам.

7.7.1. Дружественные сборки в простом случае

В .NET 1.1 можно было совершенно точно сказать, что сущность, которая определена как *внутренняя* (будь это тип, метод, свойство, переменная или событие), допускала возможность доступа только внутри сборки, где она объявлена[3]. В .NET 2.0 это по-прежнему *главным образом* справедливо, но появился новый атрибут, позволяющий слегка нарушить правила — `InternalsVisibleToAttribute`, на который обычно ссылаются как на просто `InternalsVisibleTo`. (В случае применения атрибута, имя которого заканчивается на суффикс `Attribute`, компилятор C# добавляет этот суффикс автоматически.)

Атрибут `InternalsVisibleTo` может быть применен только к сборке (не к конкретному члену), причем допускается его многократное использование для одной и той же сборки. Сборку, содержащую этот атрибут, я буду называть *исходной сборкой*, хотя это и неофициальная терминология. В случае применения атрибута `InternalsVisibleTo` должна указываться другая сборка, которая известна как *дружественная сборка*. В результате дружественная сборка может видеть все внутренние члены исходной сборки, как если бы они были открытыми. Этот факт может выглядеть как тревожный знак, но, как вскоре будет показано, он является удобным.

В листинге 7.12 представлена демонстрация сказанного на примере трех классов в трех разных сборках.

Листинг 7.12. Демонстрация дружественных сборок

```
// Компилируется в Source.dll
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("FriendAssembly")]
```

← Предоставление дополнительного доступа

[3] Использование рефлексии во время выполнения с подходящими правами доступа в расчет не принимается.

```
public class Source
{
  internal static void InternalMethod() {}
  public static void PublicMethod() {}
}

// Компилируется в FriendAssembly.dll
public class Friend
{
  static void Main()
  {
    Source.InternalMethod();  <-- Использование дополнительного доступа
                                  внутри сборки FriendAssembly
    Source.PublicMethod();
  }
}

// Компилируется в EnemyAssembly.dll
public class Enemy
{
  static void Main()
  {
    // Source.InternalMethod(); <-- ❶ Сборка EnemyAssembly не имеет специального доступа
    Source.PublicMethod();      <-- Доступ к открытому методу обычным образом
  }
}
```

В листинге 7.12 существует специальное отношение между сборками FriendAssembly.dll и Source.dll, несмотря на то, что оно установлено только в одном направлении: сборка Source.dll не имеет доступа к внутренним членам сборки FriendAssembly.dll. Если удалить символы комментария в строке ❶, класс Enemy не скомпилируется.

Для начала, с какой стати может понадобиться открывать свою успешно спроектированную сборку каким-то другим сборкам?

7.7.2. Причины использования атрибута InternalsVisibleTo

Сам я редко использую атрибут InternalsVisibleTo между двумя производственными сборками. Я понимаю, что он может быть удобным, и определенно применяю его для получения дополнительного доступа при написании инструментов, но главным случаем использования этого атрибута всегда было модульное тестирование.

Некоторые утверждают, что в коде должен тестироваться только открытый интерфейс. Лично я предпочитаю тестировать все, что только можно, в самой простой форме. Дружественные сборки существенно упрощают это: неожиданно легко тестировать код, который имеет только внутренний доступ, не предпринимая неоднозначные действия по превращению членов в открытые только ради тестирования или включения тестового кода в производственную сборку. (Иногда это означает, что в целях тестирования члены должны быть сделаны внутренними, в то время как иначе они могли бы быть закрытыми, но такая ситуация вызывает меньше беспокойства.)

Единственным недостатком приема является необходимость в указании имени тестовой сборки внутри производственной сборки. Теоретически это могло бы послужить объектом для атаки на безопасность, если сборки не подписаны, а код обычно не фун-

кционирует в условиях ограниченного набора прав доступа. (В первую очередь кто-то с полным доверием мог бы с помощью рефлексии получить доступ к членам внутри сборки. Понятно, что вы сами делаете это при модульном тестировании, но вмешательство постороннего гораздо хуже.) Если описанная проблема когда-либо действительно возникнет у кого-то, я буду очень удивлен. Но это привносит в общую картину возможность подписания сборок. Так что не следует думать, что дружественные сборки — всего лишь удобное и простое средство...

7.7.3. Атрибут `InternalsVisibleTo` и подписанные сборки

Если дружественная сборка подписана, в исходной сборке должен быть указан открытый ключ дружественной сборки, чтобы обеспечить доверие к правильному коду. Для этого нужен полный открытый ключ, а не просто маркер открытого ключа.

Например, предположим, что для выяснения открытого ключа подписанной сборки `FriendAssembly.dll` применяется следующая командная строка, которая дает показанный ниже вывод (слегка модифицированный и отформатированный согласно требованиям печатной страницы):

```
c:\Users\Jon\Test>sn -Tp FriendAssembly.dll
Microsoft (R) .NET Framework Strong Name Utility Version 3.5.21022.8
Copyright (c) Microsoft Corporation. All rights reserved.
Public key is
00240000048000009400000006020000002400005253413100040000001
000100a51372c81ccfb8fba9c5fb84180c4129e50f0facdce932cf31fe
563d0fe3cb6b1d5129e28326060a3a539f287aaf59affc5aabc4d8f981
e1a82479ab795f410eab22e3266033c633400463ee7513378bb4ef41fc
0cae5fb03986d133677c82a865b278c48d99dc251201b9c43edd7bedef
d4b5306efd0dec7787ec6b664471c2

Public key token is 647b99330b7f792c
```

Тогда исходный код класса `Source` должен был бы содержать такой атрибут:

```
[assembly:InternalsVisibleTo("FriendAssembly,PublicKey=" +
"00240000048000009400000006020000002400005253413100040000001" +
"000100a51372c81ccfb8fba9c5fb84180c4129e50f0facdce932cf31fe" +
"563d0fe3cb6b1d5129e28326060a3a539f287aaf59affc5aabc4d8f981" +
"e1a82479ab795f410eab22e3266033c633400463ee7513378bb4ef41fc" +
"0cae5fb03986d133677c82a865b278c48d99dc251201b9c43edd7bedef" +
"d4b5306efd0dec7787ec6b664471c2")]
```

К сожалению, необходимо либо указывать открытый ключ в одной строке, либо использовать конкатенацию строк — пробельные символы в открытом ключе вызовут ошибку компиляции. Было бы намного лучше, если бы вместо полного ключа допускалось указывать только маркер, но к счастью это уродство обычно ограничивается файлом `AssemblyInfo.cs`, так что часто видеть его не придется.

Теоретически возможно иметь неподписанную исходную сборку и подписанную дружественную сборку. На практике это не особенно полезно, т.к. дружественной сборке обычно нужна ссылка на исходную сборку, а ссылаться на неподписанную сборку из подписанной не разрешено. Аналогично, в подписанной сборке нельзя указывать неподписанную дружественную сборку, поэтому обычно если одна из сборок подписана, приходится подписывать и другую.

7.8. Резюме

На этом обзор новых возможностей C# 2 завершен. Темы, рассмотренные в главе, разделены на две широких категории: улучшения в стиле "хорошо иметь", которые упрощают разработку, и средства с характеристикой "надеюсь, что не понадобится", которые могут вывести из запутанных ситуаций, когда в них возникает потребность. Проводя аналогию между C# 2 и улучшениями в доме, следует отметить, что крупные средства, описанные в предшествующих главах, сравнимы с полномасштабной достройкой. С другой стороны, некоторые средства, упомянутые в настоящей главе (такие как частичные типы и статические классы), больше похожи на косметический ремонт спальни, а средства вроде псевдонимов пространств имен подобны установке датчиков дыма — вы можете никогда и не извлечь от них пользы, но гораздо спокойнее знать, что они есть, на тот случай, если они вдруг понадобятся.

Диапазон возможностей C# 2 довольно широк — проектировщики позаботились о многих областях, где разработчики испытывали трудности, не ставя перед собой каких-то всеобъемлющих целей. Это не значит, что указанные средства не функционируют слаженно вместе (к примеру, типы, допускающие `null`, не были бы возможны без обобщений), просто отсутствует общая цель, в пользу которой бы работало каждое средство, если не принимать во внимание общую продуктивность.

Теперь, когда мы закончили исследование C# 2, наступило время переходить к версии C# 3, где ситуация существенно отличается. В C# 3 почти каждая возможность направлена на формирование части грандиозной картины языка LINQ, представляющего собой конгломерат технологий, которые массово упрощают решение многих задач.

новшества в LINQ; в главе 12 будут описаны средства Parallel LINQ и Reactive Extensions, которые представляют собой два совершенно разных способа обработки данных, предусматривающие использование знакомых операций LINQ. А сверх того есть еще LINQ to Objects — простейший, самый предсказуемый и обыкновенный поставщик LINQ, который наиболее распространен в этой отрасли. Дни, когда приходилось писать очередной цикл для фильтрации, еще один фрагмент кода для поиска максимального значения, дополнительную проверку для выяснения, удовлетворяют ли элементы коллекции определенному условию, ушли — и скатертью им дорога.

Несмотря на широкое применение LINQ, я по-прежнему вижу ряд вопросов, свидетельствующих о том, что некоторые разработчики воспринимают LINQ как своего рода волшебный черный ящик. Что происходит, когда используется выражение запроса в сравнении с применением расширяющих методов напрямую? Когда данные на самом деле читаются? Каким образом обеспечить более эффективную работу? Хотя язык LINQ можно неплохо изучить, просто применяя его и прорабатывая готовые примеры, вы получите намного больше знаний, если ознакомитесь с тем, как все функционирует на языковом уровне, а также с тем, что предлагают многочисленные библиотеки.

Эта книга не посвящена LINQ — внимание будет сосредоточено на языковых средствах, которые делают LINQ возможным, а не на соглашениях о параллельном выполнении для инфраструктуры Entity Framework и тому подобном. Но после того, как вы освоите элементы языка по отдельности и узнаете, как они сочетаются друг с другом, вы будете гораздо лучше готовы к исследованию деталей конкретных поставщиков.

Часть III

C# 3: революционные изменения в доступе к данным

Значительное улучшение C# 2 по сравнению с C# 1 не вызывает никаких сомнений. В частности, обобщения являются фундаментом для других изменений, причем не только в C# 2, но также и в C# 3. Однако в некотором смысле версия C# 2 представляет собой разрозненный набор средств. Не поймите меня превратно: они довольно хорошо подогнаны друг к другу, но решают набор отдельных проблем. Это было уместным на той стадии развития языка C#, но в версии C# 3 ситуация иная.

Почти каждое средство в C# 3 направлено на то, чтобы сделать возможной одну конкретную технологию: LINQ. Многие средства полезны также за рамками этого контекста, и вы определенно не должны ограничивать себя использованием их только для написания, скажем, выражений запросов. С другой стороны, было бы не менее глупо не признавать полную картину происходящего на основе тех фрагментов головоломки, которые представлены в следующих пяти главах.

Когда я впервые писал о версии C# 3 и LINQ в 2007 году, я был крайне впечатлен довольно высоким академическим уровнем изменений. Чем более глубоко вы будете изучать язык, тем более четко сможете видеть гармонию между различными элементами, которые были введены. Элегантность выражений запросов и особенно возможность применять одинаковый синтаксис для внутрипроцессных запросов и поставщиков, подобных LINQ to SQL, была очень привлекательной. Язык LINQ обещал многое.

Теперь, спустя несколько лет, я могу снова возвратиться к обещаниям и посмотреть, какую роль они сыграли. Мой собственный опыт и сообщество, в частности Stack Overflow, позволяют утверждать, что язык LINQ совершенно очевидно был широко принят разработчиками и действительно изменил подход к решению многих задач, связанных с данными. Поставщики баз данных не ограничиваются только теми, которые предлагает Microsoft; достаточно назвать лишь два других доступных варианта — LINQ to NHibernate и LINQ to SubSonic. В Microsoft также не прекратили вводить

Глава 8

Отбрасывание мелочей с помощью интеллектуального компилятора

В этой главе...

- Автоматически реализуемые свойства
- Неявно типизированные локальные переменные
- Инициализаторы объектов и коллекций
- Неявно типизированные массивы
- Анонимные типы

Мы начнем рассмотрение C# 3 таким же способом, каким завершили анализ C# 2 — с набора относительно простых средств. Они являются лишь небольшими шагами в направлении LINQ. Каждое средство может использоваться за рамками этого контекста, но почти все они важны для упрощения кода до такой степени, которая требуется для эффективного применения LINQ.

Важно отметить, что хотя два крупнейших средства C# 2 — обобщения и типы, допускающие `null`, — требовали внесения изменений в среду CLR, в версии .NET 3.5 никаких значительных изменений не появилось. Конечно, некоторые корректировки были предприняты, но ничего фундаментального. Инфраструктура была расширена для поддержки LINQ, а в библиотеку базовых классов было добавлено несколько новых функциональных возможностей, но это совсем другое дело. Полезно четко понимать, какие изменения внесены только в *язык* C#, какие — в библиотеку, а какие — в среду CLR.

Практически все новые средства, доступные в C# 3, обусловлены желанием возложить на компилятор больший объем работы. Определенное доказательство этого было

представлено во второй части книги — особенно в форме анонимных методов и итераторных блоков — и в C# 3 такая тенденция продолжается. В настоящей главе вы ознакомитесь со следующими средствами, которые являются нововведениями версии C# 3.

- Автоматически реализуемые свойства. Избавляют от монотонной работы по написанию простых свойств, поддерживаемых напрямую полями.
- Неявно типизированные локальные переменные. Устраняют избыточность из объявлений локальных переменных за счет выведения типов переменных из их начальных значений.
- Инициализаторы объектов и коллекций. Упрощают создание и инициализацию объектов в одиночных выражениях.
- Неявно типизированные массивы. Устраняют избыточность из выражений, создающих массивы, за счет выведения типов массивов на основе их содержимого.
- Анонимные типы. Позволяют создавать разовые типы, содержащие простые свойства.

В дополнение к описанию, что каждое средство делает, будут даны рекомендации относительно его использования. Многие возможности C# 3 требуют определенной осмотрительности и сдержанности со стороны разработчика. Это не говорит о том, что они не являются мощными или полезными — как раз наоборот, — но искушение воспользоваться новейшим великолепным синтаксисом не должно брать верх над стремлением писать ясный и читабельный код.

Соображения, высказываемые в этой главе (в остальных главах книги), редко будут черно-белыми. Возможно, больше, чем когда-либо прежде, читабельность определяется личными предпочтениями, и по мере освоения новых средств код наверняка станет более понятным при чтении. Однако я должен подчеркнуть, что если нет оснований полагать, что написанный вами код будет читаться только вами, то придется принимать во внимание потребности и взгляды ваших коллег.

Итак, оставим долгие раздумывания и начнем со средства, которое не должно вызывать какие-либо споры. Простые, но эффективные автоматически реализуемые свойства всего лишь улучшают жизнь.

8.1. Автоматически реализуемые свойства

Первое обсуждаемое нами средство является, вероятно, наиболее простым во всей версии C# 3. Оно даже проще любого нового средства в C# 2. Несмотря на это — или, возможно, *благодаря* этому — оно непосредственно задействовано в очень многих ситуациях. Когда вы читали об итераторных блоках в главе 6, то могли не сразу осмыслить все области в существующем коде, которые удалось бы улучшить с их применением. Тем не менее, я был бы удивлен, если бы нашлась какая-нибудь нетривиальная программа на C# 2, которую нельзя было бы модифицировать для использования автоматически реализуемых свойств. Это невероятно простое средство позволяет выражать тривиальные свойства посредством меньшего объема кода, нежели ранее.

Что я понимаю под *тривиальным свойством*? Я подразумеваю свойство, которое читается/устанавливается и хранит свое значение в простой локальной переменной, не выполняя код проверки достоверности или любой другой специальный код. Тривиальные свойства занимают только несколько строк кода, но все равно это много для выражения такой простейшей концепции. В C# 3 это количество строк кода сокращается за счет применения простой трансформации во время компиляции, как показано на рис. 8.1.

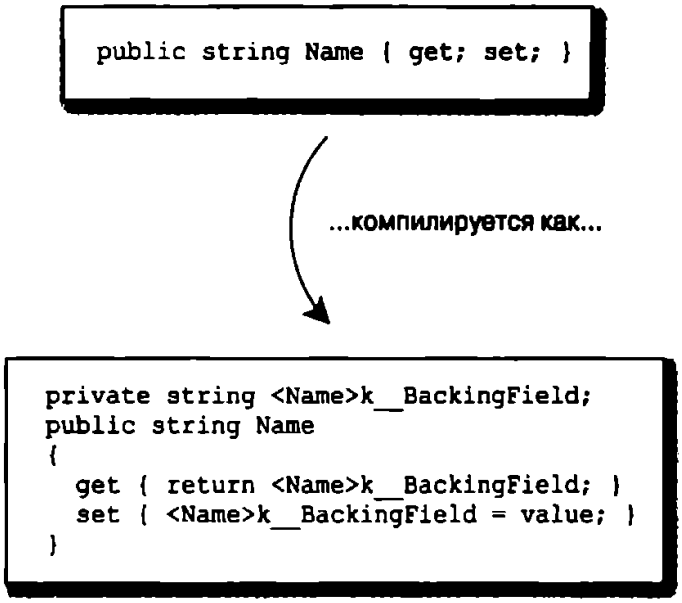

Рис. 8.1. Трансформация автоматически реализуемого свойства

Разумеется, код в нижней части рис. 8.1 не является *полностью* допустимым кодом C#. Поле получает непроизносимое имя, чтобы предотвратить конфликт имен, как это делалось ранее в случае анонимных методов и итераторных блоков. Однако так выглядит действительный код, который генерируется автоматически реализуемым свойством в верхней части рис. 8.1.

Там, где раньше ради простоты мог возникать соблазн воспользоваться открытой переменной, теперь еще меньше оправданий для отказа от применения вместо нее свойства. Это особенно верно для "временного" кода, который, как все мы знаем, существует гораздо дольше, чем предсказывалось.

Терминология: автоматическое свойство или автоматически реализуемое свойство?

Когда автоматически реализуемые свойства впервые начали обсуждаться, задолго до опубликования полной спецификации C# 3, они назывались *автоматическими свойствами*. Лично я нахожу это менее точным, чем полное название, но в сообществе оно используется более широко. Здесь нет никакого риска в плане неоднозначности, поэтому в оставшемся тексте я буду применять термины *автоматическое свойство* и *автоматически реализуемое свойство* попеременно, считая их синонимами.

По-прежнему поддерживается возможность C# 2, позволяющая указывать разный уровень доступа для средств получения и установки свойств, к тому же можно также создавать статические автоматические свойства. Но статические автоматические свойства почти всегда бессмысленны. Хотя большинство типов не требуют наличия членов экземпляра, безопасных в отношении потоков, открыто видимые статические члены обычно *должны* быть безопасными к потокам, и компилятор не делает ничего такого, что помогло бы в этом аспекте. В листинге 8.1 представлен пример безопасного, но бесполезного статического автоматического свойства, которое подсчитывает количество созданных экземпляров класса, наряду со свойствами экземпляра, хранящими имя и возраст человека.

Листинг 8.1. Неуклюжий подсчет количества экземпляров с помощью статического автоматического свойства

```
public class Person
{
  public string Name { get; private set; }

  public int Age { get; private set; }
  private static int InstanceCounter { get; set; }

  private static readonly object counterLock = new object();
  public InstanceCountingPerson(string name, int age)
  {
     Name = name;
     Age = age;
     lock (counterLock)
     {
        InstanceCounter++;
     }
  }
}
```


Блокировка в листинге 8.1 позволяет обеспечить отсутствие проблем, связанных с потоками, и одна и та же блокировка должна использоваться при каждом доступе к свойству. Существуют более удачные альтернативы, предусматривающие применение класса `Interlocked`, однако они требуют доступа к полям. Короче говоря, единственный сценарий, при котором я могу счесть статические автоматические свойства полезными, выглядит следующим образом: средство получения является открытым, средство установки — закрытым, и это средство установки вызывается *только* внутри инициализатора типа.

Ситуация с другими свойствами в листинге 8.1, представляющими имя и возраст человека, гораздо лучше — в их случае использование автоматических свойств вполне очевидно. При наличии свойств, которые были тривиально реализованы в предыдущих версиях C#, *отказ* от применения автоматических свойств не дает никаких преимуществ[1].

Небольшое затруднение возникает при использовании автоматических свойств во время написания собственных структур: все конструкторы должны явно вызывать конструктор без параметров, `this()`, чтобы компилятору было известно, что всем полям определенно присвоены значения. Устанавливать поля напрямую нельзя, т.к. они анонимные, а свойства не разрешено применять до тех пор, пока не будут установлены все поля. Единственный способ решения проблемы предполагает вызов конструктора без параметров, который присвоит полям их стандартные значения. Например, если нужно создать структуру с единственным целочисленным свойством, то следующий код не будет допустимым:

[1] Разумеется, это касается свойств, допускающих чтение/запись. Если вы создаете свойство только для чтения, то можете решить использовать поддерживающее поле, предназначенное только для чтения, и свойство с одним лишь средством получения, которое возвращает значение данного поля. Это предотвращает случайную установку свойства внутри класса, которая была бы возможной в случае автоматического свойства в стиле "открытое чтение, закрытая запись".

```
public struct Foo
{
  public int Value { get; private set; }
  public Foo(int value)
  {
    this.Value = value;
  }
}
```

Но за счет явного связывания свойства с конструктором без параметров ситуацию можно исправить:

```
public struct Foo
{
  public int Value { get; private set; }
  public Foo(int value) : this()
  {
    this.Value = value;
  }
}
```

Вот и все, что необходимо для автоматически реализуемых свойств. Никакие лишние украшения они не поддерживают. Например, не существует способа объявления таких свойств с начальными стандартными значениями, равно как приема, позволяющего сделать их подлинно допускающими только чтение (максимум, чего можно добиться — это реализовать закрытое средство установки).

Если бы все возможности C# 3 были настолько простыми, мы могли бы раскрыть *их все* в единственной главе. Конечно же, это не так, но есть средства, которые не требуют *слишком* длинных объяснений. Тема, рассматриваемая следующей, посвящена устранению дублированного кода в еще одной распространенной, хотя и специфичной ситуации — при объявлении локальных переменных.

8.2. Неявная типизация локальных переменных

В главе 2 мы обсуждали природу системы типов C# 1. В частности, я утверждал, что она была статической, явной и безопасной. Это по-прежнему справедливо в C# 2, и *почти* полностью справедливо в C# 3. Статическая и безопасная характеристики остались (не принимая во внимание код, который явно объявлен как небезопасный, что делалось в главе 2), и *большую* часть времени наблюдается явная типизация — но появилась возможность предлагать компилятору самостоятельно выводить типы локальных переменных[2].

8.2.1. Использование ключевого слова `var` для объявления локальной переменной

Для применения неявной типизации понадобится только заменить ключевым словом `var` часть, относящуюся к типу, в обычном объявлении локальной переменной.

[2] В версии C# 4 правила игры снова меняются, разрешая использование динамической типизации там, где это необходимо, как будет показано в главе 14. Один шаг за раз — язык C# по-прежнему был статически типизированным до версии C# 3 включительно.

Имеются определенные ограничения (вскоре мы до них доберемся), но по существу все сводится к простому изменению следующего оператора:

```
MyType variableName = someInitialValue;
```

на такой:

```
var variableName = someInitialValue;
```

Результаты действия этих двух строк кода (в терминах скомпилированного кода) *в точности совпадают*, предполагая, что типом `someInitialValue` является `MyType`. Компилятор просто получает тип выражения инициализации на этапе компиляции и назначает его переменной. Это может быть любой нормальный тип .NET, в том числе обобщение, делегат либо интерфейс. Переменная по-прежнему является статически типизированной; вы просто не указываете имя типа в коде.

Последнее очень важно понимать, поскольку оно служит главной причиной, по которой многие разработчики остерегаются этого нового средства, считая, что ключевое слово `var` превращает язык C# в динамический или слабо типизированный. Все совершенно не так. Лучше всего это объяснить на примере следующего недопустимого кода:

```
var stringVariable = "Hello, world.";
stringVariable = 0;
```

Показанный код не скомпилируется, т.к. типом `stringVariable` является `System.String`, а присваивать значение `0` строковой переменной нельзя. Во многих динамических языках код подобного рода *должен* был бы скомпилироваться, оставляя переменную без конкретного типа, пока об этом не позаботится компилятор, IDE-среда или исполняющая среда. Использование ключевого слова `var` *не* похоже на применение типа `VARIANT` из COM или VB6. Переменная остается статически типизированной, просто тип выводится компилятором. Возможно, я чрезмерно подробно рассматриваю этот аспект, однако он крайне важен и часто становится источником путаницы.

В среде Visual Studio можно выяснить, какой тип компилятор выбрал для переменной, наведя курсор мыши на часть `var` объявления (рис. 8.2). Обратите внимание на то, что для параметров типов обобщенного типа `Dictionary` также отображаются пояснения. Это выглядит знакомым просто потому, что оно в точности повторяет поведение среды при явном объявлении локальных переменных.

Всплывающие подсказки доступны не только в точке объявления. Как и можно было ожидать, всплывающая подсказка с типом переменной также отображается при наведении курсора мыши на имя переменной позже в коде.

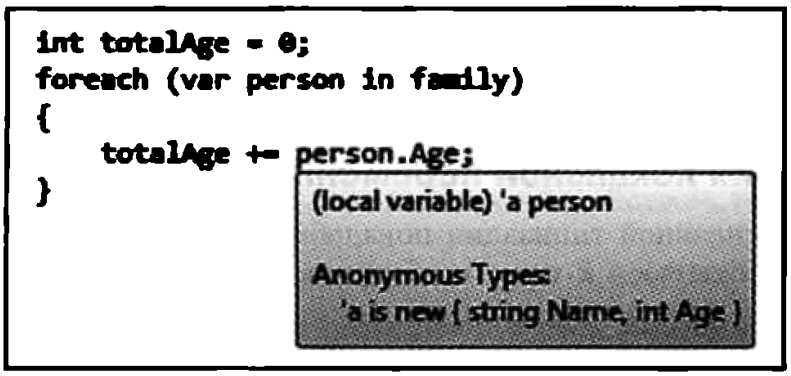

Рис. 8.2. Наведение курсора мыши на часть `var` в среде Visual Studio приводит к отображению типа объявленной переменной

Этот случай показан на рис. 8.3, где применяется то же самое объявление, но курсор наводится на место *использования* переменной. В данной ситуации поведение точно повторяет то, что можно видеть в случае обычного объявления локальной переменной.

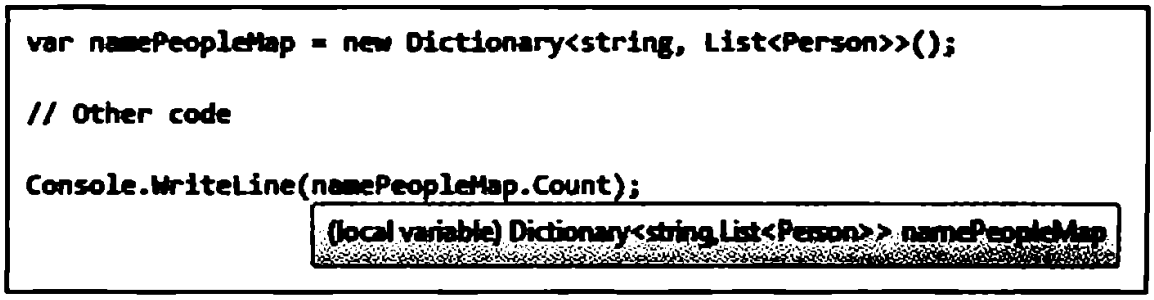

Рис. 8.3. Наведение курсора мыши на место использования переменной в среде Visual Studio приводит к отображению ее типа

Среда Visual Studio привлекается в этом контексте по двум причинам. Во-первых, в большей степени подтверждается применение статической типизации — компилятору точно известен тип переменной. Во-вторых, демонстрируется возможность выяснения типа переменной даже глубоко внутри кода метода. Это будет важно при обсуждении доводов за и против относительно использования неявной типизации. Однако сначала следует упомянуть о ряде ограничений.

8.2.2. Ограничения неявной типизации

Применять неявную типизацию для каждой переменной абсолютно в любых случаях не получится. Ее можно использовать, только если справедливы все перечисленные ниже утверждения.

- Объявляемая переменная является локальной, но не статическим полем или полем экземпляра.
- Переменная инициализируется в виде части объявления.
- Выражение инициализации не является группой методов или анонимной функцией[3] (без приведения).
- Выражение инициализации не дает в результате `null`.
- В операторе объявлена только одна переменная.
- Тип, который нужно назначить переменной, является типом выражения инициализации на этапе компиляции.
- В выражении инициализации не задействована объявляемая переменная[4].

Наиболее интересны третье и четвертое утверждения. Приведенный ниже код является недопустимым:

```
var starter = delegate() { Console.WriteLine(); };
```

[3] Термин *анонимная функция* охватывает анонимные методы и лямбда-выражения, которые будут рассматриваться в главе 9.

[4] Это крайне необычно в любом случае, но возможно в нормальных объявлениях, если приложить достаточно усилий.

Причина в том, что компилятор не может выяснить, какой тип использовать. Однако *можно* записать следующим образом:

```
var starter = (ThreadStart) delegate() { Console.WriteLine(); };
```

Но если вы собираетесь поступать так, то лучше с самого начала объявить переменную явно. То же самое верно в случае с `null` — можно было бы выполнить приведение `null` соответствующим образом, но в этом нет никакого смысла.

Обратите внимание, что в качестве выражения инициализации *допускается* применять результаты вызова методов или свойства — вы не ограничены использованием только констант и обращений к конструкторам. Например, можно было бы написать следующий код:

```
var args = Environment.GetCommandLineArgs();
```

В этом случае переменная `args` получила бы тип `string[]`. В действительности инициализация переменной результатом вызова метода является, вероятно, наиболее распространенной ситуацией, в которой применяется неявная типизация как часть LINQ. Вы увидите все это позже — просто помните о ней во время ознакомления с примерами.

Также полезно отметить, что неявную типизацию *разрешено* использовать для локальных переменных, объявляемых в первой части операторов `using`, `for` и `foreach`. Например, все приведенные ниже операторы допустимы (разумеется, с подходящими телами):

```
for (var i = 0; i < 10; i++)
using (var x = File.OpenText("test.dat"))
foreach (var s in Environment.GetCommandLineArgs())
```

Указанные переменные получают типы `int`, `StreamReader` и `string`, соответственно.

Конечно, наличие *возможности* делать что-то не означает, что вы *обязаны* это делать. Давайте рассмотрим причины в пользу и против применения неявной типизации.

8.2.3. Доводы за и против неявной типизации

Вопрос о том, когда использование неявной типизации имеет смысл, вызывает массу обсуждений в сообществе. Мнения простираются от "везде" до "нигде" с большим числом более сбалансированных подходов между этими двумя крайностями.

В разделе 8.5 будет показано, что для взаимодействия с другим средством C# 3 – анонимными типами — часто *необходимо* применять неявную типизацию. Разумеется, можно было бы также избегать анонимных типов, но это сродни тому, как вместе с водой выплеснуть и ребенка.

Главная причина *для* использования неявной типизации (оставляя на время в стороне анонимные типы) заключается не в том, что она сокращает объем вводимого кода, а в том, что делает код более упорядоченным (и, следовательно, более читабельным) на экране. В частности, когда применяются обобщения, имена типов могут становиться очень длинными. На рис. 8.2 и 8.3 присутствовал тип `Dictionary<string, List<Person>>`, имя которого содержит 33 символа. Когда этот тип понадобится указать в строке дважды (первый раз для объявления и второй раз для инициализации), получится длинная строка всего лишь для объявления и инициализации единственной переменной. В качестве альтернативы можно создать псевдоним, но это отдалит (во всяком случае, концептуально) действительный тип от кода, в котором он используется.

При чтении кода нет никакого смысла наблюдать одно и то же имя типа дважды в одной строке, когда совершенно очевидно, что они *должны* быть одинаковыми. Если объявление на экране не видно, вы находитесь в одинаковом положении независимо от того, применялась неявная типизация или нет (все способы, используемые для выяснения типа переменной, по-прежнему допустимы), а если видно, то выражение, указанное для инициализации переменной, в любом случае сообщит этот тип.

Вдобавок применение ключевого слова `var` изменяет акцент кода. Иногда необходимо, чтобы читатель обратил более пристальное внимание на то, какие точно типы задействованы, поскольку это важно. Например, несмотря на то, что обобщенные типы `SortedList` и `SortedDictionary` имеют похожие API-интерфейсы, они обладают разными характеристиками производительности, что может быть важным в отдельной части кода. Иногда вас действительно заботят только выполняемые операции; вы не задумываетесь о том, что будет, если выражение, используемое для инициализации, изменится, до тех пор, пока удается достичь тех же самых целей[5]. Применение `var` позволяет читателю сосредоточить внимание на *использовании* переменной, а не на самом объявлении, т.е. на том, *что* происходит в коде, а не *как* оно происходит.

Каковы тогда аргументы *против* неявной типизации? Как ни парадоксально, самым важным аргументом против является читабельность, и это притом что она также считается аргументом за неявную типизацию! Не указывая явно тип для объявляемой переменной, вы можете затруднить выяснение ее предназначения при чтении кода. Это нарушает образ мышления в стиле "заявить, что объявляется, а затем указать начальное значение", который сохраняет объявление и инициализацию отдельными. Насколько это окажется проблемой, зависит как от читателя кода, так и от задействованного выражения инициализации.

Если явно вызывать конструктор, всегда будет совершенно ясно, что именно создается. Если вызывается метод или применяется свойство, то все зависит от того, насколько очевидным является возвращаемый тип при простом взгляде на данный вызов. Целочисленные литералы — это пример, когда высказать предположение о типе выражения труднее, чем может показаться. Насколько быстро вы сможете выяснить типы объявленных ниже переменных?

```
var a = 2147483647;
var b = 2147483648;
var c = 4294967295;
var d = 4294967296;
var e = 9223372036854775807;
var f = 9223372036854775808;
```

Ответом будет `int`, `uint`, `uint`, `long`, `long` и `ulong`, соответственно — используемый тип зависит от значения выражения. Здесь нет ничего нового с точки зрения обработки литералов, т.к. компилятор C# всегда ведет себя подобным образом, но неявная типизация в этом случае только способствует получению непонятного кода.

Аргумент, который редко явно высказывается, но, я уверен, стоит за большинством вопросов, связанных с неявной типизацией, звучит так: "Ощущение, что что-то здесь не так". Если вы годами имели дело с языком, подобным C, то в неявной типизации есть нечто такое, что вызывает беспокойство, хотя было бы достаточно сказать себе, что

---

[5] Я понимаю, что это выглядит немного похоже на утиную типизацию: "если оно может работать, то все в порядке". Разница в том, что проверка работоспособности по-прежнему осуществляется на этапе компиляции, а не во время выполнения.

"за кулисами" по-прежнему применяется статическая типизация. Такое беспокойство нельзя считать разумным, но это не делает его менее реальным. Если вы испытываете неудобства, то, скорее всего, будете менее продуктивно работать. Вполне нормально, даже если лично для вас преимущества не перевешивают негативные ощущения. В зависимости от особенностей характера вы можете попробовать подтолкнуть себя в направлении более спокойного отношения к неявной типизации, но определенно не обязаны делать это.

8.2.4. Рекомендации

Ниже приведены некоторые рекомендации, собранные на основе моего опыта работы с неявной типизацией. Все это только рекомендации, так что относитесь к ним с изрядной долей скепсиса.

- Если важно, чтобы читатель кода определял тип переменной с первого же взгляда, используйте явную типизацию.
- Если переменная инициализируется напрямую с помощью конструктора, а имя типа имеет большую длину (что часто случается с обобщениями), подумайте о применении неявной типизации.
- Если точный тип переменной не важен, но общая природа ясна из контекста, используйте неявную типизацию, чтобы убрать акцент с того, каким образом код достигает цели, и сосредоточить внимание на более высоком уровне, связанном с тем, что именно он достигает.
- Начиная проект, посоветуйтесь с коллегами на эту тему.
- При наличии сомнений попробуйте написать строку кода обоими способами и посмотрите, какой вариант вам больше нравится.

Ранее я применял явную типизацию в производственном коде всегда кроме ситуаций, в которых использование неявной типизации сулило очевидную и значительную выгоду. Большинство случаев применения неявной типизации касалось тестового кода (а также разового кода). В наши дни я веду себя более неоднозначно и откровенно непоследовательно. Я благополучно использую неявную типизацию в производственном коде лишь ради небольшого упрощения, даже когда набирать задействованные имена типов не особо обременительно. Хотя последовательность в определенных аспектах стиля кодирования довольно-таки важна, я не обнаружил, чтобы такой подход со смешиванием и подгонкой вызывал какие-то проблемы.

В сущности, мои рекомендации сводятся к отказу от применения неявной типизации лишь потому, что она экономит несколько нажатий клавиш. Там, где неявная типизация позволяет получить более аккуратный код, давая возможность сосредоточиться на наиболее важных элементах кода, ее имеет смысл использовать. Я буду широко применять неявную типизацию в остальных материалах книги по той простой причине, что код труднее форматировать под требования печатной страницы, чем на экране монитора — доступна не такая уж большая ширина строки.

Мы снова возвратимся к неявной типизации, когда начнем рассматривать анонимные типы, т.к. они создают ситуации, в которых приходится просить компилятор вывести типы для набора переменных. Но сначала давайте посмотрим, каким образом C# 3 облегчает конструирование и наполнение нового объекта в одном выражении.

8.3. Упрощенная инициализация

Казалось бы, что объектно-ориентированные языки должны были давно упростить создание объектов. В конце концов, перед началом использования объекта *что-то* должно создать его, либо напрямую в коде, либо с помощью какого-нибудь фабричного метода. Несмотря на это, несколько языковых средств в C# 2 направлены на упрощение решения задач инициализации. Если необходимое не удается сделать с применением аргументов конструктора, значит, вам не повезло, и придется создавать объект, после чего вручную инициализировать его через свойства и предпринимать другие подобные действия.

Это особенно надоедает, когда требуется создать множество объектов за один шаг, как в случае массива или другой коллекции. Не имея способа инициализации объекта с помощью одиночного выражения, приходилось либо использовать локальные переменные для временных манипуляций, либо создавать вспомогательный метод, который выполняет нужную инициализацию на основе параметров.

В этом разделе вы увидите, что версия C# 3 приходит на помощь многими путями.

8.3.1. Определение нескольких демонстрационных типов

Выражения, которые будут применяться в этом разделе, называются *инициализаторами объектов*. Они представляют собой просто способы указания инициализации, которая должна быть выполнена после создания объекта. Можно устанавливать свойства, свойства свойств (не переживайте, это проще, чем звучит) и добавлять элементы в коллекции, которые доступны через свойства.

Для демонстрации всего этого мы опять будем использовать класс `Person`. В нем хранятся имя и возраст, доступные как записываемые свойства. Мы предоставим конструктор без параметров и конструктор, принимающий в качестве параметра имя. Кроме того, мы добавим список друзей и местоположение дома человека, которые будут доступны как свойства только для чтения, но могут быть модифицированы за счет манипулирования извлеченными объектами. Простой класс `Location` предоставляет свойства `Country` и `Town`, представляющие местоположение дома человека. В листинге 8.2 приведен полный код для упомянутых классов.

Листинг 8.2. Довольно простой класс `Person`, предназначенный для будущих демонстраций

```
public class Person
{
  public int Age { get; set; }
  public string Name { get; set; }

  List<Person> friends = new List<Person>();
  public List<Person> Friends { get { return friends; } }

  Location home = new Location();
  public Location Home { get { return home; } }

  public Person() { }

  public Person(string name)
  {
    Name = name;
  }
}
```

```
public class Location
{
  public string Country { get; set; }
  public string Town { get; set; }
}
```

Код в листинге 8.2 прямолинеен, но уместно отметить, что при создании объекта, представляющего человека, список друзей и местоположение дома создаются пустыми, а не остаются ссылками null. Вдобавок свойства, хранящие список друзей и местоположение дома, допускают только чтение. Это будет важно позже, а пока давайте посмотрим на свойства, которые представляют имя и возраст человека.

8.3.2. Установка простых свойств

Теперь, когда есть тип Person, самое время создать несколько его экземпляров с применением новых средств C# 3. В этом разделе мы рассмотрим установку свойств Name и Age, а к другим свойствам обратимся позже.

Инициализаторы объектов чаще всего используются для установки свойств, но все показанное здесь в равной степени применимо и к полям. Однако в хорошо инкапсулируемой системе вы вряд ли будете иметь доступ к полям, если только не создадите экземпляр типа внутри собственного кода этого типа. Конечно, важно учесть, что использовать поля *можно*, поэтому в остальном материале данного раздела везде, где встречается понятие *свойство*, следует воспринимать его как *свойство и поле*.

Имея все это в виду, давайте приступим к работе. Предположим, что необходимо создать объект, представляющий человека по имени Том (*Tom*) в возрасте 9 лет. До появления версии C# 3 этого можно было достичь двумя путями:

```
Person tom1 = new Person();
tom1.Name = "Tom";
tom1.Age = 9;
Person tom2 = new Person("Tom");
tom2.Age = 9;
```

В первом варианте применяется конструктор без параметров, а затем устанавливаются оба свойства. Во втором варианте используется перегруженная версия конструктора, устанавливающая имя, после чего устанавливается возраст. Оба варианта по-прежнему доступны в C# 3, но существуют также и альтернативы:

```
Person tom3 = new Person() { Name = "Tom", Age = 9 };
Person tom4 = new Person { Name = "Tom", Age = 9 };
Person tom5 = new Person("Tom") { Age = 9 };
```

Часть в фигурных скобках в конце каждой строки — это инициализатор объекта. Опять-таки, он является трюком компилятора. Код IL, применяемый для инициализации tom3 и tom4, идентичен и очень близок к коду, используемому для инициализации tom1[6]. Как и можно было ожидать, код для tom5 практически совпадает с кодом для tom2. Обратите внимание на отсутствие круглых скобок для конструктора в инициализации tom4. Такое сокращение можно применять для типов с конструктором без параметров, который и будет вызываться в скомпилированном коде.

[6] На самом деле новое значение tom1 не присваивается до тех пор, пока не будут установлены все свойства. А до этого момента используется временная локальная переменная. Этот факт редко оказывается важным, но о нем полезно знать, чтобы избежать путаницы, если вдруг отладчик остановится на середине пути выполнения инициализатора.

После вызова конструктора указанные свойства устанавливаются очевидным образом. Они устанавливаются в порядке, в котором заданы внутри инициализатора объекта, и каждое конкретное свойство можно указывать только один раз — например, не допускается устанавливать свойство Name дважды. (Можно было бы вызвать конструктор, принимающий имя в качестве параметра, а затем установить свойство Name. Хотя это бессмысленно, но компилятор не будет препятствовать подобным действиям.) Выражение, применяемое для значения свойства, может быть любым выражением, которое само не является присваиванием — можно вызывать методы, создавать новые объекты (потенциально используя другой инициализатор объекта) и делать многое другое.

Вас может интересовать, насколько это полезно — сэкономлена одна или две строки кода, но, несомненно, это не повод делать язык более сложным, не так ли? Тем не менее, здесь присутствует один тонкий момент: вы не просто создали объект в одной *строке* — вы создали его в одном *выражении*. Эта разница может быть очень важна.

Предположим, что требуется создать массив типа Person[] с некоторыми предопределенными данными. Даже без применения неявной типизации массивов, которая будет показана позже, код получается лаконичным и читабельным:

```
Person[] family = new Person[]
{
  new Person { Name = "Holly", Age = 36 },
  new Person { Name = "Jon", Age = 36 },
  new Person { Name = "Tom", Age = 9 },
  new Person { Name = "William", Age = 6 },
  new Person { Name = "Robin", Age = 6 }
};
```

В простом примере вроде этого можно было бы написать конструктор, принимающий в качестве параметров имя и возраст, и инициализировать массив способом, похожим на то, как это делалось в C# 1 или C# 2. Однако подходящие конструкторы не всегда доступны, а при наличии множества параметров часто неясен смысл того или иного параметра, а понятна только его позиция в списке. К тому времени, когда конструктор должен принимать пять-шесть параметров, я часто замечаю, что полагаюсь на средство IntelliSense больше, чем того хотел бы. В таких случаях большим благом для читабельности становится использование имен свойств[7].

Пожалуй, эта форма инициализатора объекта является одной из наиболее часто применяемых. Тем не менее, доступны еще две формы, одна из которых предназначена для установки подсвойств, а другая — для добавления элементов в коллекции. Давайте сначала посмотрим, что собой представляют подсвойства — свойства свойств.

8.3.3. Установка свойств встроенных объектов

До сих пор никаких сложностей с установкой свойств Name и Age не возникало, но установить свойство Home таким же образом не получится — оно допускает только чтение. С другой стороны, *можно* установить город и страну проживания человека, сначала извлекая свойство Home и затем устанавливая свойства полученного результата. В спецификации языка это называется установкой свойств *встроенного объекта*.

[7] В версии C# 4 предлагается альтернативный подход, предусматривающий применение именованных аргументов, который будет рассматриваться в главе 13.

Чтобы прояснить сказанное, ниже приведен код на C# 1:

```
Person tom = new Person("Tom");
tom.Age = 9;
tom.Home.Country = "UK";
tom.Home.Town = "Reading";
```

Когда заполняется местоположение дома, каждый оператор вызывает средство получения, чтобы извлечь экземпляр Location, а затем обращается к средству установки этого экземпляра. Здесь нет ничего нового, но полезно уделить этому коду большее внимание, иначе легко упустить из виду, что происходит "за кулисами".

Версия C# 3 позволяет делать все это в одном выражении, как показано ниже:

```
Person tom = new Person("Tom")
{
  Age = 9,
  Home = { Country = "UK", Town = "Reading" }
};
```

Скомпилированный код для приведенных выше фрагментов фактически одинаков. Компилятор обнаруживает справа от знака = другой инициализатор объекта и соответствующим образом применяет свойства к встроенному объекту.

Отсутствие ключевого слова new в части инициализации Home играет важную роль. Если необходимо выяснить, где компилятор собирается создавать новые объекты, а где устанавливать свойства существующих объектов, ищите вхождения new в инициализаторе. Каждый раз, когда создается новый объект, где-то присутствует ключевое слово new.

Форматирование кода инициализатора объекта

Как и с почти всеми средствами C#, инициализаторы объектов не зависят от пробельных символов. При желании вы можете устранить пробельные символы в инициализаторе объекта, поместив весь код в одну строку. Только от вас зависит, каким образом балансировать между длинными строками и большим количеством строк.

Мы имели дело со свойством Home, но как насчет друзей человека по имени Том? В типе List<Person> есть свойства, которые можно устанавливать, но ни одно из них не будет добавлять элементы в список. Самое время переходить к следующему средству — инициализаторам коллекций.

8.3.4. Инициализаторы коллекций

Создание коллекции с рядом начальных значений — исключительно распространенная задача. До появления версии C# 3 единственным языковым средством, которое хоть как-то содействовало этому, было создание массивов, даже с учетом того, что оно выглядело неуклюжим во многих ситуациях. В C# 3 имеются инициализаторы коллекций, которые позволяют использовать тот же самый вид синтаксиса, что в инициализаторах массивов, но для произвольных коллекций и с большей гибкостью.

Создание новых коллекций с помощью инициализаторов коллекций

В качестве первого примера давайте воспользуемся уже знакомым типом List<T>. В C# 2 список можно было заполнить либо путем передачи ему существующей коллекции, либо за счет многократных вызовов метода Add() после создания пустого списка. Инициализаторы коллекций в C# 3 принимают второй подход.

Предположим, что нужно заполнить список строк некоторыми именами; ниже показан код C# 2 (слева) и близкий ему эквивалент в C# 3 (справа):

```
List<string> names = new List<string>();
names.Add("Holly");
names.Add("Jon");
names.Add("Tom");
names.Add("Robin");
names.Add("William");
```

```
var names = new List<string>
{
  "Holly", "Jon", "Tom",
  "Robin", "William"
}
```

В точности как с инициализаторами объектов, при желании можно указывать аргументы либо применять конструктор без параметров, явно или неявно. Использование здесь неявной типизации частично объясняется необходимостью в сокращении кода — переменная `names` могла бы с тем же успехом быть объявлена и явно. Сокращение количества строк кода (без снижения читабельности) — это хорошо, но с инициализаторами коллекций связаны два больших преимущества.

- Часть создания и инициализации считается одним выражением.
- В коде гораздо меньше путаницы.

Первый аспект становится важным, когда коллекцию нужно передавать в виде аргумента методу или применять в качестве одного элемента в более крупной коллекции. Это случается *относительно* редко (хотя достаточно часто для того, чтобы быть полезным). Второй аспект, по моему мнению, является настоящей причиной пользоваться инициализаторами коллекций. Глядя на код справа, легко увидеть необходимую информацию, при этом каждая порция информации записывается только один раз. Один раз встречается имя переменной, один раз применяется тип, и каждый элемент инициализируемой коллекции появляется лишь однократно. Код исключительно прост и намного яснее, чем код C# 2, который содержит помимо *важной* много малозначительной информации.

Инициализаторы коллекций не ограничиваются только списками. Их можно использовать с любым типом, который реализует интерфейс `IEnumerable`, при условии, что он имеет подходящий метод `Add()` для каждого элемента в инициализаторе. Метод `Add()` можно применять с более чем одним параметром, помещая значения внутрь другого набора фигурных скобок. Самое распространенное использование такого приема связано с созданием словарей. Например, если требуется словарь, отображающий имена на возрасты, можно было бы написать следующий код:

```
Dictionary<string,int> nameAgeMap = new Dictionary<string,int>
{
  { "Holly", 36 },
  { "Jon", 36 },
  { "Tom", 9 }
};
```

В данном случае метод `Add(string, int)` будет вызван три раза. Если доступно несколько методов `Add()`, разные элементы инициализатора могут обращаться к разным перегруженным версиям. Если для указанного элемента совместимая перегруженная версия отсутствует, код не скомпилируется. В этом проектном решении присутствуют два интересных аспекта.

- Тот факт, что тип должен реализовывать интерфейс `IEnumerable`, никогда не используется компилятором.
- Метод `Add()` находится только по имени — в интерфейсе нет требований о его указании.

Оба решения прагматичны. Требование реализации `IEnumerable` — это разумная попытка проверить, что тип действительно является коллекцией какого-то вида, а применение любой доступной перегруженной версии метода `Add()` (вместо обязательной точной сигнатуры) позволяет строить простые реализации, такие как показанный выше пример словаря.

В раннем черновике спецификации C# 3 вместо этого выставлялось требование реализации интерфейса `ICollection<T>`, и вызывалась реализация метода `Add()` с одним параметром (как определено интерфейсом), а не разрешалось иметь разные перегруженные версии. Это выглядит более строгим, однако типов, которые реализуют интерфейс `IEnumerable`, существует намного больше, чем типов, реализующих `ICollection<T>`, а использование метода `Add()` с одним параметром могло быть неудобным. Например, в данном случае пришлось бы создавать экземпляр `KeyValuePair<string,int>` для каждого элемента инициализатора. Принесение в жертву некоторой академической чистоты сделало язык более удобным в реальной жизни.

Заполнение коллекций внутри других инициализаторов объектов

До сих пор рассматривались только инициализаторы коллекций, которые применялись автономным образом для создания целых новых коллекций. Они также могут быть скомбинированы с инициализаторами объектов для заполнения встроенных коллекций. В целях демонстрации сказанного мы возвратимся к примеру с типом `Person`. Свойство `Friends` допускает только чтение, поэтому нельзя создать новую коллекцию и указать ее в качестве коллекции друзей, но *можно* добавлять элементы в коллекцию, возвращаемую средством получения этого свойства. Это делается с помощью синтаксиса, похожего на тот, что использовался для установки свойств встроенных объектов, но теперь вместо последовательности объектов указывается инициализатор коллекции.

Давайте взглянем на сказанное в действии, создав еще один экземпляр `Person` для человека по имени Том, на этот раз с несколькими его друзьями.

Листинг 8.3. Построение более сложного объекта с применением инициализаторов объекта и коллекции

```
Person tom = new Person                                    ← Неявный вызов конструктора без параметров
{
   Name = "Tom",                                           ← Установка свойств напрямую
   Age = 9,
   Home = { Town = "Reading", Country = "UK" },            ← Инициализация встроенного объекта
   Friends =
   {
      new Person { Name = "Alberto" },                     ← Инициализация коллекции с помощью
                                                             добавочных инициализаторов объектов
      new Person("Max"),
      new Person { Name = "Zak", Age = 7 },
      new Person("Ben"),
      new Person("Alice")
      {
         Age = 9,
         Home = { Town = "Twyford", Country = "UK" }
      }
   }
};
```

Код в листинге 8.3 пользуется всеми описанными ранее возможностями инициализаторов объекта и коллекции. Главный интерес представляет инициализатор коллекции, который сам внутренне использует много различных форм инициализаторов коллекций. Обратите внимание на то, что здесь не создается новая коллекция, а производится добавление элементов в существующую коллекцию. (Если свойство располагает средством установки, оно *могло бы* создавать новую коллекцию и по-прежнему пользоваться синтаксисом инициализаторов коллекций.)

Можно было бы продолжить дальше, указывая друзей друзей, друзей друзей друзей и т.д. Однако невозможно указать, что человек по имени Том является другом человека по имени Альберто (Alberto). В то время как объект все еще инициализируется, доступ к нему отсутствует, поэтому выразить циклические отношения не удастся. Это может вызвать затруднения в некоторых случаях, но обычно проблемой не является.

Инициализаторы коллекций внутри инициализаторов объектов работают как своего рода гибрид автономных инициализаторов коллекций и установки свойств встроенных объектов. Для каждого элемента в инициализаторе коллекции производится обращение к средству получения свойства коллекции (в этом случае `Friends`), после чего на возвращаемом значении вызывается метод `Add()`. Перед добавлением элементов коллекция никак не очищается. Например, если вы решили, что человек должен всегда быть другом самому себе, и добавили в список друзей `this` внутри конструктора `Person`, то во время применения инициализатора коллекции должны добавляться только дополнительные друзья.

Как видите, сочетание инициализаторов коллекций и объектов можно использовать для наполнения целого дерева объектов. Но когда и где это реально происходит?

8.3.5. Использование средств инициализации

Попытка точного выяснения, где эти средства полезны, напоминает скетч Монти Пайтона об испанской инквизиции — каждый раз, когда кажется, что список завершен, обнаруживается очередной известный случай. Я упомяну лишь три примера, которые, как я надеюсь, подтолкнут вас к обдумыванию ситуаций, в которых пользоваться средствами инициализации могли бы *вы сами.*

Константные коллекции

У меня не так уж редко возникает потребность в коллекции (часто в виде карты), которая фактически является константной. Разумеется, она не может быть константной в смысле языка C#, но ее *можно* объявить статической и доступной только для чтения, и это позволит с большой уверенностью говорить, что изменяться она не должна. (Обычно коллекция является закрытой, что довольно-таки хорошо. В качестве альтернативы можно применять тип `ReadOnlyCollection<T>`.) Как правило, при этом приходится писать статический конструктор или вспомогательный метод, предназначенный для наполнения карты. Благодаря инициализаторам коллекций C# 3, все легко установить в коде.

Настройка модульных тестов

При написании модульных тестов мне нередко нужно заполнить объект всего лишь для одного теста, часто передавая его в виде аргумента методу, который тестируется в данный момент. Полный код инициализации может быть многословным и также скрывать внутреннюю структуру объекта от читателя кода, подобно тому, как код создания XML-разметки часто мешает понять внешний вид документа во время просмотра этого

кода (соответственно сформатированного) в текстовом редакторе. С помощью подходящих отступов для инициализаторов объектов вложенная структура иерархии объектов может стать более очевидной в самом коде, а также сделать значения более заметными, чем они были бы в противном случае.

Шаблон построителя

По разным причинам иногда для одиночного вызова метода или конструктора требуется указать множество значений. Самая распространенная ситуация в моей практике связана с созданием неизменяемого объекта. Вместо того чтобы иметь большой набор параметров (которые могут привести к проблеме с читабельностью, поскольку предназначение каждого аргумента становится неясным[8]), можно воспользоваться *шаблоном построителя* (Builder) — создать изменяемый тип с подходящими свойствами и затем передавать экземпляр этого построителя в конструктор или метод. Хорошим примером может служить тип `ProcessStartInfo` в инфраструктуре — проектировщики *могли бы* предусмотреть перегруженные версии для метода `Process.Start()` с множеством разных наборов параметров, но применение `ProcessStartInfo` делает все проще.

Инициализаторы объектов и коллекций позволяют создавать объект построителя в более ясной манере — при желании его можно даже указать внутри обращения к исходному члену. Общеизвестно, что вы по-прежнему должны сначала определить тип построителя, но в этом помогают автоматические свойства.

<Укажите здесь свой предпочитаемый случай использования>

Конечно, в рядовом коде кроме указанных трех встречаются и другие случаи использования новых средств, и я не собираюсь отговаривать вас применять их в тех или иных ситуациях. Помимо возможной путаницы у разработчиков, пока еще хорошо не освоивших C# 3, существует очень мало причин для *отказа* от их использования. Вы можете решить, что применение инициализатора объекта лишь для установки свойства (как противоположность явной его установки в отдельном операторе) является чрезмерным, однако это вопрос эстетического восприятия, поэтому я не могу здесь предложить сколько-нибудь объективные инструкции. Как и с неявной типизацией, полезно попытаться написать код в двух формах и научиться предугадывать собственные (а также участников команды) предпочтения в плане восприятия кода.

До сих пор мы имели дело с достаточно разрозненным набором средств: облеченная реализация свойств, упрощенные объявления локальных переменных и заполнение объектов в одиночных выражениях. В оставшемся материале главы мы будем постепенно объединять эти темы, больше применяя неявную типизацию и заполнение объектов и создавая целые *типы* без предоставления деталей реализации.

Следующее средство выглядит очень похожим на инициализаторы коллекций, если взглянуть на код, где оно используется. Ранее я упоминал, что инициализация массивов в C# 1 и C# 2 была несколько неуклюжей. Уверен, что вас не удивит тот факт, что в C# 3 это было улучшено. Давайте посмотрим, в чем суть улучшений.

8.4. Неявно типизированные массивы

В версиях C# 1 и C# 2 инициализация массива в рамках оператора объявления и инициализации переменной была довольно лаконичной, но если это требовалось делать

---

[8] Надо сказать, что именованные аргументы C# 4 оказывают помощь в этой области.

в другом месте, то необходимо было указывать точный тип массива. Например, следующий код компилируется без проблем:

```
string[] names = {"Holly", "Jon", "Tom", "Robin", "William"};
```

Тем не менее, это не работает в отношении параметров — предположим, что нужно вызвать метод MyMethod(), объявленный как void MyMethod(string[] names). Показанный ниже код не скомпилируется:

```
MyMethod({"Holly", "Jon", "Tom", "Robin", "William"});
```

Вместо этого придется сообщить компилятору тип массива, предназначенного для инициализации:

```
MyMethod(new string[] {"Holly", "Jon", "Tom", "Robin", "William"});
```

В C# 3 разрешено нечто промежуточное:

```
MyMethod(new[] {"Holly", "Jon", "Tom", "Robin", "William"});
```

Очевидно, что компилятор должен выяснить тип используемого массива. Он начинает с формирования набора всех типов выражений, известных на этапе компиляции, которые находятся внутри фигурных скобок. Если в этом наборе оказывается в точности один тип, в который могут быть неявно преобразованы все остальные типы, то он и будет типом массива. В противном случае (или если все значения являются выражениями без типов, такими как константные значения null или анонимные методы, не имеющие приведений) код не скомпилируется.

Обратите внимание, что только в качестве кандидатов на тип массива в целом учитываются только типы выражений. Это означает, что иногда может понадобиться явно приводить значение к *менее* специфичному типу. Например, следующий код не скомпилируется:

```
new[] { new MemoryStream(), new StringWriter() }
```

Преобразования из MemoryStream в StringWriter или наоборот не существует. Оба типа могут быть неявно преобразованы в object и IDisposable, но компилятор принимает во внимание только типы, которые находятся в исходном наборе, получившемся на основе самих выражений. Если в этой ситуации изменить одно из выражений так, чтобы его типом стал object или IDisposable, то код скомпилируется:

```
new[] { (IDisposable) new MemoryStream(), new StringWriter() }
```

Типом последнего выражения неявно является IDisposable[]. Разумеется, в этот момент можно также явно указать тип массива, как это бы делалось в C# 1 и C# 2, чтобы еще больше прояснить свои намерения.

По сравнению с предыдущими средствами неявно типизированные массивы вызывают небольшое разочарование. Они редко приводят меня в волнение, даже с учетом того, что *упрощают* жизнь в случаях, когда массив передается в качестве аргумента. Тем не менее, проектировщики вовсе не сошли с ума — есть она важная ситуация, при которой такая неявная типизация имеет решающее значение. Это происходит, когда вы не знаете (да и *не можете* знать) имя типа элементов в массиве. Как вообще можно попасть в такую необычную ситуацию? Об этом пойдет речь далее.

8.5. Анонимные типы

Неявная типизация, инициализаторы объектов и коллекций и неявная типизация массивов в большей или меньшей степени полезны сами по себе. Однако они также служат и более высокоуровневой цели, т.к. делают возможной работу со средством, которое рассматривается в этой главе последним — *анонимными типами*. В свою очередь, анонимные типы ориентированы на еще одну высокоуровневую цель — язык LINQ.

8.5.1. Знакомство с анонимными типами

Анонимные типы намного легче объяснять, когда уже имеется определенное представление о том, как они выглядят на примере. Я вынужден предупредить, что без применения расширяющих методов и лямбда-выражений примеры, приводимые в этом разделе, вероятно, будут выглядеть несколько надуманными. Тем не менее, здесь возникает ситуация, при которой трудно определить причину и следствие: известно, что анонимные типы наиболее полезны в рамках контекста более сложных средств, но нам необходимо раскрыть все строительные блоки, прежде чем можно будет взглянуть на широкую картину происходящего. Внимательно читайте — я обещаю, что, в конце концов, это *обретет* смысл. Давайте представим, что класса `Person` нет в наличии, а нас заботят только свойства, представляющие имя и возраст. В листинге 8.4 показано, как можно было бы строить объекты с этими свойствами, даже не объявляя тип.

Листинг 8.4. Создание объектов анонимного типа со свойствами Name и Age

```
var tom = new { Name = "Tom", Age = 9 };
var holly = new { Name = "Holly", Age = 36 };
var jon = new { Name = "Jon", Age = 36 };
Console.WriteLine("{0} is {1} years old", jon.Name, jon.Age);
```

На основе кода в листинге 8.4 можно сказать, что синтаксис для инициализации анонимного типа подобен синтаксису инициализаторов объектов, который демонстрировался в разделе 8.3.2 — только отсутствует имя типа между ключевым словом `new` и открывающей фигурной скобкой. Здесь используются неявно типизированные локальные переменные, т.к. это все, что *можно* применять (конечно, за исключением типа `object`) — имя типа, с которым должна объявляться переменная, отсутствует. Как видно в последней строке кода, тип имеет свойства `Name` и `Age`, которые можно читать и которые получают значения, указанные в *инициализаторе анонимного объекта*, используемом для создания экземпляра, поэтому в данном случае на консоль выводится строка `Jon is 36 years old`. Свойства имеют те же типы, что и выражения в инициализаторах — `string` для `Name` и `int` для `Age`. Подобно обычным инициализаторам объектов, выражения, применяемые в инициализаторах анонимных объектов, могут вызывать методы или конструкторы, извлекать значения свойств, выполнять вычисления — в общем, делать все, что необходимо. Возможно, вы уже начали понимать, почему неявно типизированные массивы настолько важны. Предположим, что требуется создать массив, содержащий целую семью, а затем пройти по нему с целью вычисления суммарного возраста членов[9].

[9] Если вы уже знакомы с LINQ, то можете счесть такой способ суммирования возрастов несколько странным. Согласен, вызов `family.Sum(p => p.Age)` был бы намного лаконичнее, но давайте двигаться постепенно.

В листинге 8.5 именно это и делается, а также одновременно демонстрируется ряд других интересных возможностей анонимных типов.

Листинг 8.5. Заполнение массива с использованием анонимных типов и вычисление суммарного возраста

```
var family = new[]                    ←❶ Использование инициализатора неявно типизированного массива
{
   new { Name = "Holly", Age = 36 }, ←❷ Пятикратное использование того же самого анонимного типа
   new { Name = "Jon", Age = 36 },
   new { Name = "Tom", Age = 9 },
   new { Name = "Robin", Age = 6 },
   new { Name = "William", Age = 6 }
};
int totalAge = 0;
foreach (var person in family) ←❸ Использование неявной типизации для переменной person
{
   totalAge += person.Age;     ←❹ Суммирование возрастов
}
Console.WriteLine("Total age: {0}", totalAge);
```

Сложив вместе код в листинге 8.5 и сведения о неявно типизированных массивах из раздела 8.4, можно сделать важный вывод: *все элементы, представляющие людей, в массиве `family` имеют одинаковый тип*. Если бы каждый случай применения инициализатора анонимного объекта ❷ ссылался на отличающийся тип, компилятору не удалось бы вывести подходящий тип для массива ❶. Внутри любой отдельно взятой сборки компилятор трактует два инициализатора анонимных объектов как относящиеся к тому же самому типу, если у них совпадает количество свойств, их имена и порядок следования. Другими словами, если поменять местами свойства `Name` и `Age` в одном из инициализаторов, будут вовлечены два типа; подобным же образом, если указать дополнительное свойство в одной из строк или воспользоваться типом `long` вместо `int` для возраста одного из объектов, будет введен еще один анонимный тип. В этот момент выведение типа для массива потерпит неудачу.

Деталь реализации: сколько всего типов?

Если вы когда-либо просмотрите код IL (или декомпилированный код C#) для анонимного типа, который сгенерирован компилятором Microsoft, то увидите, что хотя два инициализатора анонимных объектов с одинаковыми именами свойств, следующими в том же самом порядке, но имеющими отличающиеся типы, приводят к созданию двух разных типов, на самом деле они генерируются из одного обобщенного типа. Этот обобщенный тип параметризирован, но закрытые сконструированные типы будут отличаться из-за того, что они получают разные аргументы типов, указанные в разных инициализаторах.

Обратите внимание на возможность применения для прохода по массиву оператора `foreach`, как это бы делалось для любой другой коллекции. Задействованный тип выводится ❸, и переменной `person` назначается тот же самый анонимный тип, который использовался в массиве. Опять-таки, эту переменную можно применять для разных экземпляров, т.к. все они принадлежат к одному и тому же типу.

Код в листинге 8.5 также доказывает, что свойство Age действительно является строго типизированным как int — иначе попытка суммирования возрастов ❹ не прошла бы компиляцию. Компилятору известно об анонимном типе, а среда Visual Studio даже охотно делится информацией о нем через всплывающие подсказки. На рис. 8.4 показан результат наведения курсора мыши на часть person выражения person.Age в листинге 8.5.

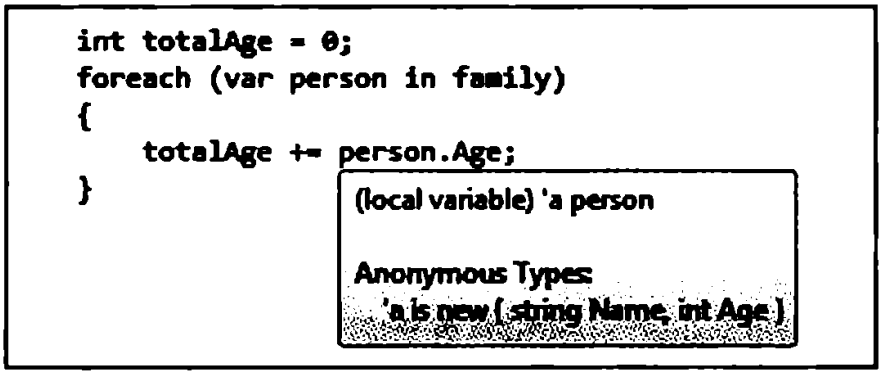

Рис. 8.4. Наведение курсора мыши на имя переменной, которая объявлена (неявно) как имеющая анонимный тип, приводит к отображению подробных сведений об этом анонимном типе

Теперь, когда вы увидели анонимные типы в работе, давайте посмотрим, что фактически делает компилятор.

8.5.2. Члены анонимного типа

Анонимные типы создаются компилятором и включаются в скомпилированную сборку таким же образом, как дополнительные типы для анонимных методов и итераторных блоков. Среда CLR трактует их как совершенно обычные методы, что так и есть — если позже вы перейдете от анонимного типа к нормальному, вручную написанному типу с поведением, описанным в этом разделе, то увидите, что ничего не изменилось.

Анонимные типы содержат перечисленные ниже члены.

- Конструктор, принимающий все инициализационные значения. Параметры следуют в том же самом порядке, как они были указаны в инициализаторе анонимного объекта, и имеют те же самые имена и типы.
- Открытые свойства только для чтения.
- Закрытые поля только для чтения, поддерживающие эти свойства.
- Переопределенные версии методов Equals(), GetHashCode() и ToString().

И это все. Не предусмотрено никаких реализованных интерфейсов, возможностей клонирования или сериализации — лишь конструктор, несколько свойств и обычные методы из типа object.

Конструктор и свойства выполняют очевидные действия. Эквивалентность двух экземпляров одного анонимного типа определяется в естественной манере, со сравнением значений свойств по очереди, используя метод Equals() типа свойства. Генерация хеш-кода работает аналогично, поочередно вызывая метод GetHashCode() на значении каждого свойства и комбинируя результаты. Точный метод объединения вместе различных хеш-кодов для формирования одного составного хеш-кода не указан, поэтому вы не должны писать код, который хоть как-то зависит от него — нужно только иметь уверен-

ность, что два эквивалентных экземпляра возвратят один и тот же хеш-код, а два неэквивалентных экземпляра, *как правило*, будут возвращать разные хеш-коды. Разумеется, это будет работать, только если реализации `Equals()` и `GetHashCode()` всех типов, которые задействованы в свойствах, соответствуют обычным правилам.

Поскольку свойства допускают только чтение, все анонимные типы являются неизменяемыми при условии неизменяемости типов, выбранных для их свойств. Это предоставляет все обычные преимущества неизменяемости — возможность передачи значений методу, не беспокоясь об их изменении, возможность простого разделения данных между потоками и тому подобное.

По умолчанию свойства анонимных типов VB являются изменяемыми

Анонимные типы также доступны в Visual Basic 9 и последующих версиях. Однако по умолчанию их свойства изменяемые; если какое-то свойство должно быть неизменяемым, его понадобится объявить с модификатором `Key`. В вычислении хеш-значений и сравнениях на предмет эквивалентности участвуют только свойства, определенные как ключи (т.е. с модификатором `Key`). При преобразовании кода из одного языка в другой данный факт довольно легко упустить из виду.

К этому моменту мы в основном закончили с анонимными типами. Но по-прежнему осталось одно небольшое ухищрение, о котором следовало бы поговорить — сокращение для ситуации, довольно распространенной в LINQ.

8.5.3. Инициализаторы проекций

Все инициализаторы анонимных объектов, показанные до сих пор, имели списки пар "имя/значение" — `Name="Jon", Age=36` и т.д. Когда подобное происходит, я всегда применяю константы, т.к. это обусловлено небольшими примерами, однако в реальном коде часто нужно копировать свойства из существующего объекта. Временами требуются какие-то манипуляции с этими значениями, но часто простой копии вполне достаточно.

И снова, не прибегая к LINQ, трудно привести убедительные примеры сказанного, но давайте возвратимся к нашему классу `Person` и *предположим*, что имеется веская причина для преобразования коллекции экземпляров `Person` в похожую коллекцию, каждый элемент которой содержит только имя и флаг, позволяющий выяснить, является ли человек совершеннолетним. При наличии подходящей переменной `person` можно было бы воспользоваться примерно таким кодом:

```
new { Name = person.Name, IsAdult = (person.Age >= 18) }
```

Код работает, и для только одного свойства синтаксис установки имени (часть, выделенная полужирным) не так уж нескладен, но если бы пришлось копировать несколько свойств, код стал бы утомительным.

В C# 3 предлагается сокращение: если вы не укажете имя свойства, а только выражение для вычисления значения, то последняя часть выражения будет использоваться в качестве имени при условии, что это простое поле или свойство. Такая конструкция называется *инициализатором проекции*. Это означает, что предыдущий код можно переписать следующим образом:

```
new { person.Name, IsAdult = (person.Age >= 18) }
```

Вы обнаружите, что все части инициализатора анонимного объекта довольно часто будут инициализаторами проекций. Как правило, подобное происходит, когда некоторые свойства берутся из одного объекта, а некоторые — из другого, зачастую в виде части операции соединения. В любом случае, я забегаю вперед.

В листинге 8.6 показан предшествующий код в действии, в котором применяется метод `List<T>.ConvertAll()` и анонимный метод.

Листинг 8.6. Трансформация из `Person` в имя и признак совершеннолетия

```
List<Person> family = new List<Person>
{
  new Person { Name = "Holly", Age = 36 },
  new Person { Name = "Jon", Age = 36 },
  new Person { Name = "Tom", Age = 9 },
  new Person { Name = "Robin", Age = 6 },
  new Person { Name = "William", Age = 6 }
};
var converted = family.ConvertAll(delegate(Person person)
  { return new { person.Name, IsAdult = (person.Age >= 18) }; }
);
foreach (var person in converted)
{
  Console.WriteLine("{0} is an adult? {1}",
                    person.Name, person.IsAdult);
}
```

В дополнение к использованию инициализатора проекции для свойства `Name` в коде листинга 8.6 демонстрируется важность выведения типа делегата и анонимных методов. Без них не удалось бы сохранить строгую типизацию переменной `converted`, поскольку не было бы возможности указать, каким должен быть параметр типа `TOutput` делегата `Converter`. В итоге можно проходить по новому списку и обращаться к свойствам `Name` и `IsAdult`, как если бы применялся любой другой тип.

Не тратьте слишком много времени на обдумывание инициализаторов проекций прямо сейчас — важно знать, что они существуют, так что вы не запутаетесь, когда увидите их позже. На самом деле этот совет касается всего раздела об анонимных типах, поэтому, особо не вникая в детали, давайте посмотрим, зачем они вообще присутствуют.

8.5.4. В чем смысл существования анонимных типов?

Надеюсь, что в этот момент вы не ощущаете себя обманутыми, и на всякий случай сочувствую вам, если это все же так. Анонимные типы предлагают довольно сложное решение проблемы, с которой мы пока еще не сталкивались. Но я действительно уверен, что вы *видели* часть этой проблемы ранее.

Если вам когда-либо приходилось выполнять реальную работу с базами данных, то вы знаете, что далеко не всегда нужны все данные, доступные во всех строках, которые соответствуют указанному критерию запроса. Часто извлечение большего объема данных, чем необходимо, не составляет проблемы, но если требуются только 2 столбца из 50 существующих в таблице, то вряд ли имеет смысл идти на затраты по извлечению всех 50 столбцов, не так ли?

Та же самая проблема возникает и в коде, не имеющем отношения к взаимодействию с базами данных. Предположим, что имеется класс, который читает журнальный файл и

генерирует последовательность строк журнала с множеством полей. Хранение всей информации может требовать слишком большого и неэффективного расхода памяти, когда интересует лишь пара полей из журнала. Язык LINQ позволяет легко отфильтровывать такую информацию.

Но чем является результат этой фильтрации? Как сохранить некоторые данные и отбросить остальные? Каким образом без труда сохранить *производные* данные, которые не представлены напрямую в исходной форме? Как объединить порции данных, которые изначально могли быть не связаны преднамеренно либо имели отношение только к определенной ситуации? В сущности, необходим новый тип данных, но создавать такой тип вручную в каждой ситуации весьма утомительно, особенно когда доступны такие инструменты, как LINQ, которые делают остаток процесса настолько простым.

На рис. 8.5 показаны три элемента, которые превращают анонимные типы в мощное средство.

Рис. 8.5. Анонимные типы позволяют хранить только данные, необходимые в конкретной ситуации, в форме, подогнанной под эту ситуацию, не требуя утомительного написания нового типа каждый раз

Если вы обнаружите, что создаете тип, который используется только в одном методе и содержит только поля и тривиальные свойства, подумайте, не подойдет ли вместо него анонимный тип. Я допускаю, что в большинстве случаев, когда принимается решение о применении анонимных типов, то помочь справиться с проблемой мог бы также и язык LINQ.

Однако если выясняется, что во множестве мест используется одинаковая последовательность свойств для одной и той же цели, то имеет смысл обдумать создание обычного типа для такой цели, даже если он будет содержать только тривиальные свойства. Анонимные типы в буквальном смысле заражают неявной типизацией любой код, в котором применяются, что часто вполне нормально, но в отдельных случаях может создавать помехи. В частности, это означает, что не получится легко создать метод для возврата экземпляра такого типа в строго типизированной манере. Как и с предшествующими средствами, используйте анонимные типы, когда они по-настоящему упрощают работу с кодом, а не лишь по причине их новизны и модности.

8.6. Резюме

До чего на вид разнородный набор средств! Вы ознакомились с четырьмя средствами, которые довольно похожи, по крайней мере, с точки зрения синтаксиса: инициализаторы объектов, инициализаторы коллекций, неявно типизированные массивы и анонимные типы. Другие два средства — автоматические свойства и неявно типизированные локальные переменные — несколько отличаются. Подобным же образом большинство средств по отдельности могли быть полезными в C# 2, в то время как затраты на изучение неявно типизированных массивов и анонимных типов окупились, только когда в игру вступили остальные средства C# 3.

Так что же эти средства в действительности имеют общего? *Все они избавляют разработчика от утомительного кодирования*. Я уверен, что вам примерно так же, как и мне, не нравится писать тривиальные свойства или устанавливать множество свойств по одному за раз, используя локальную переменную — особенно при попытках построить коллекцию схожих объектов. Мало того, что новые средства C# 3 упрощают *написание* кода, они также облегчают его чтение, во всяком случае, при разумном применении.

В следующей главе будет рассмотрено новое крупное языковое средство наряду с функциональной возможностью инфраструктуры, для прямой поддержки которой оно предназначено. Если вы думаете, что анонимные методы сделали создание делегатов легким, то просто подождите немного, пока не увидите лямбда-выражения.

Глава 9

Лямбда-выражения и деревья выражений

В этой главе...

- Синтаксис лямбда-выражений
- Преобразования из лямбда-выражений в делегаты
- Классы инфраструктуры для деревьев выражений
- Преобразования из лямбда-выражений в деревья выражений
- Сущность деревьев выражений
- Изменения в выведении типов и распознавании перегруженных версий

В главе 5 было показано, что версия C# 2 намного облегчает использование делегатов, благодаря неявным преобразованиям групп методов, анонимным методам и вариантности возвращаемого типа и параметров. Этого вполне достаточно для значительного упрощения и улучшения читабельности подписки на события, но делегаты в C# 2 по-прежнему остаются слишком громоздкими, чтобы ими можно было пользоваться на постоянной основе. Чтение страницы кода, переполненной анонимными методами, требует немалых усилий, к тому же вряд ли возникнет желание начать регулярно размещать несколько анонимных методов в одиночном операторе.

Одним из фундаментальных строительных блоков LINQ является возможность создания конвейеров операций наряду с любым состоянием, требуемым этими операциями. Операции могут выражать все виды логики для обработки данных: фильтрацию, упорядочение, соединение разных источников данных и многое другое. Когда запросы LINQ выполняются внутри одного процесса, такие операции обычно представляются посредством делегатов.

Операторы, содержащие несколько делегатов, характерны при манипулировании данными с помощью LINQ to Objects[1], и *лямбда-выражения* в C# 3 делают все это возможным, не принося в жертву читабельность.

Китайская грамота

Термин *лямбда-выражение* взят из *лямбда-исчисления*, которое также записывается как λ-исчисление, где λ — греческая буква, произносимая как "лямбда". Это область математики и вычислительной техники, имеющая отношение к определению и применению функций. Она существует довольно давно, и стала основой функциональных языков, таких как ML. Хорошо уже то, что для использования лямбда-выражений в C# 3 знать лямбда-исчисление не обязательно.

Выполнение делегатов — лишь часть сюжета, связанного с LINQ. Для эффективной работы с базами данных и другими механизмами запросов необходимо другое представление операций в конвейере — способ трактовки кода как данных, которые можно исследовать программно. Затем логика внутри этих операций может быть трансформирована в другую форму, такую как обращение к веб-службе, запрос SQL или LDAP – все, что подходит в конкретной ситуации.

Несмотря на возможность построения представлений для запросов в отдельном API-интерфейсе, как правило, усложняется чтение кода и утрачивается немалая часть поддержки со стороны компилятора. Здесь лямбда-выражения опять спасают положение: помимо того, что они могут применяться для создания экземпляров делегатов, компилятор C# также способен трансформировать их в *деревья выражений* (структуры данных, представляющие логику лямбда-выражений), которые могут быть проанализированы в другом коде. Словом, лямбда-выражения — это идиоматический способ представления операций в конвейерах данных LINQ, но мы будем рассматривать по одному аспекту за раз, исследуя их довольно изолированно и постепенно охватывая всю технологию LINQ.

В этой главе мы обсудим оба способа использования лямбда-выражений, хотя пока что описание деревьев выражений будет относительно элементарным, т.к. код SQL создаваться не будет. Освоив эту теорию, вы будете достаточно хорошо знать лямбда-выражения и деревья выражений к моменту, когда мы доберемся до действительно впечатляющих вопросов в главе 12.

Последний раздел этой главы посвящен исследованию изменений выведения типов в C# 3, которые в основном обусловлены появлением лямбда-выражений с неявными типами параметров. Это напоминает обучение завязыванию шнурков: сам процесс нельзя назвать захватывающим, но без такого умения вы споткнетесь, как только начнете бежать.

Давайте приступим к выяснению, как выглядят лямбда-выражения. Мы начнем с анонимного метода и постепенно трансформируем его во все более и более короткие формы.

---

[1] В LINQ to Objects последовательности данных обрабатываются внутри того же самого процесса. В противоположность этому, поставщики вроде LINQ to SQL выгружают такую работу во внешние по отношению к процессу системы — например, базы данных.

9.1. Лямбда-выражения как делегаты

Во многих отношениях лямбда-выражения можно рассматривать как эволюцию анонимных методов из версии C# 2. Лямбда-выражения позволяют делать почти все то, что могут делать анонимные методы, и они практически всегда более читабельны и компактны[2]. В частности, поведение захваченных переменных в лямбда-выражениях точно совпадает с их поведением в анонимных методах. В наиболее явной форме между ними нет большой разницы, но в лямбда-выражениях доступно множество сокращений, которые делают их компактными в распространенных ситуациях. Подобно анонимным методам, лямбда-выражения имеют специальные правила преобразований. Тип выражения — это не тип делегата сам по себе, но он может быть преобразован в экземпляр делегата различными путями, как неявно, так и явно. Термин *анонимная функция* охватывает анонимные методы и лямбда-выражения и во многих случаях к ним обоим применяются те же самые правила преобразований.

Мы начнем с простого примера, изначально выраженного в виде анонимного метода. Будет создан экземпляр делегата, который принимает параметр `string` и возвращает `int` (длину строки). Сначала необходимо выбрать используемый тип делегата; к счастью, в .NET 3.5 имеется целое семейство обобщенных типов делегатов, которое помогает в решении этой задачи.

9.1.1. Подготовительные работы: знакомство с типами делегатов Func<...>

В пространстве имен `System` инфраструктуры .NET 3.5 доступно пять обобщенных типов делегатов `Func`. С `Func` не связано ничего особенного — просто удобно иметь ряд предварительно определенных обобщенных типов, которые способны обрабатывать многие ситуации. Сигнатура каждого делегата принимает от нуля до четырех параметров, типы которых указаны как параметры типов[3]. Во всех случаях последний параметр типа применяется для возвращаемого типа.

Ниже перечислены сигнатуры всех типов делегатов `Func` в .NET 3.5:

```
TResult Func<TResult>()
TResult Func<T,TResult>(T arg)
TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

Например, `Func<string,double,int>` является эквивалентом типа делегата в следующей форме:

```
public delegate int SomeDelegate(string arg1, double arg2)
```

Набор делегатов `Action<...>` предоставляет эквивалентную функциональность, когда нужен возвращаемый тип `void`. Форма делегата `Action` с одним параметром существовала еще в версии .NET 2.0, но остальные появились только в .NET 3.5.

---

[2] Есть одно средство, которое доступно в анонимных методах, но не в лямбда-выражениях — возможность игнорирования параметров. Подробные сведения об этом были предоставлены в разделе 5.4.3, но на практике отсутствие данного средства для лямбда-выражений не является столь уж значимой потерей.

[3] Возможно, вы помните, что в главе 6 мы уже сталкивались с версией, вообще не принимающей параметров (но имеющей один параметр типа).

Если четырех аргументов недостаточно, версия .NET 4 дает ответ: семейства делегатов `Action<...>` и `Func<...>` в ней расширены и принимают вплоть до 16 аргументов, поэтому `Func<T1,...,T16,TResult>` имеет целых 17 параметров типов. Они главным образом призваны помочь в поддержке исполняющей среды динамического языка (Dynamic Language Runtime — DLR), с которой вы ознакомитесь в главе 14, и вам вряд ли придется иметь с ними дело напрямую.

Для рассматриваемого примера необходим тип, который принимает параметр `string` и возвращает `int`, поэтому можно воспользоваться `Func<string,int>`.

9.1.2. Первая трансформация в лямбда-выражение

Теперь, когда известен тип делегата, можно применить анонимный метод для создания экземпляра делегата. В листинге 9.1 показано создание с последующим запуском экземпляра делегата, чтобы можно было удостовериться в его работе.

Листинг 9.1. Использование анонимного метода для создания экземпляра делегата

```
Func<string,int> returnLength;
returnLength = delegate (string text) { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

Код из листинга 9.1 выводит на консоль число 5, как и можно было ожидать. Я разделил объявление и присваивание `returnLength`, чтобы уместить код делегата в одну строку — так будет проще его отслеживать. Выражение анонимного метода выделено полужирным; это та часть, которая будет преобразована в лямбда-выражение.

Самая длинная форма лямбда-выражения выглядит так:

```
(список-явно-типизированных-параметров) => { операторы }
```

Часть `=>` представляет собой нововведение версии C# 3 и сообщает компилятору о том, что применяется лямбда-выражение. В большинстве случаев лямбда-выражения используются с типом делегата, который имеет возвращаемый тип, отличный от `void`, и когда возвращаемый результат отсутствует, синтаксис несколько менее интуитивно понятен. Это является еще одним свидетельством идиоматических изменений, произошедших в языке между версиями C# 1 и C# 3. В C# 1 делегаты обычно применялись для событий и редко что-то возвращали. В LINQ они обычно использовались как часть конвейера данных, получая входные данные и возвращая результат, который сообщает о том, что собой представляет спроецированное значение, соответствует ли элемент текущему фильтру и т.д.

Благодаря явным параметрам и операторам в фигурных скобках, эта версия выглядит очень похожей на анонимный метод. В листинге 9.2 приведен код, эквивалентный коду из листинга 9.1, но в нем применяется лямбда-выражение.

Листинг 9.2. Первое длинное лямбда-выражение, похожее на анонимный метод

```
Func<string,int> returnLength;
returnLength = (string text) => { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

И снова в коде выделено полужирным выражение, используемое для создания экземпляра делегата. При чтении лямбда-выражений удобно воспринимать часть => как "идет в", поэтому пример в листинге 9.2 можно было бы прочитать как "`text` идет в `text.Length`". Так как это единственная часть листинга, которая пока что интересует, с этого момента она будет показываться одна. Текст, выделенный в листинге 9.2 полужирным, можно заменить любыми лямбда-выражениями, перечисленными в этом разделе, и результат останется таким же.

Те же самые правила, которые управляют операторами `return` в анонимных методах, применимы и к лямбда-выражениям: нельзя возвращать значение из лямбда-выражения с возвращаемым типом `void`, а когда он не `void`, то каждый путь в коде должен возвращать значение совместимого типа[4]. Все это интуитивно понятно и редко создает препятствия.

Пока что мы не сократили код особо значительно, равно как и не улучшили каким-то образом читабельность. Давайте займемся применением сокращений.

9.1.3. Использование одиночного выражения в качестве тела

Форма, которую мы видели до сих пор, предусматривала использование полного блока кода для возврата значения. Это гибкое решение, т.к. можно иметь несколько операторов, реализовывать циклы, выполнять возврат из нескольких мест внутри блока и делать тому подобное — точно как в анонимных методах. Однако большую часть времени все тело можно легко выражать в одиночном выражении, значение которого представляет собой результат лямбда-выражения[5]. В таких случаях можно указывать только это выражение безо всяких фигурных скобок, операторов `return` или точек с запятыми. Тогда формат становится следующим:

```
(список-явно-типизированных-параметров) => выражение
```

В рассматриваемом примере это означает, что лямбда-выражение принимает такой вид:

```
(string text) => text.Length
```

Начинает выглядеть проще. А что теперь можно сказать о типе параметра? Компилятору уже известно, что экземпляры `Func<string,int>` принимают единственный параметр типа `string`, поэтому должна быть возможность указания только имени данного параметра.

9.1.4. Списки неявно типизированных параметров

Большую часть времени компилятор способен угадать типы параметров без их явного указания. В таких случаях лямбда-выражение может быть записано так:

```
(список-неявно-типизированных-параметров) => expression
```

Список неявно типизированных параметров — это просто список имен, разделенных запятыми, не содержащий типов. Указывать типы для одних параметров и не указывать для других не допускается — список в целом должен содержать либо явно типизированные, либо неявно типизированные параметры.

---

[4] Разумеется, пути в коде, генерирующие исключения, не обязаны возвращать какие-то значения, и это также касается обнаруживаемых бесконечных циклов.

[5] Этот синтаксис можно применять также и для делегата с возвращаемым типом `void`, если необходим только один оператор. По существу отбрасывается точка с запятой и фигурные скобки.

Кроме того, при наличии параметров `out` или `ref` придется использовать явную типизацию. С нашим примером в этом плане все в порядке, поэтому лямбда-выражение становится следующим:

```
(text) => text.Length
```

Теперь оно довольно короткое. Осталось не так много, от чего можно было бы избавиться. Хотя круглые скобки кажутся лишними.

9.1.5. Сокращение для единственного параметра

Когда лямбда-выражению необходим только один параметр, и он может быть неявно типизирован, C# 3 позволяет опускать круглые скобки, так что форма выглядит так:

```
имя-параметра => выражение
```

Окончательная форма рассматриваемого лямбда-выражения приобретает следующий вид:

```
text => text.Length
```

Вас может интересовать, почему с лямбда-выражениями связано настолько много частных случаев — нигде больше в языке не играет роли, например, принимает метод один параметр или больше. На самом деле то, что звучит как очень частный случай, фактически оказывается *чрезвычайно* распространенной ситуацией, а улучшения в плане читабельности, получаемые по причине устранения круглых скобок из списка параметров, могут быть значительными, когда в небольшом фрагменте кода присутствует множество лямбда-выражений.

Полезно отметить, что при желании можно поместить в круглые скобки целое лямбда-выражение — в точности как другие выражения. Иногда это способствует лучшей читабельности, скажем, когда лямбда-выражение присваивается переменной или свойству — иначе символ = может вызвать путаницу, по крайней мере, сначала. Большую часть времени все должно быть полностью читабельным вообще безо всякого дополнительного синтаксиса. В листинге 9.3 это демонстрируется в контексте первоначально кода.

Листинг 9.3. Лаконичное лямбда-выражение

```
Func<string,int> returnLength;
returnLength = text => text.Length;

Console.WriteLine(returnLength("Hello"));
```

На первых порах код в листинге 9.3 может сбивать с толку при чтении, подобно тому, как анонимные методы выглядят странными для многих разработчиков до тех пор, пока они не начнут пользоваться ими. В обычных обстоятельствах вы объявляете переменную и присваиваете ей значение в одном и том же выражении, делая его еще яснее. Однако, привыкнув к лямбда-выражениям, вы сможете оценить, насколько они лаконичны. Трудно представить себе более короткий и ясный способ создания экземпляра делегата[6].

[6] Это не значит, что сделать такое невозможно. Некоторые языки позволяют определять замыкания в виде простых блоков кода с "магическим" именем переменной для представления общего случая с единственным параметром.

Можно было бы вдобавок изменить имя переменной `text` на что-то вроде `x`, и в LINQ это часто удобно, но более длинные имена предоставляют читателю полезную информацию.

Трансформация, которая была описана на нескольких страницах, кратко проиллюстрирована на рис. 9.1; это позволяет легко оценить, насколько много излишнего синтаксиса было устранено.

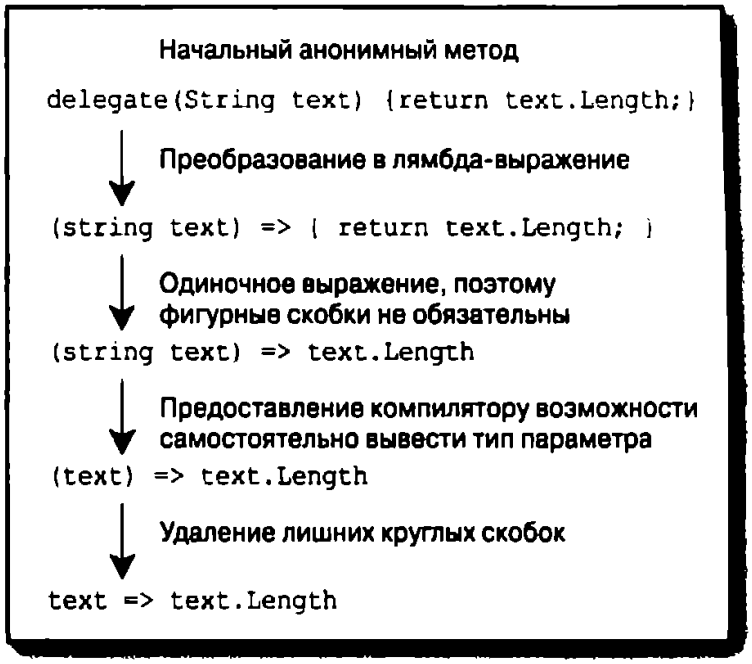

Рис. 9.1. Сокращения в синтаксисе лямбда-выражений

Решение, использовать ли короткую форму для тела лямбда-выражения, указывая только выражение вместо полного блока, совершенно не зависит от решения относительно применения явных или неявных параметров. В рассмотренном примере был избран путь сокращения лямбда-выражения, но можно было бы начать с превращения параметров в неявные. Когда вы освоитесь с лямбда-выражениями, то вообще перестанете об этом думать, а будете свободно записывать кратчайшую форму из числа доступных.

Функции высшего порядка

Тело лямбда-выражения способно само содержать лямбда-выражение, и это вполне может сбивать с толку. В качестве альтернативы параметром лямбда-выражения может быть другой делегат, что в равной степени плохо. Оба примера относятся к *функциям высшего порядка*. Если вам нравится сталкиваться с запутанными ситуациями, загляните в загружаемый исходный код, сопровождающий эту книгу. Данный подход распространен в функциональном программировании и иногда оказывается удобным. Просто он требует определенной доли настойчивости в обретении правильного образа мыслей.

До сих пор мы имели дело только с одиночным лямбда-выражением, приводя его в различные формы. Давайте рассмотрим несколько примеров, чтобы конкретизировать вопросы до того, как переходить к деталям.

9.2. Простые примеры использования типа List<T> и событий

После того, как мы приступим к исследованию расширяющих методов в главе 10, мы будем применять лямбда-выражения постоянно. Но на данный момент наилучшие примеры можно представить посредством типа `List<T>` и обработчиков событий. Мы начнем со списков, и ради краткости будем использовать автоматически реализуемые свойства, неявно типизированные локальные переменные и инициализаторы коллекций. Затем мы будем вызывать методы, принимающие параметры в форме делегатов, конечно же, создавая делегаты с помощью лямбда-выражений.

9.2.1. Фильтрация, сортировка и действия на списках

Вспомните метод `FindAll()` в типе `List<T>` — он принимает предикат `Predicate<T>` и возвращает новый список с элементами исходного списка, соответствующими предикату. Метод `Sort()` принимает экземпляр `Comparison<T>` и сортирует список указанным образом. Наконец, метод `ForEach()` принимает действие `Action<T>` и выполняет его над каждым элементом. Для предоставления экземпляра делегата каждому из этих методов в листинге 9.4 применяются лямбда-выражения. В качестве примера данных выступают названия и годы выпуска в прокат разнообразных фильмов. Сначала на консоль выводится исходный список, затем создается и выводится отфильтрованный список, содержащий только старые фильмы, и, в конце концов, исходный список сортируется по названию фильма и снова выводится на консоль. (Небезынтересно прикинуть, насколько большой объем кода пришлось бы написать для решения этой задачи в C# 1.)

Листинг 9.4. Манипулирование списком фильмов с использованием лямбда-выражений

```
class Film
{
  public string Name { get; set; }
  public int Year { get; set; }
}
...
var films = new List<Film>
{
  new Film { Name = "Jaws", Year = 1975 },
  new Film { Name = "Singing in the Rain", Year = 1952 },
  new Film { Name = "Some like it Hot", Year = 1959 },
  new Film { Name = "The Wizard of Oz", Year = 1939 },
  new Film { Name = "It's a Wonderful Life", Year = 1946 },
  new Film { Name = "American Beauty", Year = 1999 },
  new Film { Name = "High Fidelity", Year = 2000 },
  new Film { Name = "The Usual Suspects", Year = 1995 }
};
Action<Film> print = ←❶ Создание многократно используемого делегата для вывода списка на консоль
  film => Console.WriteLine("Name={0}, Year={1}",
                            film.Name, film.Year);
films.ForEach(print);                          ←❷ Вывод на консоль исходного списка
films.FindAll(film => film.Year < 1960)        ←❸ Создание отфильтрованного списка
    .ForEach(print);
films.Sort((f1, f2) => f1.Name.CompareTo(f2.Name));  ←❹ Сортировка исходного списка
films.ForEach(print);
```

Первая часть листинга 9.4 отвечает за подготовку данных. Именованный тип применяется в коде только для простоты — анонимный тип в этом конкретном случае означал бы появление еще нескольких препятствий, которые пришлось бы преодолевать.

Перед использованием построенного списка создается экземпляр делегата ❶, который будет применяться для вывода на консоль элементов списка. Этот экземпляр делегата используется три раза, и именно потому для его хранения была создана переменная вместо того, чтобы каждый раз применять отдельное лямбда-выражение. Он просто выводит на консоль одиночный элемент, но передавая экземпляр делегата методу `List<T>.ForEach()`, можно отобразить на консоли целый список. Следует отметить один тонкий, однако важный момент — точка с запятой в конце этого оператора является частью оператора присваивания, а не лямбда-выражения. Если бы то же самое лямбда-выражение использовалось в качестве аргумента при вызове метода, то сразу после `Console.WriteLine(...)` точка с запятой бы отсутствовала.

Первым на консоль выводится исходный список безо всяких модификаций ❷. Затем в списке находятся и выводятся на консоль все фильмы, снятые до 1960 года ❸. Это делается с помощью еще одного лямбда-выражения, которое выполняется для каждого фильма в списке — оно только определяет, должен ли конкретный фильм быть включен в отфильтрованный список. В исходном коде данное лямбда-выражение указано как аргумент метода, но на самом деле компилятор создает метод, подобный показанному ниже:

```
private static bool SomeAutoGeneratedName(Film film)
{
  return film.Year < 1960;
}
```

Фактически вызов метода `FindAll()` сводится к следующему:

```
films.FindAll(new Predicate<Film>(SomeAutoGeneratedName))
```

Поддержка лямбда-выражений похожа на поддержку анонимных методов в C# 2; все основано на мастерстве компилятора. (На самом деле в этом случае компилятор Microsoft оказывается даже более интеллектуальным — он полагает, что может выиграть от повторного использования экземпляра делегата, если код обратится к нему еще раз, поэтому кеширует его.)

Сортировка списка также осуществляется с применением лямбда-выражения ❹, которое сравнивает два элемента, представляющих фильмы, по их названиям. Должен признаться, что явный вызов метода `CompareTo()` выглядит немного неуклюже. В следующей главе вы увидите, каким образом расширяющий метод `OrderBy()` позволяет выразить упорядочение в более компактной манере.

Давайте обратимся к другому примеру, на этот раз демонстрирующему использование лямбда-выражений для обработки событий.

9.2.2. Регистрация внутри обработчика событий

Возвращаясь к главе 5, в разделе 5.9 был представлен простой способ применения анонимных методов для регистрации в журнале факта возникновения событий, но компактный синтаксис можно было использовать только потому, что потеря информации, передаваемой в параметрах, тогда совершенно не беспокоила. А что, если понадобится регистрировать как природу события, так и данные о его отправителе и аргументах? Лямбда-выражения позволяют делать это компактным способом, как демонстрируется в листинге 9.5.

Листинг 9.5. Регистрация событий в журнале с применением лямбда-выражений

```
static void Log(string title, object sender, EventArgs e)
{
  Console.WriteLine("Event: {0}", title);                  // Событие
  Console.WriteLine(" Sender: {0}", sender);               // Отправитель
  Console.WriteLine(" Arguments: {0}", e.GetType());       // Аргументы
  foreach (PropertyDescriptor prop in
         TypeDescriptor.GetProperties(e))
  {
    string name = prop.DisplayName;
    object value = prop.GetValue(e);
    Console.WriteLine("    {0}={1}", name, value);
  }
}
...
Button button = new Button { Text = "Click me" };
button.Click     += (src, e) => Log("Click", src, e);
button.KeyPress  += (src, e) => Log("KeyPress", src, e);
button.MouseClick += (src, e) => Log("MouseClick", src, e);

Form form = new Form { AutoSize = true, Controls = { button } };
Application.Run(form);
```

В листинге 9.5 лямбда-выражения используются для передачи имени события и *параметров* методу `Log()`, который фиксирует в журнале детали, связанные с событием. В журнал не заносятся подробные сведения об источнике события кроме данных, возвращаемых его переопределенным методом `ToString()`, поскольку с элементами управления связано слишком много информации. Однако с помощью рефлексии по дескрипторам свойств отображаются детали переданного экземпляра `EventArgs`.

Ниже показан пример вывода, получаемого в результате щелчка на кнопке:

```
Event: Click
 Sender: System.Windows.Forms.Button, Text: Click me
 Arguments: System.Windows.Forms.MouseEventArgs
   Button=Left
   Clicks=1
   X=53
   Y=17
   Delta=0
   Location={X=53,Y=17}
Event: MouseClick
 Sender: System.Windows.Forms.Button, Text: Click me
 Arguments: System.Windows.Forms.MouseEventArgs
   Button=Left
   Clicks=1
   X=53
   Y=17
   Delta=0
   Location={X=53,Y=17}
```

Конечно, все это *можно* было бы сделать и без лямбда-выражений, но за счет применения лямбда-выражений код получился намного более компактным.

После демонстрации преобразования лямбда-выражений в экземпляры делегатов самое время взглянуть на деревья выражений, которые представляют лямбда-выражения в виде данных, а не кода.

9.3. Деревья выражений

Идея *кода в форме данных* далеко не нова, но она не часто использовалась в популярных языках программирования. Вы могли бы привести в качестве довода то, что эта концепция применяются всеми программами .NET, т.к. код IL трактуется JIT-компилятором как данные, которые затем преобразуются в машинный код для выполнения центральным процессором. Тем не менее, это глубоко скрыто, и несмотря на существование библиотек для программного манипулирования кодом IL, используются они не особенно часто.

Деревья выражений в .NET 3.5 предлагают абстрактный способ представления определенного кода в виде дерева объектов. Он похож на модель CodeDOM, но функционирует на несколько более высоком уровне. Главной областью применения деревьев выражений является LINQ, и позже в этом разделе будет показано, насколько деревья выражений важны для LINQ в целом.

В C# 3 предоставляется встроенная поддержка преобразования лямбда-выражений в деревья выражений, но перед тем, как раскрывать ее, давайте изучим, почему они вписываются в инфраструктуру .NET Framework безо всяких трюков со стороны компилятора.

9.3.1. Построение деревьев выражений программным образом

Деревья выражений не являются чем-то мистическим, хотя некоторые случаи их использования выглядят подобными магии. Как следует из названия, они представляют собой деревья объектов, в которых каждый узел сам является выражением. Разные типы выражений представляют различные операции, которые можно выполнять в коде: бинарные операции, такие как сложение; унарные операции наподобие определения длины массива; вызовы методов; обращения к конструкторам и т.д.

Пространство имен `System.Linq.Expressions` содержит разнообразные классы, которые представляют выражения. Все они унаследованы от класса `Expression`, который является абстрактным и по большей части состоит из статических фабричных методов, предназначенных для создания экземпляров других классов выражений. Тем не менее, класс `Expression` предлагает два свойства.

- Свойство `Type` представляет тип .NET вычисляемого выражения – его можно рассматривать как возвращаемый тип. Например, типом выражения, которое извлекает свойство `Length` строки, будет `int`.
- Свойство `NodeType` возвращает вид выражения в форме члена перечисления `ExpressionType` со значениями вроде `LessThan`, `Multiply` и `Invoke`. Придерживаясь того же самого примера, в `myString.Length` часть, отвечающая за доступ к свойству, имела бы тип узла `MemberAccess`.

Существует множество классов, производных от `Expression`, и некоторые из них могут иметь множество узлов разных типов. Например, `BinaryExpression` представляет любую операцию с двумя операндами: арифметическую, логическую, сравнения, индексации массива и т.д. Именно здесь становится важным свойство `NodeType`, т.к. оно позволяет отделять различные виды выражений, которые представлены одним и тем же классом.

Я не намерен раскрывать здесь все классы выражений или типы узлов — их слишком много и они хорошо документированы в MSDN (http://mng.bz/3vW3). Вместо этого будет предложено общее описание того, что можно делать с деревьями выражений.

Давайте начнем с создания одного из простейших деревьев выражений, суммирующего две целочисленных константы. Код в листинге 9.6 создает дерево выражения для представления 2+3.

Листинг 9.6. Простое дерево выражения, суммирующего числа 2 и 3

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);

Console.WriteLine(add);
```

Запуск кода из листинга 9.6 приведет к получению вывода (2 + 3), который указывает на то, что различные классы выражений переопределяют метод ToString() для обеспечения вывода, воспринимаемого человеком. На рис. 9.2 изображено дерево, сгенерированное кодом.

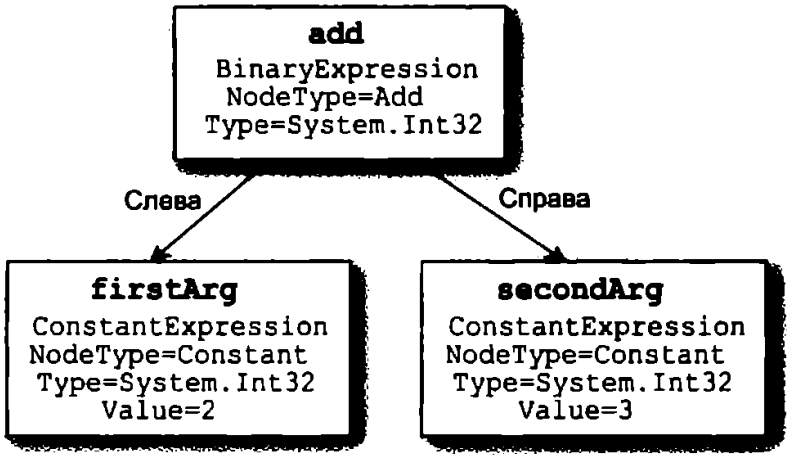

Рис. 9.2. Графическое представление дерева выражения, созданного кодом из листинга 9.6

Полезно отметить, что концевые выражения в коде создаются первыми: выражения строятся снизу вверх. Это обусловлено тем фактом, что выражения являются неизменяемыми — после того, как выражение создано, оно никогда не будет изменяться, поэтому выражения можно кешировать и многократно использовать по своему усмотрению.

Теперь, когда дерево выражения построено, наступило время его выполнить.

9.3.2. Компиляция деревьев выражений в делегаты

В число типов, производных от Expression, входит LambdaExpression. В свою очередь, от LambdaExpression унаследован обобщенный класс Expression<TDelegate>. Все это немного запутывает, поэтому с целью прояснения на рис. 9.3 показана иерархия типов.

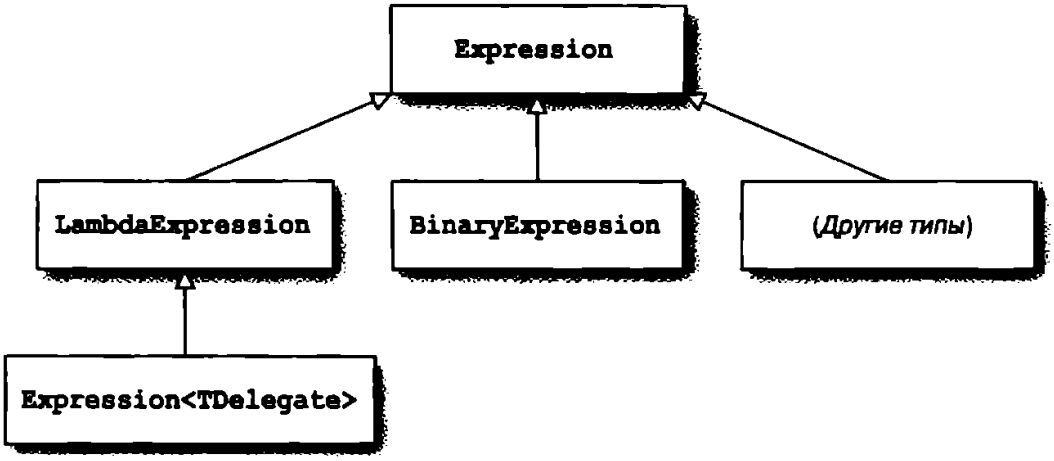

Рис. 9.3. Иерархия типов от `Expression<TDelegate>` до `Expression`

Разница между `Expression` и `Expression<TDelegate>` в том, что обобщенный класс является статически типизированным для отражения вида представляемого выражения в терминах возвращаемого типа и параметров. Очевидно, это выражается параметром типа `TDelegate`, который должен быть типом делегата. Например, простое выражение сложения не принимает параметров и возвращает целочисленное значение, что соответствует сигнатуре `Func<int>`, поэтому для представления такого выражения в статически типизированной манере можно было бы применить `Expression<Func<int>>`. Это делается с использованием метода `Expression.Lambda()`, который имеет несколько перегруженных версий. В рассматриваемых примерах применяется обобщенный метод, который использует параметр типа для указания типа представляемого делегата. С альтернативами можно ознакомиться в MSDN.

Итак, какой смысл делать все это? В классе `LambdaExpression` определен метод `Compile()`, который создает делегат подходящего типа; в `Expression<TDelegate>` имеется другой метод с таким же именем, но он статически типизирован для возвращения делегата типа `TDelegate`. Затем этот делегат может быть выполнен обычным образом, как если бы он был создан с применением нормального метода или любыми другими средствами. Сказанное иллюстрируется в листинге 9.7 на примере того же самого выражения, что и ранее.

Листинг 9.7. Компиляция и выполнение дерева выражения

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);

Func<int> compiled = Expression.Lambda<Func<int>>(add).Compile();
Console.WriteLine(compiled());
```

Код в листинге 9.7 является, вероятно, одним из самых запутанных путей вывода на консоль числа 5, какие только можно представить. В то же время он весьма выразителен. Здесь программно создаются логические блоки и представляются в виде обычных объектов, после чего у инфраструктуры запрашивается их компиляция в действительный код, который может быть выполнен. Возможно, вам никогда не придется использовать деревья выражений подобным образом или даже вообще строить их программно, но это дает полезную справочную информацию, которая поможет лучше понять функционирование LINQ.

Как упоминалось в начале этого раздела, деревья выражений не слишком далеко ушли от модели CodeDOM — к примеру, инструмент Snippy компилирует и выполняет код C#, который вводится как простой текст. Однако между CodeDOM и деревьями выражений существуют два значительных отличия.

Во-первых, в .NET 3.5 деревья выражений обладали возможностью представления только одиночных выражений. Они не были предназначены для целых классов, методов или даже просто операторов. В .NET 4 кое-что в этом плане изменилось, и в данной версии деревья выражений применяются для поддержки динамической типизации – теперь можно создавать блоки, присваивать значения переменным и т.д. Но по сравнению с CodeDOM по-прежнему существуют значительные ограничения.

Во-вторых, в C# деревья выражений поддерживаются прямо на уровне языка через лямбда-выражения. Давайте посмотрим на это прямо сейчас.

9.3.3. Преобразование лямбда-выражений C# в деревья выражений

Как уже было указано, лямбда-выражения могут быть преобразованы в соответствующие экземпляры делегатов, либо неявно, либо явно. Это не единственное доступное преобразование. Компилятору можно также предложить построить дерево выражения из заданного лямбда-выражения, создавая экземпляр `Expression<TDelegate>` во время выполнения. Например, в листинге 9.8 показан намного более короткий путь создания выражения для "возврата числа 5", его компиляции и обращения к результирующему делегату.

Листинг 9.8. Использование лямбда-выражений для создания деревьев выражений

```
Expression<Func<int>> return5 = () => 5;
Func<int> compiled = return5.Compile();
Console.WriteLine(compiled());
```

Часть `() => 5` в первой строке листинга 9.8 — это лямбда-выражение. Никакие приведения не требуются, поскольку компилятор может проверить все самостоятельно. Вместо `5` можно было бы написать `2+3`, но компилятор применил бы к этому сложению оптимизацию, заменив его суммой. Важный момент здесь в том, что лямбда-выражение было преобразовано в дерево выражения.

Ограничения преобразований

Не *все* лямбда-выражения могут быть преобразованы в деревья выражений. Преобразовать лямбда-выражение с блоком операторов (даже если это один оператор `return`) в дерево выражения не получится — оно должно иметь форму одиночного выражения, и это выражение не может содержать присваивания. Такое ограничение применимо и в версии .NET 4 с ее расширенными возможностями в отношении деревьев выражений. Несмотря на то что это самые распространенные ограничения, существуют не только они одни — приводить здесь полный список не имеет смысла, т.к. подобного рода ситуации возникают очень редко. О наличии проблемы с преобразованием вы узнаете на этапе компиляции.

Давайте рассмотрим более сложный пример, чтобы увидеть, как все работает, особенно то, что касается параметров. На этот раз будет написан предикат, который принимает две строки и проверяет, находится ли первая строка в начале второй. Код оказывается простым, когда он представлен в виде лямбда-выражения (листинг 9.9).

Листинг 9.9. Демонстрация более сложного дерева выражения

```
Expression<Func<string, string, bool>> expression =
  (x, y) => x.StartsWith(y);

var compiled = expression.Compile();

Console.WriteLine(compiled("First", "Second"));
Console.WriteLine(compiled("First", "Fir"));
```

Это дерево выражения сложнее само по себе, особенно к тому времени, как оно преобразуется в экземпляр `LambdaExpression`. В листинге 9.10 показано, как его можно было бы построить в коде.

Листинг 9.10. Построение выражения с вызовом метода в коде

```
MethodInfo method = typeof(string).GetMethod
  ("StartsWith", new[] { typeof(string) });
var target = Expression.Parameter(typeof(string), "x");
var methodArg = Expression.Parameter(typeof(string), "y");
Expression[] methodArgs = new[] { methodArg };

Expression call =
  Expression.Call(target, method, methodArgs);

var lambdaParameters = new[] { target, methodArg };
var lambda =
   Expression.Lambda<Func<string, string, bool>>
  (call, lambdaParameters);

var compiled = lambda.Compile();

Console.WriteLine(compiled("First", "Second"));
Console.WriteLine(compiled("First", "Fir"));
```

❶ Построение частей вызова метода

❷ Создание выражения `CallExpression` из частей

❸ Преобразование в `LambdaExpression`

Как видите, объем кода в листинге 9.10 значительно превышает версию с лямбда-выражением C#. Но при этом код делает более очевидным то, что в точности происходит в дереве, и показывает, как привязываются параметры.

Сначала определяется все, что необходимо знать о вызове метода, который формирует тело финального выражения ❶: цель метода (строка, на которой вызывается `StartsWith()`); сам метод (как `MethodInfo`); и список аргументов (а этом случае содержащий всего один элемент). Так получилось, что цель и аргумент нашего метода являются параметрами, передаваемыми выражению, однако они могли быть другими типами выражений — константами, результатами других вызовов методов, значениями свойств и т.д.

После построения вызова метода как выражения ❷ нужно преобразовать его в лямбда-выражение ❸, по пути привязав параметры. В качестве информации для вызова метода повторно используются те же самые ранее созданные значения `ParameterExpression`: порядок, в котором они были указаны при создании лямбда-выражения — это порядок, в котором они будут выбираться, когда в конечном итоге вызывается делегат.

На рис. 9.4 окончательное дерево выражения представлено графически. По правде говоря, хотя это по-прежнему называется деревом выражения, факт повторного использования выражений параметров (и так должно делаться — создание нового выражения параметра с тем же самым именем и попытка привязки параметров подобным образом привела бы к генерации исключения во время выполнения) означает, что в строгом смысле оно действительно деревом не является.

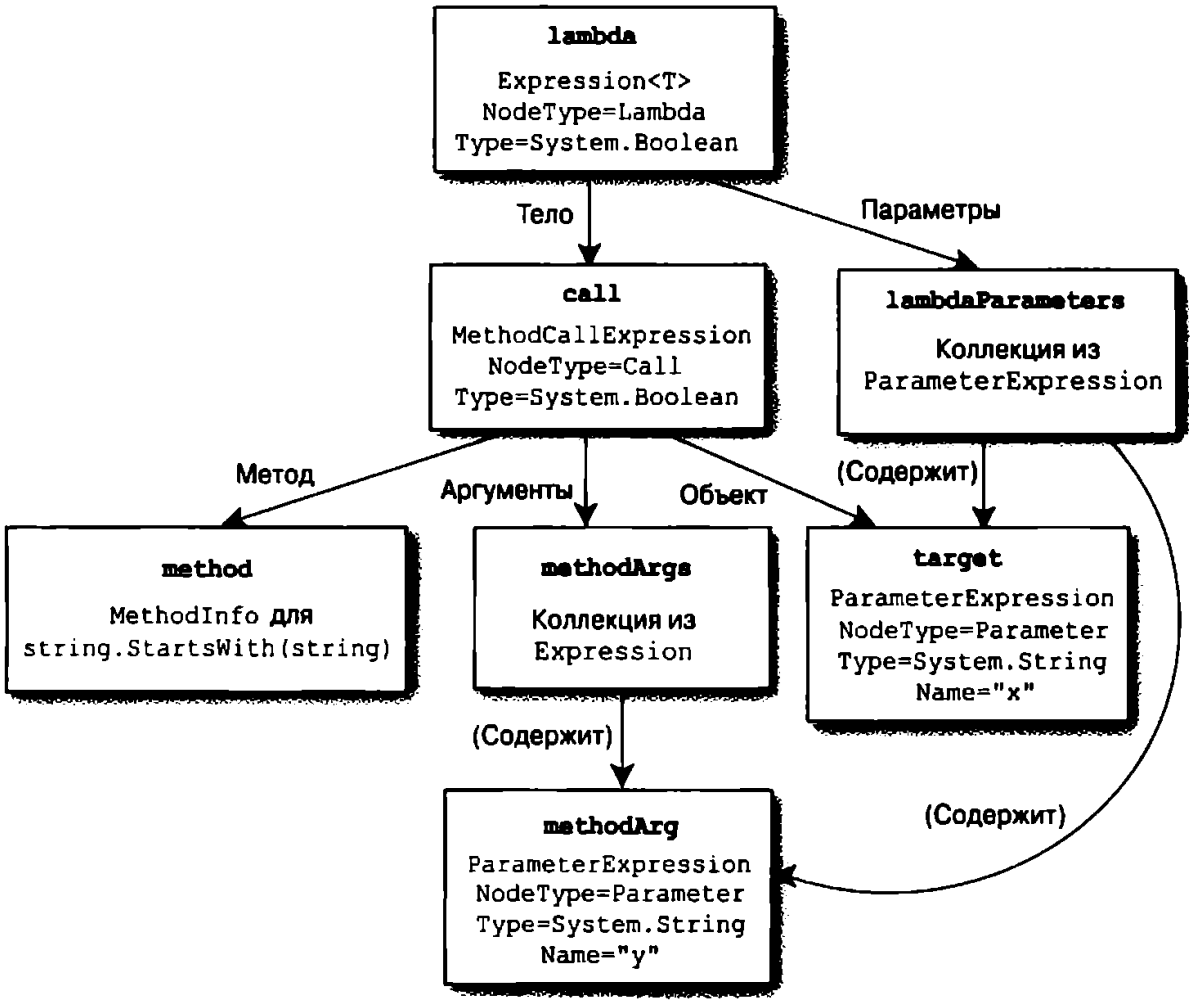

Рис. 9.4. Графическое представление дерева выражения, которое вызывает метод и использует параметры из лямбда-выражения

Бегло оценив сложность диаграммы на рис. 9.4 и кода в листинге 9.10 без попытки обратиться к деталям, можно было бы подумать, что здесь делается что-то действительно трудное, тогда как фактически это всего лишь единственный вызов метода. Представьте, как могло бы выглядеть дерево выражения для по-настоящему сложного выражения — и затем выразите благодарность за то, что версия C# 3 позволяет создавать деревья выражений из лямбда-выражений!

В качестве еще одного способа исследования той же идеи среды Visual Studio 2010 и Visual Studio 2012 предоставляют встроенный визуализатор для деревьев выражений[7]. Это может оказаться удобным, если вы пытаетесь найти способ построения дерева выражения в коде и хотите получить представление о том, как оно должно выглядеть.

[7] Если вы работаете с Visual Studio 2008, то можете загрузить из сети MSDN код примера для построения похожего визуализатора (`http://mng.bz/g6xd`), но очевидно проще воспользоваться визуализатором, входящим в состав Visual Studio, при наличии последующих версий.

Просто напишите лямбда-выражение, которое делает то, что требуется, с фиктивными данными, активизируйте визуализатор внутри отладчика и затем на основе предоставленной информации обдумайте, как построить аналогичное дерево в реальном коде. Визуализатор опирается на изменения, появившиеся в .NET 4, поэтому с проектами для целевой версии .NET 3.5 он не работает. На рис. 9.5 показано диалоговое окно визуализатора для примера с методом `StartsWith()`.

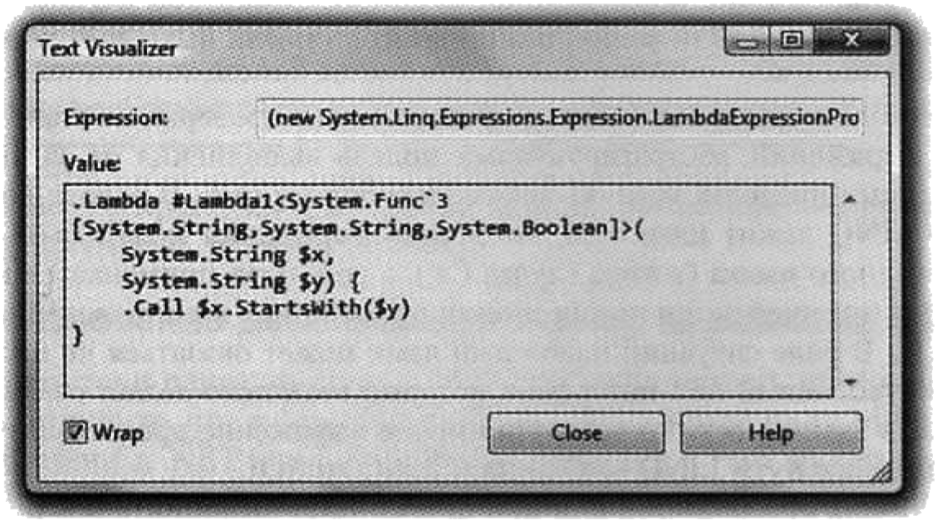

Рис. 9.5. Применение визуализатора деревьев выражений, доступного в отладчике

Части `.Lambda` и `.Call` в информации из визуализатора соответствуют вызовам методов `Expression.Lambda()` и `Expression.Call()`, а `$x` и `$y` — выражениям параметров. Результаты визуализации будут одинаковыми независимо от того, построено дерево выражения явно в коде или с использованием преобразования из лямбда-выражения.

Следует отметить один небольшой момент — хотя компилятор C# строит деревья выражений в скомпилированном коде аналогично показанному в листинге 9.10, в его арсенале имеется одно сокращение: ему не приходится применять обычную рефлексию, чтобы получить `MethodInfo` для `string.StartsWith()`. Взамен компилятор использует метод, эквивалентный операции `typeof`. Это доступно только в IL, а не в самом C#, и та же самая операция применяется для создания экземпляров делегатов из групп методов.

Разобравшись со связью между деревьями выражений и лямбда-выражениями, давайте кратко рассмотрим, по каким причинам они настолько удобны.

9.3.4. Деревья выражений являются основой LINQ

Без лямбда-выражений деревья выражений обладали бы относительно небольшой ценностью. Они могли выступать в качестве альтернативы CodeDOM в случаях, когда требовалось только моделирование одиночного выражения вместо целых операторов, методов, типов и тому подобного, но их преимущества оставались по-прежнему ограниченными.

В определенном смысле справедливо также и обратное утверждение: без деревьев выражений лямбда-выражения безусловно были бы *менее* полезными. Наличие более компактного способа создания экземпляров делегатов всегда приветствовалось, и сдвиг в сторону более функциональной формы разработки по-прежнему жизнеспособен. Как будет показано в следующей главе, лямбда-выражения особенно эффективны в сочетании с расширяющими методами, но присутствие еще и деревьев выражений делает ситуацию намного более интересной.

Что вы получите за счет комбинирования лямбда-выражений, деревьев выражений и расширяющих методов? Ответ: практически "языковую сторону LINQ". Дополнительный синтаксис, который вы увидите в главе 11, представляет собой добавочное преимущество, однако история была бы захватывающей даже только с тремя ингредиентами, упомянутыми выше. В течение долгого времени можно было иметь *либо* аккуратную проверку на этапе компиляции, *либо* возможность поручения другой платформе запуска определенного кода, обычно выражаемого в виде текста (запросы SQL являются самым очевидным примером). Тем не менее, оба средства не могли быть доступными одновременно.

Сочетая лямбда-выражения, которые предоставляют проверки на этапе компиляции, с деревьями выражений, абстрагирующими модель выполнения от заданной логики, можно в разумных пределах извлечь лучшее из обоих миров. В основе внепроцессных поставщиков LINQ лежит идея того, что дерево выражения может быть получено из знакомого исходного языка (в этом случае C#), а результат применяется как промежуточный формат, который затем преобразуется в машинный язык целевой платформы – например, SQL. В ряде ситуаций машинный язык может оказаться не настолько простым, как низкоуровневый API-интерфейс, который возможно делает разные обращения к веб-службам в зависимости от того, что именно выражение представляет. На рис. 9.6 показаны различные пути LINQ to Objects и LINQ to SQL.

В одних случаях преобразование может попробовать выполнить *всю* логику на целевой платформе, тогда как в других случаях могут использоваться возможности компиляции деревьев выражений для выполнения некоторых выражений локально и в других местах. Мы взглянем на детали этого шага преобразования в главе 12, но вы должны помнить об этой конечной цели во время исследования расширяющих методов и синтаксиса LINQ в главах 10 и 11.

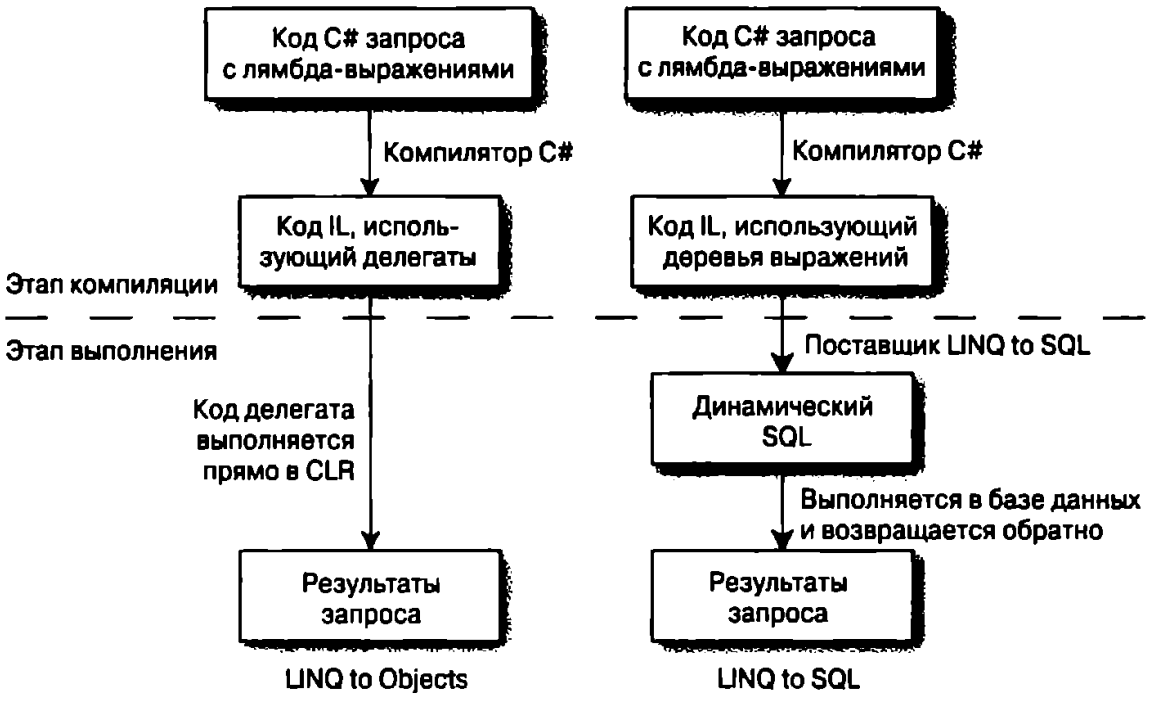

Рис. 9.6. Как LINQ to Objects, так и LINQ to SQL начинают с кода C# и в конце получают результаты запроса. Возможность выполнения кода удаленным образом появляется благодаря деревьям выражений

Не вся проверка может делаться компилятором

Когда деревья выражений анализируются преобразователем определенного вида, некоторые случаи обычно должны отбрасываться. Например, несмотря на возможность преобразования вызова метода `string.StartsWith()` в сходное SQL-выражение, вызов метода `string.IsInterned` в среде базы данных не имеет смысла. Деревья выражений обеспечивают высокую безопасность на этапе компиляции, но компилятор способен проверить только тот факт, что лямбда-выражение может быть преобразовано в допустимое дерево выражения; он не в состоянии гарантировать, что это дерево выражения окажется пригодным во всех возможных случаях его применения.

Хотя наиболее распространенные случаи использования деревьев выражений относятся к LINQ, так бывает далеко не всегда...

9.3.5. Использование деревьев выражений за рамками LINQ

Бьярне Страуструп однажды сказал: "Я бы не хотел строить инструмент, который мог бы делать только то, что я способен вообразить". Несмотря на то что деревья выражений были введены в .NET главным образом для LINQ, с того времени и сообщество разработчиков, и проектировщики из Microsoft нашли им другие применения. Этот раздел далек от полноты, но наверняка даст вам несколько идей относительно того, в чем могут помочь деревья выражений.

Оптимизация исполняющей среды динамического языка

Исполняющая среда динамического языка (DLR) будет подробно рассматриваться в главе 14, когда речь пойдет о динамической типизации в C#, однако деревья выражений являются основной частью этой архитектуры. Деревья выражений обладают тремя характеристиками, которые делают их привлекательными для DLR.

- Они неизменяемы, поэтому их можно безопасно кешировать.
- Они объединяемы, так что можно формировать сложное поведение на основе простых строительных блоков.
- Они могут быть скомпилированы в делегаты, которые являются JIT-скомпилированными в машинный код как обычные делегаты.

Среда DLR должна принимать решения о том, как обрабатывать разнообразные выражения, в которых смысл мог быть тонко изменен, на основе различных правил. Деревья выражений позволяют этим правилам (и результатам) быть трансформированными в код, который близок к тому, что вы писали бы вручную, если бы знали все правила и результаты, показанные до сих пор. Это мощная концепция, которая обеспечивает неожиданно быстрое выполнение динамического кода.

Защищенные от рефакторинга ссылки на члены

В разделе 9.3.3 я упоминал, что компилятор может выдавать ссылки на значения `MethodInfo` почти так, как это делает операция `typeof`. К сожалению, C# не обладает такой способностью. Это означает, что единственный способ сообщить фрагменту универсального, основанного на рефлексии кода о необходимости использования свойства по имени `BirthDate`, определенного в заданном типе, ранее предусматривал применение строкового литерала и обеспечение того, что при изменении имени свойс-

тва изменялся также и этот литерал. Используя версию C# 3, можно построить дерево выражения, которое представляет ссылку на свойство с помощью лямбда-выражения. Затем метод может проанализировать дерево выражения, отыскать указанное свойство и сделать с информацией все, что необходимо. Разумеется, он может также скомпилировать дерево выражения в делегат и пользоваться им напрямую.

В качестве примера, когда такое может применяться, напишем следующий код:

```
serializationContext.AddProperty(x => x.BirthDate);
```

Это позволило бы контексту сериализации (`serializationContext`) узнать, что нужно сериализировать свойство `BirthDate`, и он мог бы записать подходящие метаданные и извлечь значение. (Сериализация — это лишь одна область, где может понадобиться ссылка на свойство или метод; такая ситуация довольно распространена внутри кода, управляемого рефлексией.) Если вы проведете рефакторинг свойства `BirthDate`, назвав его `DateOfBirth`, то лямбда-выражение также изменился. Конечно, это не защищено от неправильного использования — какая-либо проверка того, что оценка выражения действительно дает простое свойство, на этапе компиляции не предпринимается; в коде метода `AddProperty()` должна быть предусмотрена соответствующая проверка времени выполнения.

Вполне вероятно, что однажды C# позволит это делать на уровне самого языка. Операция подобного рода уже получила имя: `infoof`. На протяжении некоторого времени она находилась в списке возможных средств от команды C#, и неудивительно, что Эрик Липперт написал о ней в своем блоге (`http://mng.bz/24y7`), но эта операция пока не прошла отборочный тур. Подождем выхода версии C# 6.

Более простая рефлексия

Перед тем, как погружаться в темные глубины выведения типов, следует упомянуть еще об одном случае применения деревьев выражений, который также связан с рефлексией. Как говорилось в главе 3, арифметические операции не особенно хорошо работают с обобщениями, что затрудняет написание обобщенного кода, скажем, для сложения последовательности значений. Марк Грэвелл применяет деревья выражений для получения удивительного результата в виде обобщенного класса `Operator` и необобщенного вспомогательного класса, позволяя записывать код, подобный показанному ниже:

```
T runningTotal = initialValue;
foreach (T item in values)
{
  runningTotal = Operator.Add(runningTotal, item);
}
```

Код будет функционировать даже в случаях, когда тип значений отличается от типа накапливаемой суммы (`runningTotal`), разрешая, к примеру, добавлять целую последовательность значений `TimeSpan` к `DateTime`. Это *возможно* сделать в C# 2, но оно потребует значительно больше кропотливой работы из-за способов, по которым получается доступ к операциям через рефлексию, особенно для элементарных типов. Деревья выражений позволяют реализации этой "магии" быть довольно чистой, а тот факт, что они компилируются в обычный код IL, который затем обрабатывается JIT-компилятором, обеспечивает великолепную производительность.

Были приведены только некоторые примеры, и вне всяких сомнений множество разработчиков имеют дело с совершенно другими случаями использования деревьев выражений. Однако на этом обсуждение непосредственно лямбда-выражений и деревьев выражений закончено. Вы еще увидите их немало, когда дело дойдет до LINQ, но

прежде чем двигаться дальше, осталось рассмотреть несколько изменений языка C#, которые требуют некоторых пояснений. Эти изменения касаются выведения типов и способа выбора компилятором перегруженных версий методов.

9.4. Изменения в выведении типов и распознавании перегруженных версий

В версии C# 3 шаги, предпринимаемые в рамках выведения типов и распознавании перегруженных версий, были изменены с целью приспособления к лямбда-выражениям и обеспечения большего удобства работы с анонимными методами. Это можно не рассматривать как новое средство C# по существу, но немаловажно понимать, что компилятор собирается делать. Если вы считаете детали подобного рода скучными и несущественными, можете просто сразу переходить к чтению резюме в конце главы. Однако помните о наличии данного раздела, чтобы можно было вернуться к нему в ситуации, когда вы столкнетесь с ошибкой компиляции, связанной с этой темой, и не сумеете разобраться, почему код не заработал. (Или наоборот, может возникнуть желание прочитать его, если оказалось, что код успешно скомпилировался, в то время как вы думали, что этого не должно было произойти.)

Даже в таком разделе я не буду заглядывать в каждый укромный уголок — для этого предназначена спецификация языка; подробные сведения находятся в разделе 7.5.2 ("Type inference" ("Выведение типов")) спецификации C# 5. Взамен я предложу обзор нового поведения, предоставляя примеры для распространенных случаев. Главная причина изменения спецификации связана с необходимостью обеспечения лаконичного стиля для записи лямбда-выражений, вот потому данная тема и включена в настоящую главу.

Давайте сначала немного подробнее ознакомимся с проблемами, которые бы возникли, если бы проектировщики из команды C# решили придерживаться старых правил.

9.4.1. Причины внесения изменений: упрощение вызова обобщенных методов

Выведение типов происходит в нескольких ситуациях. Вы уже видели его применение к неявно типизированным массивам, и оно также требуется при попытке неявного преобразования группы методов в тип делегата. Это может особенно запутывать, когда преобразование происходит при использовании группы методов в качестве аргумента другого метода. В случае перегрузки вызываемого метода *и* наличии перегруженных версий методов внутри группы методов, *а также* возможности участия обобщенных методов набор потенциальных преобразований может оказаться гигантским.

Безусловно, самая распространенная ситуация для выведения типов возникает при вызове обобщенного метода без указания любых аргументов типов. Это происходит в LINQ постоянно — способ, которым работают выражения запросов, сильно зависит от данной возможности. Все это обрабатывается настолько гладко, что очень легко проигнорировать факт выполнения компилятором большого объема работ ради придания написанному вами коду большей ясности и лаконичности.

В C# 2 правила были *умеренно* простыми, хотя группы методов и анонимные методы не всегда обрабатывались настолько хорошо, как возможно того хотелось. Процесс выведения типов не получал из них никакой информации, приводя к ситуациям, когда желаемое поведение было очевидным для разработчиков, но не для компилятора.

Из-за появления лямбда-выражений в C# 3 все еще больше усложнилось. Если вызвать обобщенный метод, используя лямбда-выражение со списком неявно типизированных параметров, компилятору придется выяснить, о каких типах идет речь, до того как он сможет проверить тело лямбда-выражения.

Суть намного проще понять, глядя на код, нежели описывая проблемы словами. В листинге 9.11 приведен пример подобного рода проблемы, на которую я ссылался: вызов обобщенного метода с применением лямбда-выражения.

Листинг 9.11. Пример кода, в котором требуются новые правила выведения типов

```
static void PrintConvertedValue<TInput,TOutput>
  (TInput input, Converter<TInput,TOutput> converter)
{
  Console.WriteLine(converter(input));
}
...
PrintConvertedValue("I'm a string", x => x.Length);
```

Метод `PrintConvertedValue()` в листинге 9.11 просто получает входное значение и делегат, который может преобразовать это значение в другой тип. Он полностью обобщенный — никаких предположений относительно параметров типов `TInput` и `TOutput` не делается. Теперь взгляните на типы аргументов при вызове этого метода в последней строке листинга. Первый аргумент, очевидно, имеет тип `string`, но что можно сказать о втором аргументе? Он является лямбда-выражением, поэтому его нужно преобразовать в `Converter<TInput,TOutput>`, а это означает необходимость знания типов `TInput` и `TOutput`.

Вспомните из раздела 3.3.2, что правила выведения типов в C# 2 применялись к каждому аргументу индивидуально, и не было никакого способа использования типов, выведенных для одного аргумента, во втором аргументе. В данном случае эти правила не позволили бы найти типы `TInput` и `TOutput` для второго аргумента, так что код из листинга 9.11 не смог бы компилироваться.

Наша конечная цель заключается в том, чтобы понять, что обеспечит возможность успешной компиляции кода в листинге 9.11 в C# 3, но пока начнем с чего-то более скромного.

9.4.2. Выведение возвращаемых типов анонимных функций

В листинге 9.12 представлен еще один пример кода, который по идее должен компилироваться, но это не так в условиях действия правил выведения типов C# 2.

Листинг 9.12. Попытка выведения возвращаемого типа для анонимного метода

```
delegate T MyFunc<T>();                          <— Объявление типа делегата: Func<T> отсутствует в .NET 2.0
static void WriteResult<T>(MyFunc<T> function)   <— Объявление обобщенного метода
                                                    с параметром делегата
{
  Console.WriteLine(function());
}
...
WriteResult(delegate { return 5; });             <— Для T требуется выведение типа
```

При компиляции кода из листинга 9.12 с помощью компилятора C# 2 выдается сообщение об ошибке следующего вида:

```
error CS0411: The type arguments for method
'Snippet.WriteResult<T>(Snippet.MyFunc<T>)' cannot be inferred from the
usage. Try specifying the type arguments explicitly.
```

ошибка CS0411: Аргументы типов для метода Snippet.WriteResult<T>(Snippet.MyFunc<T>) не могут быть выведены на основе использования. Попробуйте указать аргументы типов явным образом.

Исправить ошибку можно двумя путями — либо указать аргумент типа явным образом (как предлагает компилятор), либо привести анонимный метод к конкретному типу делегата:

```
WriteResult<int>(delegate { return 5; });
WriteResult((MyFunc<int>)delegate { return 5; });
```

Оба способа работают, но выглядят неуклюжими. Желательно, чтобы компилятор выполнял такой же вид выведения типов, как в случае типов, отличных от делегатов, используя тип возвращаемого выражения для выведения типа T. Именно это в C# 3 делается для анонимных методов и лямбда-выражений, однако есть одна загвоздка. Хотя во многих случаях задействован только один оператор `return`, иногда их может быть больше. В листинге 9.13 показана слегка измененная версия кода из листинга 9.12, в которой анонимный метод временами возвращает целочисленное значение, а временами — объект.

Листинг 9.13. Код, возвращающий целочисленное значение или объект в зависимости от времени суток

```
delegate T MyFunc<T>();
static void WriteResult<T>(MyFunc<T> function)
{
  Console.WriteLine(function());
}
...
WriteResult(delegate
{
  if (DateTime.Now.Hour < 12)
  {
    return 10;                    <-- Возвращаемым типом является int
  }
  else
  {
    return new object();          <-- Возвращаемым типом является object
  }
});
```

В этой ситуации для определения возвращаемого типа компилятор применяет ту же самую логику, что и в случае неявно типизированных массивов, как было описано в разделе 8.4. Он формирует набор типов из всех операторов `return`, обнаруженных в теле анонимной функции[8] (в данном случае `int` и `object`), и проверяет, есть ли среди

[8] Возвращаемые выражения, не имеющие типа, такие как `null` или другое лямбда-выражение, в этот набор не включаются. Их допустимость проверяется позже, после того как возвращаемый тип был определен, однако они не принимают участия в данном решении.

них в точности один тип, в который могут быть неявно преобразованы все остальные типы. Существует неявное преобразование из `int` в `object` (через упаковку), но не из `object` в `int`, поэтому выведенным возвращаемым типом будет `object`. Если указанному критерию не соответствует ни одного типа (или типов оказывается более одного), возвращаемый тип не может быть выведен и возникает ошибка компиляции.

Теперь вы знаете, как выяснять *возвращаемый* тип анонимной функции, но что можно сказать о лямбда-выражениях, в которых типы параметров допускают неявное определение?

9.4.3. Двухэтапное выведение типов

Детали выведения типов в C# 3 *намного* сложнее, чем в C# 2. Потребность в консультации со спецификацией относительно точного его поведения будет возникать редко, но если это все же понадобится, я рекомендую записать на бумаге все параметры типов, аргументы и прочие данные, а затем пошагово следовать спецификации, тщательно помечая каждое требуемое действие. В конечном итоге получится таблица, полная *фиксированных* и *нефиксированных* переменных типов с разными наборами *границ* для каждой из них. *Фиксированная* переменная типа — это такая переменная, для которой компилятор принял решение, касающееся ее значения; иначе она будет *нефиксированной*. *Граница* — это фрагмент информации о переменной типа. Учитывая еще и массу примечаний, я не сомневаюсь, что головная боль вам обеспечена; это не особенно интересная тема.

Я представлю менее строгий взгляд на выведение типов — он будет, скорее всего, не хуже спецификации, но намного проще для понимания. Дело в том, что если компилятор не выполняет выведение типов точно так, как вы хотите, это почти наверняка приведет к ошибке компиляции, а не к коду, который хотя и построен, но ведет себя некорректно. Если построение кода не происходит, попробуйте предоставить компилятору больше информации — это довольно просто. Ниже дано *примерное* описание того, что изменилось в версии C# 3.

Первое крупное изменение связано с тем, что в C# 3 аргументы работают слаженно, подобно единой команде. В C# 2 каждый аргумент использовался для попытки *точного* определения некоторых параметров, и компилятор предъявил бы претензии в случае, если для отдельного параметра типа любые два аргумента приводили бы к разным результатам, даже если они совместимы. В C# 3 аргументы могут нести в себе *части* информации — типы, которые должны быть неявно преобразуемыми в окончательное фиксированное значение конкретной переменной типа. Для получения этого фиксированного значения применяется та же самая логика, что и при выведении возвращаемых типов и в неявно типизированных массивах.

В листинге 9.14 показан пример, в котором не используются ни лямбда-выражения, ни даже анонимные методы.

Листинг 9.14. Гибкое выведение типов, комбинирующее информацию из множества аргументов

```
static void PrintType<T>(T first, T second)
{
  Console.WriteLine(typeof(T));
}
...
PrintType(1, new object());
```

Хотя код в листинге 9.14 *синтаксически* допустим в C# 2, он не скомпилируется; выведение типов потерпит неудачу, т.к. первый параметр предопределит, что тип T должен быть int, а второй — что T должен быть object. В C# 3 компилятор выясняет, что тип T должен быть object в точности таким же способом, как это делалось при выведении возвращаемого типа в листинге 9.13. В сущности, правила выведения возвращаемых типов являются одним примером более общего процесса в C# 3. Второе изменение заключается в том, что выведение типов теперь осуществляется в два этапа. Первый этап имеет дело с обычными аргументами, где задействованные типы известны с самого начала. Сюда входят анонимные функции со списками явно типизированных параметров.

Затем начинает действовать второй этап, на котором выводятся типы неявно типизированных лямбда-выражений и групп методов. Идея состоит в том, чтобы выяснить, достаточно ли информации, собранной компилятором из фрагментов, для определения типов параметров лямбда-выражения (или группы методов). Если ее достаточно, компилятор может приступать к исследованию тела лямбда-выражения с целью выведения возвращаемого типа, который часто является еще одним искомым параметром типа. Если второй этап предоставляет какую-то дополнительную информацию, компилятор повторяет действия второго этапа заново и так происходит до тех пор, пока не исчерпаются все возможности или не выяснятся все содержащиеся параметры типов. На рис. 9.7 сказанное представлено в виде блок-схемы, но не забывайте, что это сильно упрощенная версия алгоритма.

Давайте рассмотрим два примера, демонстрирующие работу этого алгоритма. Первым делом возьмем код, приведенный в начале этого раздела, в листинге 9.11:

```
static void PrintConvertedValue<TInput,TOutput>
  (TInput input, Converter<TInput,TOutput> converter)
{
  Console.WriteLine(converter(input));
}
...
PrintConvertedValue("I'm a string", x => x.Length);
```

Параметрами типов, которые необходимо выяснить здесь, являются TInput и TOutput. Ниже перечислены шаги, которые выполняются для этого.

1. Начинается этап 1.
2. Первый параметр имеет тип TInput, а первый аргумент — тип string. Мы делаем вывод, что должно существовать неявное преобразование из string в TInput.
3. Второй параметр имеет тип Converter<TInput,TOutput>, а вторым аргументом является неявно типизированное лямбда-выражение. Выведение не производится, т.к. информации для этого не достаточно.
4. Начинается этап 2.
5. Тип TInput не зависит от каких-либо нефиксированных параметров типов, поэтому он фиксируется как string.
6. Второй аргумент теперь имеет фиксированный *входной* тип, но нефиксированный *выходной* тип. Его можно рассматривать как (string x) => x.Length и вывести возвращаемый тип как int. Следовательно, неявное преобразование должно происходить из int в TOutput.
7. Этап 2 повторяется.
8. Тип TOutput не зависит от чего-либо нефиксированного, поэтому он фиксируется в int.
9. Теперь нефиксированных параметров типов не осталось, так что выведение успешно завершено.

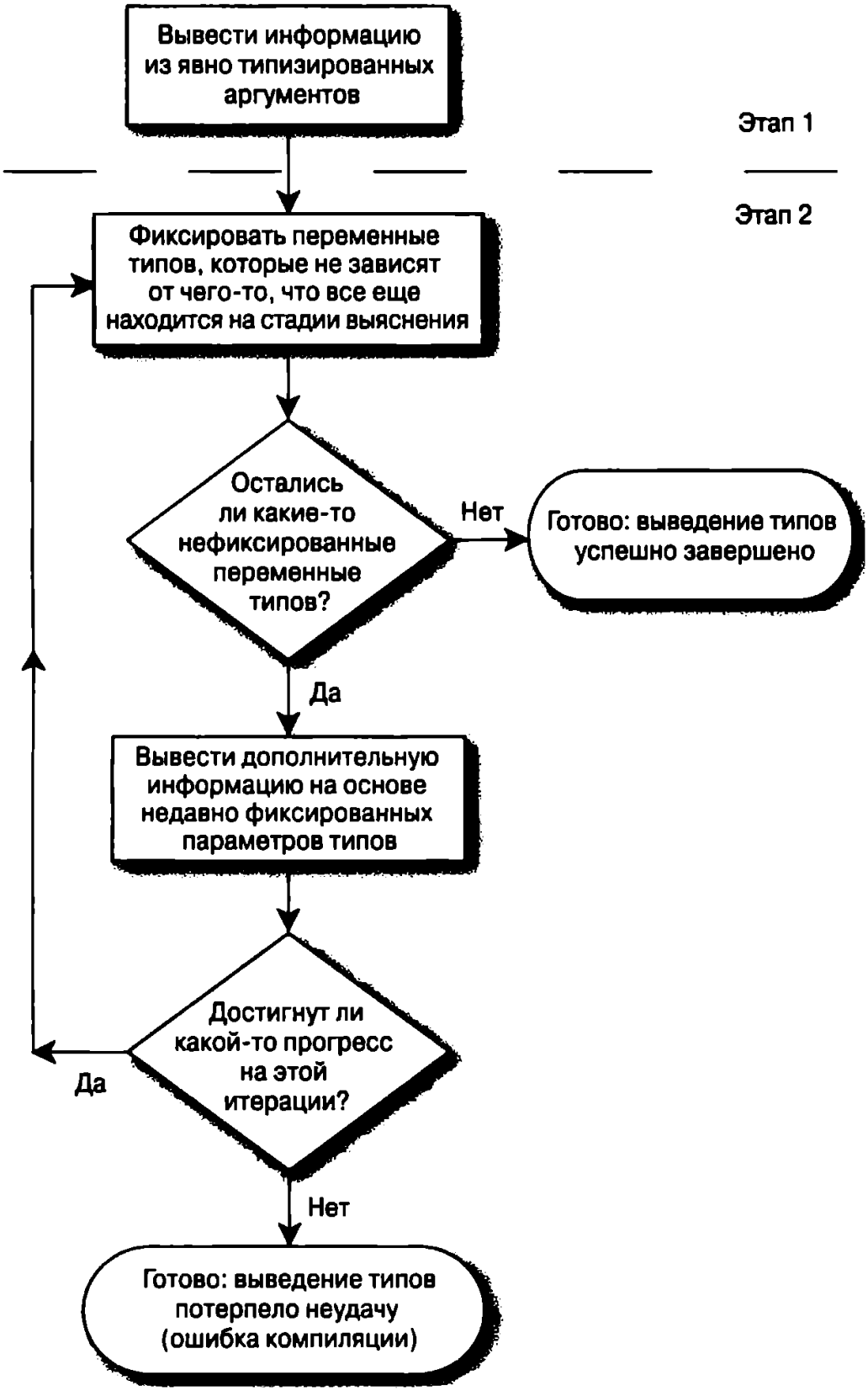

Рис. 9.7. Поток двухэтапного выведения типов

Сложно, не правда ли? Тем не менее, работа сделана — получен нужный вам результат (TInput=string, TOutput=int) и все компилируется безо всяких проблем.

Важность повторения этапа 2 лучше всего подчеркнуть с помощью еще одного примера. В листинге 9.15 демонстрируется выполнение *двух* преобразований, причем выход первого становится входом второго. До тех пор, пока не будет выяснен выходной тип первого преобразования, входной тип второго преобразования не известен, поэтому вывести его выходной тип тоже невозможно.

Листинг 9.15. Многоэтапное выведение типов

```
static void ConvertTwice<TInput,TMiddle,TOutput>
  (TInput input,
   Converter<TInput,TMiddle> firstConversion,
   Converter<TMiddle,TOutput> secondConversion)
{
  TMiddle middle = firstConversion(input);
  TOutput output = secondConversion(middle);
  Console.WriteLine(output);
}
...
ConvertTwice("Another string",
          text => text.Length,
          length => Math.Sqrt(length));
```

Первое, что следует отметить — сигнатура метода выглядит довольно устрашающе. Однако не все так плохо, если перестать бояться и взглянуть на нее внимательнее; пример применения определенно делает ее более очевидной. Здесь берется строка и над ней выполняется преобразование — то же самое преобразование, что и ранее, т.е. просто вычисление длины. Затем для этой длины строки (значение int) находится квадратный корень (значение double).

Этап 1 выведения типов сообщает компилятору о том, что должно существовать преобразование из string в TInput. Во время первого прохода этапа 2 тип TInput фиксируется в string и делается вывод о необходимости наличия преобразования из int в TMiddle. На втором проходе этапа 2 тип TMiddle фиксируется в int и делается вывод о том, что должно быть преобразование из double в TOutput. На третьем проходе этапа 2 тип TOutput фиксируется в double и выведение типов успешно завершается. По завершении выведения типов компилятор может должным образом просматривать код внутри лямбда-выражения.

Проверка тела лямбда-выражения

Тело лямбда-выражения *не может быть проверено* до тех пор, пока не станут известными типы входных параметров. Лямбда-выражение x => x.Length допустимо, если x является массивом или строкой, но недопустимо во многих других случаях. Это не проблема, когда типы параметров объявлены явно, но в случае списка неявно типизированных параметров компилятор должен подождать, пока не будет выполнено подходящее выведение типов, и только потом попытаться выяснить, в чем смысл лямбда-выражения.

Эти примеры проиллюстрировали только по одному изменению за раз, но на практике могут существовать многочисленные фрагменты информации о разных переменных типов, потенциально обнаруживаемые на различных итерациях процесса. Пытаясь спасти вашу психику (да и мою заодно), я больше не буду приводить сложные примеры, в надежде, что вы понимаете общий механизм, даже если точные детали туманны.

Хотя может показаться, что ситуация подобного рода будет возникать настолько редко, что такие сложные правила не стоит даже кратко рассматривать, на самом деле она распространена в C# 3, особенно в связи с LINQ. Вы могли бы без труда широко пользоваться выведением типов, даже не думая о нем — вполне возможно, что это войдет в вашу привычку. Но если случается отказ и вас интересует причина, вы всегда можете возвратиться к этому разделу и спецификации языка.

Необходимо ознакомиться с еще одним изменением и вас, несомненно, обрадует, что оно проще выведения типов. Давайте обратимся к перегрузке методов.

9.4.4. Выбор правильного перегруженного метода

Перегрузка предполагает наличие нескольких методов с одинаковыми именами, но разными сигнатурами. Иногда выбор метода вполне очевиден, поскольку он единственный обладает правильным количеством параметров или все аргументы в нем могут быть преобразованы в соответствующие типы параметров.

Некоторые сложности возникают в ситуации, когда подходящими *могут* быть сразу несколько методов. Правила, описанные в разделе 7.5.3 ("Overload Resolution" ("Распознавание перегруженных версий")) спецификации языка довольно сложны (да, *опять*), но основной частью является способ преобразования типов аргументов в типы параметров[9]. Например, предположим, что следующие сигнатуры методов объявлены в одном и том же типе:

```
void Write(int x)
void Write(double y)
```

Смысл вызова `Write(1.5)` очевиден, поскольку неявное преобразование из `double` в `int` отсутствует, но смысл вызова `Write(1)` определить сложнее. Неявное преобразование из `int` в `double` *существует*, поэтому оба метода допустимы. В данной точке компилятор принимает во внимание преобразование из `int` в `int` и из `int` в `double`. Преобразование типа самого в себя определяется как гораздо *лучшее*, чем преобразование в любой другой тип, так что для этого конкретного вызова метод `Write(int x)` оказывается лучше, чем `Write(double y)`.

При наличии нескольких параметров компилятор должен обеспечить применение наилучшего метода. Метод считается лучше других, если все участвующие преобразования аргументов, *по крайней мере, так же хороши*, как соответствующие преобразования в другом методе, и, по меньшей мере, одно преобразование однозначно лучше. В качестве простого примера взгляните на следующие объявления:

```
void Write(int x, double y)
void Write(double x, int y)
```

Вызов `Write(1, 1)` будет неоднозначным, и компилятор вынудит вас добавить приведение, по крайней мере, к одному параметру, чтобы прояснить, какой метод вы

[9] Предполагается, что все методы объявлены в одном классе. Когда в игру вступает наследование, все становится еще сложнее. Данный аспект в C# 3 не изменился.

намерены вызвать. Каждая перегруженная версия имеет одно лучшее преобразование аргумента, поэтому ни одна из них не может быть выбрана.

Такая логика по-прежнему применима в C# 3, но с одним дополнительным правилом, касающимся анонимных функций, в которых никогда не указывается возвращаемый тип. В этом случае в правилах наилучшего преобразования используется выведенный возвращаемый тип (как описано в разделе 9.4.2).

Давайте рассмотрим пример ситуации, когда требуется это новое правило. В листинге 9.16 содержится два метода по имени `Execute()` и вызов, в котором применяется лямбда-выражение.

Листинг 9.16. Пример выбора перегруженной версии при влиянии возвращаемого типа делегата

```
static void Execute(Func<int> action)
{
  Console.WriteLine("action returns an int: " + action());
}
static void Execute(Func<double> action)
{
  Console.WriteLine("action returns a double: " + action());
}
...
Execute(() => 1);
```

Вызов `Execute()` в листинге 9.16 взамен можно было бы записать с помощью анонимного метода или группы методов — какой бы вид преобразования не был задействован, применяются те же самые правила. Какой из методов `Execute()` должен быть вызван? Правила перегрузки гласят, что когда после преобразований аргументов оба метода применимы, эти преобразования аргументов анализируются на предмет выявления лучшего из них. Преобразования здесь осуществляются не из обычного типа .NET в тип параметра, а из лямбда-выражения в два типа делегатов. Какое из преобразований лучше?

Как ни удивительно, но эта же ситуация в C# 2 привела бы к ошибке компиляции — языковые правила для такого случая не предусмотрены. В C# 3 будет выбран метод с параметром `Func<int>`. Добавленное дополнительное правило может быть изложено своими словами следующим образом.

Если анонимная функция может быть преобразована в два типа делегатов, которые имеют одинаковые списки параметров, но отличающиеся возвращаемые типы, то преобразования делегатов оцениваются по преобразованиям из выведенного возвращаемого типа в возвращаемые типы делегатов.

Без ссылки на пример звучит как порядочная тарабарщина. Давайте снова возвратимся к листингу 9.16, в котором выполнялось преобразование из лямбда-выражения, не принимающего параметров и имеющего выведенный возвращаемый тип `int`, либо в тип `Func<int>`, либо в тип `Func<double>`. Для обоих типов делегатов списки параметров одинаковы (пустые), поэтому применяется указанное выше правило. Затем нужно просто найти лучшее преобразование: `int` в `int` или `int` в `double`. Ситуация должна выглядеть знакомой; как было упомянуто ранее, преобразование `int` в `int` считается лучшим. Таким образом, код из листинга 9.16 выводит на консоль строку `action returns an int: 1`.

9.4.5. Итоги по выведению типов и распознаванию перегруженных версий

Материал этого раздела был довольно трудным. Я хотел бы сделать его проще, но он был посвящен фундаментально сложной теме. Задействованная терминология не способствует упрощению, да еще и с учетом того, что *тип параметра* и *параметр типа* обозначают совершенно разные вещи! Примите поздравления, если вы дошли до конца и действительно все поняли. Не переживайте, если это не так; надеюсь, что когда вы будете читать данный раздел в следующий раз, он прольет больше света на эту тему — особенно после того, как вы попали в ситуацию, непосредственно касающуюся рассматриваемых здесь вопросов. Ниже перечислены наиболее важные моменты, накопившиеся к этому времени.

- Анонимные функции (анонимные методы и лямбда-выражения) имеют выведенные возвращаемые типы, основанные на типах, которые использовались во всех операторах `return`.
- Лямбда-выражения могут восприниматься компилятором, только когда известны типы всех их параметров.
- Выведение типов больше не требует, чтобы каждый аргумент независимо приходил к точно такому же заключению относительно параметров типов, при условии совместимости результатов.
- Выведение типов теперь является многоэтапным: выведенный возвращаемый тип одной анонимной функции может выступать в качестве типа параметра для другой такой функции.
- При поиске лучшей перегруженной версии метода, когда участвуют анонимные функции, принимается во внимание возвращаемый тип.

Даже такой короткий список очень плотно усеян техническими терминами. Не волнуйтесь, если вам не все в нем понятно. По моему опыту в большинстве случаев все работает так, как нужно.

9.5. Резюме

В C# 3 лямбда-выражения почти полностью заменили анонимные методы. Анонимные методы поддерживаются ради обратной совместимости, но идиоматический, заново написанный код C# 3 будет содержать их мало.

Вы видели, что лямбда-выражения — это нечто большее, чем просто компактный синтаксис для создания делегатов. Они могут быть преобразованы в деревья выражений с учетом ряда ограничений. Деревья выражений затем могут обрабатываться другим кодом, возможно выполняющим эквивалентные действия в разных исполняющих средах. Без такой характеристики язык LINQ ограничивался бы внутрипроцессными запросами.

Наше обсуждение выведения типов было в определенной степени необходимым злом; лишь немногим разработчикам действительно *нравится* говорить о такой разновидности правил, которые при этом должны применяться, но важно иметь хотя бы примерное представление о том, что происходит. Прежде чем начать чрезмерно жалеть самих себя, подумайте о бедных проектировщиках языка, которым приходилось жить и дышать этим, удостоверяясь, что правила согласованы и не разваливаются в неприятных ситуациях. Затем вспомните о тестировщиках, которые должны были пытаться нарушить работу реализации. С точки зрения *описания* лямбда-выражений на этом все, но вы еще увидите много случаев их применения в остальных главах книги. Например, в следующей главе подробно рассматриваются *расширяющие методы*. На первый взгляд, они полностью отделены от лямбда-выражений, но на деле эти два средства часто используются вместе.

Глава 10

Расширяющие методы

В этой главе...

- Написание расширяющих методов
- Вызов расширяющих методов
- Соединение методов в цепочки
- Расширяющие методы в .NET 3.5
- Другие случаи использования расширяющих методов

Я не являюсь поклонником наследования. Вернее, мне не нравятся те несколько мест, где использовалось наследование, в сопровождаемом мною коде или библиотеках классов, с которыми мне приходилось работать. Как и с очень многими средствами, наследование является мощным, когда применяется должным образом, но зачастую из виду упускаются накладные расходы по проектированию, которые со временем могут стать ощутимыми. Оно иногда используется в качестве способа добавления к классу дополнительного поведения и функциональности, даже когда никакой действительной информации об объекте не предоставляется и ничего не специализируется.

Иногда наследование оказывается подходящим, например, если объекты нового типа должны заботиться о деталях дополнительного поведения, но часто это не так. В первую очередь во многих случаях наследование просто невозможно использовать обычным образом, скажем, при работе с типом значения, запечатанным классом либо интерфейсом. Альтернатива обычно сводится к написанию набора статических методов, которые принимают экземпляр рассматриваемого типа в виде, как минимум, одного из своих параметров. Это хорошо работает, не сопровождается проектным неудобством в форме наследования, но может привести к получению неуклюже выглядящего кода.

В версии C# 3 появилась идея *расширяющих методов*, которые обладают преимуществами решения со статическими методами и также улучшают читабельность кода, в котором они вызываются. Эта идея дает возможность обращаться к статическим мето-

дам, как если бы они были методами экземпляра совершенно другого класса. Не паникуйте — это звучит не так глупо, как может показаться на первый взгляд.

В данной главе мы сначала посмотрим, как пользоваться расширяющими методами и как их писать. Затем мы проанализируем несколько расширяющих методов, предлагаемых .NET 3.5, и ознакомимся с тем, каким образом их легко соединять в цепочки. Эта возможность построения цепочек является, в первую очередь, важной составляющей самой причины добавления расширяющих методов в язык и важной частью LINQ[1]. Наконец, мы рассмотрим некоторые за и против применения расширяющих методов вместо обычных статических методов.

Тем не менее, давайте сначала более пристально посмотрим, почему расширяющие методы иногда более желательны, нежели то, что было доступно в версиях C# 1 и C# 2, особенно при создании вспомогательных классов.

10.1. Ситуация до появления вспомогательных методов

На этой стадии у вас может возникнуть ощущение дежа-вю, ведь вспомогательные классы уже упоминались в главе 7, когда мы рассматривали статические классы. Если до перехода на C# 3 вам приходилось писать немало кода в C# 2, вы должны взглянуть на свои статические классы — многие их методы могут оказаться хорошими кандидатами на преобразование в расширяющие методы. Это не говорит о том, что для такой цели подойдут все существующие статические классы, но вы можете уверенно распознать кандидатов по следующим характерным чертам.

- Вы хотите добавить к типу определенные члены.
- Вы не нуждаетесь в добавлении каких-то дополнительных данных к экземплярам типа.
- Вы не можете изменить сам тип, поскольку он находится в коде у кого-то другого.

Одна небольшая вариация этого заключается в том, чтобы при работе с интерфейсом вместо класса полезное поведение добавлялось только при вызове методов данного интерфейса. Хорошим примером является интерфейс `IList<T>`. Разве не была бы замечательной возможность сортировки любой (изменяемой) реализации `IList<T>`? Было бы неприятно заставлять в каждой реализации интерфейса принудительно реализовывать сортировку, но это неплохо с точки зрения *пользователя* готового списка.

Дело в том, что `IList<T>` предлагает все строительные блоки для полностью обобщенной процедуры сортировки (фактически для нескольких), но поместить эту реализацию в интерфейс нельзя. Вместо этого `IList<T>` можно было бы указать как абстрактный класс, и функциональность сортировки тогда была бы включена, но поскольку C# и .NET поддерживается одиночное наследование реализации, это наложило бы значительное ограничение на производные от него типы. Расширяющий метод на `IList<T>` позволит сортировать любую реализацию `IList<T>`, делая так, чтобы *казалось*, что эту функциональность предоставляет сам список.

---

[1] Если вы сыты по горло утверждениями о том, что то или иное средство является "важной частью LINQ", я вас не осуждаю, но это все же часть его великолепия. Существует очень много небольших частей, однако их сумма дает блестящий результат. Тот факт, что каждое средство может быть использовано и независимо — лишь дополнительная награда.

Позже вы увидите, что много функциональности LINQ построено на основе расширяющих методов поверх интерфейсов. Тем не менее, в данный момент мы будем использовать в примерах другой тип: `System.IO.Stream`, краеугольный камень бинарных коммуникаций в .NET. Сам тип `Stream` является абстрактным классом, имеющим несколько конкретных производных классов, таких как `NetworkStream`, `FileStream` и `MemoryStream`. К сожалению, есть несколько порций функциональности, которые было бы полезно включить в `Stream`, но которые в нем отсутствуют.

Недостающие средства, которые мне чаще всего нужны, включают возможность чтения целого массива в память в виде байтового массива и возможность копирования содержимого одного массива в другой[2]. Оба средства часто реализуются неудачно, делая в отношении потоков предположения, которые просто не являются допустимыми — самое распространенное недоразумение заключается в том, что метод `Stream.Read()` будет полностью заполнять буфер, если данные не закончились раньше.

Тем не менее, не так уж много отсутствующих средств

Одно из таких средств было добавлено в .NET 4: тип `Stream` теперь имеет метод `CopyTo()`. Это удобно с точки зрения демонстрации одного довольно тонкого аспекта расширяющих методов, и мы вернемся к нему в разделе 10.2.3. Метод `ReadFully()` по-прежнему отсутствует, но его в любом случае следовало бы применять осмотрительно: поток может быть прочитан полностью только тогда, когда точно известно, что у него есть признак конца и все данные умещаются в памяти. Потоки не связаны обязательством иметь конечный объем данных.

Было бы неплохо иметь эту функциональность в одном месте, а не дублировать во многих проектах. Именно по этой причине я создал класс `StreamUtil` в своей смешанной вспомогательной библиотеке. Действительный код содержит большое количество проверок на предмет ошибок и другую функциональность, но в листинге 10.1 представлена усеченная версия, которой более чем достаточно для текущих потребностей.

Листинг 10.1. Простой вспомогательный класс, предоставляющий дополнительную функциональность потокам

```
using System.IO;
public static class StreamUtil
{
  const int BufferSize = 8192;
  public static void Copy(Stream input, Stream output)
  {
    byte[] buffer = new byte[BufferSize];
    int read;
    while ((read = input.Read(buffer, 0, buffer.Length)) > 0)
    {
      output.Write(buffer, 0, read);
    }
  }
```

[2] Из-за природы потоков это копирование не обязательно должно *дублировать* данные — они просто читаются из одного потока и записываются в другой. Хотя *копирование* не является совершенно точным термином в этом смысле, разница обычно не имеет значения.

```
    public static byte[] ReadFully(Stream input)
    {
      using (MemoryStream tempStream = new MemoryStream())
      {
        Copy(input, tempStream);
        return tempStream.ToArray();
      }
    }
}
```

Детали реализации не играют особой роли, хотя полезно отметить, что в методе `ReadFully()` вызывается метод `Copy()` — это будет полезно впоследствии для демонстрации одной особенности, связанной с расширяющими методами.

Класс `StreamUtil` использовать легко; например, в листинге 10.2 показано, как можно записать на диск ответ, получаемый в результате запроса веб-страницы.

Листинг 10.2. Применение класса `StreamUtil` для копирования в файл потока ответа, получаемого в результате запроса веб-страницы

```
WebRequest request = WebRequest.Create("http://manning.com");
using (WebResponse response = request.GetResponse())
using (Stream responseStream = response.GetResponseStream())
using (FileStream output = File.Create("response.dat"))
{
  StreamUtil.Copy(responseStream, output);
}
```

Код в листинге 10.2 довольно компактен, и класс `StreamUtil` позаботился об организации цикла и запрашивании данных из потока ответа до тех пор, пока все они не были получены. Как вспомогательный класс, он совершенно корректно выполняет свою работу. Но даже при этих условиях он не выглядит особенно объектно-ориентированным. Было бы лучше указывать потоку ответа на необходимость копирования его содержимого в выходной поток, подобно тому, для чего в классе `MemoryStream` предусмотрен метод `WriteTo()`. Это не *крупная* проблема, но способ ее решения выглядит несколько неуклюжим.

Поскольку в данной ситуации наследование не помогает (это поведение должно быть доступно для всех потоков, а не только для тех, за которые вы несете ответственность), а изменить сам класс `Stream` невозможно, то как тогда поступить? В C# 2 вариантов не было — приходилось придерживаться статических методов и мириться с неудобным кодом. В C# 3 появилась возможность изменить статический класс, чтобы представить его члены в виде расширяющих методов, так что можно сделать вид, что эти методы всегда были частью `Stream`. Давайте посмотрим, какие изменения потребуется внести.

10.2. Синтаксис расширяющих методов

Расширяющие методы на удивление легко создавать и также просто использовать. Соображения относительно того, когда и как их применять, значительно серьезнее, чем трудности, сопровождающие изучение особенностей их написания. Начнем с преобразования класса `StreamUtil`, чтобы он имел пару расширяющих методов.

10.2.1. Объявление расширяющих методов

Использовать в качестве расширяющего метода абсолютно любой метод не получится – метод должен обладать перечисленными ниже характеристиками.

- Он должен находиться в невложенном и необобщенном статическом классе (и, следовательно, должен быть статическим методом).
- Он должен принимать хотя бы один параметр.
- Первый параметр должен предваряться ключевым словом `this`.
- Первый параметр не может иметь какие-то другие модификаторы (такие как `out` или `ref`).
- Типом первого параметра не должен быть указатель.

Это весь перечень требований; метод может быть обобщенным, возвращать значение, иметь параметры `ref/out` кроме первого, быть реализованным посредством итераторного блока, быть частью частичного класса, использовать типы, допускающие `null` — в общем, все, что угодно, при условии удовлетворения указанных выше ограничений.

Мы будем называть тип первого параметра *расширенным типом* метода и говорить, что данный метод *расширяет* этот тип — в рассматриваемом случае мы расширяем тип `Stream`. Это не официальная терминология из спецификации, но она является удобным сокращением.

В предшествующем списке не только описаны все ограничения, но также указаны действия, которые необходимо предпринять для превращения обычного статического метода в расширяющий метод — нужно просто добавить ключевое слово `this`. В листинге 10.3 приведен код того же самого класса, что и в листинге 10.1, но этот раз оба метода являются расширяющими.

Листинг 10.3. Класс `StreamUtil`, но уже с расширяющими методами

```
public static class StreamUtil
{
  const int BufferSize = 8192;
  public static void CopyTo(this Stream input, Stream output)
  {
    byte[] buffer = new byte[BufferSize];
    int read;
    while ((read = input.Read(buffer, 0, buffer.Length)) > 0)
    {
      output.Write(buffer, 0, read);
    }
  }
  public static byte[] ReadFully(this Stream input)
  {
    using (MemoryStream tempStream = new MemoryStream())
    {
      CopyTo(input, tempStream);
      return tempStream.ToArray();
    }
  }
}
```

Действительно, единственным крупным изменением в листинге 10.3 стало добавление двух модификаторов, выделенных полужирным. Кроме того, имя метода было изменено с `Copy()` на `CopyTo()`. Как вскоре будет показано, это позволит более естественно воспринимать вызывающий код, хотя в данный момент его вызов внутри метода `ReadFully()` выглядит несколько странно.

В наличии расширяющих методов не было бы особого смысла, если бы их нельзя было использовать...

10.2.2. Вызов расширяющих методов

Я упомянул об этом мимоходом, но вы пока еще не видели, что в действительности *делает* расширяющий метод. Попросту говоря, он делает вид, что является методом экземпляра другого типа — типа, заданного для первого параметра метода.

Трансформация кода, в котором применяется класс `StreamUtil`, не сложнее трансформации самого вспомогательного класса. На этот раз вместо добавления чего-либо мы кое-что уберем. Код в листинге 10.4 почти повторяет код из листинга 10.2, но в нем используется новый синтаксис для вызова `CopyTo()`. Синтаксис назван "новым", но на самом деле он совершенно не нов — это тот же самый синтаксис, который всегда применяется для вызова методов экземпляра.

Листинг 10.4. Копирование потока с использованием расширяющего метода

```
WebRequest request = WebRequest.Create("http://manning.com");
using (WebResponse response = request.GetResponse())
using (Stream responseStream = response.GetResponseStream())
using (FileStream output = File.Create("response.dat"))
{
  responseStream.CopyTo(output);
}
```

В листинге 10.4 вызов только *выглядит* так, как будто у потока ответа запрашивается выполнение копирования. Всю работу "за кулисами" по-прежнему делает `StreamUtil`, но такой код воспринимается более естественно. В сущности, компилятор преобразует обращение к `CopyTo()` в вызов обычного статического метода `StreamUtil.CopyTo()`, передавая значение `responseStream` в качестве первого аргумента (за которым следует `output`).

Теперь, когда вы видели код, я надеюсь, что вы понимаете причину изменения имени метода с `Copy()` ("копировать") на `CopyTo()` ("копировать в"). Одни имена одинаково хорошо подходят как для статических методов, так и для методов экземпляра, но вы обнаружите, что другие имена нуждаются в корректировке ради обеспечения максимально возможной читабельности.

Если вы хотите сделать код `StreamUtil` еще более симпатичным, можете изменить строку с вызовом метода `CopyTo()` внутри `ReadFully()` следующим образом:

```
input.CopyTo(tempStream);
```

В данный момент измененное имя полностью соответствует всем случаям его применения — хотя ничего не может воспрепятствовать использованию расширяющего метода как обычного статического метода, что может быть удобным при переносе большого объема кода.

Возможно, вы заметили, что в приведенных выше вызовах методов ничего не указывало на применение именно расширяющего метода, а не обычного метода экземпляра `Stream`. Данный факт следует рассматривать с двух сторон: с одной стороны это хорошо, если ваше намерение заключается в том, чтобы сделать расширяющие методы как можно более гармоничными и не причиняющими беспокойства, но с другой стороны это плохо, если нужна возможность непосредственно видеть, что происходит *в действительности*.

Если вы работаете в Visual Studio, то можете навести курсор мыши на вызов метода и получить внутри всплывающей подсказки указание на то, что вызывается расширяющий метод (рис. 10.1). Средство IntelliSense также обозначает расширяющий метод, как в значке для метода, так и во всплывающей подсказке, когда он выбран. Разумеется, вы не должны наводить курсор на каждый вызов метода или слишком часто пользоваться IntelliSense, т.к. большую часть времени не имеет значения, вызывается метод экземпляра или расширяющий метод.

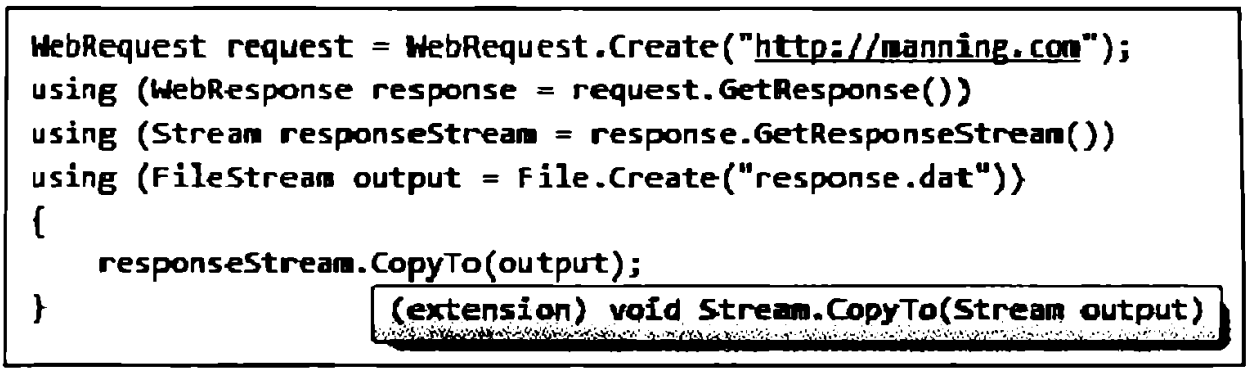

Рис. 10.1. Наведение курсора мыши на вызов метода в Visual Studio позволяет выяснить, является ли метод расширяющим

В этом вызываемом коде все еще остается одна странность — в нем совершенно нигде не упоминается класс `StreamUtil`! Как компилятор узнает, что должен применяться расширяющий метод?

10.2.3. Обнаружение расширяющих методов

Важно знать, каким образом вызываются расширяющие методы, но не менее важно также понимать, как обеспечить, чтобы они не вызывались, т.е. как избежать выбора нежелательных вариантов. Для этого, прежде всего, необходимо знать, каким образом компилятор принимает решение о том, какие расширяющие методы использовать.

Расширяющие методы делаются доступными коду тем же самым способом, которым делаются доступными классы, указанные без пространства имен — с помощью директив `using`. Когда компилятор сталкивается с выражением, которое выглядит как попытка применения метода экземпляра, но ни один из методов экземпляра с этим вызовом метода не совместим (к примеру, отсутствует метод с таким именем или нет перегруженной версии, которая бы соответствовала заданным аргументам), он приступает к поиску подходящего расширяющего метода. Компилятор просматривает все расширяющие методы во всех импортированных пространствах имен, а также в текущем пространстве имен, и выполняет сопоставление с теми из них, для которых существует неявное преобразование из типа выражения в расширенный тип.

Деталь реализации: как компилятор обнаруживает расширяющий метод?

Чтобы выяснить, должен ли использоваться расширяющий метод, компилятору необходима возможность определения разницы между расширяющим методом и другими методами внутри статического класса, которые имеют подходящую сигнатуру. Он делает это путем проверки, применялся ли атрибут `System.Runtime.CompilerServices.ExtensionAttribute` к методу и классу. Этот атрибут был введен в .NET 3.5, но компилятор не проверяет, из какой сборки этот атрибут поступает. В результате расширяющими методами можно по-прежнему пользоваться, даже если целевой платформой проекта является .NET 2.0, просто нужно определить собственный атрибут с корректным именем в правильном пространстве имен. Затем можно объявлять свои расширяющие методы обычным образом и атрибут будет применен автоматически. Компилятор также применяет этот атрибут к сборке, которая содержит расширяющий метод, но в текущий момент это не требуется при поиске расширяющих методов.

Введение собственных копий системных типов может стать проблематичным, если впоследствии понадобится применять версию инфраструктуры, в которой эти типы уже определены. Если вы используете такой прием, полезно с помощью символов препроцессора объявлять атрибут условно. Это позволит строить одну версию кода для целевой платформы .NET 2.0, а другую — для целевой платформы .NET 3.5 и последующих версий.

Если доступно сразу несколько пригодных расширяющих методов для разных расширенных типов (с использованием неявных преобразований), то наиболее подходящий из них выбирается посредством правил лучшего преобразования, которые задействованы при выборе перегруженной версии метода. Например, если интерфейс `IDerived` унаследован от `IBase`, и для обоих имеется расширяющий метод с тем же самым именем, то расширяющему методу в интерфейсе `IDerived` отдается предпочтение перед таким методом в `IBase`. Опять-таки, это средство применяется в LINQ, как будет показано в разделе 12.2, где вы встретитесь с интерфейсом `IQueryable<T>`.

Важно отметить, что если доступен пригодный метод экземпляра, он всегда будет использоваться до поиска расширяющих методов, однако компилятор не выдает предупреждения, когда расширяющий метод также соответствует существующему методу экземпляра. Например, в .NET 4 класс `Stream` имеет новый метод, который также называется `CopyTo()`. Для него определены две перегруженных версии, одна из которых конфликтует с только что созданным расширяющим методом. В итоге предпочтение отдается новому методу, а не расширяющему, поэтому в результате компиляции кода из листинга 10.4 для .NET 4 будет применяться `Stream.CopyTo()`, а не `StreamUtil.CopyTo()`. Метод класса `StreamUtil` по-прежнему можно вызывать статически, используя нормальный синтаксис `StreamUtil.CopyTo(input, output)`, но он никогда не будет выбран как расширяющий метод. В данном случае существующему коду никакого вреда не приносится: новый метод экземпляра имеет тот же смысл, что и ваш расширяющий метод, поэтому не имеет значения, какой из них применяется. В других случаях могут быть тонкие отличия в семантике, которые иногда трудно обнаружить до нарушения работы кода.

Еще одна потенциальная проблема заключается в том, что способ, которым расширяющие методы делаются доступными коду, чрезвычайно широкомасштабен. Если пространство имен содержит два класса, которые имеют методы с тем же самым рас-

ширенным типом, то нет никакого приема, который бы позволил работать только с расширяющими методами из какого-то одного класса. Аналогично, не существует способа импортирования пространства имен ради того, чтобы типы стали доступными через их простые имена, не делая одновременно доступными также и расширяющие методы, определенные внутри этого пространства имен. Чтобы смягчить эту проблему, можно применять пространство имен, которое содержит исключительно статические классы с расширяющими методами, если только не окажется, что остальная функциональность этого пространства имен уже сильно зависит от расширяющих методов (как в ситуации с `System.Linq`, например).

Один аспект расширяющих методов может довольно-таки удивить, когда вы впервые столкнетесь с ним, однако он также удобен в некоторых ситуациях. Речь идет о ссылках `null` — давайте взглянем на них.

10.2.4. Вызов метода на ссылке `null`

Любой, у кого есть хоть сколько-нибудь значимый опыт программирования для .NET, должен был иметь дело с исключением `NullReferenceException`, возникающим из-за вызова метода через переменную, значением которой оказывалась ссылка `null`. Вызывать методы экземпляра на ссылках `null` в C# нельзя (хотя сам язык IL поддерживает это для невиртуальных вызовов), но расширяющие методы *могут* быть вызваны со ссылкой `null`. Это демонстрируется в листинге 10.5. Обратите внимание, что это не фрагмент кода, т.к. вложенные классы не могут содержать расширяющие методы.

Листинг 10.5. Расширяющий метод, вызываемый на ссылке `null`

```
using System;

public static class NullUtil
{
  public static bool IsNull(this object x)
  {
    return x == null;
  }
}

public class Test
{
  static void Main()
  {
    object y = null;
    Console.WriteLine(y.IsNull());
    y = new object();
    Console.WriteLine(y.IsNull());
  }
}
```

Код из листинга 10.5 выведет на консоль `True` и затем `False`. Если бы `IsNull()` был обычным методом экземпляра, то во второй строке `Main()` сгенерировалось бы исключение; взамен `IsNull()` вызывается с передачей `null` в качестве аргумента. До появления расширяющих методов в C# отсутствовали безопасные способы написания кода в форме, которая была бы более читабельной, чем `y.IsNull()`; вместо нее требовалось использовать `NullUtil.IsNull(y)`.

В рамках инфраструктуры имеется один особенно очевидный пример, где этот аспект поведения расширяющих методов мог бы оказаться полезным: `string.IsNullOrEmpty()`. В C# 3 разрешено определять расширяющий метод, который имеет такую же сигнатуру (кроме наличия дополнительного параметра для расширенного типа), как и существующий статический метод в расширенном типе. Чтобы не заставлять вас читать эту фразу много раз, ниже приведен пример — даже хотя класс `string` имеет статический метод без параметров `IsNullOrEmpty()`, по-прежнему можно создавать и пользоваться следующим расширяющим методом:

```
public static bool IsNullOrEmpty(this string text)
{
  return string.IsNullOrEmpty(text);
}
```

На первых порах возможность вызова метода `IsNullOrEmpty()` на переменной, имеющей значение `null`, без генерации исключения кажется странной, особенно если вы знакомы с этим по статическим методам из .NET 2.0. Однако, на мой взгляд, код, в котором применяется расширяющий метод, намного проще для понимания. Например, читая вслух `if (name.IsNullOrEmpty())`, можно точно сказать, что оно делает (зная английский язык).

Как всегда, эксперименты помогут увидеть, что именно работает, и во время отладки кода помните о возможности использования этого приема другими. Не следует предполагать, что при вызове метода будет сгенерировано исключение, если только вы не уверены в том, что это расширяющий метод. Кроме того, хорошо подумайте, прежде чем повторно применять существующее имя для расширяющего метода — предыдущий расширяющий метод может сбить с толку читателей кода, которые знакомы только со статическим методом из инфраструктуры.

Проверка на предмет равенства null

Я уверен, что вы, будучи добросовестным разработчиком, всегда обеспечиваете в производственных методах проверку допустимости аргументов перед их обработкой. Из-за необычной характеристики расширяющих методов естественным образом возникает один вопрос: какое исключение должно быть сгенерировано, когда первым аргументом является `null` (при условии, что такое не допускается). Должно ли это быть исключение `ArgumentNullException`, как в случае обычного аргумента, или же `NullReferenceException`, которое происходит, если расширяющий метод, как назло, оказался методом экземпляра? Я рекомендую первый вариант: это по-прежнему аргумент, несмотря на то, что синтаксис расширяющего метода делает данный факт неочевидным. Такой маршрут был избран Microsoft в отношении расширяющих методов внутри инфраструктуры, поэтому с ним связано дополнительное преимущество согласованности. Наконец, имейте в виду, что расширяющие методы могут также вызываться как обычные статические методы, и в этой ситуации очевидным результатом является исключение `ArgumentNullException`.

Теперь, когда известны синтаксис и поведение расширяющих методов, мы можем перейти к рассмотрению ряда примеров этих методов, предлагаемых в виде части инфраструктуры .NET 3.5.

10.3. Расширяющие методы в .NET 3.5

Расширяющие методы инфраструктуры чаще всего используются в LINQ. Некоторые поставщики LINQ предоставляют по нескольку расширяющих методов, чтобы помочь во взаимодействии с ними, но в пространстве имен `System.Linq` есть два класса, которые заметно выделяются из общей массы: `Enumerable` и `Queryable`. Они содержат многочисленные расширяющие методы; большинство расширяющих методов в `Enumerable` оперируют на `IEnumerable<T>`, а большинство таких методов в `Queryable` работают на `IQueryable<T>`. Назначение интерфейса `IQueryable<T>` будет обсуждаться в главе 12, а пока мы сосредоточим внимание на классе `Enumerable`.

10.3.1. Первые шаги в работе с классом `Enumerable`

Класс `Enumerable` имеет *массу* методов, и цель этого раздела заключается вовсе не в том, чтобы охватить их все, а в том, чтобы предоставить достаточную информацию для самостоятельного перехода к экспериментированию с ними. Пробное использование всех возможностей, доступных в `Enumerable`, принесет немало удовольствия, и однозначно имеет смысл задействовать в своих экспериментах Visual Studio или LINQPad (вместо Snippy), т.к. средство IntelliSense очень удобно в действиях подобного рода. Кроме того, в приложении A предлагается краткое описание поведения всех методов класса `Enumerable`.

Все завершенные примеры в этом разделе имеют дело с простой ситуацией: мы начинаем с коллекции целых чисел и трансформируем ее различными способами. Реальные ситуации, скорее всего, окажутся более сложными и, как правило, будут предусматривать работу с типами, связанными с бизнес-правилами. По этой причине в конце раздела будет представлена пара примеров трансформаций, применяемых в возможных реальных случаях, а их полный исходный код доступен для загрузки на веб-сайте. Однако данные примеры труднее разбирать, чем примеры с простой коллекцией чисел.

По мере чтения этой главы полезно заглядывать в недавно выполненные проекты; подумайте, можно ли было сделать их код проще или читабельнее за счет использования описываемых здесь операций.

В классе `Enumerable` есть несколько методов, не являющихся расширяемыми, и мы будем применять один из них в примерах, приводимых далее в этой главе. Метод `Range()` принимает два параметра типа `int`: начальное число и количество выдаваемых результатов. Результатом будет экземпляр `IEnumerable<int>`, возвращающий по одному числу за раз в очевидной манере.

Для демонстрации функционирования метода `Range()` и создания рабочей инфраструктуры давайте выведем на консоль числа от 0 до 9, как показано в листинге 10.6.

Листинг 10.6. Использование метода `Enumerable.Range()` для вывода на консоль чисел от 0 до 9

```
var collection = Enumerable.Range(0, 10);
foreach (var element in collection)
{
  Console.WriteLine(element);
}
```

В листинге 10.6 расширяющие методы не вызываются, а производится обращение только к простому статическому методу. Действительно, этот код только выводит на консоль числа от 0 до 9 — я никогда и не заявлял, что он делает что-то совершенно необычное.

Отложенное выполнение

Метод `Range()` не строит список с необходимыми числами — он просто выдает их в подходящие моменты времени. Другими словами, конструирование перечислимого экземпляра не требует большой работы; он приведен в готовность, поэтому данные могут быть оперативно предоставлены в нужной точке. Такое поведение называется *отложенным выполнением* — с аналогичным видом поведения вы уже сталкивались при рассмотрении итераторных блоков в главе 6, а в следующей главе вы встретитесь с еще большим количеством случаев.

Пожалуй, самое простое, что можно сделать с последовательностью чисел, которая уже упорядочена — это изменить порядок на обратный. В листинге 10.7 для этого применяется расширяющий метод `Reverse()` — он возвращает экземпляр `IEnumerable<T>`, выдающий те же самые элементы, что и исходная последовательность, но в обратном порядке.

Листинг 10.7. Изменение порядка следования элементов в коллекции на обратный с помощью метода `Reverse()`

```
var collection = Enumerable.Range(0, 10).Reverse();
foreach (var element in collection)
{
  Console.WriteLine(element);
}
```

Код вполне предсказуемо выводит на консоль 9, 8, 7 и так далее до 0 включительно. Здесь вызывается метод `Reverse()` (по внешнему виду) на `IEnumerable<int>` и тот же самый тип возвращается. Такой шаблон возвращения одного перечислимого экземпляра, основанного на другом, характерен для класса `Enumerable`.

Эффективность: сравнение буферизации и организации потока

Расширяющие методы, предоставляемые инфраструктурой, организуют поток или конвейер данных везде, где это возможно. Когда у итератора запрашивается следующий элемент, он часто получается от связанного с ним итератора, обрабатывается и затем возвращается, предпочтительно без использования дополнительного хранилища. С таким подходом хорошо сочетаются простые трансформации и фильтры, которые представляют собой мощное средство обработки данных, когда это возможно, но некоторые операции, такие как изменение порядка на обратный или сортировка, требуют доступности всех данных, поэтому данные загружаются в память полностью с целью массовой обработки. Отличие между таким буферизованным подходом и организацией конвейера похоже на отличие между чтением данных путем загрузки целого объекта `DataSet` и применением экземпляра `DataReader` для обработки по одной записи за раз. Важно принимать во внимание, что именно требуется во время использования LINQ — единственный вызов метода может привести к значительным последствиям в плане производительности. Организацию потока также называют *ленивым выполнением*, а буферизацию — *энергичным выполнением*. Например, метод `Reverse()` применяет отложенное выполнение (ничего не делая до первого вызова `MoveNext()`), но затем энергично обрабатывает свой источник данных. Лично мне термины *ленивое* и *энергичное* не нравятся, поскольку для разных людей они означают разные вещи (тема, которую я поднимаю в статье "Just how lazy are you?" ("Так насколько вы ленивы?") своего блога: `http://mng.bz/3LLM`).

Давайте теперь сделаем что-то необычное — воспользуемся лямбда-выражением для удаления четных чисел.

10.3.2. Фильтрация с помощью метода Where() и соединение обращений к методам в цепочку

Расширяющий метод `Where()` предлагает простой, но мощный способ фильтрации коллекций. Он принимает предикат, который применяет к каждому элементу исходной коллекции. Метод возвращает результирующую коллекцию `IEnumerable<T>`, содержащую все элементы, которые соответствовали предикату.

В листинге 10.8 это демонстрируется на примере применения фильтра "четный/нечетный" к коллекции целых чисел перед изменением порядка следования ее элементов на противоположный. Вы не *обязаны* использовать здесь лямбда-выражение; например, можно было бы применить созданный ранее делегат или анонимный метод. В этом случае (и во многих других реальных ситуациях) логику фильтрации проще реализовать внутристрочно, и лямбда-выражение обеспечит коду лаконичность.

Листинг 10.8. Использование метода Where() с лямбда-выражением для поиска нечетных чисел

```
var collection = Enumerable.Range(0, 10)
                           .Where(x => x % 2 != 0)
                           .Reverse();
foreach (var element in collection)
{
  Console.WriteLine(element);
}
```

Код из листинга 10.8 выводит на консоль числа 9, 7, 5, 3 и 1. Надеюсь, вы заметили сформированный шаблон — вызовы методов соединяются в цепочку. Сама идея такого соединения в цепочку не нова. Например, метод `StringBuilder.Replace()` всегда возвращает экземпляр, на котором он вызван, позволяя записывать код вроде следующего:

```
builder = builder.Replace("<", "&lt;")
                 .Replace(">", "&gt;")
                 ...
```

В отличие от него, метод `String.Replace()` возвращает строку, однако каждый раз новую — это дает возможность соединения в цепочку, но несколько отличающимся способом. Оба шаблона полезно знать; шаблон "возврат той же самой ссылки" хорошо работает для изменяемых типов, тогда как для неизменяемых типов требуется шаблон "возврат нового экземпляра, который является копией исходного с некоторыми изменениями".

Соединение в цепочку методов экземпляра наподобие `String.Replace()` и `StringBuilder.Replace()` всегда было простым, но расширяющие методы позволяют строить цепочки из вызовов *статических* методов. *Это одна из основных причин существования расширяющих методов.* Так, они *полезны* в других вспомогательных классов, однако их истинная мощь связана с этой возможностью естественного соединения в цепочки статических методов. Именно потому расширяющие методы первоначально обнаружились в классах `Enumerable` и `Queryable` внутри .NET: язык LINQ приспособил этот подход к обработке данных, при которой информация эффективно перемещается по конвейерам, созданным цепочками отдельных операций.

Соображение по поводу эффективности: изменяйте порядок вызова методов во избежание нежелательных накладных расходов

Я не являюсь сторонником микрооптимизации без веских причин, но в листинге 10.8 полезно взглянуть на порядок вызова методов. Вызов `Where()` можно было бы поместить после вызова `Reverse()` и получить те же самые результаты, но некоторые усилия затрачивались бы впустую — вызов `Reverse()` должен был бы обрабатывать четные числа, поступающие из последовательности, несмотря на то, что они будут отброшены из конечного результата. В этом случае разница была бы не особенно большой, но в реальных ситуациях на производительность могло быть оказано значительное влияние; если имеется возможность сократить объем ненужной работы без нанесения ущерба читабельности, то разумно так и поступать. Тем не менее, это не означает, что фильтры всегда должны помещаться в начало конвейера; любое переупорядочивание должно тщательно обдумываться, чтобы было гарантировано получение корректных результатов.

Существуют два очевидных подхода к написанию первой части листинга 10.8, не учитывающие тот факт, что `Reverse()` и `Where()` являются расширяющими методами. Один из них предусматривает применение временной переменной, которая сохраняет коллекцию незатронутой:

```
var collection = Enumerable.Range(0, 10);
collection     = Enumerable.Where(collection, x => x % 2 != 0)
collection     = Enumerable.Reverse(collection);
```

Надеюсь, вы согласитесь с тем, что смысл этого кода намного менее очевиден, чем смысл кода в листинге 10.8.

Второй подход, при котором все записывается в стиле единственного оператора, еще более ухудшает положение:

```
var collection = Enumerable.Reverse
                 (Enumerable.Where(Enumerable.Range(0, 10), x => x % 2 != 0));
```

Порядок вызова методов кажется обратным, поскольку первым будет выполнен самый внутренний вызов (`Range()`), а затем остальные, и выполнение в таком случае направлено изнутри наружу. Даже всего лишь с тремя вызовами методов код выглядит неуклюже, но с ростом количества операций все становится намного хуже.

Прежде чем двигаться дальше, давайте подумаем о том, что делает метод `Where()`.

10.3.3. Антракт: разве мы не видели метод `Where()` раньше?

Если метод `Where()` кажется знакомым, то это потому, что он был реализован в главе 6. Необходимо только преобразовать код в листинге 6.9 в расширяющий метод и изменить тип делегата из `Predicate<T>` на `Func<T,bool>`, в результате чего получается вполне приличная альтернативная реализация метода `Enumerable.Where()`:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                       Func<T, bool> predicate)
{
   if (source == null || predicate == null)
   {
      throw new ArgumentNullException();
   }
   return WhereImpl(source, predicate);
}
```

```
private static IEnumerable<T> WhereImpl<T>(IEnumerable<T> source,
                                           Func<T, bool> predicate)
{
  foreach (T item in source)
  {
    if (predicate(item))
    {
      yield return item;
    }
  }
}
```

Можно также изменить последнюю часть листинга 6.9, сделав ее больше похожей на стиль LINQ:

```
foreach (string line in LineReader.ReadLines("../../FakeLinq.cs")
                                  .Where(line => line.StartsWith("using")))
{
  Console.WriteLine(line);
}
```

В сущности это запрос LINQ, не использующий пространство имен `System.Linq`. Он будет довольно хорошо работать в .NET 2.0, если объявить соответствующий делегат `Func` и атрибут `[ExtensionAttribute]`. Как будет показано в следующей главе, эту реализацию можно было бы даже применять для конструкции `where` в выражении запроса (по-прежнему имея в качестве целевой платформу .NET 2.0), но давайте не будем забегать вперед.

Фильтрация является одной из простейших операций в запросе, а другой такой операцией считается трансформирование или *проецирование* результатов.

10.3.4. Проецирование с использованием метода `Select()` и анонимных типов

Наиболее часто используемым методом проецирования в `Enumerable` является `Select()`. Он оперирует на экземпляре `IEnumerable<TSource>` и проецирует его в `IEnumerable<TResult>` с применением делегата `Func<TSource,TResult>`, который представляет собой выраженную в виде делегата трансформацию, используемую в отношении каждого элемента. Метод `Select()` очень похож на метод `ConvertAll()` в `List<T>`, но работает с любыми перечислимыми коллекциями и использует отложенное выполнение для построения проекции, когда требуется каждый элемент.

Во время представления анонимных типов я утверждал, что их удобно применять с лямбда-выражениями и LINQ — и ниже приведен пример того, что с ними можно делать. В настоящий момент имеются нечетные числа от 0 до 9 (в обратном порядке) — так давайте создадим тип, который инкапсулирует квадратный корень вместе с исходным числом. В листинге 10.9 демонстрируется проецирование и несколько модифицированный способ вывода результатов на консоль. Отступы подкорректированы исключительно из-за ограниченного пространства печатной страницы.

Листинг 10.9. Проецирование с использованием лямбда-выражения и анонимного типа

```
var collection = Enumerable.Range(0, 10)
     .Where(x => x % 2 != 0)
     .Reverse()
     .Select(x => new { Original = x, SquareRoot = Math.Sqrt(x) } );
```

```
foreach (var element in collection)
{
  Console.WriteLine("sqrt({0})={1}",
                    element.Original,
                    element.SquareRoot);
}
```

На этот раз типом collection является не IEnumerable<int>, а IEnumerable <Something>, где Something — анонимный тип, созданный компилятором. Переменной collection нельзя назначить явный тип, отличный от необобщенного типа IEnumerable или object. Неявная типизация (с помощью var) — это то, что позволяет применять свойства Original и SquareRoot при выводе результатов на консоль.

Вывод кода из листинга 10.9 выглядит следующим образом:

```
sqrt(9)=3
sqrt(7)=2.64575131106459
sqrt(5)=2.23606797749979
sqrt(3)=1.73205080756888
sqrt(1)=1
```

Разумеется, метод Select() не *обязан* использовать анонимный тип вообще — можно было бы выбрать только квадратный корень числа, отбросив исходное число. В этом случае результатом был бы экземпляр IEnumerable<double>. В качестве альтернативы можно было бы вручную написать тип, инкапсулирующий целое число и квадратный корень — просто в приведенном случае легче было воспользоваться анонимным типом.

Давайте рассмотрим еще один метод, чтобы оставив на время обзор класса Enumerable: OrderBy().

10.3.5. Сортировка с использованием метода OrderBy()

Сортировка относится к распространенным требованиям при обработке данных, и в LINQ она обычно осуществляется с применением метода OrderBy() или OrderByDescending(). Иногда за этим первым вызовом следует обращение к ThenBy() либо ThenByDescending(), если необходимо сортировать по более чем одному свойству данных. Сортировка по нескольким свойствам всегда давалась нелегко, с использованием сложного сравнения, но возможность представления вместо этого последовательности простых сравнений сделала бы ее намного яснее.

Чтобы продемонстрировать сказанное, внесем небольшое изменение в задействованные операции. Мы начнем с целых чисел от -5 до 5 (включительно, так что всего имеется 11 элементов), после чего спроецируем их на анонимный тип, содержащий исходное число и его квадрат (вместо квадратного корня). Наконец, мы выполним сортировку последовательности по квадрату и затем по исходному числу. Все это показано в листинге 10.10.

Листинг 10.10. Упорядочение последовательности по двум свойствам

```
var collection = Enumerable.Range(-5, 11)
     .Select(x => new { Original = x, Square = x * x })
     .OrderBy(x => x.Square)
     .ThenBy(x => x.Original);
foreach (var element in collection)
{
  Console.WriteLine(element);
}
```

Обратите внимание, что кроме вызова `Enumerable.Range()` код читается почти как текстовое описание (на английском языке). На этот раз реализация метода `ToString()` анонимного типа осуществляет форматирование, так что результат выглядит следующим образом:

```
{ Original = 0, Square = 0 }
{ Original = -1, Square = 1 }
{ Original = 1, Square = 1 }
{ Original = -2, Square = 4 }
{ Original = 2, Square = 4 }
{ Original = -3, Square = 9 }
{ Original = 3, Square = 9 }
{ Original = -4, Square = 16 }
{ Original = 4, Square = 16 }
{ Original = -5, Square = 25 }
{ Original = 5, Square = 25 }
```

Как и предполагалось, главным свойством, по которому производится сортировка, является `Square`, но когда два значения дают одинаковый результат при возведении в квадрат, после сортировки отрицательное значение всегда будет находиться перед положительным. Написание одиночного сравнения, которое делает то же самое (в общем случае — существуют математические трюки, позволяющие справиться с этим конкретным примером), оказалось бы настолько сложнее, что вы отказались бы от помещения кода внутрь лямбда-выражения.

Следует отметить один момент — упорядочение не изменяет существующую коллекцию, а возвращает новую последовательность, которая выдает те же самые данные, что и исходная последовательность, но только отсортированные. Сравните это с методами `List<T>.Sort()` или `Array.Sort()`, которые оба изменяют порядок следования элементов внутри списка или массива.

Операции LINQ спроектированы как *свободные от побочных эффектов*: они не оказывают влияния на свои входные данные и не привносят любые другие изменения в среду, если только не производится проход по естественно поддерживающей состояние последовательности (вроде чтения из сетевого потока) или аргумент делегата не имеет побочных эффектов. Этот подход взят из функционального программирования, и он приводит к получению кода, который является более читабельным, тестируемым, компонуемым, предсказуемым, безопасным в отношении потоков и надежным в работе.

Мы рассмотрели лишь несколько из множества расширяющих методов, доступных в классе `Enumerable`, но есть надежда, что вы смогли по достоинству оценить, насколько аккуратно они могут соединяться в цепочки. В следующей главе вы увидите, как это может быть выражено по-другому с применением дополнительного синтаксиса, предоставляемого C# 3 (выражения запросов), и будут показаны другие операции, которые здесь не рассматривались. Стоит запомнить, что вы не *обязаны* использовать выражения запросов — часто проще сделать пару обращений к методам `Enumerable` и с помощью расширяющих методов соединить операции в цепочку.

Теперь, когда вы ознакомились с примером, в котором участвовала коллекция чисел, наступило время исполнить обещание, касающееся демонстрации нескольких реальных бизнес-примеров.

10.3.6. Бизнес-примеры, предусматривающие соединение вызовов в цепочки

Большинство из того, что мы делаем как разработчики, так или иначе связано с перемещением данных. На самом деле во многих приложениях это является *единственным* значимым действием — пользовательский интерфейс, веб-службы, база данных и другие компоненты часто существуют исключительно ради получения данных в одном месте и помещения их в другое место или преобразования их из одной формы в другую. Не должен быть удивительным тот факт, что рассматриваемые в этом разделе расширяющие методы хорошо подходят для решения многих бизнес-задач.

Я приведу здесь только пару примеров. Я уверен, что вы сумеете понять, каким образом с помощью C# 3 и класса `Enumerable` более выразительно решить задачи, касающиеся ваших бизнес-требований. Для каждого примера будет включен только запрос — его должно быть достаточно, чтобы понять назначение кода. Полный рабочий код доступен на веб-сайте, посвященном книге.

Агрегирование: подведение итогов по заработной плате

В первом примере задействована компания, состоящая из нескольких отделов. В каждом отделе работает определенное количество сотрудников, которые получают заработную плату. Предположим, что необходимо сформировать отчет по суммарной заработной плате для всех отделов, с указанием более затратных отделов первыми. Запрос выглядит следующим образом:

```
company.Departments
       .Select(dept => new
       {
         dept.Name,
         Cost = dept.Employees.Sum(person => person.Salary)
       })
       .OrderByDescending(deptWithCost => deptWithCost.Cost);
```

В этом запросе для хранения названия отдела (с помощью инициализатора проекции) и суммарной заработной платы всех его сотрудников используется анонимный тип. Суммирование заработной платы осуществляется с применением расширяющего метода `Sum()`, доступного в классе `Enumerable`.

В результате название отдела и общая заработная плата могут извлекаться как свойства. Если необходима ссылка на экземпляр отдела, понадобится изменить анонимный тип в методе `Select()` на соответствующий именованный тип.

Группирование: подсчет дефектов, подлежащих исправлению разработчиками

Если вы — профессиональный разработчик, то наверняка сталкивались со многими инструментами управления проектами, которые выдают различные метрики. При наличии доступа к низкоуровневым данным LINQ может помочь трансформировать их практически любым необходимым способом.

В качестве простого примера давайте рассмотрим список разработчиков и количество подлежащих исправлению ими дефектов к настоящему моменту:

```
bugs.GroupBy(bug => bug.AssignedTo)
    .Select(list => new { Developer = list.Key, Count = list.Count() })
    .OrderByDescending(x => x.Count);
```

В этом запросе применяется расширяющий метод `GroupBy()`, который группирует исходную коллекцию посредством проецирования (в данном случае разработчик, которому поручено исправить дефект), давая в результате экземпляр `IGrouping<TKey,TElement>`. Существует множество перегруженных версий метода `GroupBy()`, но в этом примере используется простейшая из них, которая выбирает только ключ (имя разработчика) и количество подлежащих исправлению дефектов. После этого результат упорядочивается, чтобы разработчики с большим числом дефектов оказались в начале списка.

Одной из проблем, возникающих при работе с классом `Enumerable`, является выяснение того, что в точности происходит; например, среди перегруженных версий метода `GroupBy()` есть такая, которая принимает четыре параметра типов и пять обычных параметров (три из которых представляют собой делегаты). Сохраняйте спокойствие — просто следуйте шагам, описанным в главе 3, назначая разные типы параметрам типов до тех пор, пока не будет получен конкретный пример того, как должен выглядеть метод. Обычно это значительно облегчает понимание происходящего.

Приведенные примеры нельзя назвать особо увлекательными, но я надеюсь, что вы смогли оценить мощь соединения обращений к методам в цепочки, где каждый метод принимает исходную коллекцию и возвращает другую коллекцию в несколько иной форме, фильтруя значения, упорядочивая их, трансформируя элементы, выполняя агрегирование некоторых значений или применяя другие варианты. Во многих случаях результирующий код можно читать вслух и сразу же понимать, а в других ситуациях он по-прежнему намного проще эквивалентного кода, который пришлось бы писать в предшествующих версиях C#.

Пример данных по отслеживанию дефектов будет использоваться в следующей главе при рассмотрении выражений запросов. Теперь, после ознакомления с некоторыми расширяющими методами, давайте подумаем, когда имеет смысл создавать такие методы самостоятельно.

10.4. Идеи и руководство по использованию

Подобно неявной типизации локальных переменных, расширяющие методы вызывают споры. Нельзя сказать, что во многих случаях они затрудняют понимание общей цели кода, но в то же время они *скрывают* детали того, какой метод будет вызван. Один из лекторов в моем университете говорил примерно так: "Я скрываю правду, чтобы показать вас *большую* правду". Если вы считаете, что наиболее важным аспектом кода является его результат, то расширяющие методы великолепны. Если для вас важнее реализация, то явный вызов статических методов дает более ясный код. В сущности это сводится к разнице между *что* и *как*.

Мы уже видели применение расширяющих методов в служебных классах и при соединении вызовов в цепочки, но прежде чем перейти к обсуждению всех за и против, полезно рассмотреть пару аспектов, которые могут быть неочевидными.

10.4.1. "Расширение мира" и совершенствование интерфейсов

Вес Дайер, бывший разработчик в команде, занимающейся созданием компилятора C#, вел превосходный блог, в котором раскрывались все связанные с этим темы (`http://blogs.msdn.com/b/wesdyer/`). Одна из статей в его блоге, посвященная расширяющим методам (`http://mng.bz/I4F2`), привлекла мое особое внимание. Статья называлась "Extending the World" ("Расширение мира") и в ней шла речь о том,

как расширяющие методы могут сделать код проще для чтения, фактически приспосабливая среду к своим потребностям.

Для заданной задачи типичным является то, что программист приучен строить решение до тех пор, пока оно, наконец, не начнет удовлетворять требованиям. Теперь стало возможным расширить мир с целью удовлетворения требованиям решения вместо одного лишь построения для удовлетворения требований мира. Если данная библиотека не предоставляет того, что нужно, просто расширьте ее, чтобы она удовлетворяла существующим требованиям.

Последствия этого выходят за рамки ситуаций, в которых использовался бы вспомогательный класс. Обычно разработчики начинают создавать вспомогательные классы только после того, когда обнаруживают, что определенная разновидность кода воспроизводится в десятках мест, но расширение библиотеки направлено не столько ради ясности выражения, сколько ради избегания дублирования кода. Расширяющие методы могут обеспечить для вызывающего кода впечатление того, что библиотека богаче, чем есть в действительности.

Вы уже видели это на примере интерфейса `IEnumerable<T>`, где даже простейшая реализация *выглядит* как имеющая широкий набор доступных операций, таких как сортировка, группирование и фильтрация. Преимущества не ограничены интерфейсами – "расширить мир" можно также с помощью перечислений, абстрактных классов и т.д.

Платформа .NET Framework также предлагает хороший пример еще одного применения расширяющих методов: текучие интерфейсы.

10.4.2. Текучие интерфейсы

В Великобритании есть телевизионная программа под названием *Catchphrase* ("Броская фраза"). Идея заключается в том, что участники соревнования просматривают мультфильм с зашифрованной версией фразы или поговорки, которую они должны угадать. Ведущий программы часто пытается помочь, инструктируя: "Скажите, что вы видите". Почти такая же идея лежит в основе *текучих интерфейсов* — если вы читаете код дословно, его предназначение станет ясным, как если бы он был написан на обычном английском языке. Термин "текучие интерфейсы" был впервые применен Мартином Фаулером (см. статью в его блоге по адресу `http://mng.bz/3T9T`) и Эриком Эвансом.

Если вы знакомы с *предметно-ориентированными языками* (domain-specific language – DSL), то вас может заинтересовать, в чем состоят отличия между текучим интерфейсом и DSL. По этой теме написано немало, но кажется, что выработанное общими усилиями мнение заключается в том, что DSL обладает большей свободой в создании собственного синтаксиса и грамматики, тогда как текучий интерфейс ограничен базовым языком (в данном случае C#).

Хорошими примерами текучих интерфейсов внутри инфраструктуры являются методы `OrderBy()` и `ThenBy()`: благодаря некоторому участию лямбда-выражений, код точно представляет то, что он делает. Код, показанный ранее в листинге 10.10, можно было без особого труда прочитать как "упорядочить по значению квадрата, а затем по исходному числу". В итоге операторы читаются как цельные предложения, а не отдельные конструкции в виде существительных и глаголов (английского языка).

Написание текучих интерфейсов может требовать изменения типа мышления. Для имен методов не должна обязательно применяться обычная форма описательных глаголов, т.к. в текучем интерфейсе методам иногда больше подходят имена вроде `And`, `Then` и `If`.

Сами методы часто всего лишь настраивают контекст для будущих вызовов, возвращая тип, единственное назначение которого заключается в том, чтобы действовать в качестве шлюза между вызовами. На рис. 10.2 иллюстрируется работа такого моста. Несмотря на использование всего двух расширяющих методов (на `int` и `TimeSpan`), разницу в читабельности заметить легко.

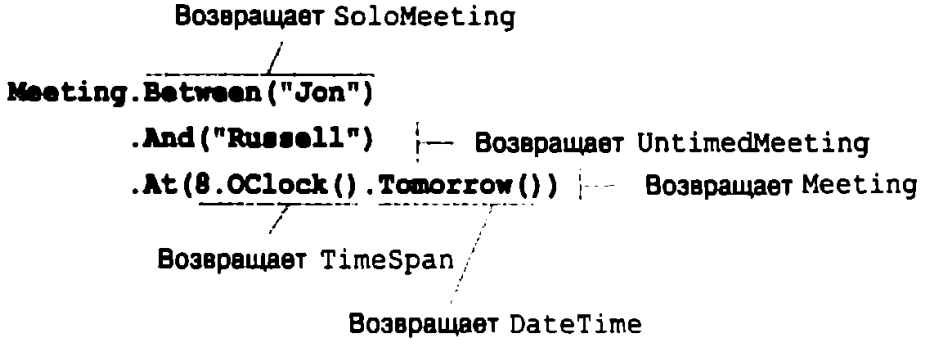

Рис. 10.2. Разбор выражения с текучим интерфейсом, которое создает объект, представляющий встречу. Время встречи указано с применением расширяющих методов для создания `TimeSpan` из `int` и `DateTime` из `TimeSpan`

Грамматически пример на рис. 10.2 мог бы принимать множество разных форм; скажем, есть возможность добавить к экземпляру `UntimedMeeting` дополнительные объекты участников или создать экземпляр `UnattendedMeeting` в определенное время до указания участников. За более подробными сведениями о языках DSL обращайтесь к книге Орена Эйни (псевдоним Ayende Rahien) *DSLs in Boo: Domain-Specific Languages in .NET* (Manning, 2010 г.).

В C# 3 поддерживаются только расширяющие *методы*, но не расширяющие *свойства*, что слегка ограничивает текучие интерфейсы. Это означает невозможность иметь выражения, подобные `1.week.from.now` или `2.days + 10.hours` (которые оба являются допустимыми в Groovy с подходящим пакетом: `ttp://groovy.codehaus.org/Google+Data+Supportttp://groovy.codehaus.org/Google+Data+Support`), но с помощью нескольких избыточных круглых скобок можно достигнуть аналогичных результатов. Поначалу вызов метода на числе выглядит странно (например, `2.Dollars()` или `3.Meters()`), но трудно отрицать, что смысл такого вызова очевиден. Без расширяющих методов подобная ясность невозможна, когда необходимо действовать на таких типах, как числа, которые не находятся под вашим контролем.

На момент написания этих строк в отношении текучих интерфейсов сообщество разработчиков по-прежнему занимало выжидательную позицию: они относительно редко применяются в большинстве областей, хотя многие библиотеки имитации и модульного тестирования обладают хотя бы минимальным аспектом текучести. Текучие интерфейсы определенно не являются универсально применимыми, но в подходящих ситуациях они могут радикально улучшить читабельность вызываемого кода. В качестве примера с помощью соответствующих расширяющих методов из моей библиотеки MiscUtil я могу реализовать проход по всем дням в читабельной форме:

```
foreach (DateTime day in 19.June(1976).To(DateTime.Today).Step(1.Days()))
```

Хотя детали реализации, связанные с диапазоном, сложны, расширяющие методы, позволяющие записывать `19.June(1976)` и `1.Days()`, исключительно просты. Этот специфичный к культуре код вряд ли появится в производственном решении, однако он может значительно облегчить написание модульных тестов.

Разумеется, это не единственная область использования для расширяющих методов. Я применяю их для проверки достоверности аргументов, реализации подходов, альтернативных LINQ, добавления собственных операций к LINQ to Objects, упрощения способов построения сложных сравнений, добавления к перечислениям функциональности, связанной с флагами, и многого другого. Я не перестаю удивляться, как такое простое средство может настолько глубоко влиять на читабельность при условии его подходящего использования. Ключевым словом здесь является "подходящего", которое легче произнести, чем объяснить.

10.4.3. Разумное использование расширяющих методов

Я не собираюсь указывать вам, как писать свой код. Может и возможно написать тесты, объективно измеряющие читабельность для усредненного разработчика, но это будет иметь значение только для тех, кто планирует использовать и сопровождать ваш код. Необходимо по возможности проконсультироваться с соответствующими людьми, представив разные варианты и получив по ним отзывы. Расширяющие методы во многих случаях делают это довольно простым, поскольку в рабочем коде можно одновременно продемонстрировать оба варианта — превращение метода в расширяющий не препятствует его явному вызову тем же способом, как это делалось ранее.

Главный вопрос, который следует задать, и который упоминался в начале данного раздела, звучит так: является ли аспект "что он делает", связанный с кодом, более важным, чем аспект "как он это делает"? Ответ зависит от человека и ситуации, но ниже перечислены руководящие указания, которые необходимо иметь в виду.

- Каждый член команды разработчиков должен знать расширяющие методы и где они могут применяться. По возможности старайтесь не преподносить сюрпризы тем, кто будет сопровождать код.
- Помещая расширения в собственное пространство имен, вы затрудняете случайное их использование. Даже если это не очевидно при чтении кода, разработчики, пишущие код, должны отдавать себе отчет в том, что они делают. При назначении названия этому пространству имен применяйте соглашение на уровне проекта или всей компании. Вы можете пойти еще дальше и использовать единое пространство имен для каждого расширенного типа. Например, можно было бы создать пространство имен `TypeExtensions` для классов, которые расширяют `System.Type`.
- Хорошо подумайте, прежде чем расширять повсеместно применяемые типы, такие как числовые типы или `object`, либо писать метод, в котором расширенный тип является параметром типа. В ряде руководств даже рекомендуется никогда не поступать подобным образом; я считаю, что такие расширения имеют право на существование, но они должны действительно заслужить свое место в вашей библиотеке. В этой ситуации становится даже еще более важным то, чтобы расширяющий метод был внутренним или находился в собственном пространстве имен. К примеру, я не хотел бы, чтобы средство IntelliSense предлагало мне расширяющий метод `June()` везде, где я использую целое число, а только в классах, в которых применяются хоть какие-то расширяющие методы, связанные с датой и временем.
- Решение о написании расширяющего метода всегда должно быть осознанным. Оно не должно превращаться в привычку. Не каждый статический метод заслуживает того, чтобы стать расширяющим методом.

- Документируйте, разрешено ли первому параметру (значение, на котором, как кажется, метод вызывается) принимать значение `null` — если нет, проверяйте значение в методе и при необходимости генерируйте исключение `ArgumentNullException`.
- Будьте осторожны, чтобы не использовать имя метода, которое уже имеет определенный смысл в расширенном типе. Если расширенный тип является типом из инфраструктуры или поступает из независимой библиотеки, проверяйте имена всех своих расширяющих методов всякий раз, когда меняете версии библиотеки. Если вам повезет (как мне с методом `Stream.CopyTo()`), то новый смысл сохранится таким же, как ранее, но даже при этих условиях может понадобиться объявить расширяющий метод устаревшим.
- Прислушайтесь к своим инстинктам и проверьте, влияют ли расширяющие методы на продуктивность вашей работы. Подобно неявной типизации, применение средства, которое инстинктивно отталкивает, является элементом принуждения.
- Попытайтесь сгруппировать расширяющие методы в статические классы, имеющие дело с одним и тем же расширенным типом. Иногда связанные классы (такие как `DateTime` и `TimeSpan`) могут быть разумно сгруппированы вместе, но избегайте группирования внутри одного класса расширяющих методов, нацеленных на несопоставимые типы вроде `Stream` и `string`.
- По-настоящему крепко подумайте, прежде чем помещать расширяющие методы с одним и тем же расширенным типом и одинаковыми именами в два разных пространства имен, особенно если существуют ситуации, при которых эти разные методы из двух пространств имен оказываются подходящими (из-за одинакового количества параметров). Вполне нормально, если добавление или удаление директивы `using` приводит к отказу компиляции программы, но весьма неприятно, когда программа компилируется, но меняет свое поведение.

Предельно четкими являются лишь несколько описанных руководящих указаний; в некотором роде вы должны самостоятельно выработать свой стиль использования расширяющих методов или вовсе отказаться от них. Совершенно справедливо вообще никогда не писать расширяющие методы и применять только те методы, которые связаны с LINQ, для обеспечения лучшей читабельности. Однако полезно хотя бы *думать* о том, *что* с их помощью становится возможным.

10.5. Резюме

Технический аспект расширяющих методов прямолинеен — эту возможность просто описать и продемонстрировать. С другой стороны, рассуждать об их преимуществах (и плате за них) в категоричной манере труднее — это довольно эмоциональная тема, и разные люди просто обязаны иметь отличающиеся точки зрения на ценность расширяющих методов.

В этой главе я попытался показать все понемногу. В самом начале мы взглянули на то, что данное средство привносит в язык, а затем посмотрели на некоторые возможности, доступные в инфраструктуре. В каком-то смысле это было плавным введением в LINQ; мы еще вернемся к рассмотрению ряда методов, которые вы видели до сих пор, и ознакомимся с новыми методами, когда углубимся в исследование выражений запросов в следующей главе.

В рамках класса `Enumerable` доступно широкое разнообразие методов, и в этой главе мы лишь слегка коснулись поверхности. Забавно придумывать сценарий собственного изобретения (будь то гипотетический или реальный проект) и просмотреть MSDN, чтобы узнать, что именно из доступного могло бы помочь. Я призываю вас воспользоваться каким-нибудь искусственным проектом и поиграть с описанными в главе расширяющими методами — это действительно больше похоже на игру, нежели на работу, и вы, скорее всего, не захотите себя ограничивать только методами, которые нужны для достижения самой непосредственной цели. В приложении А приведен список стандартных операций запросов из LINQ, которые покрывают многие методы из класса `Enumerable`.

В отрасли разработки программного обеспечения продолжают появляться новые шаблоны и приемы, поэтому идеи из одних систем часто являются источниками идей в других системах. Это одна из характерных особенностей, которые сохраняют процесс разработки настолько захватывающим. Расширяющие методы позволяют записывать код таким способом, который ранее был невозможным в C#, создавая текучие интерфейсы и изменяя среду для удовлетворения потребностей кода, а не наоборот. Это лишь те несколько приемов, которые рассматривались в данной главе — всенепременно будут возникать интересные будущие разработки, использующие новые средства C#, по отдельности или в комбинации.

Очевидно, революционное развитие на этом не заканчивается. Для некоторых вызовов расширяющие методы хороши. В следующей главе мы займемся действительно мощными инструментами: выражениями запросов и полномасштабным LINQ.

Глава 11

Выражения запросов и LINQ to Objects

В этой главе...

- Потоковые последовательности данных и отложенное выполнение
- Стандартные операции запросов и трансляция выражений запросов
- Переменные диапазонов и прозрачные идентификаторы
- Проецирование, фильтрация и сортировка
- Соединение и группирование
- Выбор используемого синтаксиса

Возможно, к данному моменту вы уже устали от всех этих хвалебных од в адрес LINQ. Вы уже видели некоторые примеры в книге и почти наверняка многое читали о LINQ в Интернете. Именно здесь мы отделим мифы от реальности.

- LINQ не превращает самые сложные запросы в однострочные.
- LINQ не обещает, что вам никогда больше не придется снова видеть низкоуровневый код SQL.
- LINQ не может волшебным образом сделать из вас архитектурного гения.

С учетом всего этого LINQ по-прежнему является наилучшим способом выражения запросов, какой только мне приходилось видеть в рамках объектно-ориентированной среды. Конечно, это не символ технологического прорыва, однако *очень* мощный инструмент, который стоит иметь в своем арсенале средств разработки. Мы исследуем два отдельных аспекта LINQ: поддержку со стороны инфраструктуры и трансляцию компилятором *выражений запросов*. Поначалу выражения запросов могут выглядеть странными, но я уверен, что вы научитесь их любить.

Выражения запросов, в сущности, преобразуются компилятором в "нормальный" код C# 3, который затем компилируется обычным образом. Это аккуратный способ интег-

рации запросов в язык, требующий изменения всего лишь одного небольшого раздела спецификации. В большей части этой главы приводится список предварительных трансляций, выполняемых компилятором, а также эффектов, которые достигаются, когда результат использует расширяющие методы класса `Enumerable`.

Здесь вы не встретите код SQL или XML — это отложено до главы 12. Но, применяя эту главу в качестве основы, вы должны быть в состоянии понять, *что* делают остальные поставщики LINQ, когда вы столкнетесь с ними. Можете называть меня тем, кто портит настроение другим, но я хочу разоблачить часть их магии. Технология LINQ остается великолепной даже без атмосферы таинственности.

Давайте сначала обсудим основы языка LINQ и подход к его исследованию.

11.1. Введение в LINQ

Такая обширная тема, как LINQ, требует предварительного ознакомления с определенными сведениями, прежде чем вы будете готовы увидеть ее в действии. В этом разделе мы взглянем на ряд принципов, лежащих в основе LINQ, и на модель данных, которая будет использоваться в примерах, приводимых в этой и следующей главах. Скорее всего, вы испытываете непреодолимое желание с головой погрузиться в код, поэтому введение будет довольно кратким.

11.1.1. Фундаментальные концепции LINQ

Одна из задач уменьшения рассогласования между двумя моделями данных обычно предусматривает создание еще одной модели, которая будет действовать в качестве шлюза. В этом разделе приводится описание модели LINQ, начиная с наиболее важного аспекта — последовательностей.

Последовательности

Вы уже знакомы с концепцией последовательности: она инкапсулирована интерфейсами `IEnumerable` и `IEnumerable<T>` и довольно подробно рассматривалась в главе 6 при изучении итераторов. Последовательность похожа на ленту конвейера с элементами — вы извлекаете по одному элементу за раз до тех пор, пока в этом не исчезнет интерес или не закончатся данные.

Ключевое отличие между последовательностью и другими структурами коллекций данных, такими как списки и массивы, состоит в том, что при чтении последовательности количество ожидающих элементов, как правило, неизвестно, к тому же нельзя получить доступ к произвольным элементам, а только к одному текущему. Более того, некоторые последовательности могут оказаться бесконечными; например, вполне может существовать бесконечная последовательность случайных чисел. Списки и массивы могут *выступать* в качестве последовательностей, подобно тому, как `List<T>` реализует `IEnumerable<T>`, однако обратное утверждение не всегда верно. Скажем, невозможно иметь бесконечный массив или список.

Последовательности — это средства к существованию LINQ. Когда вы читаете выражение запроса, то должны думать о задействованных последовательностях; вначале всегда имеется хотя бы одна последовательность, которая обычно по ходу дела трансформируется в другие последовательности, возможно соединяясь еще с какими-то последовательностями. Примеры запросов LINQ, доступные в Интернете, часто снабжаются слабыми объяснениями, но когда вы начнете разбирать их на части, глядя на каждую последовательность по очереди, все станет более осмысленным.

Помимо помощи в *чтении* кода, такой подход хорошо помогает и при его *написании*. Мышление в терминах последовательностей может быть сложным, т.к. иногда приходится слегка менять образ мыслей, но если вы сумеете достичь цели, это окажет неизмеримую помощь при работе с LINQ.

Рассмотрим простой пример выражения запроса, выполняемого в отношении списка людей. Мы сначала применим фильтрацию, а затем проецирование, чтобы в итоге получить последовательность имен совершеннолетних людей:

```
var adultNames = from person in people
                 where person.Age >= 18
                 select person.Name;
```

На рис. 11.1 показано графическое представление этого выражения запроса, с разбиением его на отдельные шаги.

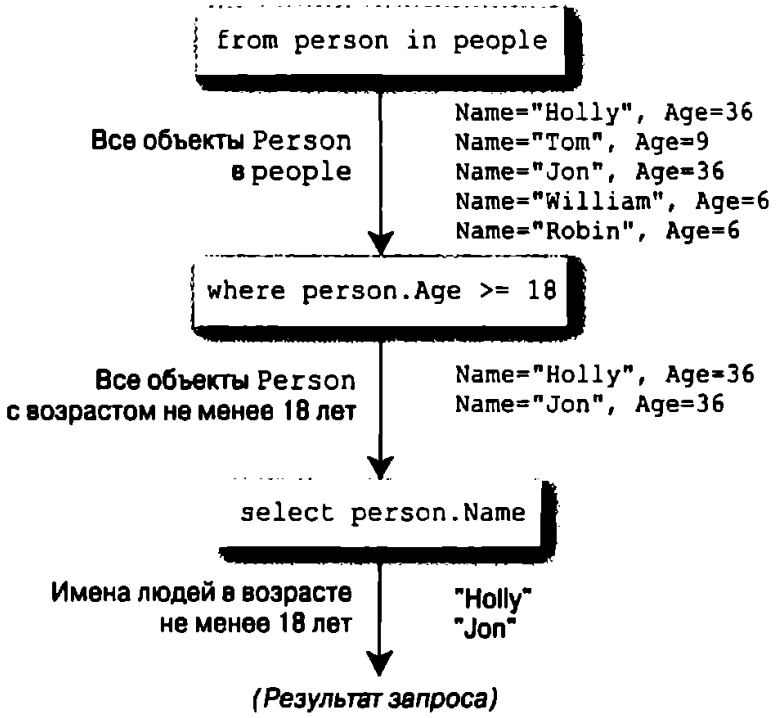

Рис. 11.1. Простое выражение запроса, разбитое на задействованные последовательности и трансформации

Каждая стрелка представляет последовательность, описание которой приведено слева, а образец данных справа. Каждый блок — это шаг внутри выражения запроса. Изначально имеется целое семейство (в виде объектов `Person`); затем после фильтрации последовательность содержит только совершеннолетних (снова в виде объектов `Person`); и, наконец, финальный результат состоит из имен этих совершеннолетних в виде строк. Каждый раз берется одна последовательность и к ней применяется операция для получения новой последовательности. Результатом являются не строки `Holly` и `Jon`, а экземпляр `IEnumerable<string>`, который при последующих запросах элементов друг за другом выдаст сначала `Holly` и затем `Jon`.

Описанный пример был прямолинеен, но мы будем применять тот же самый прием позже к более сложным выражениям запросов, что позволит их легче понять. Некоторые расширенные операции предусматривают наличие более одной последовательности на входе, но об этом по-прежнему не стоит сильно беспокоиться, как и пытаться понять весь запрос за один присест.

Так почему же последовательности настолько важны? Они являются основой потоковой модели для обработки данных — модели, которая позволяет извлекать и обрабатывать данные, только когда они необходимы.

Отложенное выполнение и организация потока

Когда выражение запроса, показанное на рис. 11.1, только создается, никакие данные не обрабатываются. Доступ к исходному списку людей не производится *вообще*[1]. Вместо этого в памяти строится представление запроса. Для представления предиката, проверяющего совершеннолетие, и преобразования объекта, хранящего сведения о человеке, в имя человека используются экземпляры делегатов. Механизм начинает работать, только когда у результирующего экземпляра `IEnumerable<string>` запрашивается первый его элемент.

Такой аспект LINQ называется *отложенным выполнением*. Когда запрашивается первый элемент результата, трансформация `Select()` обращается за первым элементом к трансформации `Where()`. Трансформация `Where()` запрашивает первый элемент у списка, проверяет его на соответствие предикату (который в данном случае дает соответствие) и возвращает этот элемент обратно `Select()`. В свою очередь, `Select()` извлекает имя и возвращает его в качестве результата.

Разве мы не видели это раньше?

Вы можете испытывать чувство, близкое к дежа-вю, поскольку все это уже упоминалось в главе 10. Однако это настолько важная тема, что полезно раскрыть ее еще раз, но уже с большими подробностями.

Как обычно, диаграмма последовательностей делает все намного яснее. Я свернул вызовы `MoveNext()` и `Current()` в единственную операцию извлечения; это намного упрощает диаграмму. Просто запомните, что каждый раз, когда происходит извлечение, оно фактически также проверяет, не закончилась ли последовательность. На рис. 11.2 представлено несколько этапов выполнения выражения запроса при выводе элементов на консоль с помощью цикла `foreach`.

Как показано на рис. 11.2, производится обработка только одного элемента за раз. Если вы решите остановить вывод на консоль после отображения строки `Holly`, то никаких операций над другими элементами исходной последовательности выполняться не будет. Несмотря на наличие здесь нескольких этапов, обработка данных в *потоковой* манере подобного рода является эффективной и гибкой. Безотносительно к объему исходных данных, в любой момент времени вы не обязаны знать больше, чем об одном элементе.

Это наилучший сценарий. Временами извлечение первого результата запроса требует оценки *всех* данных из источника. Мы уже видели один такой пример в предыдущей главе: методу `Enumerable.Reverse()` нужно было извлечь все доступные данные, чтобы возвратить последний элемент исходной последовательности в качестве первого элемента результирующей последовательности. Это делает метод `Reverse()` *буферизирующей* операцией, которая может оказывать большое влияние на эффективность (или даже осуществимость) всего действия. Если вы не можете позволить себе держать все данные в памяти одновременно, то использовать буферизирующие операции не получится.

[1] Хотя проверяются на предмет `null` различные задействованные параметры. Это важно иметь в виду при реализации собственных операций LINQ, как вы увидите в главе 12.

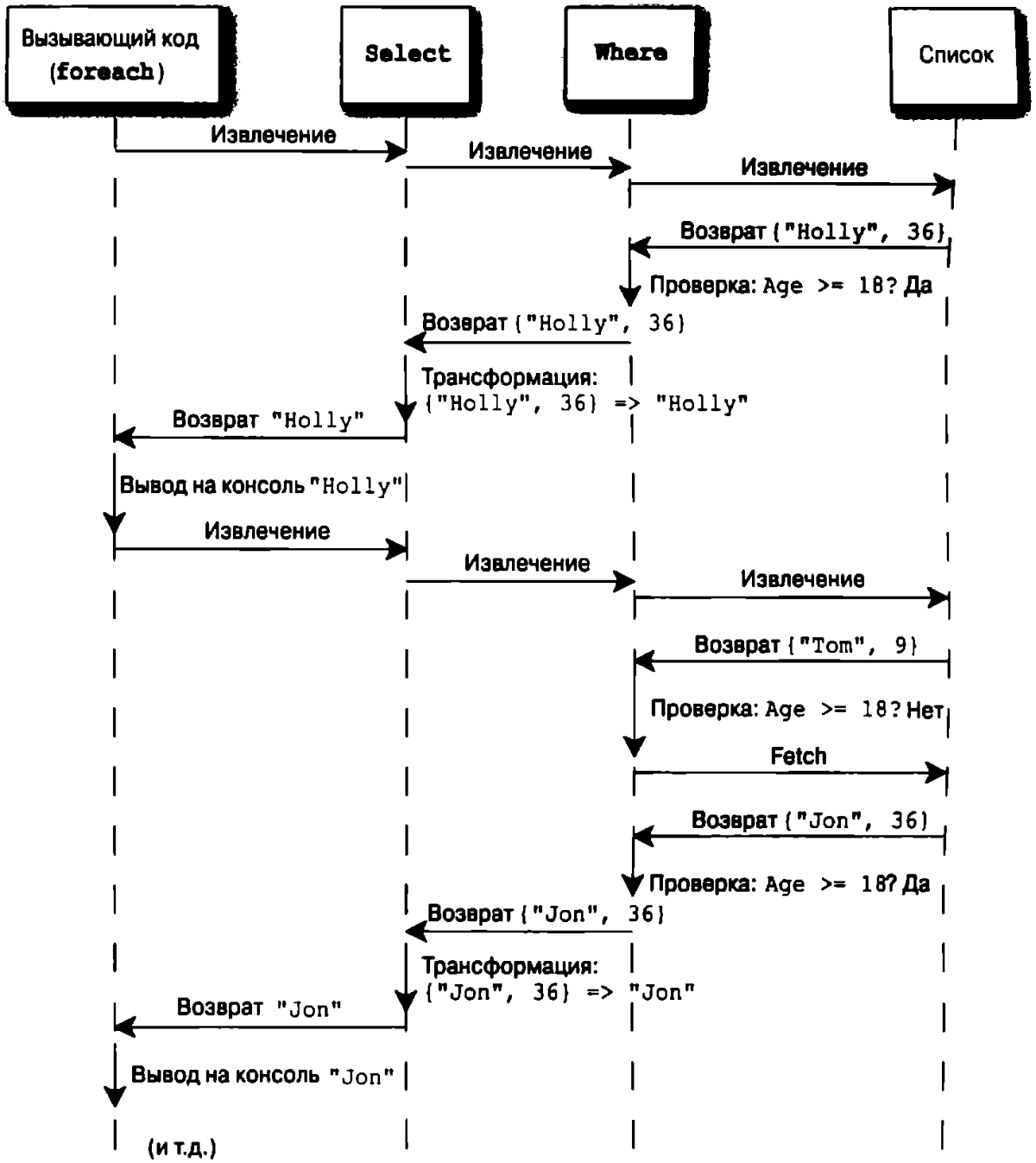

Рис. 11.2. Диаграмма последовательностей, иллюстрирующая выполнение выражения запроса

Подобно тому, как организация потока зависит от интересующей операции, некоторые трансформации будут осуществляться по мере обращения к ним, а не с применением отложенного выполнения. Это называется *немедленным выполнением*. Говоря в общем, операции, которые возвращают другую последовательность (обычно `IEnumerable<T>` или `IQueryable<T>`) используют отложенное выполнение, тогда как операции, возвращающие одиночное значение, применяют немедленное выполнение.

Операции, широко доступные в LINQ, известны под названием стандартных операций запросов — давайте кратко взглянем на них.

Стандартные операции запросов

Стандартные операции запросов LINQ — это коллекция трансформаций с хорошо понятным смыслом. Поставщики LINQ реализуют столько таких операций, сколько в принципе возможно, обеспечивая для реализации ожидаемое поведение. Это критически важно в плане предоставления согласованной инфраструктуры запросов в отношении множества источников данных. Конечно, некоторые поставщики LINQ могут предлагать дополнительную функциональность, а некоторые операции могут не отображаться должным образом на целевую предметную область поставщика, но во всяком случае существует возможность для обеспечения согласованности.

Специфичные для реализации детали стандартных операций

Только тот факт, что стандартные операции запросов имеют общий универсальный смысл, вовсе не означает, что они будут работать в точности одинаковым образом в каждой реализации. Например, некоторые поставщики LINQ могут загружать данные для всего запроса, когда им нужен первый элемент — при обращении к веб-службе это вполне адекватно. Аналогично, запрос, который работает в LINQ to Objects, может обладать слегка отличающейся семантикой в LINQ to SQL. Это не значит, что LINQ обманул ожидания, просто при написании запроса необходимо принимать во внимание, к какому источнику данных производится доступ. По-прежнему существует большое преимущество в наличии единого набора операций запросов и согласованного синтаксиса самих запросов, хотя это отнюдь не является панацеей.

В C# 3 имеется поддержка для ряда стандартных операций запросов, встроенная в язык посредством выражений запросов, но вы всегда можете принять решение вызывать их вручную, если считаете, что это делает код яснее. Возможно, вам интересно будет узнать, что в языке VB9 присутствует большее количество операций; как всегда, существует компромисс между добавочной сложностью из-за включения средства в язык и преимуществами, которые привносит это средство. Лично я думаю, что команда проектировщиков C# проделала замечательную работу; я всегда был сторонником относительно небольшого языка и крупной библиотеки, сопровождающей его.

Перегрузка операций

Термин *операция* используется для описания и операций запросов (методов вроде `Select()` и `Where()`), и знакомых операций, таких как сложение, равенство и т.д. Обычно то, какая разновидность операций имеется в виду, должно быть ясным из контекста — если речь идет о LINQ, то *операция* почти всегда будет относиться к методу, применяемому как часть запроса.

Вы увидите ряд таких операций в примерах, приводимых в этой и следующей главах, но я не намерен предоставлять здесь исчерпывающее руководство по ним; данная книга посвящена в первую очередь C#, а не целому LINQ. Для продуктивной работы с LINQ знать абсолютно все операции не обязательно, однако ваш опыт с течением времени наверняка будет расти. В приложении A даны краткие описания всех стандартных операций запросов, а в MSDN можно узнать дополнительные сведения о каждой конкретной перегруженной версии. Когда возникнет проблема, проверьте этот список: если

вы чувствуете, что *должен* быть встроенный метод, который вам поможет, то вполне вероятно, что он есть! Тем не менее, так бывает не всегда — я создал проект с открытым кодом MoreLINQ для добавления нескольких дополнительных операций к LINQ to Objects (`http://code.google.com/p/morelinq/`). Подобным образом пакет Reactive Extensions (`http://mng.bz/R7ip`) имеет добавления для модели с пассивным источником данных LINQ to Objects, а также модели с активным источником, которую мы рассмотрим позже. Если стандартных операций не хватает, проверьте указанные два проекта, прежде чем строить собственное решение. Хотя не так уж страшно, если вам *придется* писать собственную операцию; вы получите немало удовольствия. В главе 12 будет дано несколько советов по этой теме.

Учитывая упомянутые примеры, пришло время представить модель данных, которая будет использоваться в остальных примерах кода в этой главе.

11.1.2. Определение эталонной модели данных

В разделе 10.3.4 был приведен краткий пример отслеживания дефектов как реальный случай применения расширяющий методов и лямбда-выражений. Мы будем использовать ту же самую идею почти во всех примерах кода в настоящей главе — это довольно простая модель, но ею можно манипулировать множеством разных способов, предоставляя полезную информацию. Кроме того, отслеживание дефектов является той областью, с которой, к сожалению, слишком хорошо знакомо большинство профессиональных разработчиков.

Вымышленным местом действия будет SkeetySoft, небольшая компания по разработке программного обеспечения, но с большими амбициями. Основатели компании решили создать офисный пакет, медиа-проигрыватель и приложение для мгновенного обмена сообщениями. В конце концов, разве в этих рыночных нишах есть крупные игроки?

Отдел разработки в SkeetySoft состоит из пяти человек: двух разработчиков (Дебора (Deborah) и Даррен (Darren)), двух тестировщиков (Тара (Tara) и Тим (Tim)) и менеджера (Мэри (Mary)). В настоящее время есть единственный заказчик: Колин (Colin). Упомянутыми продуктами являются, соответственно, SkeetyOffice, SkeetyMediaPlayer и SkeetyTalk[2]. Мы собираемся рассматривать дефекты, зафиксированные в течение мая 2013 года, используя модель данных, которая показана на рис. 11.3.

Как видите, записи подлежат не так уж много данных. В частности, не ведется реальная хронология дефектов, но данных вполне достаточно, чтобы позволить нам поработать со средствами выражений запросов C# 3.

Для целей этой главы все данные хранятся в памяти. Имеется в наличии класс по имени `SampleData` со свойствами `AllDefects`, `AllUsers`, `AllProjects` и `AllSubscriptions`, которые все возвращают подходящий тип `IEnumerable<T>`. Свойства `Start` и `End` возвращают экземпляры `DateTime` для начала и конца мая, соответственно, и внутри `SampleData` предусмотрены вложенные классы `Users` и `Projects` для обеспечения простого доступа к отдельному пользователю и проекту. Одним типом, который может не сразу стать очевидным, является `NotificationSubscription`; в его основе лежит идея отправки сообщения электронной почты по указанному адресу каждый раз, когда обнаружен дефект или в соответствующий проект внесено изменение.

В эталонных данных зарегистрированы сведения о 41 дефекте, созданные с применением инициализаторов объектов C# 3. Весь код доступен для загрузки на веб-сайте вместе с эталонными данными.

---

[2] Отдел маркетинга в SkeetySoft не особенно креативен.

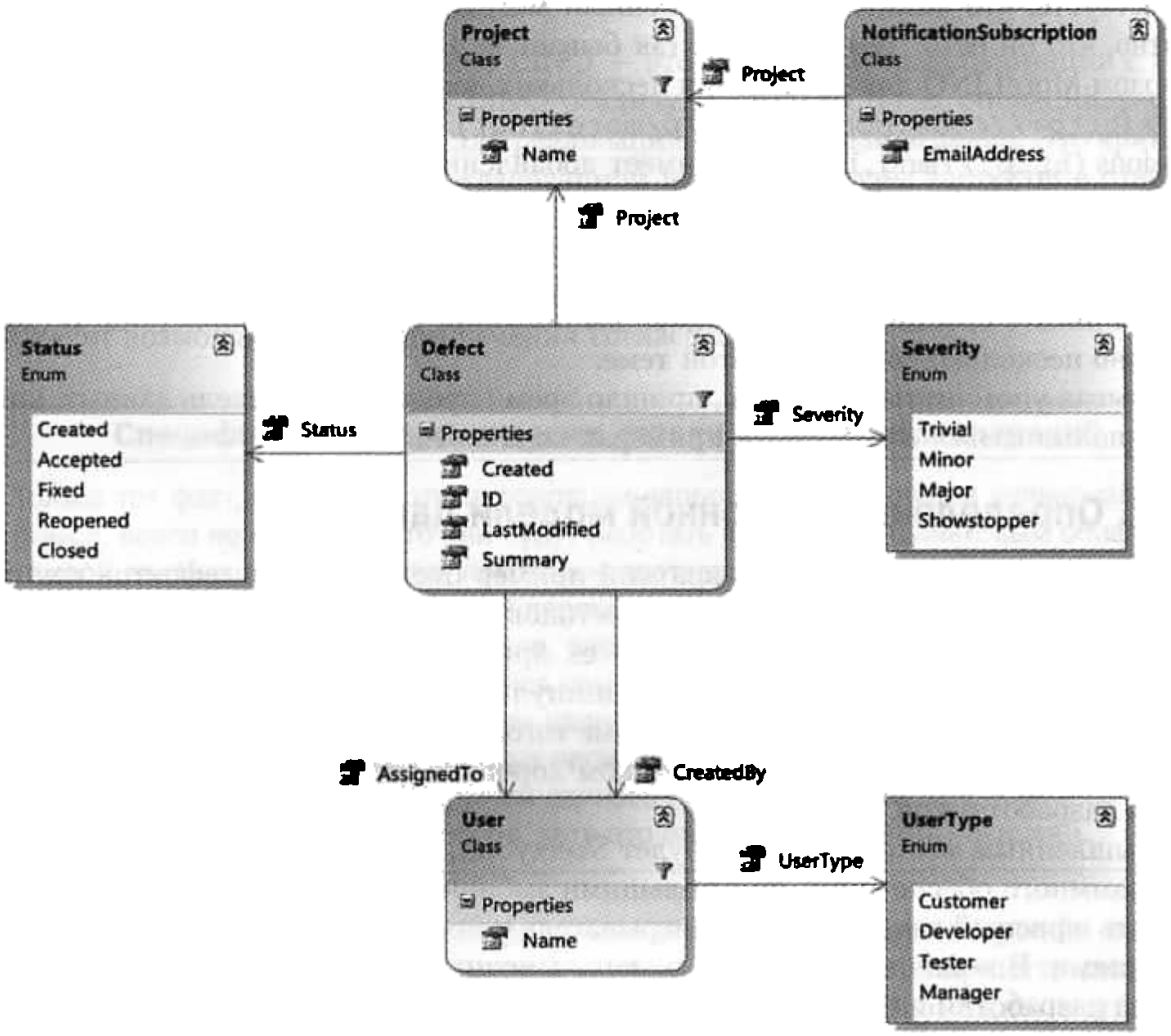

Рис. 11.3. Диаграмма классов для модели данных о дефектах SkeetySoft

Теперь, когда подготовка завершена, давайте займемся вплотную некоторыми запросами.

11.2. Простое начало: выборка элементов

Мы уже обсуждали несколько общих концепций LINQ — я буду представлять концепции, специфичные для C# 3, по мере того, как эти вопросы будут подниматься в ходе материала главы. Мы начнем с простого запроса (который даже проще, чем показанный ранее) и разработаем более сложные запросы, не только обретая понимание того, что делает компилятор C# 3, но также и обучаясь тому, как читать код LINQ.

Все примеры будут следовать шаблону определения запроса и затем вывода результатов на консоль. Мы не будем касаться привязки запросов к сеткам данных или чего-то подобного — все это важно, однако не имеет прямого отношения к изучению C# 3.

В качестве стартовой точки для выяснения, что компилятор делает "за кулисами", а также изучения *переменных диапазонов*, может выступить простое выражение, которое выводит на консоль всех пользователей.

11.2.1. Превращение начального источника в выборку

Каждое выражение запроса в C# 3 начинается одинаково — с указания источника последовательности данных:

```
from элемент in источник
```

Часть *элемент* — это просто идентификатор с необязательным именем типа перед ним. В большинстве случаев имя типа указывать не требуется, и в первом примере оно отсутствует. Часть *источник* является обычным выражением. После этой первой конструкции может много чего происходить, но рано или поздно встретится конструкция `select` или `group`. В целях простоты мы начнем с конструкции `select`. Синтаксис конструкции `select` также прост:

```
select выражение
```

Конструкция `select` известна как *проекция*.

Объединение всего описанного и применение элементарного выражения дает простой (и практически бесполезный) запрос, который показан в листинге 11.1.

Листинг 11.1. Элементарный запрос для вывода на консоль списка пользователей

```
var query = from user in SampleData.AllUsers
            select user;
foreach (var user in query)
{
  Console.WriteLine(user);
}
```

Выражение запроса выделено полужирным. Поскольку метод `ToString()` для всех сущностей в модели был переопределен, результаты выполнения кода из листинга 11.1 выглядят следующим образом:

```
User: Tim Trotter (Tester)
User: Tara Tutu (Tester)
User: Deborah Denton (Developer)
User: Darren Dahlia (Developer)
User: Mary Malcop (Manager)
User: Colin Carton (Customer)
```

Вас может интересовать, насколько полезен этот пример; в конце концов, можно было бы воспользоваться свойством `SampleData.AllUsers` напрямую внутри оператора `foreach`. Однако мы будем применять такое, пусть и простое, выражение запроса для представления двух новых концепций. Сначала мы взглянем на общую природу процесса *трансляции*, который компилятор использует, когда сталкивается с выражением запроса, а затем обсудим переменные диапазонов.

11.2.2. Трансляция компилятором как основа выражений запросов

Поддержка выражений запросов в C# 3 основана на трансляции компилятором этих выражений в обычный код C#. Трансляция делается механически без попытки анализа кода, применения выведения типов, проверки допустимости обращений к методам или выполнения любой другой работы, характерной для компилятора. Все это происходит позже, после трансляции. Во многих отношениях данный первый этап можно рассматривать как шаг препроцессора.

Перед *действительной* компиляцией компилятор транслирует код из листинга 11.1 в код, приведенный в листинге 11.2.

Листинг 11.2. Выражение запроса из листинга 11.1, транслированное в вызов метода

```
var query = SampleData.AllUsers.Select(user => user);
foreach (var user in query)
{
  Console.WriteLine(user);
}
```

Компилятор C# 3 транслирует выражение запроса в *именно* такой код, прежде чем приступить к его компиляции. В частности, он не предполагает, что должен использоваться метод `Enumerable.Select()` или что тип `List<T>` должен содержать метод по имени `Select()`. Пока код просто транслируется, а второй этап компиляции будет иметь дело с поиском подходящего метода — будь это обычный член класса или расширяющий метод[3]. Параметр может иметь подходящий тип делегата либо `Expression<T>` для соответствующего типа `T`.

Именно здесь становится важным тот факт, что лямбда-выражения могут быть преобразованы в экземпляры делегатов и деревья выражений. Во всех примерах этой главы будут применяться делегаты, но вы увидите, как использовать деревья выражений, при рассмотрении других поставщиков LINQ в главе 12. Когда я представляю сигнатуры для методов, вызываемых компилятором позже, помните, что они относятся только к LINQ to Objects — всякий раз, когда параметр имеет тип делегата (как в большинстве случаев), компилятор будет применять в качестве аргумента лямбда-выражение и затем пытаться отыскать метод с подходящей сигнатурой.

Также важно помнить, что везде, где в лямбда-выражении обнаруживается обычная переменная (такая как локальная переменная внутри метода) после того, как трансляция выполнена, она станет захваченной переменной в том же стиле, как было показано в главе 5. Это нормальное поведение лямбда-выражения, но если вы не понимаете, какие переменные будут захватываться, то возникнет риск получить неожиданные результаты из запросов.

В спецификации языка предоставляются подробные сведения о *шаблоне выражений запросов*, который должен быть реализован всеми выражениями запросов, чтобы они были работоспособными, однако он не определен в виде интерфейса, как можно было ожидать. Это очень разумно: такой подход позволяет LINQ быть применимым к интерфейсам наподобие `IEnumerable<T>` с использованием расширяющих методов. В этой главе по очереди рассматриваются все элементы шаблона выражений запросов. Если вы хотите ознакомиться с точным определением трансляции внутри спецификации языка, обратитесь к ее разделу 7.16 ("Query Expressions" ("Выражения запросов")).

---

[3] На самом деле все даже более универсально — компилятор не требует, чтобы `Select()` был методом или `SampleData.AllUsers` — свойством. До тех пор, пока транслированный код компилируется, все в порядке. Почти в каждом разумном случае вы будете получать доступ либо к стандартному, либо к расширяющему методу, но в своем блоге я описал несколько довольно необычных запросов, которые вполне устраивают компилятор (`http://mng.bz/7E3i`). Я не считаю, что запросы подобного рода полезны на практике, однако мне нравится приводить этот пример в качестве иллюстрации, насколько механическим является сам процесс трансляции и до какой степени компилятор не заботит смысл транслированного кода.

В листинге 11.3 иллюстрируется работа трансляции, выполняемая компилятором: в нем предоставлены фиктивные реализации методов `Select()` и `Where()`, причем `Select()` является обычным методом экземпляра, а `Where()` — расширяющим методом. Наше исходное простое выражение запроса содержит только конструкцию `select`, но она включает конструкцию `where`, чтобы продемонстрировать применение обоих методов. В листинге 11.3 приведен полный код, а не фрагмент, т.к. расширяющие методы могут быть объявлены только в высокоуровневых статических классах.

Листинг 11.3. Трансляция компилятором вызывающих методов для фиктивной реализации LINQ

```
static class Extensions
{
  public static Dummy<T> Where<T>(this Dummy<T> dummy,
              Func<T,bool> predicate)   <-- 1 Объявление расширяющего метода Where()
  {
    Console.WriteLine("Where called");
    return dummy;
  }
}
class Dummy<T>
{
  public Dummy<U> Select<U>(Func<T,U> selector)   <-- 2 Объявление метода экземпляра Select()
  {
    Console.WriteLine("Select called");
    return new Dummy<U>();
  }
}
class TranslationExample
{
  static void Main()
  {
    var source = new Dummy<string>();   <-- 3 Создание источника для отправки запросов
    var query = from dummy in source    <-- 4 Вызов методов через выражение запроса
                where dummy.ToString() == "Ignored"
                select "Anything";
  }
}
```

В результате запуска кода из листинга 11.3 на консоль выводится строка `Where called` и затем строка `Select called`, как и можно было ожидать, поскольку выражение запроса было транслировано в следующий код:

```
var query = source.Where(dummy => dummy.ToString() == "Ignored")
                  .Select(dummy => "Anything");
```

Разумеется, здесь не выполняется какой-то запрос или трансформация, но можно понять, как компилятор транслирует выражение запроса. Если затрудняетесь сказать, почему лямбда-выражение в вызове `Select()` возвращает `"Anything"`, а не просто `dummy`, то это потому, что в данном конкретном случае компилятор удалил проекцию `dummy` (как ничего не делающую). Мы разберем это в разделе 11.3.2, а пока важная идея касается в целом вида задействованной трансляции. Необходимо только знать, какие

трансляции компилятор C# будет использовать, после чего вы сможете взять любое выражение запроса, преобразовать его в форму, в которой лямбда-выражения не применяются, и затем посмотреть на то, что оно делает, с этой точки зрения.

Обратите внимание, что интерфейс `IEnumerable<T>` вообще не реализован в `Dummy<T>`. Трансляция выражений запросов в нормальный код от этого не зависит, но на практике большинство поставщиков LINQ будут открывать доступ к данным либо через `IEnumerable<T>`, либо через `IQueryable<T>` (последний из двух будет описан в главе 12). Тот факт, что трансляция не зависит от каких-нибудь конкретных типов, а полагается только на имена и параметры методов, можно считать разновидностью утиной типизации на этапе компиляции. Похожим образом инициализаторы коллекций, представленные в главе 8, ищут открытый метод по имени `Add()`, используя обычное распознавание перегруженных версий, а не интерфейс, который содержит метод `Add()` с определенной сигнатурой. Выражения запросов продвигают эту идею еще дальше – трансляция происходит на раннем этапе процесса компиляции, чтобы позволить компилятору выбрать либо методы экземпляра, либо расширяющие методы. Можете даже считать трансляцию как работу отдельного препроцессорного механизма.

Вам может показаться, что я слишком надоедливо веду речь об этом, но все делается ради снятия завесы, которая иногда покрывает LINQ. Если вы перепишете выражение запросов в виде последовательности обращений к методам, фактически сделав то, что предпринял бы компилятор, то тем самым никак не измените производительность или поведение запроса. Это просто два разных способа представления одного и того же кода.

Почему вместо `select...from...where` записывается `from...where...select`?

Многие разработчики с самого начала находят порядок следования конструкций в выражениях запросов сбивающим с толку. Порядок выглядит похожим на SQL — но только задом наперед. Если вы посмотрите снова на трансляцию в методы, то увидите главную причину такого положения вещей. Выражение запроса обрабатывается в том же самом порядке, в котором оно записано: вы начинаете с указания источника в конструкции `from`, затем фильтруете его в конструкции `where` и, наконец, проецируете его в конструкции `select`. Взглянуть на это по-другому можно, обратившись к диаграммам, приводимым в настоящей главе. Данные текут сверху вниз, а блоки в диаграмме расположены в том же самом порядке, что и соответствующие им конструкции внутри выражения запроса. Как только пройдет начальный дискомфорт из-за недостаточного знакомства, вы можете счесть такой подход привлекательным — как это получилось у меня. Вы можете даже обнаружить, что готовы задать похожий вопрос в отношении SQL.

Теперь вам известно о том, что задействуется трансляция на уровне исходного кода, однако есть еще одна критически важная концепция, которую следует понять, прежде чем можно будет двигаться дальше.

11.2.3. Переменные диапазонов и нетривиальные проекции

Давайте рассмотрим исходное выражение запроса, представленное в начале этой главы, более тщательно. Мы не исследовали идентификатор в конструкции `from` и выражение в конструкции `select`. На рис. 11.4 снова показано это выражение запроса с объяснениями назначения каждой его части.

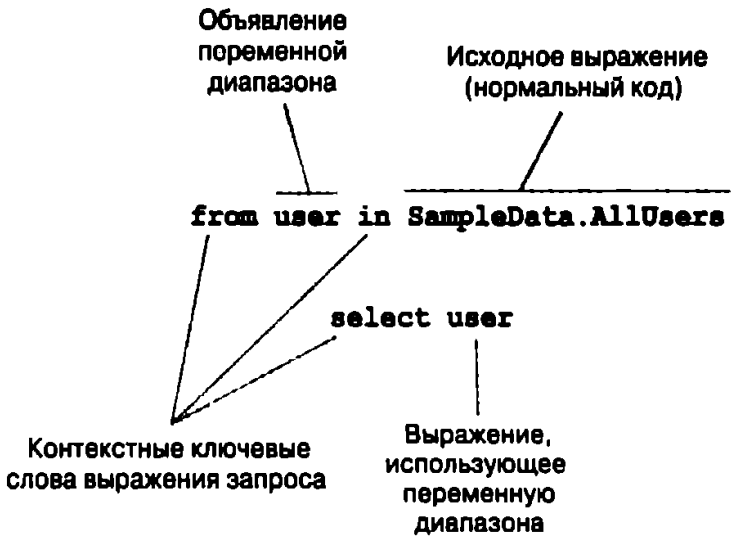

Рис. 11.4. Простое выражение запроса, разбитое на составляющие части

Контекстные ключевые слова объяснить легко — они указывают компилятору, что именно нужно делать с данными. Подобным же образом исходное выражение является обычным выражением C# — в данном случае свойством, но с той же легкостью оно могло бы быть вызовом метода или переменной.

Сложности начинаются, когда дело доходит до объявления переменной диапазона и выражения проекции. Переменные диапазонов не похожи на переменные других видов. В некотором смысле это вообще не переменные! Они доступны только в выражениях запросов и фактически предназначены для распространения контекста из одного выражения в другое. Они представляют по одному элементу отдельной последовательности за раз и применяются при трансляции, выполняемой компилятором, для облегчения перевода других выражений в лямбда-выражения.

Вы уже видели, что исходное выражение запроса было преобразовано следующим образом:

```
SampleData.AllUsers.Select(user => user)
```

Левая сторона лямбда-выражения — часть, предоставляющая имя параметра, — взята из объявления переменной диапазона. Правая сторона поступает из конструкции `select`. Трансляция столь же проста (в этом случае). Все вычисляется нормально, поскольку на обеих сторонах используется одно и то же имя.

Предположим, что написано выражение запроса вроде приведенного ниже:

```
from user in SampleData.AllUsers
select person
```

Транслированная версия будет выглядеть так:

```
SampleData.AllUsers.Select(user => person)
```

В этот момент компилятор откажется компилировать код, т.к. ему не известно, на что ссылается `person`.

Теперь, когда вы знаете, насколько прост процесс, становится легче понять выражение запроса, которое содержит несколько более сложную проекцию. Код в листинге 11.4 выводит на консоль только имена пользователей.

Листинг 11.4. Запрос, выбирающий только имена пользователей

```
IEnumerable<string> query = from user in SampleData.AllUsers select user.Name;
foreach (string name in query)
{
  Console.WriteLine(name);
}
```

На этот раз в качестве проекции применяется `user.Name`, и результатом является последовательность строк, а не объектов `User`. (Чтобы акцентировать внимание на этом, использовалась явно типизированная переменная.) Трансляция выражения запроса следует тем же правилам, что и ранее, и приводит к получению такого кода:

```
SampleData.AllUsers.Select(user => user.Name)
```

Компилятор разрешает это, т.к. выбранный расширяющий метод `Select()` из класса `Enumerable` имеет такую сигнатуру[4]:

```
static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source,
   Func<TSource,TResult> selector)
```

Выведение типов, описанное в главе 9, преобразует лямбда-выражение в `Func<TSource,TResult>`. Сначала делается вывод, что `TSource` является `User` вследствие типа `SampleData.AllUsers`. В этот момент известен тип параметра для лямбда-выражения, поэтому `user.Name` может быть распознано как выражение доступа к свойству, возвращающему тип `string`, так что для типа `TResult` выводится `string`. Вот почему лямбда-выражения допускают неявно типизированные параметры, а правила выведения типов настолько сложны; все это — компоненты механизма LINQ.

Зачем все это нужно знать?

На протяжении долгого времени на переменные диапазонов можно не обращать внимания. Результаты выполнения очень многих запросов вполне понятны и без знания того, что происходит "за кулисами". Такой подход хорош, когда все работает должным образом (подобно примерам в руководствах), но когда что-то идет не так, полезно знать детали. Столкнувшись с ситуацией, когда выражение запроса не компилируется из-за неизвестного идентификатора, необходимо взглянуть на задействованные переменные диапазонов.

До сих пор мы видели только неявно типизированные переменные диапазонов. А что произойдет, если в объявление включить тип? Ответ кроется в стандартных операциях запросов `Cast()` и `OfType()`.

11.2.4. `Cast()`, `OfType()` и явно типизированные переменные диапазонов

В большинстве случаев переменные диапазонов могут быть типизированы неявно; скорее всего, вы будете работать с обобщенными коллекциями, в которых все, что требуется — это указанный тип. А что, если это не так? Что, если запрос необходимо выполнить на `ArrayList` или, возможно, `object[]`? Жаль, если LINQ не смог бы

[4] Чтобы сигнатуры всех методов, упоминаемых в этой главе, умещались в печатную страницу, я не указываю в них модификатор `public`. На деле все они являются открытыми.

применяться в таких ситуациях. К счастью, существуют две стандартных операции запросов, которые приходят на помощь: `Cast()` и `OfType()`. Непосредственно синтаксисом выражений запросов поддерживается только `Cast()`, но в настоящем разделе мы рассмотрим обе операции.

Эти две операции похожи: обе принимают произвольную нетипизированную последовательность (они являются расширяющими методами для необобщенного типа `IEnumerable`) и возвращают строго типизированную последовательность. Операция `Cast()` делает это за счет приведения каждого элемента к целевому типу (или отказа в случае, если элемент имеет неподходящий тип), а `OfType()` сначала выполняет проверку, пропуская любые элементы с неправильным типом.

В листинге 11.5 демонстрируется работа обеих этих операций, используемых как простые расширяющие методы из `Enumerable`. Для разнообразия в качестве данных система регистрации дефектов SkeetySoft применяться не будет — в конце концов, все данные в ней строго типизированы! Взамен будут использоваться два объекта `ArrayList`.

Листинг 11.5. Применение операций `Cast()` и `OfType()` для работы со слабо типизированными коллекциями

```
ArrayList list = new ArrayList { "First", "Second", "Third" };
IEnumerable<string> strings = list.Cast<string>();
foreach (string item in strings)
{
  Console.WriteLine(item);
}
list = new ArrayList { 1, "not an int", 2, 3 };
IEnumerable<int> ints = list.OfType<int>();
foreach (int item in ints)
{
  Console.WriteLine(item);
}
```

Первый список содержит только строки, поэтому использовать `Cast<string>()` для получения последовательности строк вполне безопасно. Второй список имеет смешанное содержимое, так что для извлечения из него только целых чисел применяется `OfType<int>()`. Использование `Cast<int>()` для второго списка привело бы к генерации исключения при попытке приведения "не `int`" к `int`. Обратите внимание, что это произошло бы только *после* вывода на консоль `1` — обе операции организуют поток для данных, преобразуя элементы по мере их извлечения.

Только преобразования идентичности, ссылочные и распаковывающие преобразования

В .NET 3.5 SP1 поведение операции `Cast()` слегка изменилось. В исходной платформе .NET 3.5 она выполнила бы больше преобразований, поэтому применение `Cast<int>()` к `List<short>` преобразовало бы каждый элемент типа `short` в тип `int` при его извлечении. В .NET 3.5 SP1 и всех последующих выпусках это привело бы к генерации исключения. Если необходимо любое преобразование, отличное от ссылочного или распаковывающего (или преобразования идентичности, не выполняющего никаких действий), используйте вместо операции `Cast()` проекцию `Select()`. Операция `OfType()` также выполняет только эти преобразования, но не генерирует исключение в случае отказа.

Когда вводится переменная диапазона с явным типом, компилятор применяет вызов `Cast()`, чтобы обеспечить подходящий тип для последовательности, используемой остальной частью выражения запроса. Это иллюстрируется кодом в листинге 11.6, в котором проекция применяет метод `Substring()` для доказательства того, что последовательность, генерируемая конструкцией `from`, является последовательностью строк.

Листинг 11.6. Использование явно типизированной переменной диапазона для автоматического вызова `Cast()`

```
ArrayList list = new ArrayList { "First", "Second", "Third");
var strings = from string entry in list
              select entry.Substring(0, 3);
foreach (string start in strings)
{
  Console.WriteLine(start);
}
```

Код из листинга 11.6 выводит на консоль `Fir`, `Sec`, `Thi`, но более интересно взглянуть на транслированное выражение запроса:

```
list.Cast<string>().Select(entry => entry.Substring(0,3));
```

Без приведения вызов `Select()` был бы вообще невозможен, т.к. расширяющий метод определен только для `IEnumerable<T>`, но не `IEnumerable`. Даже когда применяется строго типизированная коллекция, может по-прежнему требоваться указывать явно типизированную переменную диапазона. Например, при наличии коллекции, определенной как `List<ISomeInterface>`, может быть известно, что все ее элементы являются экземплярами типа `MyImplementation`. Использование переменной диапазона с явным типом `MyImplementation` позволяет получать доступ ко всем членам `MyImplementation` без ручной вставки приведений повсеместно в коде.

К настоящему моменту было раскрыто множество важных концепций, даже при условии, что пока еще не получены сколько-нибудь впечатляющие результаты. Ниже кратко описаны наиболее важные из них.

- Язык LINQ основан на последовательностях данных, которые организуются в потоки всегда, когда это возможно.
- Создание запроса обычно не означает его выполнение; большинство операций используют отложенное выполнение.
- Выражения запросов в C# 3 задействуют этап предварительной обработки, на котором они преобразуются в нормальный код C#. Затем этот код соответствующим образом компилируется с применением всех обычных правил выведения типов, перегрузки, лямбда-выражений и т.д.
- Переменные, объявленные внутри выражений запросов, не действуют подобно переменным, определенным в любом другом месте; они являются переменными диапазонов, которые позволяют согласованно ссылаться на данные в рамках выражений запросов.

Я понимаю, что здесь присутствует немало в каком-то смысле абстрактной информации. Не переживайте, если поначалу будет возникать вопрос, стоит ли LINQ всех этих неприятностей. Обещаю, что стоит. После серьезной подготовительной работы мы можем приступить к выполнению по-настоящему полезных действий, таких как фильтрация данных с последующим их упорядочением.

11.3. Фильтрация и упорядочение последовательности

Возможно, вас удивит, что фильтрация и упорядочение являются двумя простейшими операциями для объяснения в терминах трансляций, предпринимаемых компилятором. Это объясняется тем, что они всегда возвращают последовательность с элементами того же самого типа, что и во входной последовательности, т.е. беспокоиться о введении новых переменных диапазонов не нужно. Помогает также и тот факт, что вы уже видели соответствующие расширяющие методы в главе 10.

11.3.1. Фильтрация с использованием конструкции where

Понять конструкцию where на удивление легко. Ее синтаксис выглядит следующим образом:

```
where выражение-фильтра
```

Компилятор транслирует это в вызов метода Where() с лямбда-выражением, которое в качестве параметра использует соответствующую переменную диапазона, а в качестве тела — выражение фильтра. Выражение фильтра применяется как предикат к каждому элементу входящего потока данных, при этом в результирующую последовательность попадают только те элементы, для которых предикат возвращает значение true.

Использование нескольких конструкций where дает в результате цепочку вызовов Where() — частью окончательной последовательности будут только элементы, которые соответствуют *всем* предикатам. В листинге 11.7 показано выражение запроса, находящее все открытые дефекты, исправление которых назначено Тиму.

Листинг 11.7. Выражение запроса с несколькими конструкциями where

```
User tim = SampleData.Users.TesterTim;
var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            select defect.Summary;
foreach (var summary in query)
{
  Console.WriteLine(summary);
}
```

Выражение запроса из листинга 11.7 транслируется следующим образом:

```
SampleData.AllDefects.Where(defect => defect.Status != Status.Closed)
                     .Where(defect => defect.AssignedTo == tim)
                     .Select(defect => defect.Summary)
```

Ниже представлен вывод кода из листинга 11.7:

```
Installation is slow
Subtitles only work in Welsh
Play button points the wrong way
Webcam makes me look bald
Network is saturated when playing WAV file
```

Разумеется, в качестве альтернативы применению нескольких where можно было бы написать единственную конструкцию where, которая объединяла бы эти два условия.

В некоторых случаях это может улучшить производительность, но следует принимать во внимание также и читабельность выражения запроса, поэтому решение, скорее всего, будет индивидуальным. Лично я склонен объединять те условия, которые связаны друг с другом логически, и оставлять отдельными другие. В данном случае обе части выражения имеют дело непосредственно с дефектом (поскольку это все, что содержится в последовательности), так что было бы разумно объединить их. Как и ранее, полезно написать обе формы и посмотреть, какая из них выглядит более ясной.

Вскоре мы приступим к применению в отношении запроса ряда правил упорядочения, но сначала мы должны взглянуть на небольшую деталь, связанную с работой конструкции `select`.

11.3.2. Вырожденные выражения запросов

Несмотря на наличие довольно простой трансляции для работы, давайте возвратимся к одной особенности, упомянутой ранее в разделе 11.2.2, когда впервые были представлены трансляции компилятора. До сих пор все транслированные выражения запросов включали вызов `Select()`. Что произойдет, если конструкция `select` ничего не делает, фактически возвращая ту же самую последовательность, которую она получает? Ответ состоит в том, что компилятор удалит такой вызов `Select()`, но только если имеются другие операции, выполняемые в пределах выражения запроса.

Например, следующее выражение запроса просто выбирает все дефекты в системе:

```
from defect in SampleData.AllDefects
select defect
```

Это называется *вырожденным выражением запроса*. Компилятор преднамеренно генерирует вызов `Select`, даже если кажется, что ничего делать не нужно:

```
SampleData.AllDefects.Select(defect => defect)
```

Однако между этим и использованием `SampleData.AllDefects` в качестве простого выражения существует большая разница. Элементы, возвращаемые обеими последовательностями, будут одинаковыми, но результатом метода `Select()` является *просто* последовательность элементов, но не сам источник. Результатом выражения запроса никогда не должен быть тот же самый объект, что и источник данных, если только не окажется, что поставщик LINQ неудачно спроектирован. Это может быть важно с точки зрения целостности данных — поставщик может возвращать изменяемый результирующий объект, зная, что изменения в возвращаемых данных не затронут оригинал, даже в случае вырожденного запроса.

Когда участвуют другие операции, компилятору нет необходимости в сохранении конструкций `select`, которые ничего не делают. Например, предположим, что выражение запроса из листинга 11.7 изменено для выборки всех сведений о дефекте, а не только названия:

```
from defect in SampleData.AllDefects
where defect.Status != Status.Closed
where defect.AssignedTo == SampleData.Users.TesterTim
select defect
```

Теперь нет нужды в финальном вызове `Select()`, поэтому транслированный код становится таким:

```
SampleData.AllDefects.Where(defect => defect.Status != Status.Closed)
                     .Where(defect => defect.AssignedTo == tim)
```

Данные правила редко становятся препятствием при написании выражений запросов, но они могут вызвать путаницу, если вы декомпилируете код с помощью инструмента, подобного Reflector, и обнаружите, что обращение к `Select()` по непонятной причине отсутствует.

Вооружившись этими знаниями, самое время усовершенствовать запрос, чтобы узнать, над чем Тим должен работать далее.

11.3.3. Упорядочение с использованием конструкции `orderby`

Не так уж редко разработчиков и тестировщиков просят устранить сначала наиболее критичные дефекты, прежде чем они с усердием возьмутся за более тривиальные дефекты. Можно воспользоваться простым запросом, чтобы сообщить Тиму порядок, в котором он должен заниматься назначенными ему открытыми дефектами. Именно это делается в листинге 11.8 с применением конструкции `orderby` и вывода на консоль подробных сведений о дефектах по убыванию их приоритета.

Листинг 11.8. Сортировка дефектов по степени серьезности, от больших приоритетов к меньшим

```
User tim = SampleData.Users.TesterTim;

var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            orderby defect.Severity descending
            select defect;

foreach (var defect in query)
{
  Console.WriteLine("{0}: {1}", defect.Severity, defect.Summary);
}
```

Вывод кода из листинга 11.8 показывает, что результат был отсортирован нужным образом:

```
Showstopper: Webcam makes me look bald
Major: Subtitles only work in Welsh
Major: Play button points the wrong way
Minor: Network is saturated when playing WAV file
Trivial: Installation is slow
```

В наличии два крупных дефекта. В каком порядке их устранять? Пока что никакого четкого порядка не предусмотрено.

Давайте изменим запрос так, чтобы после нисходящей сортировки по степени серьезности выполнялась *восходящая* сортировка по времени последнего изменения. Это означает, что Тим будет проверять сначала дефекты, которые были зафиксированы давным-давно, и только затем относительно недавние дефекты. Для этого потребуется лишь одно дополнительное выражение в конструкции `orderby`, как показано в листинге 11.9.

Листинг 11.9. Упорядочение по степени серьезности и затем по времени последнего изменения

```
User tim = SampleData.Users.TesterTim;

var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            orderby defect.Severity descending, defect.LastModified
            select defect;

foreach (var defect in query)
{
  Console.WriteLine("{0}: {1} ({2:d})",
              defect.Severity, defect.Summary, defect.LastModified);
}
```

Результаты выполнения кода из листинга 11.9 приведены ниже. Обратите внимание, что порядок следования двух крупных дефектов изменился на противоположный:

```
Showstopper: Webcam makes me look bald (05/27/2013)
Major: Play button points the wrong way (05/17/2013)
Major: Subtitles only work in Welsh (05/23/2013)
Minor: Network is saturated when playing WAV file (05/31/2013)
Trivial: Installation is slow (05/15/2013)
```

Вот так выглядит выражение запроса, но что конкретно делает компилятор? Он просто вызывает методы `OrderBy()` и `ThenBy()` (или `OrderByDescending()`/`ThenByDescending()` для упорядочения по убыванию). Выражение запроса транслируется следующим образом:

```
SampleData.AllDefects.Where(defect => defect.Status != Status.Closed)
                     .Where(defect => defect.AssignedTo == tim)
                     .OrderByDescending(defect => defect.Severity)
                     .ThenBy(defect => defect.LastModified)
```

Теперь, после ознакомления с примером, можно переходить к рассмотрению общего синтаксиса конструкций `orderby`. По существу они представляются с помощью контекстного ключевого слова `orderby`, за которым следует один или более порядков. *Порядок* — это всего лишь выражение (в котором могут присутствовать переменные диапазонов), за которым необязательно может быть указано слово `ascending` или `descending`, имеющее вполне очевидный смысл. (Стандартным порядком является возрастающий (т.е. `ascending`).) Главный порядок транслируется в вызов метода `OrderBy()` или `OrderByDescending`, а за ним следует столько обращений к методу `ThenBy` или `ThenByDescending`, сколько было задано последующих порядков.

Разница между `OrderBy()` и `ThenBy` проста: метод `OrderBy()` принимает на себя ответственность за первичный контроль над упорядочением, тогда как метод `ThenBy()` служит средством для выполнения одного или более последующих упорядочений. В LINQ to Objects метод `ThenBy()` определен только как расширяющий для интерфейса `IOrderedEnumerable<T>`, который представляет собой тип, возвращаемый методом `OrderBy()` (и самим методом `ThenBy()`, чтобы позволить дальнейшее выстраивание в цепочку).

Важно отметить, что хотя допускается использовать несколько конструкций `orderby`, каждая из них будет начинаться с собственного вызова `OrderBy()` или

`OrderByDescending()`, и это означает, что последняя, в сущности, выиграет. Я еще не сталкивался с ситуацией, когда это бы пригодилось, если только не предположить, что между конструкциями `orderby` внутри запроса делается что-то еще; взамен вы почти всегда должны применять единственную конструкцию, содержащую несколько порядков.

Как было показано в главе 10, выполнение упорядочения требует загрузки всех данных (во всяком случае, для LINQ to Objects) — к примеру, упорядочить бесконечную последовательность невозможно. Надеюсь, причина для вас очевидна: нельзя узнать, какой элемент должен попасть в самое начало, до тех пор, пока не станут доступными все элементы.

Мы находимся примерно на середине пути по изучению выражений запросов, и вас может удивить, что мы пока еще встречали ни одного *соединения*. Очевидно, что соединения важны в LINQ, равным образом как они важны в SQL, однако они также и сложны. Я обещаю, что в свое время мы до них доберемся, но чтобы представлять по одной концепции за раз, мы выберем окольный путь и рассмотрим сначала конструкции `let`. Это позволит обсудить прозрачные идентификаторы, прежде чем мы вплотную займемся соединениями.

11.4. Конструкции `let` и прозрачные идентификаторы

Большинство оставшихся операций, на которые все еще предстоит взглянуть, задействуют *прозрачные идентификаторы*. В точности как с переменными диапазона, без понимания прозрачных идентификаторов можно прекрасно обойтись, если вам достаточно лишь поверхностных знаний выражений запросов. Но раз уж вы приобрели эту книгу, то вероятно хотите изучить язык C# на более глубоком уровне, который позволит (помимо прочего) смело смотреть в лицо ошибкам компиляции и понимать, о чем они говорят.

Вам не понадобится знать о прозрачных идентификаторах *абсолютно все*, но я обучу вас достаточно для того, чтобы при встрече с ними в спецификации языка вы не испытывали желание убежать и спрятаться. Вдобавок вы будете понимать, зачем они вообще нужны — и вот здесь как раз пригодится пример. Конструкция `let` — это простейшая трансформация из числа доступных, в которой используются прозрачные идентификаторы.

11.4.1. Добавление промежуточных вычислений с помощью конструкции `let`

Конструкция `let` вводит новую переменную диапазона со значением, которое может быть основано на других переменных диапазона. Ее синтаксис проще простого:

```
let идентификатор = выражение
```

Чтобы объяснить эту операцию в терминах, не касающихся более сложных операций, я прибегну к весьма надуманному примеру. Оставьте в покое недоверие и вообразите, что нахождение длины строки является дорогостоящей в плане ресурсов операцией. А теперь представьте, что имеется совершенно странное системное требование о том, что пользователи должны упорядочиваться по именам и вместе с именем отображалось его длина. Да, я знаю, что подобное маловероятно. Тем не менее, в листинге 11.10 продемонстрирован способ реализации этого без применения конструкции `let`.

Листинг 11.10. Сортировка по длине имени пользователя без использования конструкции `let`

```
var query = from user in SampleData.AllUsers
            orderby user.Name.Length
            select user.Name;
foreach (var name in query)
{
  Console.WriteLine("{0}: {1}", name.Length, name);
}
```

Код работает нормально, но дорогостоящее свойство `Length` в нем применяется дважды — один раз для сортировки пользователей и еще раз для вывода на консоль. Несомненно, даже самый быстрый суперкомпьютер не смог бы справится с задачей нахождения длин шести строк *целых два раза*! Нет-нет-нет, нужно срочно избавиться от таких избыточных вычислений.

Это можно сделать с помощью конструкции `let`, которая вычисляет выражение и вводит результат как новую переменную диапазона. В листинге 11.11 приведен код, обеспечивающий получение того же результата, что и код в листинге 11.10, но с использованием свойства `Length` только один раз для каждого пользователя.

Листинг 11.11. Применение конструкции `let` для устранения избыточных вычислений

```
var query = from user in SampleData.AllUsers
            let length = user.Name.Length
            orderby length
            select new { Name = user.Name, Length = length };
foreach (var entry in query)
{
  Console.WriteLine("{0}: {1}", entry.Length, entry.Name);
}
```

В листинге 11.11 вводится новая переменная диапазона по имени `length`, которая содержит длину имени пользователя (для текущего пользователя в исходной последовательности). Затем эта новая переменная диапазона используется для сортировки и в конце для проецирования. Вы уже заметили проблему? Необходимо применять две переменные диапазонов, но лямбда-выражение, передаваемое в `Select()`, принимает только один параметр! Здесь и вступают в игру прозрачные идентификаторы.

11.4.2. Прозрачные идентификаторы

В листинге 11.11 в финальное проецирование вовлечены *две* переменные диапазонов, но метод `Select()` действует только на одной последовательности. Как компоновать переменные диапазонов?

Ответ заключается в создании анонимного типа, содержащего обе переменных, и применении умелой трансляции, которая сделает его *выглядящим* так, как будто действительно есть два параметра для конструкций `select` и `orderby`. На рис. 11.5 показаны задействованные последовательности.

Конструкция `let` достигает своих целей за счет использования еще одного вызова `Select()`, создания анонимного типа для результирующей последовательности и фактически создания новой переменной диапазона, имя которой никогда не встретится в исходном коде.

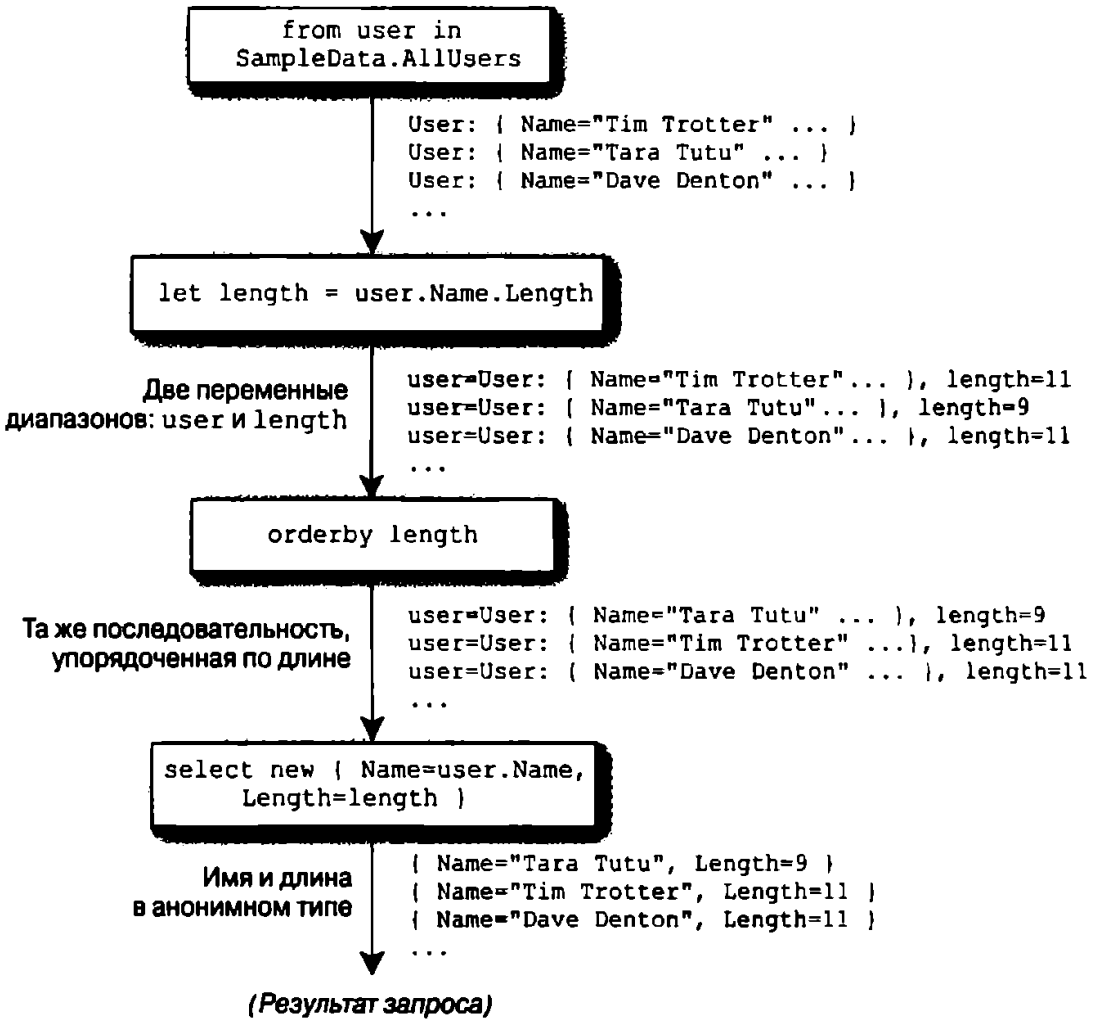

Рис. 11.5. Последовательности, задействованные в листинге 11.11, где конструкция `let` вводит переменную диапазона `length`

Выражение запроса из листинга 11.11 транслируется примерно так:

```
SampleData.AllUsers
        .Select(user => new { user, length = user.Name.Length })
        .OrderBy(z => z.length)
        .Select(z => new { Name = z.user.Name, Length = z.length })
```

Каждая часть запроса была соответствующим образом подкорректирована: там, где в исходном выражении запроса присутствовала ссылка непосредственно на `user` или `length`, но после конструкции `let`, она была заменена ссылкой `z.user` или `z.length`. Выбор `z` в качестве имени совершенно произволен — это имя скрывается компилятором.

Анонимные типы являются деталью реализации

Строго говоря, решение о группировании различных переменных диапазонов для обеспечения работы прозрачных идентификаторов зависит от реализации компилятора C#. Реализация от Microsoft предусматривает применение анонимных типов, а в спецификации также показаны трансляции в этих терминах, так что я следую сложившейся тенденции. Даже если в другом компиляторе избран другой подход, это не должно влиять на результаты.

Если вы обратитесь к описанию конструкций `let` в спецификации языка (раздел 7.16.2.4), то увидите, что трансляция осуществляется из одного выражения запроса в другое. Для представления вводимого прозрачного идентификатора в ней используется звездочка (`*`). Затем в качестве заключительного шага трансляции прозрачный идентификатор *уничтожается*. Я не собираюсь применять данную нотацию в этой главе, поскольку ухватить ее суть довольно трудно, к тому же такой уровень подробностей совершенно не нужен. Надеюсь, благодаря описанным здесь основам, спецификация не будет казаться настолько малопонятной, как могло быть в противном случае.

Хорошая новость состоит в том, что теперь мы можем взглянуть на оставшиеся трансляции, входящие в состав поддержки выражений запросов C# 3. Я не буду вдаваться в детали каждого вводимого прозрачного идентификатора, но упомяну ситуации, в которых это происходит. Давайте начнем с поддержки соединений.

11.5. Соединения

Если вам когда-либо приходилось читать что-нибудь о языке SQL, то вероятно вы представляете себе, что такое операция соединения в базе данных. Она берет две таблицы (или представления, табличные функции и т.д.) и создает результат, сопоставляя один набор строк с другим таким набором. Соединение в LINQ выглядит похожим за исключением того, что работает на последовательностях. Доступны три типа соединений, хотя не все они используют ключевое слово `join` в выражении запроса. Первым делом мы рассмотрим соединение, которое очень близко напоминает внутреннее соединение в SQL.

11.5.1. Внутренние соединения с использованием конструкций `join`

Внутренние соединения задействуют две последовательности. Одно выражение *селектора ключей* применяется к каждому элементу первой последовательности, а другое такое выражение (которое может быть совершенно другим) применяется к каждому элементу второй последовательности. Результатом соединения является последовательность всех пар элементов, в которых ключ из первого элемента совпадает с ключом из второго элемента.

Конфликт терминологии! Внутренние и внешние последовательности

В документации MSDN по методу `Join()`, используемому для вычисления внутренних соединений, задействованные последовательности называются `inner` (внутренняя) и `outer` (внешняя), и действительные параметры метода также основаны на указанных именах. Это не имеет ничего общего с внутренними и внешними соединениями, а просто является способом различения данных последовательностей. Вы можете называть их первой и второй, левой и правой, Власом и Еником (из детской передачи "Улица Сезам" — *прим.ред.*) — в общем, как угодно, если это поможет их отличать. В этой главе я буду применять понятия "левая" и "правая", чтобы сделать понятными диаграммы. Обычно *внешняя* последовательность соответствует *левой*, а *внутренняя* — *правой*.

Упомянутые две последовательности могут быть какими угодно; правая последовательность может даже быть той же самой, что и левая, и это весьма удобно.

(В качестве примера подумайте о поиске пар людей, родившихся в один и тот же день.) Единственное, что имеет значение — оба выражения селекторов ключей должны возвращать ключи одинакового типа[5].

Невозможно соединить последовательность людей и последовательность городов, указав, что дата рождения человека соответствует численности населения города — это совершенно не имеет смысла. Однако одной важной возможностью является использование анонимного типа для ключа; это работает из-за того, что анонимные типы подходящим образом реализуют эквивалентность и хеширование. Если необходимо создать ключ из нескольких столбцов, то в этом помогут анонимные типы. Как мы увидим позже, это также применимо в отношении операций группирования.

Синтаксис внутреннего соединения выглядит сложнее, чем есть на самом деле:

```
[запрос, выбирающий левую последовательность]
join правая-переменная-диапазона in правая-последовательность
  on левый-селектор-ключей equals правый-селектор-ключей
```

Использование equals как контекстного ключевого слова, а не условного обозначения может сбивать с толку, но это упрощает отделение левого селектора ключей от правого селектора ключей. Часто (но не всегда), по меньшей мере, один из селекторов ключей является простейшим, который всего лишь выбирает точный элемент из своей последовательности. Контекстное ключевое слово применяется компилятором для выделения селекторов ключей в разные лямбда-выражения. Способность процессора запросов получать ключи для каждого значения (с обеих сторон соединения) важна как в отношении производительности LINQ to Objects, так и в плане осуществимости трансляции запроса в другие формы, такие как SQL.

Давайте рассмотрим пример из нашей системы регистрации дефектов. Предположим, что добавлено средство уведомления и требуется отправить первый пакет сообщений электронной почты по всем существующим дефектам. Для этого необходимо соединить список подписок на уведомления и список дефектов, выбрав в качестве соответствия проекты. Такое соединение представлено в листинге 11.12.

Листинг 11.12. Соединение списка дефектов и списка подписок на уведомления на основе проектов

```
var query = from defect in SampleData.AllDefects
            join subscription in SampleData.AllSubscriptions
              on defect.Project equals subscription.Project
            select new { defect.Summary, subscription.EmailAddress };
foreach (var entry in query)
{
  Console.WriteLine("{0}: {1}", entry.EmailAddress, entry.Summary);
}
```

Код из листинга 11.12 выведет на консоль каждый дефект в медиа-проигрывателе дважды — один раз для mediabugs@skeetysoft.com и еще раз для theboss@skeetysoft.com (поскольку шеф действительно проявляет интерес к проекту медиа-проигрывателя).

[5] Также допускается использование двух типов ключей, если доступно неявное преобразование из одного типа в другой. Один из типов должен быть лучшим выбором, чем другой, как это делается при выведении компилятором типа для неявно типизированного массива. Хотя в моей практике редко приходилось преднамеренно учитывать эту деталь.

В этом конкретном случае можно было бы легко сделать соединение обратным путем, поменяв местами левую и правую последовательности. Результатом были бы те же самые записи, но в другом порядке. Реализация в LINQ to Objects возвращает записи таким образом, что сначала возвращаются все пары, использующие первый элемент левой последовательности (в порядке, применяемом правой последовательностью), затем все пары, использующие второй элемент левой последовательности, и т.д. Правая последовательность буферизируется, но для левой организуется поток, поэтому если нужно соединить крупную последовательность с мелкой, то полезно по возможности указывать мелкую последовательность как правую.

Операция по-прежнему выполняется отложенным образом: прежде чем читать данные из любой последовательности, она ожидает до тех пор, пока не будет запрошена первая пара. В этот момент осуществляется чтение правой последовательности полностью с целью построения таблицы соответствий между ключами и значениями, дающими эти ключи. В дальнейшем читать правую последовательность заново не понадобится и можно начинать проход по левой последовательности с выдачей подходящих пар.

Одна ошибка, которая может сбивать с толку, связана с неправильным размещением селекторов ключей. В левом селекторе ключей в области действия находится только переменная диапазона из левой последовательности; в правом селекторе ключей в области видимости будет находиться только переменная диапазона из правой последовательности. Меняя местами левую и правую последовательности, вы должны также поменять местами левый и правый селекторы ключей. К счастью, компилятору известна эта распространенная ошибка, так что он предложит необходимый выход из положения.

Просто чтобы прояснить происходящее, на рис. 11.6 демонстрируется процесс обработки последовательностей.

Часто будет требоваться фильтрация последовательности, и фильтрация до выполнения соединения происходит более эффективно, чем после. На этом этапе выражение запроса оказывается проще, если левой последовательностью является та, которую нужно фильтровать. Например, если необходимо отобразить только закрытые дефекты, можно было бы воспользоваться следующим выражением запроса:

```
from defect in SampleData.AllDefects
where defect.Status == Status.Closed
join subscription in SampleData.AllSubscriptions
  on defect.Project equals subscription.Project
select new { defect.Summary, subscription.EmailAddress }
```

Выполнить тот же самый запрос можно и с переставленными последовательностями, но он станет более запутанным:

```
from subscription in SampleData.AllSubscriptions
join defect in (from defect in SampleData.AllDefects
                where defect.Status == Status.Closed
                select defect)
  on subscription.Project equals defect.Project
select new { defect.Summary, subscription.EmailAddress }
```

Обратите внимание на возможность применения одного выражения запроса внутри другого — многие трансляции компилятора описаны в спецификации языка в этих терминах. Вложенные выражения запросов удобны, но они также наносят урон читабельности; часто лучше искать альтернативу либо использовать переменную для последовательности справа, чтобы сделать код яснее.

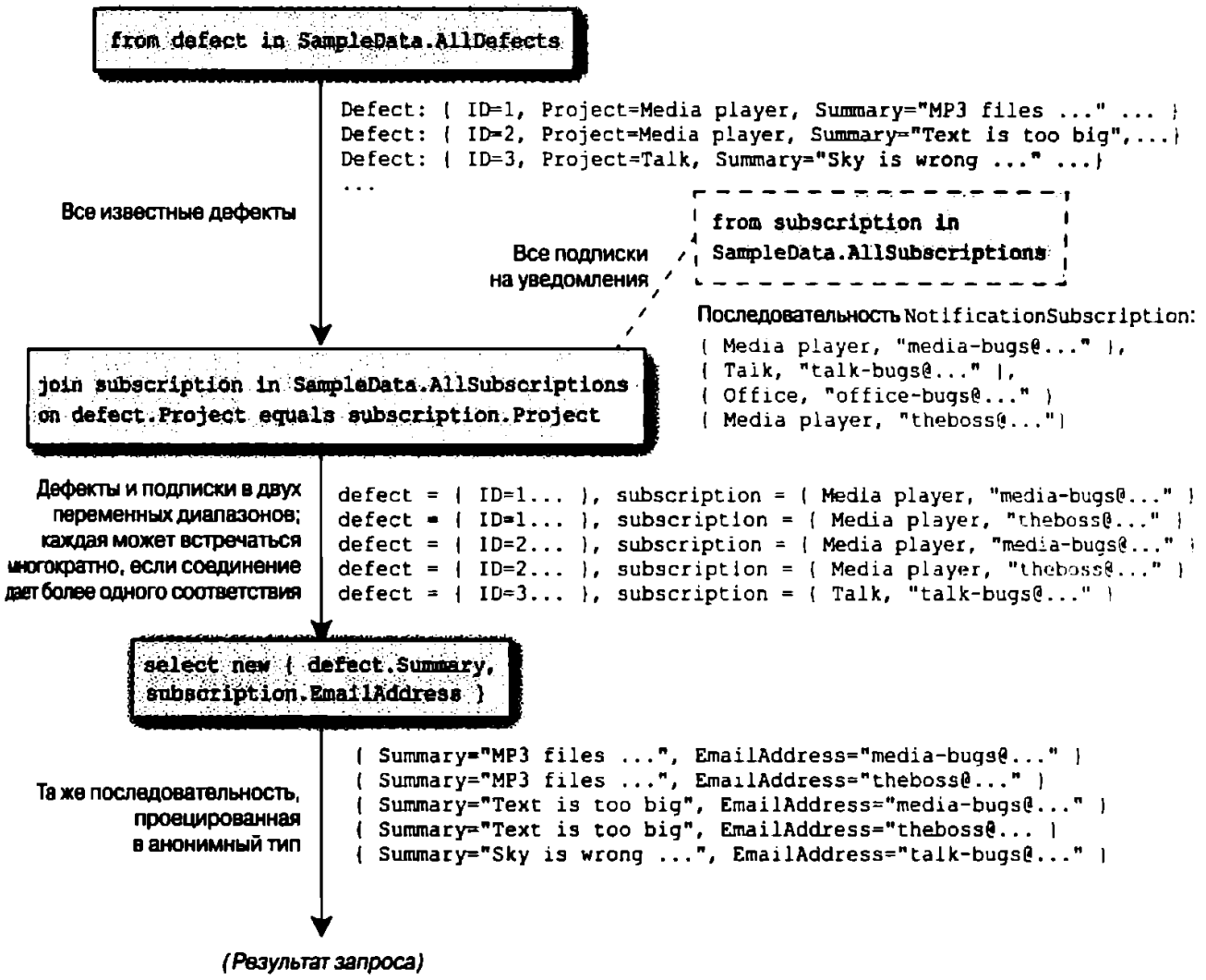

Рис. 11.6. Соединение из листинга 11.12 в графической форме, иллюстрирующей использование двух разных последовательностей (дефектов и подписок на уведомления) как источников данных

Полезны ли внутренние соединения в LINQ to Objects?

В SQL внутренние соединения применяются постоянно. Фактически они представляют собой способ навигации от одной сущности к другой, связанной с ней сущности, обычно с выполнением соединения внешнего ключа одной таблицы с первичным ключом другой. В объектно-ориентированной модели навигация между объектами, как правило, осуществляется с помощью ссылок. Например, извлечение сводной информации по дефекту и имени пользователя, который назначен для работы над ним, в SQL потребовало бы соединения — с другой стороны, в C# часто можно использовать цепочку свойств. Если бы в модели имелась обратная ассоциация между свойством `Project` и списком связанных с ним объектов `NotificationSubscription`, то для достижения цели этого примера не пришлось бы применять соединение. Это не говорит о том, что внутренние соединения временами не оказываются полезными внутри объектно-ориентированных моделей, но они не возникают естественным образом настолько часто, как это бывает в реляционных моделях.

Внутренние соединения транслируются компилятором в вызовы метода Join(), например:

```
leftSequence.Join(rightSequence,
                  leftKeySelector,
                  rightKeySelector,
                  resultSelector)
```

Сигнатура перегруженной версии, которая используется в LINQ to Objects, выглядит следующим образом (это реальная сигнатура, с реальными именами параметров – отсюда и ссылки на *inner* и *outer*):

```
static IEnumerable<TResult> Join<TOuter,TInner,TKey,TResult> (
  this IEnumerable<TOuter> outer,
  IEnumerable<TInner> inner,
  Func<TOuter,TKey> outerKeySelector,
  Func<TInner,TKey> innerKeySelector,
  Func<TOuter,TInner,TResult> resultSelector
)
```

Первые три параметра самоочевидны, если вы не забыли трактовать *inner* (внутренняя последовательность) и *outer* (внешняя последовательность) как *правую* и *левую* последовательности, но последний параметр более интересен. Он является проекцией из двух элементов (одного из левой последовательности и одного из правой последовательности) в единственный элемент результирующей последовательности.

Когда за соединением следует что-то, отличное от конструкции select, то компилятор C# 3 вводит прозрачный идентификатор, чтобы сделать переменные диапазонов, применяемые в обеих последовательностях, доступными для последующих конструкций, а также создает анонимный тип и простое отображение для использования с параметром resultSelector.

Но если следующей частью выражения запроса является конструкция select, проекция из конструкции select применяется напрямую как параметр resultSelector – не имеет смысла создавать пару и затем вызывать метод Select(), в то время как трансформацию можно сделать за один шаг. Вы по-прежнему можете *воспринимать* это как шаг "соединения", за которым следует шаг "выборки", несмотря на то, что они спрессованы в одиночный вызов метода. На мой взгляд, это приводит к более согласованной умозрительной модели, рассуждать о которой намного проще. Если вы не просматриваете сгенерированный код, просто проигнорируйте эту оптимизацию, выполненную компилятором.

Хорошая новость заключается в том, что после изучения внутренних соединений вы сочтете следующий тип соединения намного более простым в освоении.

11.5.2. Групповые соединения с использованием конструкций join...into

Вы видели, что результирующая последовательность, получаемая с помощью обычной конструкции join, состоит из пар элементов, по одному из каждой входной последовательности. *Групповое соединение* выглядит похожим в терминах выражения запроса, но дает совершенно другой результат. Каждый элемент группового соединения состоит из элемента левой последовательности (доступного с использованием исходной переменной диапазона) и *последовательности* всех подходящих элементов из правой после-

довательности. Последовательность подходящих элементов представлена в виде новой переменной диапазона, указанной с помощью идентификатора, который находится после into в конструкции join.

Давайте изменим предыдущий пример с целью применения группового соединения. В листинге 11.13 снова выводятся на консоль все дефекты и подписки на уведомления, затребованные для них, но с разделением по дефектам. Особое внимание обратите на то, что результаты отображаются с помощью вложенного цикла foreach.

Листинг 11.13. Групповое соединение дефектов и подписок на уведомления

```
var query = from defect in SampleData.AllDefects
            join subscription in SampleData.AllSubscriptions
              on defect.Project equals subscription.Project
              into groupedSubscriptions
            select new { Defect = defect,
                         Subscriptions = groupedSubscriptions };

foreach (var entry in query)
{
  Console.WriteLine(entry.Defect.Summary);
  foreach (var subscription in entry.Subscriptions)
  {
    Console.WriteLine (" {0}", subscription.EmailAddress);
  }
}
```

Свойство Subscriptions каждой записи — это вложенная последовательность подписок на уведомления, соответствующих дефекту из данной записи. На рис. 11.7 показано, как выполняется соединение двух начальных последовательностей.

Между внутренним и групповым соединением, а также между групповым соединением и обычным группированием, имеется одно важное отличие — групповое соединение имеет дело с соответствием "один к одному" между левой и результирующей последовательностями, даже если некоторые элементы в левой последовательности не соответствуют ни одному из элементов в правой последовательности. Это может оказаться важным и временами используется для эмуляции *левого внешнего соединения* из SQL. Когда для левого элемента отсутствуют подходящие правые элементы, вложенная последовательность пуста. Как и с внутренним соединением, групповое соединение буферизирует правую последовательность, но организует поток для левой последовательности.

В листинге 11.14 приведен пример запроса, который подсчитывает количество дефектов, обнаруженных ежедневно в мае. В нем применяется тип DateTimeRange для генерации левой последовательности дат в мае и проекция, которая вызывает метод Count() на встроенной последовательности внутри результата группового соединения[6].

[6] Эта простая реализация может выступать только в качестве примера и не является полнофункциональной или универсальной версией.

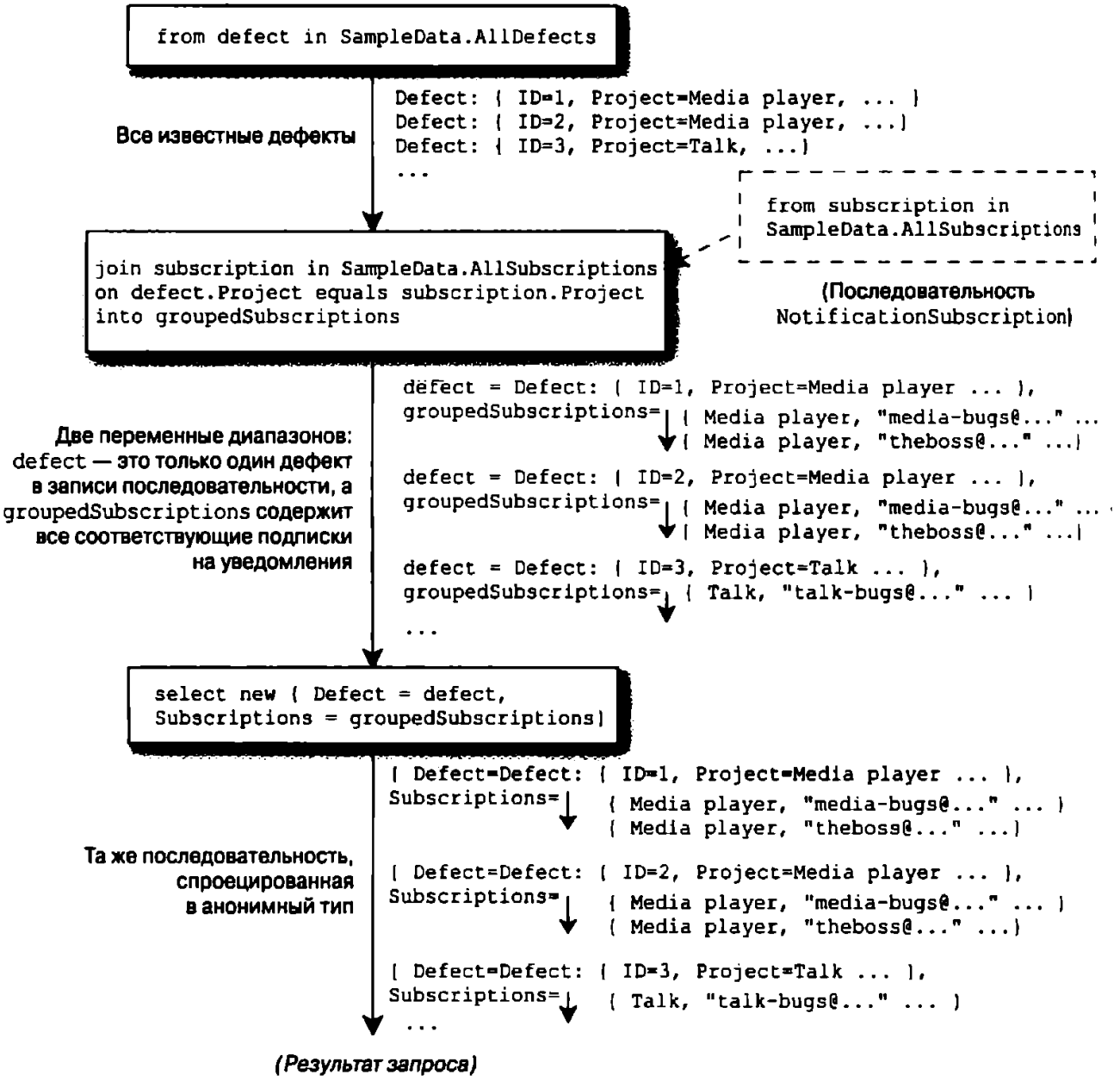

Рис. 11.7. Последовательности, задействованные в групповом соединении из листинга 11.13. Короткими линиями со стрелками отмечены внутренние последовательности в результирующих записях. Некоторые записи в выводе содержат множество адресов электронной почты для одного дефекта

Листинг 11.14. Подсчет количества дефектов, обнаруженных ежедневно в мае

```
var dates = new DateTimeRange(SampleData.Start, SampleData.End);
var query = from date in dates
            join defect in SampleData.AllDefects
              on date equals defect.Created.Date
              into joined
            select new { Date = date, Count = joined.Count() };
foreach (var grouped in query)
{
  Console.WriteLine("{0:d}: {1}", grouped.Date, grouped.Count);
}
```

Метод Count() использует немедленное выполнение, проходя по всем элементам последовательности, на которой он вызывается — однако он вызывается только в части проецирования выражения запроса, поэтому становится частью лямбда-выражения. Это означает, что мы по-прежнему имеем отложенное выполнение; до тех пор, пока не начнется цикл foreach, ничего выполняться не будет. Ниже показана первая часть результатов, генерируемых кодом из листинга 11.14, в которых отображено число обнаруженных дефектов ежедневно в течение первой недели мая:

```
05/01/2013: 1
05/02/2013: 0
05/03/2013: 2
05/04/2013: 1
05/05/2013: 0
05/06/2013: 1
05/07/2013: 1
```

Компилятор транслирует групповое соединение в вызов метода GroupJoin() аналогично тому, как внутреннее соединение транслируется в вызов Join(). Вот как выглядит сигнатура метода Enumerable.GroupJoin():

```
static IEnumerable<TResult> GroupJoin<TOuter,TInner,TKey,TResult>(
  this IEnumerable<TOuter> outer,
  IEnumerable<TInner> inner,
  Func<TOuter,TKey> outerKeySelector,
  Func<TInner,TKey> innerKeySelector,
  Func<TOuter,IEnumerable<TInner>,TResult> resultSelector
)
```

Сигнатура в точности такая же, как для внутренних соединений, за исключением того, что параметр resultSelector должен работать с последовательностью элементов, находящихся справа, а не с единственным элементом. Как и во внутренних соединениях, если за групповым соединением следует конструкция select, в качестве селектора результата вызова GroupJoin() применяется проекция; в противном случае вводится прозрачный идентификатор. В рассматриваемой ситуации конструкция select находится непосредственно после группового соединения, поэтому транслированный запрос выглядит следующим образом:

```
dates.GroupJoin(SampleData.AllDefects,
                date => date,
                defect => defect.Created.Date,
                (date, joined) => new  { Date = date,
                                         Count = joined.Count() })
```

Последний тип соединений называется *перекрестным соединением*, и он не так прост, как может показаться на первый взгляд.

11.5.3. Перекрестные соединения и выравнивание последовательностей с использованием нескольких конструкций from

До сих пор все соединения были *эквисоединениями* (т.е. соединениями по эквивалентности) — сопоставление выполнялось между элементами левой и правой последовательностей. Перекрестные соединения не делают никаких сопоставлений между данными последовательностями; результат содержит все возможные пары элементов. Это дости-

гается за счет применения двух (или более) конструкций from. Для простоты мы будем пока рассматривать только две конструкции from — когда их больше, представляйте себе перекрестное соединение на первых двух конструкциях from, затем перекрестное соединение на результирующей последовательности и следующей конструкции from и т.д. Дополнительная конструкция from добавляет свою переменную диапазона через прозрачный идентификатор.

В листинге 11.15 демонстрируется простое (и бесполезное) перекрестное соединение в действии, генерируя последовательность, в которой каждая запись содержит информацию о пользователе и проекте. Я намеренно выбрал две совершенно несвязанных друг с другом начальных последовательности, чтобы доказать, что никакого сопоставления не выполняется.

Листинг 11.15. Перекрестное соединение пользователей и проектов

```
var query = from user in SampleData.AllUsers
            from project in SampleData.AllProjects
            select new { User = user, Project = project };

foreach (var pair in query)
{
  Console.WriteLine("{0}/{1}",
                    pair.User.Name,
                    pair.Project.Name);
}
```

Ниже показан вывод кода из листинга 11.15:

```
Tim Trotter/Skeety Media Player
Tim Trotter/Skeety Talk
Tim Trotter/Skeety Office
Tara Tutu/Skeety Media Player
Tara Tutu/Skeety Talk
Tara Tutu/Skeety Office
```

На рис. 11.8 представлены последовательности, участвующие в получении такого результата.

Если вы знакомы с языком SQL, то, скорее всего, хорошо понимаете, что происходило до сих пор, т.к. это выглядит подобно декартову произведению, полученному из запроса, в котором указано несколько таблиц. Но здесь при необходимости доступна и более мощная возможность: правая последовательность может зависеть от текущего значения в левой последовательности. Другими словами, каждый элемент в левой последовательности используется для генерации правой последовательности, а затем этот элемент из левой последовательности образует пары с каждым элементом в новой последовательности. Когда дело обстоит именно так, это перестает быть перекрестным соединением в обычном смысле данного термина. Фактически оно выравнивает последовательность из последовательностей в единственную последовательность. Трансляция выражения запроса будет той же самой, применяется настоящее перекрестное соединение или нет, поэтому для понимания процесса трансляции необходимо разобраться с более сложным сценарием.

Перед тем, как углубляться в детали, давайте посмотрим на производимый результат. В листинге 11.16 приведен простой пример, в котором используются последовательности целых чисел.

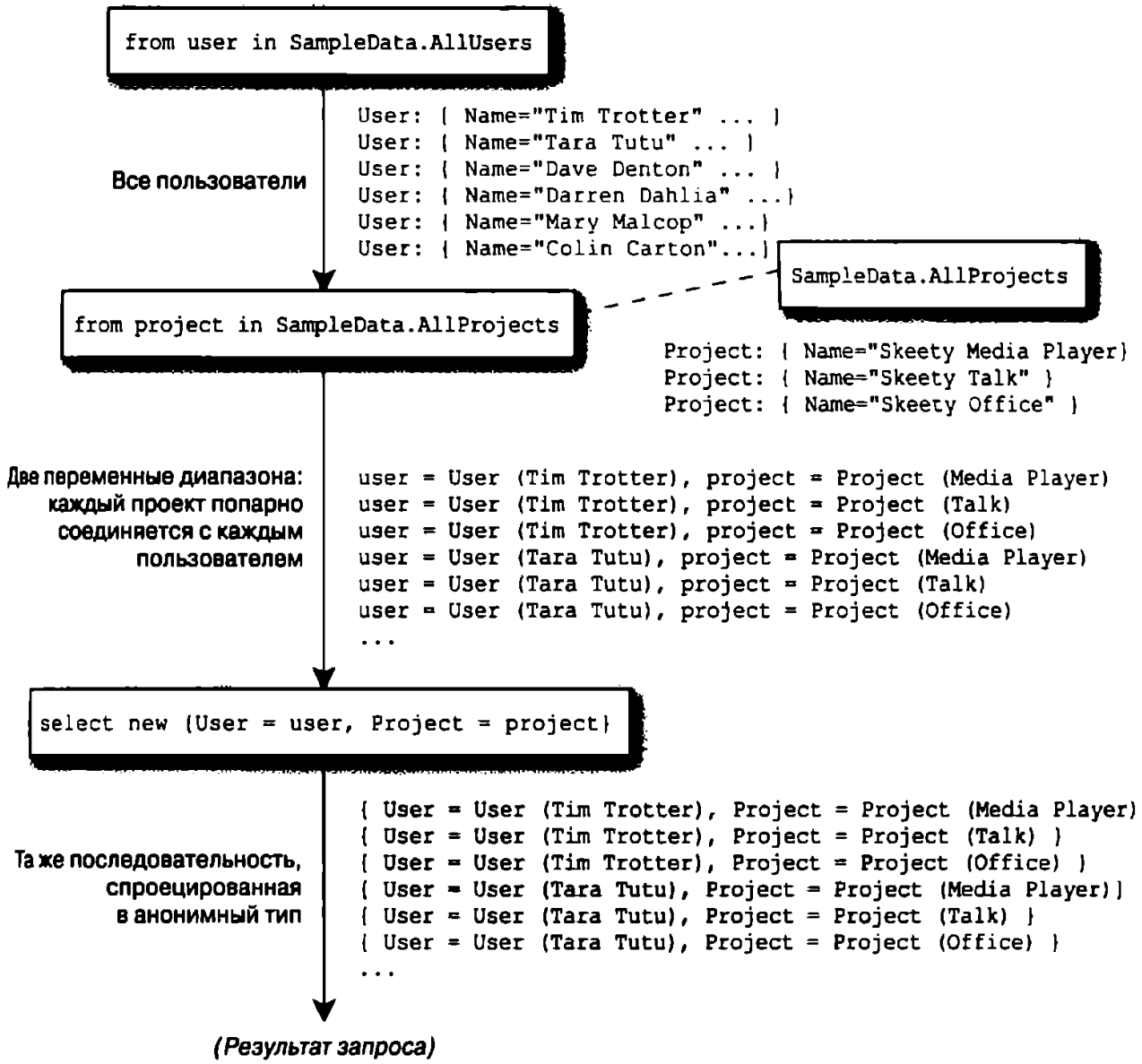

Рис. 11.8. Последовательности из листинга 11.15 во время выполнения перекрестного соединения пользователей и проектов. В результате возвращаются все возможные комбинации

Листинг 11.16. Перекрестное соединение, при котором правая последовательность зависит от элемента из левой последовательности

```
var query = from left in Enumerable.Range(1, 4)
            from right in Enumerable.Range(11, left)
            select new { Left = left, Right = right };

foreach (var pair in query)
{
  Console.WriteLine("Left={0}; Right={1}",
                    pair.Left, pair.Right);
}
```

Код в листинге 11.16 начинается с создания простого диапазона целых чисел от 1 до 4. Для каждого числа создается другой диапазон, который начинается с 11 и имеет количество элементов, равное исходному целому числу. С помощью нескольких конс-

трукций from левая последовательность соединяется с каждой генерируемой правой последовательностью, давая в результате следующий вывод:

```
Left=1; Right=11
Left=2; Right=11
Left=2; Right=12
Left=3; Right=11
Left=3; Right=12
Left=3; Right=13
Left=4; Right=11
Left=4; Right=12
Left=4; Right=13
Left=4; Right=14
```

Для генерации этой последовательности компилятор вызывает метод SelectMany(). Он принимает единственную входную последовательность (*левая* последовательность в нашей терминологии), делегат для *генерации* другой последовательности из любого элемента левой последовательности и делегат для генерации результирующего элемента на основе элемента из каждой последовательности. Вот как выглядит сигнатура метода Enumerable.SelectMany():

```
static IEnumerable<TResult> SelectMany<TSource,TCollection,TResult>(
  this IEnumerable<TSource> source,
  Func<TSource,IEnumerable<TCollection>> collectionSelector,
  Func<TSource,TCollection,TResult> resultSelector
)
```

Как и с другими видами соединений, если за частью, касающейся соединения, в выражении запроса следует конструкция select, эта проекция используется в качестве последнего аргумента; иначе вводится прозрачный идентификатор, чтобы сделать переменные диапазонов левой и правой последовательностей доступными позже в запросе.

Просто чтобы чуть конкретизировать сказанное, ниже приведено выражение запроса из листинга 11.16 в виде транслированного исходного кода:

```
Enumerable.Range(1, 4)
          .SelectMany(left => Enumerable.Range(11, left),
                      (left, right) => new {Left = left, Right = right})
```

Одной интересной особенностью метода SelectMany() является полностью потоковое выполнение — в любой момент времени необходимо обрабатывать только один элемент, поскольку для каждого отдельного элемента из левой последовательности применяется заново сгенерированная правая последовательность. Сравните это с внутренними и групповыми соединениями: перед началом возвращения результатов указанные виды соединений полностью загружают правую последовательность.

Поведение выравнивания SelectMany() может оказаться весьма удобным. Возьмем ситуацию, когда нужно обработать множество журнальных файлов построчно. Бесшовную последовательность строк можно обрабатывать практически любым способом. Показанный ниже псевдокод более подробно представлен в загружаемом коде, но его общий смысл и польза должны быть вполне очевидными:

```
var query = from file in Directory.GetFiles(logDirectory, "*.log")
            from line in ReadLines(file)
            let entry = new LogEntry(line)
            where entry.Type == EntryType.Error
            select entry;
```

С помощью всего пяти строк кода имеется возможность извлечь, проанализировать и отфильтровать целую коллекцию журнальных файлов, возвращая последовательность записей, которые представляют ошибки. Важно отметить, что ни один журнальный файл не пришлось загружать полностью в память за раз, не говоря уже о загрузке всех файлов — все данные организованы в поток.

По сравнению с соединениями последние элементы, которые необходимо рассмотреть, несколько проще для понимания. Мы взглянем на элементы группирования по ключу и на продолжение выражения запроса после конструкции `group...by` или `select`.

11.6. Группирование и продолжение

Группирование последовательности элементов по одному из их свойств является распространенным требованием. Это легко делается в LINQ с помощью конструкции `group...by`. В дополнение к описанию этого финального типа конструкции в настоящем разделе мы также возвратимся к конструкции `select`, чтобы ознакомится со средством, которое называется *продолжением запроса* и может применяться как к группированию, так и к проецированию. Давайте начнем с простого группирования.

11.6.1. Группирование с помощью конструкции `group...by`

Группирование должно быть интуитивно понятным, а LINQ делает его простым. Чтобы сгруппировать последовательность в выражении запроса, понадобится всего лишь воспользоваться конструкцией `group...by` со следующим синтаксисом:

```
group проекция by группирование
```

Подобно `select`, эта конструкция располагается в конце выражения запроса. Сходства между данными конструкциями на этом не заканчиваются: выражение *проекция* является тем же видом проекции, которая применяется в конструкциях `select`. Тем не менее, результат получается несколько иной.

Выражение *группирование* определяет, каким образом последовательность группируется — это *селектор ключей* операции группирования. Общий результат представляет собой последовательность, в которой каждый элемент является группой. Каждая группа — это последовательность спроецированных элементов, которые также имеют свойство `Key`, выступающее в роли ключа для данной группы; такая комбинация инкапсулирована в интерфейсе `IGrouping<TKey,TElement>`, расширяющем `IEnumerable<TElement>`. Опять-таки, если нужно сгруппировать по нескольким значениям, для ключа можно использовать анонимный тип.

Давайте рассмотрим простой пример из системы регистрации дефектов SkeetySoft: группирование дефектов по лицам, которым назначено их исправление.

В листинге 11.17 это делается с помощью простейшей формы проецирования, так что результирующая последовательность в каждой записи содержит назначенного исполнителя в качестве ключа и встроенную последовательность дефектов.

Листинг 11.17. Группирование дефектов по назначенному исполнителю — тривиальная проекция

```
var query = from defect in SampleData.AllDefects
            where defect.AssignedTo != null          <-- (1) Фильтрация дефектов, которым не назначены исполнители
            group defect by defect.AssignedTo;       <-- (2) Группирование по назначенному исполнителю
```

```
foreach (var entry in query)
{
  Console.WriteLine(entry.Key.Name);   <-- ❷ Использование ключа каждой записи: назначенного исполнителя
  foreach (var defect in entry)        <-- ❶ Проход по встроенной последовательности
  {
    Console.WriteLine("  ({0}) {1}",
                  defect.Severity, defect.Summary);
  }
  Console.WriteLine();
}
```

Код в листинге 11.17 может оказаться удобным для построения ежедневных отчетов, позволяющих быстро просмотреть, на какие дефекты должны обращать внимание ответственные за их устранение. Он отфильтровывает дефекты, которые больше не нуждаются во внимании ❶, и затем осуществляет группирование с применением свойства `AssignedTo`. Хотя в этот раз используется свойство, выражение группирования может быть любым — оно применяется к каждой записи во входящей последовательности, а последовательность группируется на основе результата этого выражения. Обратите внимание, что группирование не может обеспечить организацию потоков для результатов; оно применяет селектор ключей и проекцию к каждому элементу во входной последовательности и буферизирует сгруппированные последовательности проецированных элементов. Несмотря на то что для этого не организуется поток, выполнение по-прежнему откладывается до тех пор, пока не начнется извлечение результатов.

Применяемое при группировании ❷ проецирование тривиально — оно лишь выбирает исходный элемент. По мере прохождения по результирующей последовательности выясняется, что каждая запись имеет свойство `Key` типа `User` ❷, а также реализует интерфейс `IEnumerable<Defect>`, который представляет последовательность дефектов, назначенных для исправления данному пользователю ❶.

Результаты выполнения кода из листинга 11.17 начинаются так:

```
Darren Dahlia
  (Showstopper) MP3 files crash system
  (Major) Can't play files more than 200 bytes long
  (Major) DivX is choppy on Pentium 100
  (Trivial) User interface should be more caramelly
```

После вывода на консоль всех дефектов, назначенных для исправления Даррену, будут выведены дефекты, назначенные Таре, Тиму и т.д. В сущности, реализация сохраняет список назначенных исполнителей, показанный до сих пор, и добавляет нового назначенного исполнителя каждый раз, когда это необходимо. На рис. 11.9 показаны последовательности, сгенерированные на протяжении всего выражения запроса, что может прояснить это упорядочение.

Внутри вложенной последовательности каждой записи порядок следования дефектов совпадает с порядком их следования в исходной последовательности дефектов. Если вы всерьез проявляете интерес к упорядочению, подумайте о явном его указании в выражении запроса, сделав его более читабельным.

Если вы запустите код из листинга 11.17, то заметите, что записи для Мэри (Mary Malcop) в выводе вообще отсутствуют, поскольку ей не назначены какие-либо дефекты для исправления. Если необходимо построить полный список пользователей и дефектов, назначенных им для исправления, придется применять групповое соединение, подобное показанному в листинге 11.14.

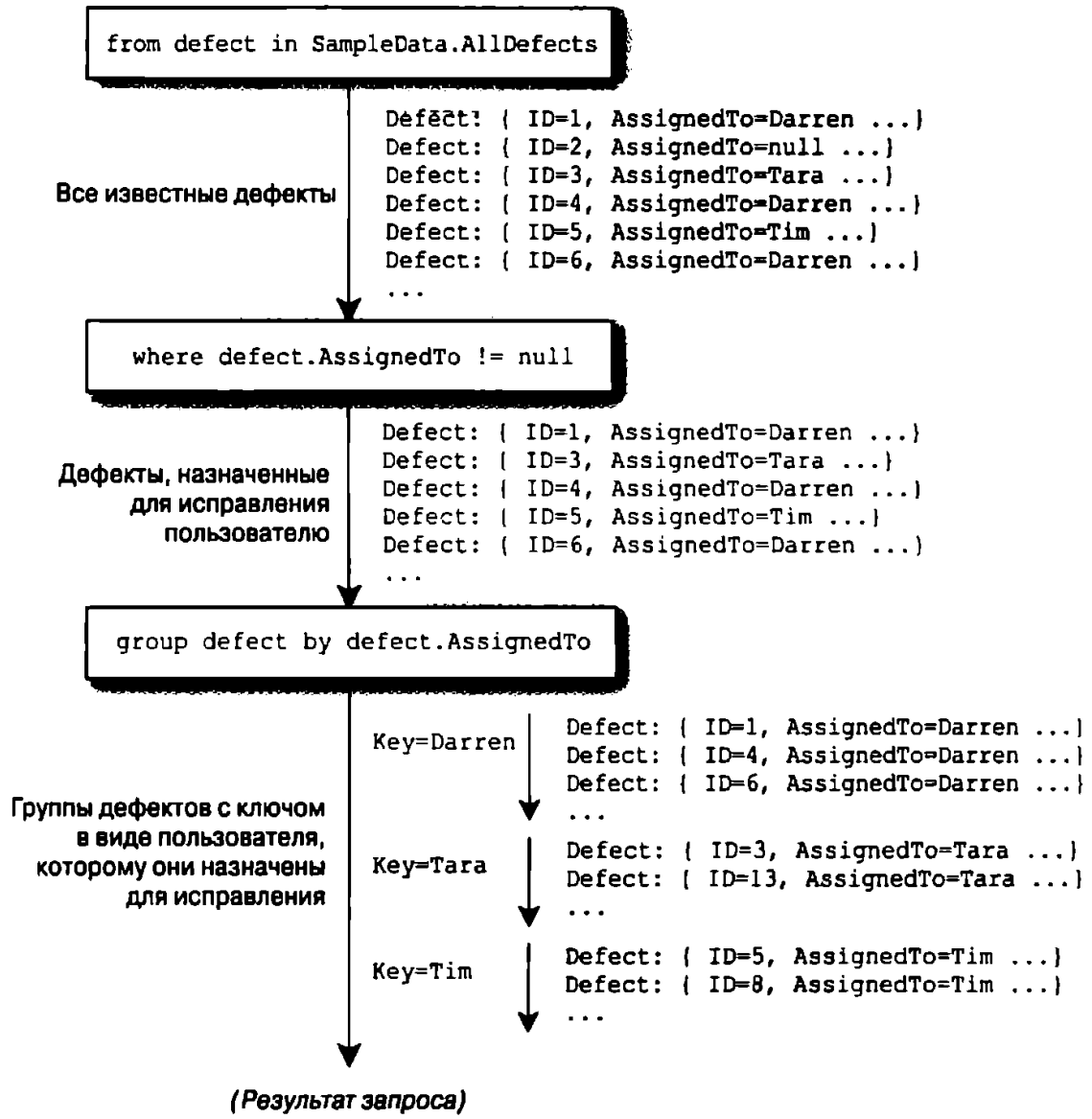

Рис. 11.9. Последовательности, используемые при группировании дефектов по назначенному исполнителю. Каждая запись результата имеет свойство `Key` и также содержит последовательность записей о дефектах

Для конструкций группирования компилятор всегда использует метод по имени `GroupBy()`. Когда проекция в конструкции группирования тривиальна, т.е. каждая запись в исходной последовательности отображается прямо на тот же самый объект во вложенной последовательности, компилятор применяет простой вызов метода, который требует только выражения группирования, поэтому ему известно, как отображать каждый элемент на ключ. Например, выражение запроса из листинга 11.17 транслируется следующим образом:

```
SampleData.AllDefects.Where(defect => defect.AssignedTo != null)
                     .GroupBy(defect => defect.AssignedTo)
```

Когда проекция не является тривиальной, используется несколько более сложная версия. В листинге 11.18 приведен пример проекции, при которой захватывается только сводная информация по каждому дефекту, а не сам объект `Defect`.

Листинг 11.18. Группирование дефектов по назначенному исполнителю — проекция оставляет только сводную информацию

```
var query = from defect in SampleData.AllDefects
            where defect.AssignedTo != null
            group defect.Summary by defect.AssignedTo;
foreach (var entry in query)
{
  Console.WriteLine(entry.Key.Name);
  foreach (var summary in entry)
  {
    Console.WriteLine(" {0}", summary);
  }
  Console.WriteLine();
}
```

Отличия между листингами 11.18 и 11.17 выделены полужирным. Поскольку каждый дефект проецируется в его сводную информацию, встроенной последовательностью внутри каждой записи будет IEnumerable<string>. В этом случае компилятор применяет перегруженную версию метода GroupBy() с другим параметром, представляющим проекцию. Выражение запроса из листинга 11.18 транслируется в следующее выражение:

```
SampleData.AllDefects.Where(defect => defect.AssignedTo != null)
                     .GroupBy(defect => defect.AssignedTo,
                              defect => defect.Summary)
```

Конструкции группирования относительно просты, но вместе с тем полезны. Даже в системе регистрации дефектов можно легко обосновать желание группировать дефекты по проекту, создателю, уровню серьезности или состоянию, равно как и по назначенному исполнителю, как было в рассмотренных примерах.

До сих пор каждое выражение завершалось конструкцией `select` или `group...by`, и это было концом выражения. Однако иногда требуется выполнить дополнительные действия над результатами, для чего и предназначено *продолжение запроса*.

11.6.2. Продолжение запроса

Продолжение запроса предоставляет способ использования результата из одного выражения запроса в качестве начальной последовательности для другого выражения запроса. Продолжение применяется к конструкциям `group...by` и `select`, и для обеих конструкций синтаксис одинаков — нужно просто указать контекстное ключевое слово `into` и затем имя новой переменной диапазона. Далее эта переменная диапазона может использоваться в следующей части выражения запроса.

В спецификации такое продолжение объясняется в терминах трансляции из одного выражения в другое, с превращением

первый-запрос `into` *идентификатор*
тело-второго-запроса

в такой вид:

`from` *идентификатор* `in` (*первый-запрос*)
тело-второго-запроса

Прояснить это поможет пример. Давайте возвратимся к группированию дефектов по назначенному исполнителю, но на этот раз представим, что необходимо только подсчитать количество дефектов, назначенных для исправления каждому лицу. Это не удастся сделать с помощью проекции в конструкции группирования, потому что она применяется только к каждому отдельному дефекту. Требуется спроецировать каждую группу, содержащую назначенного исполнителя и последовательность дефектов, за исправление которых он отвечает, в единственный элемент, состоящий из назначенного исполнителя и количества дефектов в группе. Необходимый запрос показан в листинге 11.19.

Листинг 11.19. Продолжение группирования другой проекцией

```
var query = from defect in SampleData.AllDefects
            where defect.AssignedTo != null
            group defect by defect.AssignedTo into grouped
            select new { Assignee = grouped.Key,
                         Count = grouped.Count() };
foreach (var entry in query)
{
  Console.WriteLine("{0}: {1}",
                    entry.Assignee.Name, entry.Count);
}
```

Изменения в выражении запроса выделены полужирным. Переменную диапазона grouped можно использовать во второй части запроса, но переменная диапазона defect больше не доступна — ее можно считать покинувшей область действия. Эта проекция просто создает анонимный тип со свойствами Assignee и Count, применяя ключ каждой группы как назначенного исполнителя и подсчитывая количество дефектов в последовательности, которая связана с каждой группой.

Ниже показаны результаты выполнения запроса из листинга 11.19:

```
Darren Dahlia: 14
Tara Tutu: 5
Tim Trotter: 5
Deborah Denton: 9
Colin Carton: 2
```

Согласно спецификации, выражение запроса из листинга 11.19 транслируется так:

```
from grouped in (from defect in SampleData.AllDefects
                 where defect.AssignedTo != null
                 group defect by defect.AssignedTo)
select new { Assignee = grouped.Key, Count = grouped.Count() }
```

Затем выполняются остальные трансляции, давая в итоге следующий код:

```
SampleData.AllDefects
          .Where(defect => defect.AssignedTo != null)
          .GroupBy(defect => defect.AssignedTo)
          .Select(grouped => new { Assignee = grouped.Key,
                                   Count = grouped.Count() })
```

Альтернативный подход к пониманию продолжения предусматривает его восприятие как двух отдельных операторов. Это не настолько точно в плане терминов действительной трансляции, выполняемой компилятором, но я считаю, что такой подход упрощает выяснение происходящего. В данном случае выражение запроса (и присваивание переменной query) можно трактовать как два показанных ниже оператора:

```
var tmp = from defect in SampleData.AllDefects
          where defect.AssignedTo != null
          group defect by defect.AssignedTo;
var query = from grouped in tmp
          select new { Assignee = grouped.Key,
                       Count = grouped.Count() };
```

Разумеется, если вы находите это более простым в чтении, то ничего не препятствует тому, чтобы привести первоначальное выражение в такую форму в своем исходном коде. Из-за отложенного выполнения ничего не будет оцениваться до тех пор, пока не начнется проход каким-нибудь образом по результатам запроса.

Конструкция join...into не является продолжением

Очень легко попасть в ловушку, думая, что всегда, когда встречается контекстное ключевое слово `into`, оно обозначает продолжение. В случае соединений это неверно — конструкция `join...into` (используемая для групповых соединений) не формирует продолжение. Важное отличие заключается в том, что при групповом соединении все ранее введенные переменные диапазонов (кроме той, что применяется для именования правой стороны соединения) могут по-прежнему использоваться. Сравните это с запросами, рассматриваемыми в настоящем разделе, где продолжение начинает все с начала; впоследствии доступна *только* переменная диапазона, объявленная продолжением.

Давайте расширим пример, чтобы посмотреть, как работать с несколькими продолжениями. Результаты в данный момент не отсортированы — изменим это, чтобы первым выводился исполнитель, которому назначено исправление наибольшего числа дефектов. Можно было бы воспользоваться конструкцией `let` после первого продолжения, но в листинге 11.20 продемонстрирован альтернативный подход с применением второго продолжения после текущего выражения.

Листинг 11.20. Продолжения в выражении запроса из group и select

```
var query = from defect in SampleData.AllDefects
          where defect.AssignedTo != null
          group defect by defect.AssignedTo into grouped
          select new { Assignee = grouped.Key,
                       Count = grouped.Count() } into result
          orderby result.Count descending
          select result;

foreach (var entry in query)
{
  Console.WriteLine("{0}: {1}",
                    entry.Assignee.Name,
                    entry.Count);
}
```

Отличия между листингами 11.19 и 11.20 выделены полужирным. Необходимость в изменении кода вывода на консоль отсутствует, поскольку последовательность осталась той же самой, а к ней просто было применено упорядочение. На этот раз транслированное выражение запроса выглядит следующим образом:

```
SampleData.AllDefects
          .Where(defect => defect.AssignedTo != null)
          .GroupBy(defect => defect.AssignedTo)
          .Select(grouped => new { Assignee = grouped.Key,
                                   Count = grouped.Count() })
      .OrderByDescending(result => result.Count);
```

По чисто случайному стечению обстоятельств это удивительно похоже на первый запрос по отслеживанию дефектов, который приводился в разделе 10.3.6. Финальная конструкция `select` фактически ничего не делает, поэтому компилятор C# ее игнорирует. Тем не менее, эта конструкция обязательна в выражении запроса, т.к. все выражения запросов должны заканчиваться либо на `select`, либо на `group...by`. Ничего не мешает использовать отличающееся проецирование или выполнять другие операции после продолжения запроса — соединения, дальнейшие группирования и т.д. Просто по мере роста выражения запроса присматривайте за читабельностью.

Говоря о читабельности, стоит упомянуть о возможностях, которые доступны при написании запросов LINQ.

11.7. Выбор между выражениями запросов и точечной нотацией

Как вы видели повсеместно в главе, перед компиляцией выражения запросов транслируются в нормальный код C#. Хотя официальное название для способа обращения к операциям запросов LINQ, который предусматривает написание обычного кода C# вместо выражения запроса, отсутствует, многие разработчики ссылаются на него как на *точечную нотацию*[7]. Каждое выражение запроса может быть записано с помощью точечной нотации, однако обратное утверждение несправедливо: многие операции LINQ не имеют эквивалентов в выражениях запросов на C#. Главный вопрос вот в чем: когда должен использоваться тот или другой синтаксис?

11.7.1. Операции, которые требуют точечной нотации

Наиболее очевидной ситуацией, когда вынужденно приходится применять точечную нотацию, является вызов такого метода, как `Reverse()` или `ToDictionary()`, который вообще не представлен в рамках синтаксиса выражений запросов. Но даже когда используется операция, поддерживаемая выражениями запросов, вполне возможно, что желаемая перегруженная версия окажется недоступной.

Например, метод `Enumerable.Where()` имеет перегруженную версию, которая принимает индекс в родительской последовательности как еще один аргумент делегата. В ситуации такого рода для получения каждого второго элемента последовательности можно было бы написать следующий код:

```
sequence.Where((item, index) => index % 2 == 0)
```

Существует аналогичная перегруженная версия и для метода `Select()`, поэтому если вам необходима возможность получить исходный индекс в последовательности после упорядочения, понадобится написать такой код:

```
sequence.Select((Item, Index) => new { Item, Index })
        .OrderBy(x => x.Item.Name)
```

[7] С этого момента я буду использовать данный термин, но если вы слышите, как другие говорят о *текучей нотации*, скорее всего, они имеют в виду то же самое.

Этот пример показывает еще один вариант, который может быть принят во внимание: если вы собираетесь применять параметр с лямбда-выражением непосредственно в анонимном типе, то могли бы отказаться от нормального соглашения о начале имени параметра с буквы нижнего регистра и затем использовать инициализатор проекции во избежание написания кода `new { Item = item, Index = index }`, который приводит к путанице. Конечно, взамен можно проигнорировать соглашение об именовании свойств и определить в анонимном типе свойства с именами, начинающимися с буквы нижнего регистра (`item` и `index`, например). Решение всецело зависит от вас и экспериментирование делу не повредит. Хотя согласованность обычно важна, здесь она не играет слишком большой роли, т.к. влияние несогласованности ограничено данным методом; эти имена не доступны через открытый API-интерфейс или в других местах класса.

Многие операции запросов также поддерживают специальные сравнения — самыми очевидными примерами могут служить упорядочение и соединение. По моему опыту они вряд ли будут требоваться часто, но иногда это бесценно. Например, если необходимо выполнить соединение по имени человека в независимой от регистра манере, то в качестве последнего аргумента при вызове `Join()` можно указать значение свойства `StringComparer.OrdinalIgnoreCase` (или специфичный к культуре компаратор). Опять-таки, если есть ощущение, что действие операции *приблизительно* соответствует нужному, но не полностью, проверьте документацию на предмет наличия других перегруженных версий этой операции. Когда применение точечной нотации неизбежно, решение об ее использовании принимается легко, но что можно сказать о случаях, при которых *могло бы* применяться выражение запроса?

11.7.2. Использование выражений запросов в ситуациях, когда точечная нотация может быть проще

Некоторые разработчики применяют выражения запросов везде, где это сходит с рук; лично я сначала смотрю на то, что запрос делает, и затем решаю, какой подход будет более читабельным.

Например, возьмем следующее выражение запроса, которое похоже на выражение, представленное ближе к началу этой главы:

```
var adults = from person in people
             where person.Age >= 18
             select person;
```

Показанные три строки кода несут в себе немалый багаж, хотя все, что они делают – это фильтрация. В данном случае я бы использовал точечную нотацию:

```
var adults = people.Where(person => person.Age >= 18);
```

Я нахожу такой код более ясным — каждая его часть ссылается на что-то, в чем вы действительно заинтересованы.

Еще одной областью, в которой применение точечной нотации во всем выражении запроса может обеспечить большую ясность, является ситуация, когда она должна использоваться для части выражения в любом случае. К примеру, предположим, что вы собираетесь применять расширяющий метод `ToList()` для получения списка имен совершеннолетних. (В данном случае я использую также и проекцию, так что это более сбалансированное сравнение.) Вот как выглядит выражение запроса:

```
var adultNames = (from person in people
                  where person.Age >= 18
                  select person.Name).ToList();
```

А ниже показан эквивалент в точечной нотации:

```
var adultNames = people.Where(person => person.Age >= 18)
                       .Select(person => person.Name)
                       .ToList();
```

В первом случае кое-что, связанное с необходимостью помещения выражения запроса в круглые скобки, делает его более неприглядным *для меня*. Во многом это дело персонального выбора — данный раздел действительно укрепляет осознание того, что выбор *существует*, и вы можете самостоятельно принять решение. Если вы планируете применять LINQ в сколько-нибудь значительной степени, то должны на самом деле хорошо освоить обе нотации и не испытывать страх перед переключением стиля на основе конкретного запроса. Как вы видели, генерируемый код совершенно одинаков. Разумеется, ничего из всего этого не говорит о том, что мне не нравятся выражения запросов.

11.7.3. Ситуации, когда выражения запросов блестящи

После объяснения ситуаций, в которых точечная нотация может оказаться более выгодной, я должен обратить внимание на то, что когда дело доходит до операций, где выражение запроса использовало бы прозрачные идентификаторы — особенно это касается соединений — точечная нотация начинает негативно сказываться на читабельности. Красота прозрачных идентификаторов кроется именно в их прозрачности — они настолько прозрачны, что вы вообще не сможете их заметить, мельком взглянув на выражение запроса. Даже простой конструкции `let` может быть достаточно, чтобы вынести решение в пользу выражений запросов; введение нового анонимного типа лишь для распространения контекста через запрос может быстро утомить.

Другой областью, в которой выигрывают выражения запросов, являются ситуации, когда требуется сразу несколько лямбда-выражений или даже множество вызовов методов. Опять-таки, сюда входят соединения, в которых необходимо указать селектор ключей для каждой стороны соединения, а также селектор результатов. Например, ниже приведена усеченная версия ранее показанного запроса, когда давалось введение во внутренние соединения:

```
from defect in SampleData.AllDefects
join subscription in SampleData.AllSubscriptions
  on defect.Project equals subscription.Project
select new { defect.Summary, subscription.EmailAddress }
```

В IDE-среде было бы разумно разместить всю конструкцию `join` в одной строке, получив тем самым более читабельный код. С другой стороны, эквивалент запроса в точечной нотации выглядит довольно непривлекательно:

```
SampleData.AllDefects.Join(SampleData.AllSubscriptions,
  defect => defect.Project,
  subscription => subscription.Project,
  (defect, subscription) => new { defect.Summary,
                                  subscription.EmailAddress })
```

Последний аргумент мог бы уместиться в одну строку в IDE-среде, но он все равно бы выглядел неуклюже, поскольку лямбда-выражения не содержат особенно много контекста; вы не сумеете немедленно сказать, что означает тот или иной аргумент. Здесь могут помочь именованные аргументы из C# 4, но это даже больше увеличивает объем запроса. Сложные упорядочения могут быть аналогично неприглядными в точечной нотации.

Посмотрите, что лучше читается — следующая конструкция `orderby`:

```
orderby item.Rating descending, item.Price, item.Name
```

или три вызова методов:

```
.OrderByDescending(item => item.Rating)
.ThenBy(item => item.Price)
.ThenBy(item => item.Name)
```

Изменение приоритета этих упорядочений в выражении запроса осуществляется просто — достаточно поменять их местами. В точечной нотации может также потребоваться переключиться с `OrderBy()` на `ThenBy()` или наоборот.

Повторюсь, я не пытаюсь навязать свои персональные предпочтения по написанию кода. Я просто хочу, чтобы вы знали, что именно доступно, и обдумывали выбираемые варианты. Конечно, это только один аспект написания читабельного кода, но он представляет собой совершенно новую область языка C#, которую следует иметь в виду.

11.8. Резюме

В этой главе мы рассмотрели взаимодействие LINQ to Objects и C# 3, уделив основное внимание первоначальной трансляции выражений запросов в код, в котором выражения запросов *не* задействованы и который затем компилируется обычным способом. Вы видели, что все выражения запросов формируют набор последовательностей, на каждом шаге преобразуя некоторые описания. Во многих случаях эти последовательности оцениваются с применением отложенного выполнения, при котором данные извлекаются только при первом их запрашивании.

По сравнению со всеми остальными средствами C# 3 выражения запросов выглядят несколько чужеродными — они больше похожи на SQL, чем на привычный код C#. Одна из причин, по которым выражения запросов имеют настолько странный вид, связана с тем, что они являются *декларативными*, а не *императивными* — запрос сообщает о характеристиках результата, а не о точных шагах, необходимых для его получения. Это идет рука об руку с более функциональным стилем мышления. Его понимание может потребовать определенного времени, к тому же оно не подходит абсолютно во всех ситуациях, но там, где декларативный синтаксис уместен, значительно улучшается читабельность, а также упрощается тестирование и распараллеливание кода.

Не вводите себя в заблуждение, думая, что язык LINQ должен использоваться только с базами данных. Распространенным его применением является манипулирование коллекциями в памяти, вдобавок вы видели, насколько хорошо он поддерживается выражениями запросов и расширяющими методами в классе `Enumerable`.

На самом деле вы ознакомились со всеми средствами, появившимися в C# 3! Пока еще не были показаны какие-то другие поставщики LINQ, но вы уже лучше понимаете, что компилятор будет делать, когда вы запросите у него обработку XML и SQL. Самому компилятору ничего не известно о разнице между LINQ to Objects, LINQ to SQL или любыми другими поставщиками; он всего лишь вслепую следует одним и тем же правилам.

В следующей главе вы увидите, как эти правила формируют финальный фрагмент головоломки LINQ, когда они преобразуют лямбда-выражения в деревья выражений, позволяя различным конструкциям выражений запросов выполняться на разных платформах. Вы также ознакомитесь с другими примерами того, что может делать LINQ.

Глава 12

LINQ за рамками коллекций

В этой главе...

- LINQ to SQL
- Интерфейс `IQueryable` и деревья выражений запросов
- LINQ to XML
- Parallel LINQ
- Проект Reactive Extensions for .NET
- Написание собственных операций

Предположим, что инопланетянин попросил вас описать свою *культуру*. Каким образом можно было бы охватить все многообразие человеческой культуры за короткий промежуток времени? Вы могли бы решить, что лучше потратить это время на то, чтобы *показать* ему культуру, а не описывать ее абстрактными понятиями: посетить новоорлеанский джаз-клуб, оперу Ла Скала, выставку картин в Лувре, пьесу Шекспира в Стратфорде-на-Эйвоне и т.д.

Знал ли бы инопланетянин человеческую культуру впоследствии? Смог ли бы он сочинить мелодию, написать книгу, танцевать балет, изваять скульптуру? Безусловно, нет. Но возможно у него осталось бы *восприятие* культуры — ее богатства и разнообразия, ее способности оживлять жизнь людей.

Так и с этой главой. Вы уже видели все средства C# 3, но без дополнительного материала по LINQ вы не имеете достаточного контекста, чтобы по-настоящему оценить их. На время публикации первого издания этой книги было доступно не особенно много технологий LINQ, но теперь их в избытке, как от Microsoft, так и от других производителей. Сам по себе этот факт меня не удивляет, но я восхищен тем, насколько различается *природа* этих технологий.

Мы рассмотрим различные пути, которыми LINQ проявляет себя, и ознакомимся с соответствующими примерами. Я решил демонстрировать главным образом технологии

от Microsoft, поскольку они являются наиболее типовыми. Однако из этого не стоит делать вывод, что продукты других производителей не приветствуются в экосистеме LINQ: имеется несколько проектов, и коммерческих, и с открытым кодом, которые предоставляют доступ к разнообразным источникам данных и строят дополнительные средства на основе существующих поставщиков.

В противовес остальным материалам этой книги здесь мы будем описывать все темы лишь поверхностно — цель не в том, чтобы изучить детали, а в том, чтобы погрузиться в *дух* LINQ. Для дельнейшего исследования любой из этих технологий я рекомендую обратиться к отдельным книгам по нужной теме или к соответствующей документации. Мне удалось преодолеть искушение приписать в конце каждого раздела "Однако по сравнению с этой технологией в LINQ для [того-то и того-то] предлагается гораздо больше средств", но помните об этом во время чтения. Каждая технология обладает многими возможностями помимо запрашивания, но в главе внимание будет уделяться только областям, которые напрямую связаны с LINQ.

Давайте начнем с поставщика, к которому было приковано наибольшее внимание, когда LINQ только появился: LINQ to SQL.

12.1. Запрашивание базы данных с помощью LINQ to SQL

Уверен, что к этому моменту вы уже правильно предположили, что LINQ to SQL преобразует выражения запросов в код SQL, который затем выполняется в базе данных. Более того, он является полноценным решением ORM, но внимание будет сосредоточено на той стороне LINQ to SQL, которая касается запросов, а не на обработке распараллеливания и других деталях, реализуемых ORM. Будет показано достаточно для того, чтобы было приступить к самостоятельному экспериментированию — база данных и код доступны на веб-сайтах книги (`http://csharpindepth.com`) и издательства. Для простоты работы с ней база данных имеет формат SQL Server 2005, хотя вполне очевидно, что в Microsoft позаботились о том, чтобы технология LINQ to SQL успешно функционировала также с более новыми версиями.

Почему LINQ to SQL, а не Entity Framework?

Увидев упоминание "новых версий", вас может удивить, по какой причине я выбрал для демонстрации LINQ to SQL, а не инфраструктуру Entity Framework, которая теперь является предпочтительным решением Microsoft (к тому же она также поддерживает LINQ). Причина связана единственно с простотой; инфраструктура Entity Framework во многих отношениях бесспорно мощнее, чем LINQ to SQL, однако она требует знания дополнительных концепций, объяснение которых заняло бы здесь слишком большой объем пространства. Я стараюсь дать вам представление о согласованности (а временами и о несогласованности), обеспечиваемой LINQ, и это применимо к LINQ to SQL, а также к Entity Framework.

Перед тем, как приступить к написанию запросов, необходима база данных и модель для ее представления в коде.

12.1.1. Начало работы: база данных и модель

Для базы данных в LINQ to SQL нужны метаданные, которые позволяют видеть соответствие между классами и таблицами базы данных, а также другую информацию. Существуют разнообразные способы представления этих метаданных, и в настоящем разделе мы будем пользоваться визуальным конструктором LINQ to SQL, встроенным в Visual Studio. Можно сначала спроектировать сущности и предложить LINQ создать базу данных или поступить наоборот – спроектировать базу данных и позволить Visual Studio выяснить, как должны выглядеть сущности. Лично я предпочитаю второй подход, но оба подхода обладают своими преимуществами и недостатками.

Создание схемы базы данных

Отображение классов из главы 11 на таблицы базы данных довольно прямолинейно. Каждая таблица имеет автоинкрементный целочисленный столбец идентификатора с подходящим именем: `ProjectID`, `DefectID` и т.д. Ссылки между таблицами просто используют такие же имена, так что таблица `Defect`, например, содержит столбец `ProjectID` с ограничением внешнего ключа.

Из этого набора несложных правил есть несколько исключений.

- `User` является зарезервированным словом в T-SQL, поэтому класс `User` отображается на таблицу `DefectUser`.
- Перечисления (состояние, уровень серьезности и тип пользователя) не имеют таблиц. Их значения отображаются на столбцы `tinyint` в таблицах `Defect` и `DefectUser`.
- Таблица `Defect` содержит две ссылки на таблицу `DefectUser`: одну для пользователя, зарегистрировавшего дефект, и одну для текущего назначенного исполнителя. Они представлены с помощью столбцов `CreatedByUserId` и `AssignedToUserId`, соответственно.

Создание сущностных классов

После того, как таблицы созданы, генерация сущностных классов в Visual Studio выполняется легко. Просто откройте окно **Server Explorer** (Проводник сервера), выбрав пункт меню **View⇨Server Explorer** (Вид⇨Проводник сервера), и добавьте источник данных для базы данных `SkeetySoftDefects` (щелкнув правой кнопкой мыши на элементе **Data Connections** (Подключения к данным) и выбрав в контекстном меню пункт **Add Connection** (Добавить подключение)). Вы должны видеть четыре таблицы: `Defect`, `DefectUser`, `Project` и `NotificationSubscription`.

Затем добавьте в проект новый элемент типа **LINQ to SQL classes** (Классы LINQ to SQL). Его имя будет основой для сгенерированного класса, представляющего общую модель базы данных; я применил имя `DefectModel`, которое приводит к созданию класса `DefectModelDataContext`. После создания этого нового элемента откроется визуальный конструктор. Затем можно перетащить четыре таблицы из окна **Server Explorer** на поверхность визуального конструктора, который выявит все ассоциации. После этого можно перепланировать диаграмму и подстроить различные свойства сущностных классов. Ниже приведен список внесенных мною изменений.

- Я переименовал свойство `DefectID` в `ID`, чтобы оно соответствовало предыдущей модели.
- Я переименовал свойство `DefectUser` в `User` (так что хотя таблица по-прежнему называется `DefectUser`, будет сгенерирован класс по имени `User`, как и ранее).

- Я изменил типы свойств `Severity`, `Status` и `UserType` на эквиваленты их перечислений (скопировав эти перечисления в проект).
- Я переименовал родительские и дочерние свойства, используемые для ассоциаций между классами `Defect` и `DefectUser` — для других ассоциаций визуальный конструктор вывел подходящие имена, но здесь потерпел неудачу, т.к. между одной парой таблиц оказалось две ассоциации. Эти отношения получили имена `AssignedTo/AssignedDefects` и `CreatedBy/CreatedDefects`.

На рис. 12.1 показана диаграмма классов в визуальном конструкторе после всех описанных изменений. Как видите, она выглядит очень похожей на диаграмму классов на рис. 11.3, но без перечислений.

Если вы заглянете в код C#, сгенерированный визуальным конструктором (`DefectModel.designer.cs`), то обнаружите пять частичных классов: по одному классу для каждой сущности и упомянутый ранее класс `DefectModelDataContext`. Тот факт, что они являются частичными, удобен; в этом случае я добавил дополнительные конструкторы, соответствующие конструкторам в исходных классах, которые создавали экземпляры в памяти. В результате код из главы 11, предназначенный для создания образца данных, можно использовать повторно без необходимости в большом объеме дополнительной работы. Ради краткости этот код здесь не приводится, но если вы просмотрите файл `PopulateDatabase.cs` в исходном коде, то легко найдете в нем все нужные сведения. Разумеется, запускать его необязательно — загружаемая база данных уже наполнена.

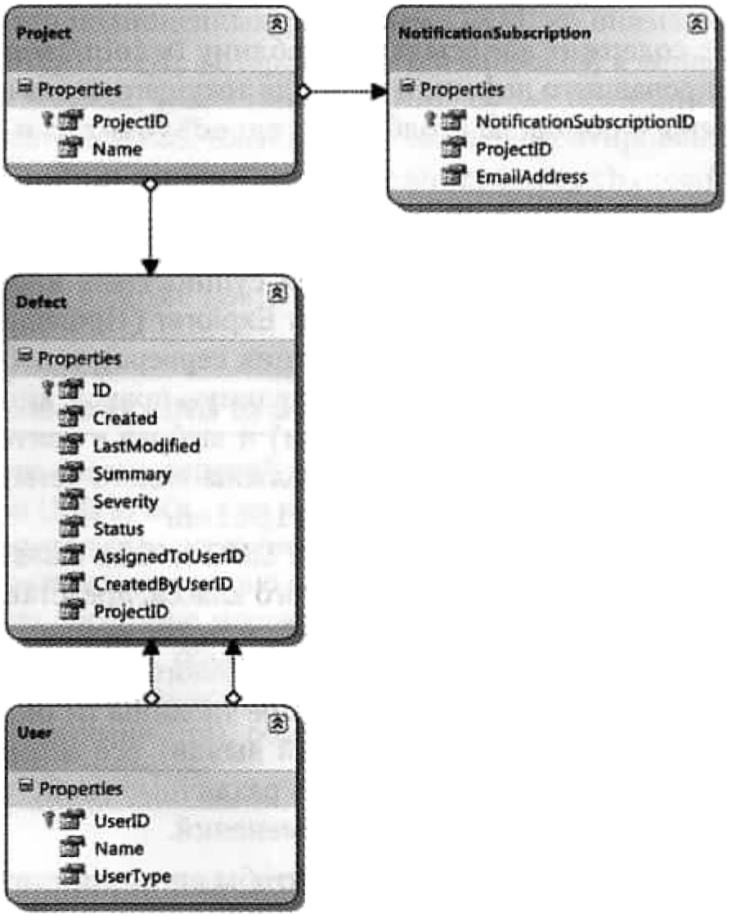

Рис. 12.1. Визуальный конструктор классов LINQ to SQL отображает переделанные и модифицированные сущности

Теперь, имея схему в SQL, сущностную модель в C# и образец данных, самое время приступать к запросам.

12.1.2. Начальные запросы

Наверняка вы уже догадались, что произойдет дальше, но я надеюсь, что это не сделает его менее впечатляющим. Мы будем запускать выражения запросов в отношении источника данных, на лету отслеживая преобразование запросов LINQ to SQL в код SQL. Мы будем применять знакомые запросы, которые в главе 11 выполнялись на коллекциях, расположенных в памяти.

Первый запрос: нахождение дефектов, назначенных для исправления Тиму

Тривиальные примеры, приводимые в начале главы 11, здесь рассматриваться не будут, а начнем мы сразу с запроса из листинга 11.7, который ищет открытые дефекты, назначенные для исправления Тиму (Tim). В целях сравнения ниже представлена часть листинга 11.7, касающаяся запроса:

```
User tim = SampleData.Users.TesterTim;

var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            select defect.Summary;
```

Полный эквивалент кода из листинга 11.7, реализованного с помощью LINQ to SQL, показан в листинге 12.1.

Листинг 12.1. Запрашивание базы данных для нахождения всех открытых дефектов, назначенных для исправления Тиму

```
using (var context = new DefectModelDataContext()) ←❶ Создание рабочего контекста
{
  context.Log = Console.Out; ←❷ Включение регистрации на консоли

  User tim = context.Users ←❸ Запрашивание базы данных с целью нахождения записи для Тима
                    .Where(user => user.Name == "Tim Trotter")
                    .Single();

  var query = from defect in context.Defects ←❹ Запрашивание базы данных с целью нахождения открытых дефектов, назначенных для исправления Тиму
              where defect.Status != Status.Closed
              where defect.AssignedTo == tim
              select defect.Summary;

  foreach (var summary in query)
  {
    Console.WriteLine(summary);
  }
}
```

Код в листинге 12.1 требует определенного объема объяснений, поскольку он целиком новый. Сначала создается новый *контекст данных*, с которым будет осуществляться работа ❶. Контексты данных достаточно многофункциональны и отвечают за управление подключениями и транзакциями, трансляцию запросов, отслеживание изменений в сущностях и обработку идентичности. В рамках этой главы контекст данных можно

рассматривать как точку соприкосновения с базой данных. Здесь не демонстрируются более сложные средства, а будет показана только одна удобная возможность: сообщение контексту данных о необходимости вывода на консоль всех команд SQL, которые он выполняет ❶. Все свойства, связанные с моделью, которые используются в коде внутри этого раздела (`Defects`, `Users` и т.д.), имеют тип `Table<T>` для соответствующего сущностного типа. Они действуют в качестве источников данных для запросов.

Применять `SampleData.Users.TesterTim` для идентификации Тима в главном запросе невозможно, т.к. этому объекту не известен идентификатор нужной строки в таблице `DefectUser`. Взамен используется отдельный запрос, загружающий пользовательскую сущность, которая представляет Тима ❶. Для этого применяется точечная нотация, но не менее успешно работало бы и выражение запроса. Метод `Single()` просто возвращает из запроса одиночный результат, генерируя исключение, если присутствует не точно один элемент. В реальной ситуации сущность может быть результатом выполнения других операций, таких как вход в систему, и если нет полной сущности, то может быть в наличии ее идентификатор, который в равной степени может использоваться в рамках главного запроса. В этом случае в качестве альтернативы можно было бы изменить запрос открытых дефектов для осуществления фильтрации на основе имени назначенного исполнителя. Тем не менее, это не соответствовало в должной мере духу исходного запроса.

Внутри выражения запроса ❶ единственное отличие между запросом в памяти и запросом связано с источником данных — вместо `SampleData.AllDefects` применяется свойство `context.Defects`. Окончательные результаты те же (хотя упорядочение не гарантируется), но работа выполнялась в отношении базы данных.

Поскольку контексту данных было указано на необходимость вывода на консоль сгенерированного кода SQL, можно точно наблюдать, что происходит во время выполнения кода. Консольный вывод отражает как запросы, выполняемые в базе данных, так и значения параметров запросов[1]:

```
SELECT [t0].[UserID], [t0].[Name], [t0].[UserType]
FROM [dbo].[DefectUser] AS [t0]
WHERE [t0].[Name] = @p0
-- @p0: Input String (Size = 11; Prec = 0; Scale = 0) [Tim Trotter]

SELECT [t0].[Summary]
FROM [dbo].[Defect] AS [t0]
WHERE ([t0].[AssignedToUserID] = @p0) AND ([t0].[Status] <> @p1)
-- @p0: Input Int32 (Size = 0; Prec = 0; Scale = 0) [2]
-- @p1: Input Int32 (Size = 0; Prec = 0; Scale = 0) [4]
```

Следует отметить, что первый запрос извлекает все свойства, описывающие пользователя, т.к. заполняется целая сущность, но второй запрос извлекает только сводку, поскольку только она и нужна. Кроме того, во втором запросе LINQ to SQL преобразует две отдельных конструкции `where` в единственный фильтр в базе данных.

Средства LINQ to SQL позволяют транслировать широкий спектр выражений. Давайте слегка усложним запрос из главы 11 и посмотрим, какой код SQL для него сгенерируется.

---

[1] Чтобы не отвлекаться от сути запросов SQL, сгенерированный дополнительный вывод, содержащий некоторые детали контекста данных, здесь не показан. Естественно, в консольном выводе также присутствуют сводки, выводимые циклом `foreach`.

Генерация кода SQL для более сложного запроса: конструкция `let`

Следующий запрос демонстрирует, что происходит, когда с помощью конструкции `let` вводится некий вид временной переменной. Если помните, в главе 11 мы обсуждали неестественную ситуацию — вообразили, что вычисление длины строки занимает длительное время. И снова рассматриваемое здесь выражение запроса в точности повторяет приведенное в листинге 11.11, но с одним исключением — в нем используется источник данных. В листинге 12.2 показан код LINQ to SQL.

Листинг 12.2. Применение конструкции `let` в LINQ to SQL

```
using (var context = new DefectModelDataContext())
{
  context.Log = Console.Out;

  var query = from user in context.Users
              let length = user.Name.Length
              orderby length
              select new { Name = user.Name, Length = length };

  foreach (var entry in query)
  {
    Console.WriteLine("{0}: {1}", entry.Length, entry.Name);
  }
}
```

Сгенерированный код SQL близок по духу к последовательностям, которые приводились на рис. 11.5. Самая внутренняя последовательность (первая на упомянутой диаграмме) — это список пользователей; она трансформируется в последовательность пар "имя/длина" (с помощью вложенной операции `select`) и затем применяется ничего не делающее проецирование с упорядочением по длине:

```
SELECT [t1].[Name], [t1].[value]
FROM (
  SELECT LEN([t0].[Name]) AS [value], [t0].[Name]
  FROM [dbo].[DefectUser] AS [t0]
  ) AS [t1]
ORDER BY [t1].[value]
```

Это хороший пример ситуации, когда сгенерированный код SQL оказывается многословнее, чем должен быть. Хотя ссылаться на элементы финальной выходной последовательности при выполнении упорядочения в выражении запроса нельзя, это можно делать в SQL. Следующий более простой запрос будет вполне нормально работать:

```
SELECT LEN([t0].[Name]) AS [value], [t0].[Name]
FROM [dbo].[DefectUser] AS [t0]
ORDER BY [value]
```

Разумеется, важно то, что именно делает оптимизатор запросов в базе данных — планы выполнения, отображаемые с помощью инструмента SQL Server Management Studio Express, для обоих запросов совпадают, поэтому не похоже, чтобы что-то терялось.

Последний набор запросов LINQ to SQL, который мы рассмотрим, задействует соединения.

12.1.3. Запросы, в которых задействованы соединения

Мы опробуем внутренние и групповые соединения, используя примеры с соединением подписок на уведомления и проектов. Подозреваю, что вы уже привыкли к практическим работам — для каждого запроса применяется один и тот же шаблон написания кода, поэтому в дальнейшем, если ничего особо интересного происходить не будет, я собираюсь показывать только выражение запроса и сгенерированный код SQL.

Явные соединения: сопоставление дефектов и подписок на уведомления

Первый запрос демонстрирует простейший вид соединения — внутреннее эквисоединение с использованием конструкции `join` языка LINQ:

```
// Выражение запроса (модификация выражения из листинга 11.12)
from defect in context.Defects
join subscription in context.NotificationSubscriptions
     on defect.Project equals subscription.Project
select new { defect.Summary, subscription.EmailAddress }
-- Сгенерированный код SQL
SELECT [t0].[Summary], [t1].[EmailAddress]
FROM [dbo].[Defect] AS [t0]
INNER JOIN [dbo].[NotificationSubscription] AS [t1]
ON [t0].[ProjectID] = [t1].[ProjectID]
```

Не удивительно, что в SQL применяется внутреннее соединение. В этом случае предугадать сгенерированный код SQL несложно. А как насчет группового соединения? Здесь все становится несколько более запутанным:

```
// Выражение запроса (модификация выражения из листинга 11.13)
from defect in context.Defects
join subscription in context.NotificationSubscriptions
     on defect.Project equals subscription.Project
     into groupedSubscriptions
select new { Defect = defect, Subscriptions = groupedSubscriptions }
-- Сгенерированный код SQL
SELECT [t0].[DefectID] AS [ID], [t0].[Created],
[t0].[LastModified], [t0].[Summary], [t0].[Severity],
[t0].[Status], [t0].[AssignedToUserID],
[t0].[CreatedByUserID], [t0].[ProjectID],
[t1].[NotificationSubscriptionID],
[t1].[ProjectID] AS [ProjectID2], [t1].[EmailAddress],
   (SELECT COUNT(*)
    FROM [dbo].[NotificationSubscription] AS [t2]
    WHERE [t0].[ProjectID] = [t2].[ProjectID]) AS [count]
FROM [dbo].[Defect] AS [t0]
LEFT OUTER JOIN [dbo].[NotificationSubscription] AS [t1]
ON [t0].[ProjectID] = [t1].[ProjectID]
ORDER BY [t0].[DefectID], [t1].[NotificationSubscriptionID]
```

Объем сгенерированного кода SQL значительно увеличился! В этом коде следует отметить два важных аспекта. Во-первых, вместо внутреннего соединения в нем используется *левое внешнее соединение*, так что вы будете по-прежнему видеть дефект, даже когда никто не подписался на уведомления об изменениях его проекта. Если требуется левое внешнее соединение, но без группирования, это удобно выразить с применением группового соединения и дополнительной конструкции `from`, использующей расширя-

ющий метод DefaultIfEmpty() на встроенной последовательности. Выглядит странно, но работает хорошо.

Во-вторых, еще один необычный момент в предшествующем запросе связан с тем, что он подсчитывает количество элементов для каждой группы внутри базы данных. Фактически это трюк, предпринятый LINQ to SQL, чтобы обеспечить возможность выполнения всей обработки на сервере. Бесхитростная реализация выполняла бы группирование в памяти после выборки всех результатов. В ряде случаев поставщик мог бы за счет трюков избегать необходимости в подсчете, просто отслеживая изменение идентификатора группы, но такой подход испытывает трудности с некоторыми запросами. Возможно, что будущая реализация LINQ to SQL будет способна менять образ действий в зависимости от точного запроса.

Чтобы увидеть соединение в коде SQL, записывать его явно в выражении запроса вовсе не обязательно. Запросы, которые рассматриваются последними, продемонстрируют неявное создание соединений посредством выражений доступа к свойствам.

Неявные соединения: отображение сводок по дефектам и названий проектов

Давайте обратимся к простому примеру. Предположим, что необходимо вывести список дефектов, отображая их сводки и названия проектов, к которым они относятся. Выражение запроса сводится всего лишь к проецированию:

```
// Выражение запроса
from defect in context.Defects
select new { defect.Summary, ProjectName = defect.Project.Name }
-- Сгенерированный код SQL
SELECT [t0].[Summary], [t1].[Name]
FROM [dbo].[Defect] AS [t0]
INNER JOIN [dbo].[Project] AS [t1]
ON [t1].[ProjectID] = [t0].[ProjectID]
```

Обратите внимание на навигацию от дефекта к проекту, выполненную с помощью свойства — LINQ to SQL преобразует такую навигацию во внутреннее соединение. Внутреннее соединение здесь может применяться из-за того, что схема имеет ограничение, не допускающее значений null, на столбце ProjectID таблицы Defect — каждый дефект связан с каким-то проектом. Тем не менее, не каждый дефект имеет назначенного исполнителя, потому что поле AssignedToUserID допускает значения null, поэтому если вместо этого использовать в проекции назначенного исполнителя, то сгенерируется левое внешнее соединение:

```
// Выражение запроса
from defect in context.Defects
select new { defect.Summary, Assignee = defect.AssignedTo.Name }
-- Сгенерированный код SQL
SELECT [t0].[Summary], [t1].[Name]
FROM [dbo].[Defect] AS [t0]
LEFT OUTER JOIN [dbo].[DefectUser] AS [t1]
ON [t1].[UserID] = [t0].[AssignedToUserID]
```

Конечно, если навигация осуществляется по большему числу свойств, соединения становятся все более и более сложными. Я не собираюсь вдаваться здесь в особые детали — важно знать только то, что для выяснения требуемого кода SQL механизм LINQ to SQL проводит глубокий анализ выражения запроса. Для выполнения такого анализа вполне очевидно, что должна быть возможность просмотра указанного запроса.

Давайте отложим в сторону специфику LINQ to SQL и подумаем в общих терминах о том, что должны делать поставщики LINQ такого рода. Это будет применимо к любому поставщику, которому необходимо анализировать запрос, а не просто предоставлять делегат. В конце концов, самое время разобраться, по какой причине были добавлены деревья выражений в качестве средства в C# 3.

12.2. Трансляция с использованием IQueryable и IQueryProvider

В настоящем разделе мы рассмотрим основы того, как LINQ to SQL преобразует выражения запросов в код SQL. Это отправная точка для реализации собственного поставщика LINQ, которую придется учитывать. (Ни в коем случае не следует недооценивать сопровождающие данную процедуру технические сложности; но если вам нравится заниматься сложными задачами, то реализация поставщика LINQ непременно будет интересной.) Этот раздел главы имеет в большей степени теоретический характер, однако полезно иметь определенное представление о том, каким образом в LINQ принимается решение об использовании обработки внутри памяти, в базе данных или в каком-то другом механизме запросов.

Во всех выражениях запросов, с которыми мы сталкивались в LINQ to SQL, источником был `Table<T>`. Но если взглянуть на `Table<T>`, можно заметить, что он не имеет ни метода `Where()`, ни `Select()`, ни `Join()`, ни любой другой стандартной операции запроса. Взамен применяется тот же трюк, который предпринимается в LINQ to Objects. Подобно тому, как источник в LINQ to Objects всегда реализует интерфейс `IEnumerable<T>` (возможно, после вызова `Cast()` или `OfType()`) и затем использует расширяющие методы из класса `Enumerable`, так и `Table<T>` реализует интерфейс `IQueryable<T>` и позже применяет расширяющие методы из класса `Queryable`. Вы увидите, что LINQ строит дерево выражения и затем позволяет поставщику выполнить его в надлежащее время. Давайте начнем с рассмотрения состава `IQueryable<T>`.

12.2.1. Введение в IQueryable<T> и связанные интерфейсы

Если вы заглянете в документацию по интерфейсу `IQueryable<T>` и посмотрите, какие члены он содержит непосредственно (а не наследует), то можете быть разочарованы. Их попросту нет. Вместо этого данный интерфейс унаследован от `IEnumerable<T>` и необобщенного `IQueryable`, а те, в свою очередь, унаследованы от необобщенного `IEnumerable`. Итак, `IQueryable` — это место, где находятся новые члены, правильно? Да, почти. На самом деле интерфейс `IQueryable` имеет только три свойства: `QueryProvider`, `ElementType` и `Expression`. Типом свойства `QueryProvider` является `IQueryProvider` — еще один интерфейс, который нужно рассмотреть.

Пока неясно? Возможно, пониманию поспособствует рис. 12.2, на котором показана диаграмма напрямую задействованных интерфейсов.

Проще всего воспринимать интерфейс `IQueryable` как представляющий запрос, который будет выдавать последовательность результатов, когда он запущен на выполнение. Детали запроса в терминах LINQ хранятся в дереве выражения, которое возвращается свойством `Expression` интерфейса `IQueryable`. Запрос выполняется путем прохода по `IQueryable` (другими словами, вызовом метода `GetEnumerator()` и затем метода `MoveNext()` на полученном результате) или за счет вызова метода `Execute()` на `IQueryProvider` с передачей ему дерева выражения.

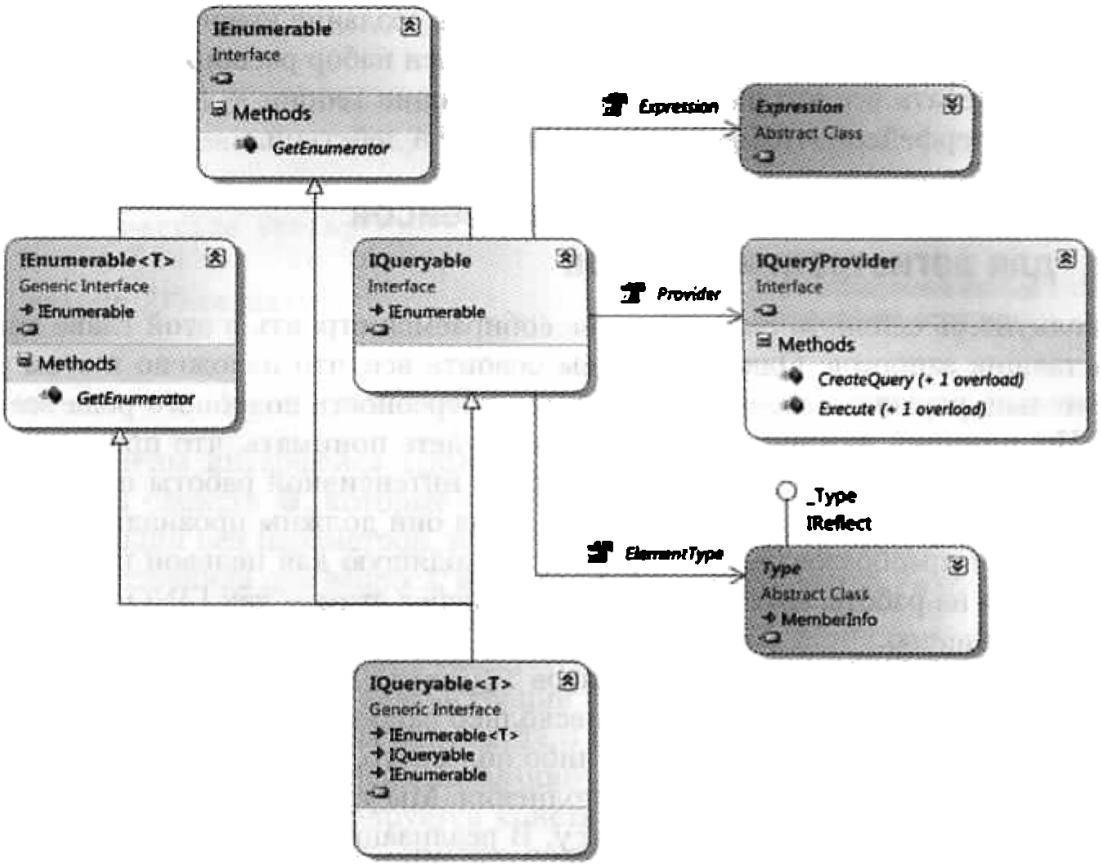

Рис. 12.2. Диаграмма интерфейсов, задействованных в `IQueryable<T>`

Теперь, когда у вас есть хотя бы некоторое понимание, для чего предназначен интерфейс `IQueryable`, возникает вопрос: что собой представляет `IQueryProvider`? С запросом можно выполнять больше действий, чем просто запускать его; его можно использовать для построения более объемного запроса, для чего предназначены стандартные операции запросов в LINQ[2]. Для построения запроса необходимо применять метод `CreateQuery()` на подходящем `IQueryProvider`[3].

Воспринимайте источник данных как простой запрос (например, `SELECT * FROM SomeTable` в SQL) — вызов `Where()`, `Select()`, `OrderBy()` и аналогичных методов дает в результате другой запрос, основанный на первом. Имея любой запрос `IQueryable`, создать новый запрос можно за счет выполнения следующих шагов.

1. Получить у существующего запроса его дерево выражения (с помощью свойства `Expression`).
2. Построить новое дерево выражения, которое содержит исходное выражение и нужную дополнительную функциональность (скажем, фильтр, проекция или упорядочение).
3. Получить у существующего запроса его поставщик (через свойство `Provider`).
4. Вызвать метод `CreateQuery()` на поставщике, передав ему новое дерево выражения.

---

[2] Точнее, те из них, которые сохраняют отложенное выполнение, вроде `Where()` и `Join()`. Вскоре вы увидите, что произойдет с операциями агрегирования, такими как `Count()`.

[3] Методы `Execute()` и `CreateQuery()` имеют обобщенные и необобщенные перегруженные версии. Необобщенные версии делают более простым создание запросов динамически в коде. В выражениях запросов этапа компиляции используется обобщенная версия.

Из приведенных шагов сложным является лишь создание нового дерева выражения. К счастью, в статическом классе `Queryable` доступен набор расширяющих методов, которые могут сделать все это вместо вас. И достаточно теории — давайте приступим к реализации интерфейсов, чтобы увидеть сказанное в действии.

12.2.2. Имитация: реализация интерфейсов для регистрации вызовов

Не волнуйтесь слишком сильно, мы не собираемся строить в этой главе полноценный поставщик запросов. Однако если вы освоите все, что изложено в этом разделе, то значительно улучшите свои позиции, когда потребность подобного рода все же возникнет. Но, пожалуй, важнее всего то, что вы будете понимать, что происходит, когда выдаются запросы LINQ to SQL. Большая часть интенсивной работы поставщиков запросов производится в момент выполнения, когда они должны проанализировать дерево выражения и преобразовать его в форму, подходящую для целевой платформы. Мы сосредоточимся на работе, которая выполняется перед этим — как LINQ подготавливает запрос к выполнению.

Мы взглянем на реализации интерфейсов `IQueryable` и `IQueryProvider`, а затем попробуем запустить в отношении них несколько запросов. Самым интересным будет не результат, т.к. запросы не делают чего-либо полезного, а последовательность вызовов, совершенных вплоть до данной точки выполнения. Мы уделим основное внимание двум типам: `FakeQueryProvider` и `FakeQuery`. В реализации каждого метода интерфейса на консоль выводится текущее задействованное выражение с применением простого метода регистрации (здесь не показанного).

В листинге 12.3 представлен код класса `FakeQuery`.

Листинг 12.3. Простая реализация интерфейса `IQueryable`, регистрирующая вызовы методов

```
class FakeQuery<T> : IQueryable<T>
{
  public Expression Expression { get; private set; }   <-- 1 Объявление простых автоматических свойств
  public IQueryProvider Provider { get; private set; }
  public Type ElementType { get; private set; }

  internal FakeQuery(IQueryProvider provider,
                     Expression expression)
  {
    Expression = expression;
    Provider = provider;
    ElementType = typeof(T);
  }

  internal FakeQuery() : this(new FakeQueryProvider(), null)
  {
    Expression = Expression.Constant(this);   <-- 2 Использование этого запроса в качестве начального выражения
  }

  public IEnumerator<T> GetEnumerator()
  {
    Logger.Log(this, Expression);
    return Enumerable.Empty<T>().GetEnumerator();
  }
```

```
    IEnumerator IEnumerable.GetEnumerator()
    {
      Logger.Log(this, Expression);
      return Enumerable.Empty<T>().GetEnumerator();
    }
    public override string ToString()
    {
      return "FakeQuery";
    }
  }
```

❹ Возвращение пустой результирующей последовательности

❺ Переопределение метода `ToString()` для целей регистрации

Свойства-члены интерфейса `IQueryable` реализованы в `FakeQuery` с помощью автоматических свойств ❶, которые устанавливаются в конструкторах. Доступны два конструктора: один без параметров, используемый главной программой с целью создания простого источника для запроса, и один, который вызывается `FakeQueryProvider` с передачей текущего выражения запроса.

Применение `Expression.Constant(this)` в качестве начального исходного выражения ❷ — это просто способ демонстрации того, что запрос изначально представляет исходный объект. (Для примера подумайте о реализации, представляющей таблицу; до тех пор, пока не будет применена какая-нибудь операция запроса, запрос возвращает целую таблицу.) Когда регистрируется константное выражение, оно использует переопределенный метод `ToString()`; именно по этой причине было выбрано короткое строковое описание ❶. Благодаря такому переопределению, окончательное выражение будет намного более ясным. С целью упрощения при проходе по результатам запроса всегда возвращается пустая последовательность ❹. Производственные реализации в этом случае анализировали бы выражение или (более вероятно) вызывали бы метод `Execute()` своего поставщика запросов и возвращали результат.

Как видите, в классе `FakeQuery` происходит не так уж много действий. В листинге 12.4 представлен класс `FakeQueryProvider`, который тоже прост.

Листинг 12.4. Реализация интерфейса `IQueryProvider`, которая использует `FakeQuery`

```
class FakeQueryProvider : IQueryProvider
{
  public IQueryable<T> CreateQuery<T>(Expression expression)
  {
    Logger.Log(this, expression);
    return new FakeQuery<T>(this, expression);
  }
  public IQueryable CreateQuery(Expression expression)
  {
    Type queryType = typeof(FakeQuery<>).MakeGenericType(expression.Type);
    object[] constructorArgs = new object[] { this, expression };
    return (IQueryable)Activator.CreateInstance(queryType, constructorArgs);
  }
  public T Execute<T>(Expression expression)
  {
    Logger.Log(this, expression);
    return default(T);
  }
```

```
    public object Execute(Expression expression)
    {
      Logger.Log(this, expression);
      return null;
    }
}
```

О реализации FakeQueryProvider можно сказать даже меньше, чем о FakeQuery<T>. Методы CreateQuery() не выполняют реальной обработки, а действуют как фабричные методы для запроса. Единственная сложность заключается в том, что необобщенной перегруженной версии по-прежнему нужно предоставлять правильный аргумент типа для FakeQuery<T> на основе свойства Type конкретного выражения. Перегруженные версии метода Execute() после регистрации вызова возвращают пустые результаты. Именно в них *обычно* проводится интенсивный анализ наряду с действительным обращением к веб-службе, базе данных или другой целевой платформе.

Хотя никакой реальной работы не было сделано, интересные вещи начинают происходить, когда класс FakeQuery применяется в качестве источника в выражении запроса. Я уже упоминал о наличии возможности писать выражения запросов без явной реализации методов для обработки стандартных операций запросов: имеются в виду расширяющие методы — на этот раз из класса Queryable.

12.2.3. Интеграция выражений: расширяющие методы из класса Queryable

Подобно тому, как тип Enumerable содержит расширяющие методы интерфейса IEnumerable<T> для реализации стандартных операций запросов LINQ, тип Queryable располагает расширяющими методами интерфейса IQueryable<T>. Существуют два крупных отличия между реализациями в Enumerable и в Queryable.

Первое отличие в том, что все методы Enumerable используют в качестве своих параметров делегаты — например, метод Select() принимает Func<TSource,TResult>. Это удобно для манипуляций в памяти, но для поставщиков LINQ, которые выполняют запросы где-то в других местах, необходим формат, который можно исследовать более тщательно — деревья выражений.

К примеру, соответствующая перегруженная версия метода Select() в Queryable принимает параметр типа Expression<Func<TSource,TResult>>. Компилятор совершенно не против этого — после трансляции запроса получается лямбда-выражение, которое должно быть передано в виде аргумента методу, и лямбда-выражения могут быть преобразованы либо в экземпляры делегатов, либо в деревья выражений.

Вот почему возможна настолько гладкая работа LINQ to SQL. Все четыре задействованных ключевых элемента являются новыми средствами C# 3: лямбда-выражения, трансляция выражений запросов в нормальные выражения, которые *используют* лямбда-выражения, расширяющие методы и деревья выражений. Не было бы их всех четырех, возникли бы проблемы. К примеру, если бы выражения запросов всегда транслировались в делегаты, они не могли бы применяться с поставщиком наподобие LINQ to SQL, который требует деревья выражений. На рис. 12.3 показаны два возможных пути следования выражений запросов; они отличаются только в том, какие интерфейсы реализуют их источник данных.

На рис. 12.3 видно, что ранние этапы процесса компиляции не зависят от источника данных. Применяется одно и то же выражение запроса, которое транслируется одинаковым образом.

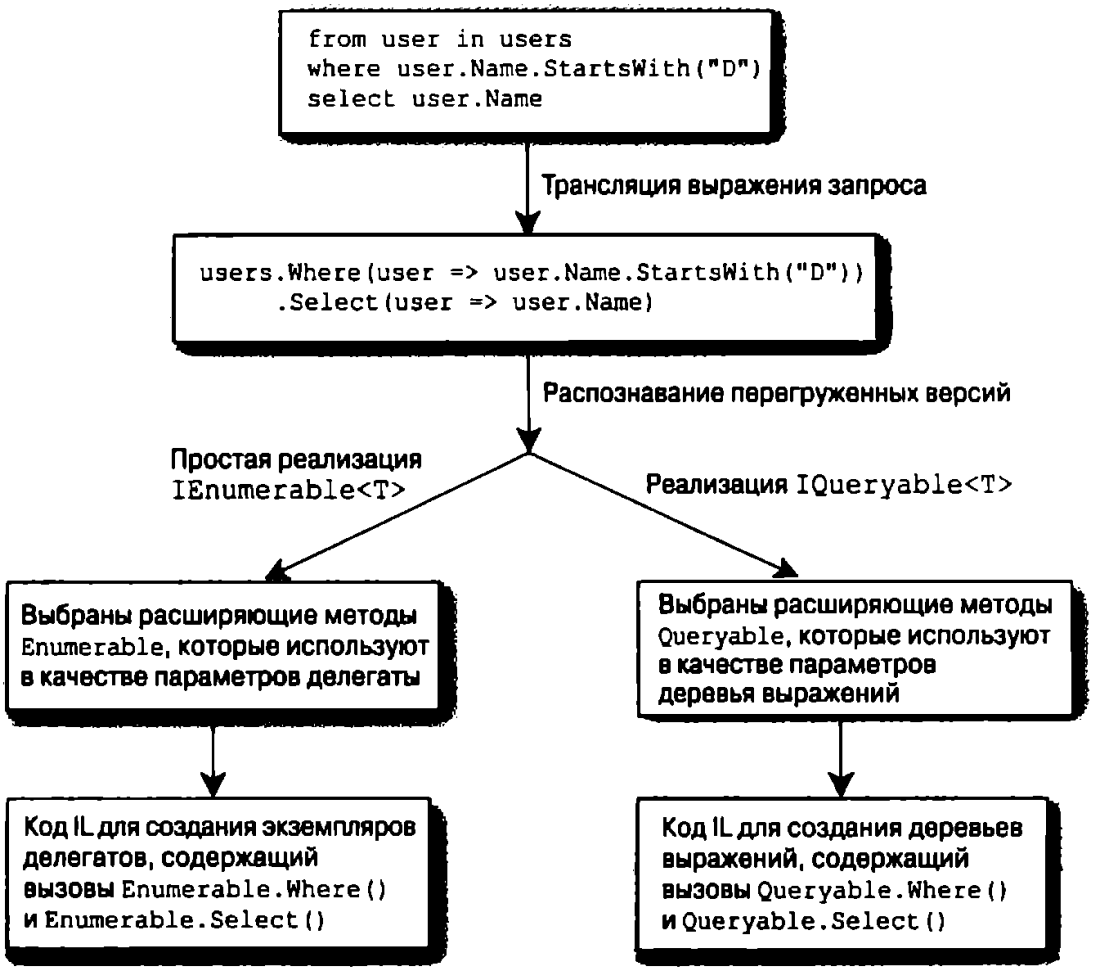

Рис. 12.3. Запрос выбирает один из двух путей в зависимости от того, какие интерфейсы реализует источник данных — `IQueryable` или только `IEnumerable`

Источник данных становится действительно важным, только когда компилятор просматривает транслированный запрос с целью нахождения подходящих для использования методов `Select()` и `Where()`. В этот момент лямбда-выражения могут быть преобразованы либо в экземпляры делегатов, либо в деревья выражений, потенциально давая совершенно разные реализации: обычно внутри памяти для пути слева и код SQL, выполняющийся в базе данных, для пути справа.

Просто чтобы было ясно: решение относительно выбора `Enumerable` или `Queryable`, показанное на рис. 12.3, не имеет явной поддержки в компиляторе C#. Как вы увидите при ознакомлении с Parallel LINQ и Reactive LINQ, это не единственные два возможных пути. Можно создать собственный интерфейс и реализовать расширяющие методы, следуя шаблону запросов, или даже создать тип с подходящими методами экземпляра.

Второе крупное отличие между `Enumerable` и `Queryable` заключается в том, что расширяющие методы `Enumerable` выполняют действительную работу, связанную с соответствующей операцией запроса (или, по крайней мере, строят итераторы, которые делают эту работу). Например, в методе `Enumerable.Where()` предусмотрен код для выполнения указанного фильтра и выдачи только подходящих элементов в качестве результирующей последовательности. По контрасту с этим реализации операций запросов в `Queryable` делают немногое: они лишь создают новый запрос на основе параметров или вызывают метод `Execute()` поставщика запросов, как было описано в конце раздела 12.2.1. Другими словами, эти расширяющие методы применяются только для постро-

ния запросов и требования их выполнения — они не содержат логику, лежащую в основе операций. Это означает, что такие расширяющие методы подходят для любого поставщика LINQ, который использует деревья выражений, но сами по себе они бесполезны. Они являются связующими элементами между вашим кодом и деталями поставщика.

Имея доступные расширяющие методы `Queryable`, а также готовые к применению реализации `IQueryable` и `IQueryProvider`, можно, наконец, посмотреть на то, что происходит, когда выражение запроса используется со специальным поставщиком.

12.2.4. Имитированный поставщик запросов в действии

В листинге 12.5 приведено простое выражение запроса, которое (предположительно) находит в имитированном источнике все строки, начинающиеся с `abc`, и проецирует результаты в последовательность, перечисляющую длины подходящих строк. Проход по результатам выполняется, но безо всяких действий с ними, т.к. уже известно, что они пусты. Причина связана с отсутствием исходных данных и какого-либо кода для реальной фильтрации — вы просто регистрируете вызовы методов, которые делает LINQ в ходе создания выражения запроса, и проходите по результатам.

Листинг 12.5. Простое выражение запроса, использующее классы имитированного запроса

```
var query = from x in new FakeQuery<string>()
            where x.StartsWith("abc")
            select x.Length;
foreach (int i in query) { }
```

Какими, по вашему мнению, должны быть результаты запуска кода из листинга 12.5? В частности, что должно быть зарегистрировано *последним*, в точке, когда обычно ожидается выполнение какой-то реальной работы с деревом выражения? Ниже показаны результаты, слегка переформатированные для большей наглядности:

```
FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))

FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)

FakeQuery<Int32>.GetEnumerator
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)
```

Здесь следует отметить два важных момента: метод `GetEnumerator()` вызывается только в конце, а не на промежуточных запросах; ко времени вызова `GetEnumerator()` в наличии вся информация, представленная в исходном выражении запроса. Вы не обязаны вручную отслеживать начальные части выражения на каждом шаге — единственное дерево выражения содержит всю необходимую информацию.

Кстати, не позволяйте столь лаконичному выводу вводить вас в заблуждение — действительное дерево выражения будет глубоким и сложным, особенно из-за конструкции `where`, включающей дополнительный вызов метода. Такое дерево выражения является именно тем, что LINQ to SQL будет исследовать с целью выяснения, какой запрос выполнять. Поставщики LINQ *могли бы* строить собственные запросы (в любой необходимой им форме), когда совершены вызовы `CreateQuery()`, но обычно проще просматривать финальное дерево, когда вызывается `GetEnumerator()`, поскольку вся нужная информация доступна в одном месте.

Последним вызовом, зарегистрированным запросом из листинга 12.5, был `FakeQuery.GetEnumerator()`, и вас может заинтересовать, почему в `IQueryProvider` *также* необходим метод `Execute()`. Дело в том, что не все выражения запросов генерируют последовательности. Если вы применяете операцию агрегирования, такую как `Sum()`, `Count()` или `Average()`, то в действительности больше не создаете источник, а немедленно вычисляете результат. Именно тогда и вызывается метод `Execute()`, как продемонстрирует вывод запроса из листинга 12.6.

Листинг 12.6. Вызов метода `IQueryProvider.Execute()`

```
var query = from x in new FakeQuery<string>()
            where x.StartsWith("abc")
            select x.Length;
double mean = query.Average();

// Вывод
FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)
FakeQueryProvider.Execute
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)
                    .Average()
```

Класс `FakeQueryProvider` может оказаться довольно удобным, когда нужно понять, что компилятор C# делает "за кулисами" с выражениями запросов. Он отобразит прозрачные идентификаторы, введенные внутри выражения запроса, транслированные вызовы методов `SelectMany()`, `GroupJoin()` и т.д.

12.2.5. Итоги по интерфейсу `IQueryable`

Пока еще не было написано сколько-нибудь значимого кода, который бы требовался в реальном поставщике запросов для выполнения полезной работы, но надо надеяться, что рассмотренный имитированный поставщик дал представление о том, как поставщики LINQ получают необходимую информацию из выражений запросов. Все это построено на основе расширяющих методов `Queryable` с учетом подходящей реализации `IQueryable` и `IQueryProvider`.

В этом разделе мы погружались в несколько большие подробности, чем в остальных разделах главы, т.к. пришлось иметь дело с основами, поддерживающими код LINQ to SQL, который был показан ранее. Хотя вам, скорее всего, не понадобится реализовать интерфейсы запросов самостоятельно, задействованные шаги по получению выражения запроса на C# и (во время выполнения) запуску некоторого кода SQL в базе данных довольно глубоки и лежат в основе важных средств C# 3. Понимание причин добавления этих средств в C# способствует лучшему владению этим языком программирования.

На этом исследование использования деревьев выражений в LINQ завершено. Остаток главы посвящен внутрипроцессным запросам, применяющим делегаты, но вы увидите, что использование LINQ по-прежнему допускает большое разнообразие и инновационные подходы. Нашей следующей гаванью будет LINQ to XML, который является "просто" API-интерфейсом для работы с XML, спроектированным в целях хорошей интеграции с LINQ to Objects.

12.3. API-интерфейсы, дружественные к LINQ, и LINQ to XML

Вне всяких сомнений LINQ to XML – это наиболее приятный API-интерфейс для XML из числа тех, с которыми мне приходилось сталкиваться. Работаете ли вы с существующим XML-документом, генерируете новый документ или делаете то и другое, этот API-интерфейс прост в применении и понимании. Часть этого совершенно не зависит от LINQ, но большинство функций ожидаемо хорошо взаимодействует с LINQ. Как и в разделе 12.1, я предоставлю такой объем вводной информации, которого достаточно для понимания примеров, после чего вы увидите, как LINQ to XML смешивает свои операции запросов с такими операциями в LINQ to Objects. К концу этого раздела у вас могут появиться идеи о том, каким образом привести собственные API-интерфейсы в гармонию с инфраструктурой.

12.3.1. Основные типы в LINQ to XML

Технология LINQ to XML сосредоточена в сборке `System.Xml.Linq`, а большинство типов находятся в пространстве имен `System.Xml.Linq`[4]. Почти все типы в этом пространстве имен имеют префикс `X`, поэтому с учетом того, что обычный API-интерфейс DOM содержит тип `XmlElement`, его эквивалентом в LINQ to XML будет `XElement`. Это упрощает выяснение того факта, что в коде используется LINQ to XML, даже если вы непосредственно не знакомы с задействованным типом. На рис. 12.4 приведена диаграмма с типами, которые будут применяться наиболее часто.

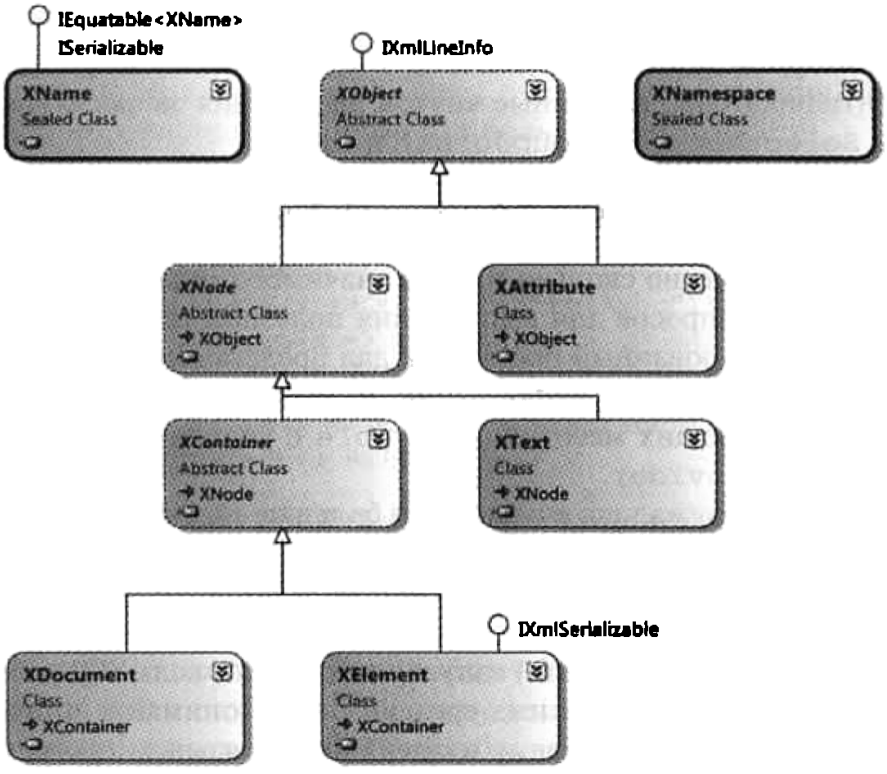

Рис. 12.4. Диаграмма классов для LINQ to XML, отражающая наиболее часто используемые типы

[4] Я регулярно забываю, как выглядит название этого пространства имен – `System.Xml.Linq` или `System.Linq.Xml`. Если вы запомните, что в первую очередь он является API-интерфейсом для XML, то все должно быть в порядке.

Ниже представлен краткий обзор показанных на рис. 12.4 типов.

- Тип `XName` используется для представления имен элементов и атрибутов. Его экземпляры обычно создаются с применением неявного преобразования из строки (в этом случае пространство имен не используется) или через перегруженную операцию `+(XNamespace, string)`.
- Тип `XNamespace` представляет пространство имен XML — в сущности, URI-указатель. Его экземпляры обычно создаются посредством неявного преобразования из строки.
- Тип `XObject` — это общий предок для `XNode` и `XAttribute`; в отличие от API-интерфейса DOM, в LINQ to XML атрибут не является узлом. Например, методы, возвращающие дочерние узлы, не включают атрибуты.
- Тип `XNode` представляет узел в дереве XML. В нем определены разнообразные члены, предназначенные для манипулирования и выдачи запросов к дереву. Доступно несколько других производных от `XNode` классов, которые не были показаны на рис. 12.4, наподобие `XComment` и `XDeclaration`. Они применяются относительно редко — самыми распространенными типами узлов являются документы, элементы и текст.
- Тип `XAttribute` — это атрибут с именем и значением. Значением, по сути, является текст, но предусмотрены явные преобразования во множество других типов, таких как `int` и `DateTime`.
- Тип `XContainer` представляет узел в дереве XML, который может иметь дочернее содержимое — по существу это элемент или документ.
- Тип `XText` — это текстовый узел, и для представления текстовых узлов CDATA используется дополнительный производный тип `XCData`. (Узел CDATA является грубым эквивалентом дословного строкового литерала, в котором требуется меньшее количество отмен.) Экземпляры типа `XText` редко создаются напрямую в пользовательском коде; вместо этого, когда в качестве содержимого элемента или документа используется строка, она преобразуется в экземпляр `XText`.
- Тип `XElement` представляет элемент. Наряду с `XAttribute` он является наиболее часто используемым классом в LINQ to XML. В отличие от API-интерфейса DOM, экземпляр `XElement` можно создавать без необходимости в создании содержащего его документа. Если отсутствует реальная необходимость в наличии объекта документа (возможно, для специального объявления XML), часто достаточно применять только элементы.
- Тип `XDocument` — это документ. Доступ к его корневому элементу осуществляется с использованием свойства `Root`, которое является эквивалентом `XmlDocument.DocumentElement`. Как отмечалось ранее, часто документ не требуется.

В рамках документной модели доступно даже большее число типов, и существует несколько других типов для таких возможностей, как опции загрузки и сохранения, но этот список отражает только самые важные средства. Из перечисленных выше типов придется регулярно явно ссылаться только на `XElement` и `XAttribute`. Если вы работаете с пространствами имен, то будете также использовать `XNamespace`, но в основном большинство оставшихся типов можно проигнорировать. Просто удивительно, насколько много можно делать с таким небольшим количеством типов.

Раз уж речь зашла об удивительном, не могу удержаться, чтобы не продемонстрировать вам, как работает поддержка пространств имен в LINQ to XML. Мы не собираемся применять пространства имен где-либо еще, но это удачный пример того, как правильно спроектированный набор преобразований может значительно упростить дело. Это также облегчит освоение следующей темы: конструирование элементов.

Если необходимо только указать имя элемента или атрибута без пространства имен, можно использовать строку. Тем не менее, вы не обнаружите в каких-либо типах конструкторы с параметрами `string` — все они принимают `XName`. Существует неявное преобразование из `string` в `XName`, а также из `string` в `XNamespace`. Объединение пространства имен и строки также дает `XName`. Граница между некорректным и удачным применением операции чрезвычайно тонка, но в данном случае LINQ to XML действительно воплощает ее в жизнь. Ниже показан код для создания двух элементов — одного внутри пространства имен, а другого нет:

```
XElement noNamespace = new XElement("no-namespace");
XNamespace ns = "http://csharpindepth.com/sample/namespace";
XElement withNamespace = new XElement(ns + "in-namespace");
```

Это делает код более читабельным, даже когда задействованы пространства имен, что поступило как долгожданное облегчение из ряда других API-интерфейсов. Но мы лишь создали два пустых элемента. Как предоставить им какое-то содержимое?

12.3.2. Декларативное конструирование

Обычно в рамках API-интерфейса DOM сначала создается элемент, а затем к нему добавляется содержимое. В LINQ to XML это можно делать через метод `Add()`, унаследованный от `XContainer`, но такой способ не является идиоматическим для технологии LINQ to XML[5]. Тем не менее, взглянуть на сигнатуру метода `XContainer.Add()` все же полезно, т.к. он вводит понятие модели содержимого. Возможно, вы ожидали увидеть сигнатуру `Add(XNode)` или `Add(XObject)`, однако она имеет вид `Add(object)`. Тот же самый шаблон используется для сигнатур конструкторов `XElement` (и `XDocument`). Во всех конструкторах `XElement` предусмотрен один параметр для имени элемента, но затем можно ничего не указывать (для создания пустого элемента), указать одиночный объект (для создания элемента с одним дочерним узлом) или задать массив объектов для создания множества дочерних узлов. В случае с множеством дочерних узлов применяется массив параметров (ключевое слово `params` в C#), т.е. компилятор создаст массив самостоятельно — вам нужно лишь предоставить список параметров. Использование простого типа `object` для типа содержимого может выглядеть странным, но на самом деле это исключительно удобно. При добавлении содержимого — либо с помощью конструктора, либо через метод `Add()` — необходимо учитывать следующие моменты.

- Ссылки `null` игнорируются.
- Экземпляры `XNode` и `XAttribute` добавляются в относительно прямолинейной манере; они клонируются, если уже имеют родительские экземпляры, но в противном случае никакие преобразования не требуются. (При этом выполняется ряд других проверок корректности, таких как выявление возможных дублированных атрибутов в одном и том же элементе.)

---

[5] В некотором смысле неприятно, что тип `XElement` не реализует интерфейс `IEnumerable`, поскольку в противном случае возможен был бы еще один подход к конструированию, предусматривающий использование инициализаторов коллекций. Тем не менее, конструкторы и без этого работают довольно искусно.

- Строки, числа, даты, время и тому подобные значения добавляются за счет их преобразования в узлы `XText` с применением стандартного форматирования XML.
- Если аргумент реализует интерфейс `IEnumerable` (и он не перекрывается чем-то еще), метод `Add()` будет проходить по содержимому аргумента и добавлять каждое значение по очереди, используя рекурсию там, где это необходимо.
- Все, что не подразумевает специальную обработку, преобразуется в текст с помощью простого вызова `ToString()`.

Это означает, что содержимое часто не приходится специальным образом готовить, прежде чем его можно будет добавить в элемент — все необходимое сделает LINQ to XML. Детали явно документированы, поэтому переживать о том, что это сверхъестественно, не нужно — все действительно работает.

Конструирование вложенных элементов приводит к коду, который естественным образом воспроизводит иерархическую структуру дерева. Это лучше продемонстрировать на примере. Ниже показан фрагмент кода LINQ to XML:

```
new XElement("root",
  new XElement("child",
    new XElement("grandchild", "text")),
  new XElement("other-child"));
```

А вот разметка XML созданного элемента — обратите внимание на визуальное сходство между кодом и выводом:

```
<root>
  <child>
    <grandchild>text</grandchild>
  </child>
  <other-child />
</root>
```

Пока все идет хорошо, но в приведенном выше списке важной частью является четвертый пункт, упоминающий о рекурсивной обработке, поскольку это позволяет строить структуру XML из запроса LINQ естественным образом. Например, на веб-сайте книги доступен код для генерации RSS-ленты из базы данных. Оператор для конструирования XML-документа занимает 28 строк, что обычно я счел бы отвратительным, однако он удивительно удобен для чтения[6]. Этот оператор содержит два запроса LINQ — один для заполнения значения атрибута и один для предоставления последовательности элементов, каждый из которых представляет элемент новостей. Читая такой код, становится очевидным, как будет выглядеть результирующая разметка XML.

Для большей конкретики давайте возьмем два простых примера из системы отслеживания дефектов. При этом будет применяться образец данных LINQ to Objects, но вы можете воспользоваться почти идентичными запросами для работы с другим поставщиком LINQ. Сначала необходимо построить элемент, содержащий всех пользователей в системе. Для этого понадобится просто проекция, потому в листинге 12.7 применяется точечная нотация.

---

[6] Одним фактором, содействующим читабельности, является расширяющий метод, который я создал для преобразования анонимных типов в элементы, используя свойства для дочерних элементов. Если вам интересно, то код свободно доступен как часть моего проекта `MiscUtil` (`http://mng.bz/xDMt`). Он помогает, только когда необходимая структура XML вписывается в определенный шаблон, но в рассматриваемом случае этот код может значительно сократить нагромождение вызовов конструкторов `XElement`.

Листинг 12.7. Создание элементов из пользователей системы

```
var users = new XElement("users",
    SampleData.AllUsers.Select(user => new XElement("user",
      new XAttribute("name", user.Name),
      new XAttribute("type", user.UserType)))
);
Console.WriteLine(users);

// Вывод
<users>
  <user name="Tim Trotter" type="Tester" />
  <user name="Tara Tutu" type="Tester" />
  <user name="Deborah Denton" type="Developer" />
  <user name="Darren Dahlia" type="Developer" />
  <user name="Mary Malcop" type="Manager" />
  <user name="Colin Carton" type="Customer" />
</users>
```

Если вы хотите сделать несколько более сложный запрос, возможно, стоит воспользоваться выражением запроса. В листинге 12.8 создается еще один список пользователей, но на этот раз он включает только разработчиков из SkeetySoft. Ради небольшого разнообразия имя каждого разработчика делается текстовым узлом внутри элемента, а не значением атрибута.

Листинг 12.8. Создание элементов с текстовыми узлами

```
var developers = new XElement("developers",
   from user in SampleData.AllUsers
   where user.UserType == UserType.Developer
   select new XElement("developer", user.Name)
);
Console.WriteLine(developers);

// Вывод
<developers>
  <developer>Deborah Denton</developer>
  <developer>Darren Dahlia</developer>
</developers>
```

Подобный прием может быть применен ко всем данным в образце, давая в результате документ следующего вида:

```
<defect-system>
  <projects>
    <project name="..." id="...">
      <subscription email="..." />
    </project>
  </projects>
  <users>
    <user name="..." id="..." type="..." />
  </users>
  <defects>
    <defect id="..." summary="..." created="..." project="..."
```

```
        assigned-to="..." created-by="..." status="..."
        severity="..." last-modified="..." />
  </defects>
</defect-system>
```

Код для генерации всего этого находится в файле `XmlSampleData.cs` внутри загружаемого решения. Он демонстрирует альтернативу подходу с одним крупным оператором: каждый элемент, находящийся ниже верхнего уровня, создается отдельно, а затем они компонуются примерно так:

```
XElement root = new XElement("defect-system", projects, users, defects);
```

Мы будем использовать эту разметку XML при иллюстрации следующего средства интеграции LINQ: запросов. Давайте начнем с методов запросов, доступных для одиночного узла.

12.3.3. Запросы для одиночных узлов

Возможно, вы ожидаете, что я раскрою факт реализации типом `XElement` интерфейса `IEnumerable` и доступности запросов LINQ без лишних хлопот. Но это не *настолько* просто, поскольку существует очень много вещей, по которым мог бы проходить `XElement`. Тип `XElement` содержит несколько *осевых методов*, которые применяются в качестве источников запросов. Если вы знакомы с XPath, то идея оси без сомнений должна быть вам известна.

Ниже перечислены осевые методы, используемые непосредственно для запрашивания одиночного узла, каждый из которых возвращает подходящую реализацию `IEnumerable<T>`:

- `Ancestors`
- `AncestorsAndSelf`
- `Annotations`
- `Attributes`
- `Descendants`
- `DescendantsAndSelf`
- `DescendantNodes`
- `DescendantNodesAndSelf`
- `Elements`
- `ElementsAfterSelf`
- `ElementsBeforeSelf`
- `Nodes`

Все методы не требуют особых объяснений (дополнительные сведения можно найти в документации MSDN). Существуют удобные перегруженные версии, предназначенные для извлечения только узлов с указанным именем; к примеру, вызов `Descendants("user")` на экземпляре `XElement` возвратит все элементы `user`, расположенные ниже элемента, на котором был произведен вызов.

В дополнение к методам, возвращающим последовательности, некоторые методы возвращают одиночный результат — наиболее важными являются `Attribute()` и `Element()`, возвращающие, соответственно, именованный атрибут и первый дочерний элемент с указанным именем.

Кроме того, доступны явные преобразования из XAttribute или XElement в другие типы, такие как int, string и DateTime. Это важно для результатов фильтрации и проецирования. Каждое преобразование в тип значения, не допускающий null, также имеет преобразование в его эквивалент, значение null допускающий; такие преобразования (и преобразование в string) возвращают значение null, если вызываются на ссылке null. Подобное распространение null означает, что вы не обязаны проверять наличие или отсутствие атрибутов или элементов внутри запроса — вместо этого вы можете применять результаты запроса.

Как это касается LINQ? Действительно, тот факт, что множественные результаты поиска возвращаются в виде IEnumerable<T>, означает возможность использования обычных методов LINQ to Objects после нахождения некоторых элементов. В листинге 12.9 приведен пример поиска имен и типов для пользователей, на этот раз в образце данных XML.

Листинг 12.9. Отображение пользователей внутри структуры XML

```
XElement root = XmlSampleData.GetElement();
var query = root.Element("users").Elements().Select(user => new
          {
            Name = (string) user.Attribute("name"),
            UserType = (string) user.Attribute("type")
          });
foreach (var user in query)
{
  Console.WriteLine ("{0}: {1}", user.Name, user.UserType);
}
```

После создания данных в начале производится переход к элементу users и запрашивание у него непосредственных дочерних элементов. Такое двухшаговое извлечение можно было бы сократить до root.Descendants("user"), но полезно знать о более жесткой навигации, чтобы применять ее по мере необходимости. Это также более надежно с точки зрения изменений в структуре документа, таких как добавление еще одного (несвязанного) элемента user где-то в рамках документа.

Остаток выражения запроса представляет собой просто проекцию XElement в анонимный тип. Соглашусь с тем, что я немного схитрил с типом пользователя: он сохраняется в виде строки вместо того, чтобы вызывать метод Enum.Parse() для преобразования его в соответствующее значение UserType. Последний подход работает великолепно, но он довольно многословен, когда нужна лишь строковая форма, и получающийся в результате код трудно подогнать под требования печатной страницы.

В конце концов, здесь нет ничего особенного — возвращение результатов запроса в виде последовательности весьма распространено. Полезно обратить внимание на то, насколько гладко можно переходить от операций запросов, специфичных для предметной области, к универсальным операциям. Однако на этом история не заканчивается. В LINQ to XML также есть несколько дополнительных расширяющих методов.

12.3.4. Выравнивающие операции запросов

Вы видели, что результатом одной части запроса часто оказывается последовательность, а в LINQ to XML зачастую это последовательность элементов. Что, если затем на каждом таком элементе необходимо выполнить специфичный для XML запрос?

В качестве несколько надуманного примера можно привести поиск всех проектов в образце данных с помощью `root.Element("projects").Elements()`, но как найти внутри них элементы `subscription`? К каждому элементу понадобится применить еще один запрос и затем выровнять результаты. (Опять-таки, можно было бы использовать `root.Descendants("subscription")`, но вообразите более сложную модель документа, в котором это не сработает.)

Описанное выше может выглядеть знакомым, и не зря — технология LINQ to Objects уже предлагает операцию `SelectMany()` (представляемую множеством конструкций `from` в выражении запроса), которая делает это. Запрос можно было бы написать следующим образом:

```
from project in root.Element("projects").Elements()
from subscription in project.Elements("subscription")
select subscription
```

Поскольку внутри проекта нет никаких других элементов кроме `subscription`, можно было бы воспользоваться перегруженной версией метода `Elements()`, чтобы не указывать имя. Лично я считаю указание имени элемента в данном случае более ясным, но это дело вкуса. (Надо сказать, что аналогичный довод можно было бы привести и в пользу вызова `Element("projects").Elements("project")`.)

Ниже показан тот же запрос, записанный с применением точечной нотации и перегруженной версии метода `SelectMany()`, которая только возвращает выровненную последовательность, не выполняя никакого дальнейшего проецирования:

```
root.Element("projects").Elements()
    .SelectMany(project => project.Elements("subscription"))
```

Ни один из этих запросов нельзя назвать полностью нечитабельным, однако они и не идеальны. В LINQ to XML предлагается несколько расширяющих методов (в классе `System.Xml.Linq.Extensions`), которые либо действуют на специфическом типе последовательности, либо является обобщенным с ограниченным аргументом типа, чтобы справиться с отсутствием ковариантности обобщенных интерфейсов в версиях, предшествующих C# 4. Имеется метод `InDocumentOrder()`, возвращающий все узлы в порядке, заданном в документе, и большинство осевых методов, упомянутых в разделе 12.4.3, также доступны в виде расширяющих методов. Это означает, что предыдущий запрос можно преобразовать в следующую более простую форму:

```
root.Element("projects").Elements().Elements("subscription")
```

Конструкция такого вида упрощает запись XPath-подобных запросов в LINQ to XML без требования, чтобы все было строками. Если вы хотите использовать язык XPath, он доступен через дополнительные расширяющие методы, но методы запросов чаще меня устраивали, чем не устраивали. Кроме того, допускается смешивать осевые методы и операции LINQ to Objects. Например, чтобы найти все подписки на уведомления для проектов с названием, включающем строку *Media*, можно было бы записать так:

```
root.Element("projects").Elements()
    .Where(project => ((string) project.Attribute("name"))
                                .Contains("Media"))
    .Elements("subscription")
```

Прежде чем переходить к Parallel LINQ, давайте подумаем о том, как проект LINQ to XML заслужил указания части "LINQ" в своем названии, и каким образом можно было бы потенциально применить те же самые приемы к собственным API-интерфейсам.

12.3.5. Работа в гармонии с LINQ

Некоторые из проектных решений в LINQ to XML выглядят странными, если рассматривать их изолированно как часть API-интерфейса XML, но в контексте LINQ они обретают настоящий смысл. Проектировщики четко представляли себе, как их типы могли бы использоваться внутри запросов LINQ, и каким образом взаимодействовать с другими источниками данных. Если вы разрабатываете собственный API-интерфейс доступа к данным, то в каком бы контексте он не находился, полезно принять во внимание аналогичные аспекты. Если кто-то применит ваши методы в середине выражения запроса, принесут ли они хоть какую-то пользу? Будет ли возможность использовать некоторые из ваших методов запросов, затем некоторые методы из LINQ to Objects и в конце снова ваши методы в одном текучем выражении?

Мы видели три пути, по которым технология LINQ to XML приспосабливается к остальным частям LINQ.

- Она способна потреблять последовательности благодаря своему подходу к конструированию. Язык LINQ изначально был декларативным, и LINQ to XML поддерживает это посредством декларативного способа создания структур XML.
- Она возвращает последовательности из своих методов запросов. Пожалуй, это наиболее очевидный шаг, который API-интерфейсы доступа к данным должны были бы уже предпринимать: возвращение результатов запроса в виде `IEnumerable<T>` или реализующего его класса — далеко не бином Ньютона.
- Она расширяет набор запросов, которые можно выполнять в отношении последовательностей типов XML. Это делает ее похожей на унифицированный API-интерфейс запросов, несмотря на то, что некоторые ее части являются специфичными для XML.

Вы можете придумать и другие пути взаимодействия ваших библиотек с LINQ; это не единственные варианты, которые вы должны рассмотреть, но они являются хорошей отправной точкой. Прежде всего, я настоятельно рекомендую поставить себя на место разработчика, желающего использовать ваш API-интерфейс внутри кода, в котором уже применяется LINQ. К чему такой разработчик может стремиться? Можно ли легко смешивать в коде взаимодействие с LINQ и вашим API-интерфейсом, или же они действительно предназначены для разных целей?

Мы находимся примерно на середине пути скоростного обзора разнообразных подходов, обеспечиваемых LINQ. Наша следующая остановка в некоторой степени обнадеживает, но в некоторой — устрашает: мы снова возвращаемся к запрашиванию простых последовательностей, но на этот раз параллельно...

12.4. Замена LINQ to Objects технологией Parallel LINQ

Я следил за Parallel LINQ на протяжении долгого времени. Впервые я узнал о нем, когда Джо Даффи представил его в своем блоге в сентябре 2006 года (`http://mng.bz/vYCO`). Первая предварительная версия (Community Technology Preview – CTP) была выпущена в ноябре 2007 года, и с течением времени сформировался полный набор средств. Теперь это часть обширного проекта под названием Parallel Extensions, который входит в состав .NET 4 и нацелен на предоставление строительных блоков для параллельного программирования более высокого уровня, чем относительно неболь-

шой набор примитивов, с которыми приходилось работать до сих пор. Проект Parallel Extensions включает много других средств помимо Parallel LINQ (или PLINQ, как часто на него ссылаются), но мы рассмотрим здесь только аспекты, касающиеся LINQ.

Идея, положенная в основу Parallel LINQ, состоит в том, что должна быть возможность взять запрос LINQ to Objects, отнимающий длительное время, и заставить выполняться быстрее за счет использования множества потоков в нескольких ядрах процессора, причем с минимальными изменениями самого запроса. Как и все, что связано с параллелизмом, это не так просто, но вас может удивить, насколько многого можно достигнуть. Разумеется, мы по-прежнему стараемся мыслить более широкими категориями, чем отдельные технологии LINQ — мы будем оперировать различными моделями взаимодействия, которые при этом задействуются, а не точными деталями реализации. Однако если вас интересует параллелизм, я настоятельно рекомендую глубже исследовать Parallel Extensions — один из самых многообещающих подходов к решению задач распараллеливания, которые только мне встречались.

В данном разделе будет рассматриваться единственный пример: визуализация изображения множества Мандельброта (за объяснением обращайтесь в Википедию: `http://ru.wikipedia.org/wiki/Множество_Мандельброта`). Давайте начнем с попытки сделать это в одном потоке, а после этого займемся более сложным решением.

12.4.1. Отображение множества Мандельброта с помощью одного потока

Прежде чем меня начнут атаковать математики, я должен отметить, что термин *множество Мандельброта* здесь используется нестрого. Детали не особенно важны, но важны следующие аспекты.

- Будет создаваться прямоугольное изображение на основе разнообразных параметров, таких как ширина, высота, начало координат и глубина поиска.
- Для каждого пикселя в изображении будет вычисляться байтовое значение, которое представляет собой индекс внутри палитры из 256 записей.
- Вычисление значения одного пикселя не основано на любых других результатах.

Последний аспект из перечисленных критически важен — он означает, что эта задача *совершенно параллельна*. Другими словами, в самой задаче нет ничего такого, что затруднило бы ее распараллеливание. По-прежнему необходим механизм для распределения рабочей нагрузки по потокам с последующим сбором результатов, но остальное должно быть простым. Ответственным за распределение и сбор будет PLINQ (с небольшой помощью ему); вам всего лишь понадобится выразить диапазон пикселей и способ вычисления цвета каждого пикселя.

В целях демонстрации нескольких подходов я построил абстрактный базовый класс, который отвечает за надлежащую настройку, запуск запроса и отображение результатов; он также имеет метод для вычисления цвета отдельного пикселя. Абстрактный метод должен создавать байтовый массив значений, которые затем преобразуются в изображение. Сначала идет первая строка пикселей, слева направо, потом вторая строка и т.д. Каждый приводимый здесь пример сводится просто к реализации этого метода.

Должен заметить, что применение в такой ситуации LINQ на самом деле не является идеальным решением — данный подход по разным причинам неэффективен. Не заостряйте внимания на этой стороне дела: сосредоточьтесь на идее того, что мы имеем совершенно параллельный запрос, и хотим выполнить его на нескольких ядрах.

В листинге 12.10 показана однопоточная версия метода во всей своей изящной простоте.

Листинг 12.10. Однопоточная версия запроса, генерирующего множество Мандельброта

```
var query = from row in Enumerable.Range(0, Height)
            from column in Enumerable.Range(0, Width)
            select ComputeIndex(row, column);
return query.ToArray();
```

Запрос проходит по всем строкам и позициям в каждой строке, вычисляя индекс соответствующего пикселя. Вызов `ToArray()` вычисляет результирующую последовательность, преобразуя ее в массив. На рис. 12.5 можно видеть симпатичные результаты.

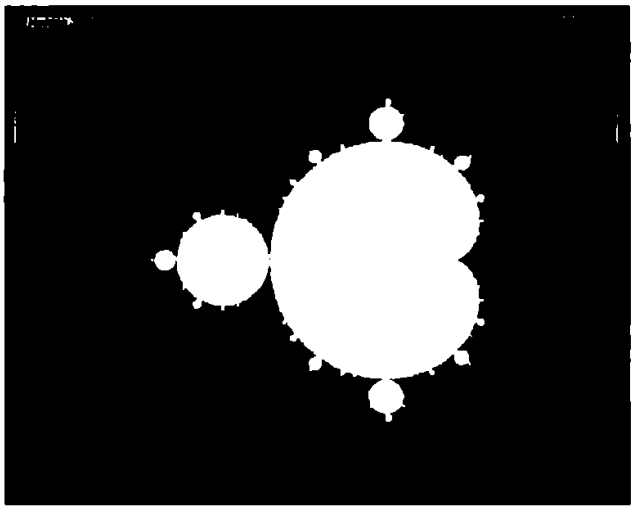

Рис. 12.5. Изображение множества Мандельброта, сгенерированное в одном потоке

На моем стареньком двухядерном лаптопе генерация заняла около 5,5 секунды. Метод `ComputeIndex()` выполняет больше итераций, чем действительно необходимо, но это позволяет сделать разницу в оценках времени более очевидной[7]. Теперь, когда имеется эталон в терминах измерения времени и внешнего вида результатов, можно заняться распараллеливанием запроса.

12.4.2. Введение в `ParallelEnumerable`, `ParallelQuery` и `AsParallel()`

Технология Parallel LINQ предлагает несколько новых типов, но во многих случаях вы никогда не встретите упоминания их имен. Они находятся в пространстве имен `System.Linq`, поэтому не придется даже изменять директивы `using`. Статический класс `ParallelEnumerable` похож на `Enumerable`; он содержит главным образом расширяющие методы, большинство из которых расширяют новый тип `ParallelQuery`.

Тип `ParallelQuery` имеет необобщенную и обобщенную формы (`ParallelQuery` и `ParallelQuery<TSource>`), но большую часть времени вы будете использовать его

[7] Надлежащее эталонное тестирование выполнять нелегко, особенно в условиях многопоточности. Я не пытался здесь проводить точные измерения. Приводимые оценки времени предназначены просто для отражения факта более быстрого или медленного выполнения; относитесь к этим числам скептически.

обобщенную форму, в точности как IEnumerable<T> применяется более широко, чем IEnumerable. Вдобавок имеется тип OrderedParallelQuery<TSource>, который представляет собой параллельный эквивалент IOrderedEnumerable<T>. Отношения между всеми этими типами показаны на рис. 12.6.

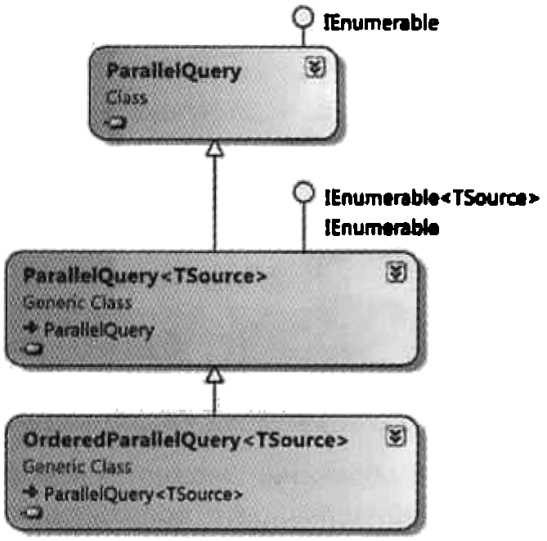

Рис. 12.6. Диаграмма классов для Parallel LINQ, включая отношения с обычными интерфейсами LINQ

Как видите, ParallelQuery<TSource> реализует IEnumerable<TSource>, так что после конструирования запроса можно проходить по результатам обычным образом. При наличии параллельного запроса расширяющие методы в классе ParallelEnumerable получают преимущество перед такими методами в Enumerable (поскольку ParallelQuery<T> более специфичен, чем IEnumerable<T>; обратитесь в раздел 10.2.3, если вы подзабыли эти правила); именно так параллелизм поддерживается на протяжении всего запроса. Существуют параллельные эквиваленты всех стандартных операций запросов LINQ, но вы должны проявлять осторожность, создавая собственные расширяющие методы. Возможность обращения к ним остается, однако после этого запрос перейдет в однопоточный режим.

Для начала, как получить параллельный запрос? За счет вызова расширяющего метода AsParallel() из класса ParallelEnumerable, который расширяет IEnumerable<T>. Это означает, что распараллелить запрос, генерирующий множество Мандельброта, невероятно просто, что и демонстрируется в листинге 12.11.

Листинг 12.11. Первая попытка выполнения многопоточного запроса, генерирующего множество Мандельброта

```
var query = from row in Enumerable.Range(0, Height)
                                  .AsParallel()
            from column in Enumerable.Range(0, Width)
            select ComputeIndex(row, column);

return query.ToArray();
```

Работа сделана? Не совсем. Этот запрос *действительно* выполняется параллельно, но результаты не вполне удовлетворительны: порядок, в котором обрабатываются строки, не поддерживается.

Вместо симпатичного изображения множества Мандельброта мы получаем нечто, подобное показанному на рис. 12.7, причем каждый раз мелкие детали меняются.

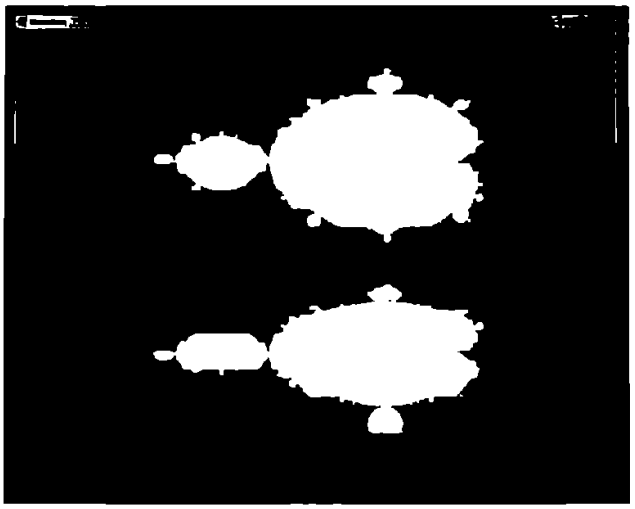

Рис. 12.7. Изображение множества Мандельброта, сгенерированное с использованием неупорядоченного запроса, имеет неправильно расположенные участки

Ох! Но есть и оптимистичная новость: изображение визуализируется примерно за 3,2 секунды, так что на моей машине было определенно задействовано второе ядро. Но нам очень важно получить правильный результат.

Вас может удивить тот факт, что это является заранее спланированной *особенностью* Parallel LINQ. Упорядочение параллельного запроса требует большей координации между потоками, а итоговая цель распараллеливания заключается в улучшении производительности, поэтому по умолчанию запрос PLINQ не упорядочивается. Хотя в данном случае это досадно.

12.4.3. Подстройка параллельных запросов

К счастью, выход из положения существует — нужно просто заставить запрос трактоваться как упорядоченный, что может быть сделано с помощью расширяющего метода `AsOrdered()`. В листинге 12.12 приведен скорректированный код, генерирующий первоначальное изображение. Он работает немного медленнее, чем неупорядоченный запрос, но по-прежнему значительно быстрее однопоточной версии.

Листинг 12.12. Многопоточный запрос, генерирующий изображение множества Мандельброта и поддерживающий упорядочение

```
var query = from row in Enumerable.Range(0, Height)
                            .AsParallel().AsOrdered()
            from column in Enumerable.Range(0, Width)
            select ComputeIndex(row, column);

return query.ToArray();
```

Нюансы упорядочения выходят за рамки настоящей книги, но я рекомендую почитать статью "PLINQ Ordering" ("Упорядочение в PLINQ") в блоге MSDN (`http://mng.bz/9x9U`), в которой раскрываются все жуткие подробности.

Поведение запроса можно изменять с применением других методов, которые кратко описаны ниже.

- `AsUnordered()`. Делает упорядоченный запрос неупорядоченным; если упорядоченные результаты нужны только в первой части запроса, то этот метод позволит последующим этапам запроса выполняться более эффективно.
- `WithCancellation()`. Указывает маркер отмены для использования с этим запросом. Маркеры отмены применяются в Parallel Extensions, позволяя отменять задачи в безопасной и контролируемой манере.
- `WithDegreeOfParallelism()`. Позволяет указывать максимальное количество параллельных задач, используемых для выполнения запроса. Этот метод можно применять для ограничения количества используемых потоков во избежание перегрузки машины работой или наоборот — увеличения числа потоков в случае, если запрос не задействует интенсивно центральный процессор.
- `WithExecutionMode()`. Может применяться для принудительного выполнения запроса в параллельном режиме, даже если Parallel LINQ считает, что запрос будет выполняться быстрее в однопоточном режиме.
- `WithMergeOptions()`. Позволяет подстраивать буферизацию результатов. Отключение буферизации сокращает промежуток времени перед возвращением первого результата, но также уменьшает пропускную способность; полная буферизация обеспечивает наивысшую пропускную способность, но результаты не будут возвращаться до тех пор, пока запрос не выполнится полностью. Стандартная настройка является компромиссом между этими двумя случаями.

Важный момент заключается в том, что помимо упорядочения эти методы не должны влиять на *результаты* запроса. Вы можете спроектировать свой запрос и протестировать его в LINQ to Objects, затем распараллелить, сформулировать требования к упорядочению и при необходимости подстроить запрос, чтобы он делал именно то, что нужно. Если вы покажете окончательный запрос кому-то, кто знает LINQ, но не PLINQ, то вам придется объяснить только вызовы методов, специфичных для PLINQ, а остальные части запроса должны быть ясны. Приходилось ли вам видеть настолько простой путь обеспечения параллелизма? (Остальные аспекты Parallel Extensions также направлены на достижение простоты, где только это возможно.)

Экспериментируйте с кодом самостоятельно

В загружаемом исходном коде демонстрируется пара дополнительных моментов. Если распараллелить целый запрос для пикселей, а не только по строкам, то результаты неупорядоченного запроса будут выглядеть даже еще более странными. Кроме того, существует метод `ParallelEnumerable.Range()`, который предоставляет PLINQ немного больше информации, чем вызов `Enumerable.Range(...).AsParallel()`. В этом разделе используется метод `AsParallel()`, т.к. он является более общим способом распараллеливания запроса; большинство запросов не начинаются с диапазона.

Смена модели внутрипроцессных запросов с однопоточной на параллельную — это на самом деле довольно небольшой концептуальный скачок. В следующем разделе мы перевернем эту модель с ног на голову.

12.5. Инвертирование модели запросов с помощью LINQ to Rx

Все библиотеки LINQ, которые вы видели до сих пор, имели одну общую черту: данные доставлялись пассивно с применением `IEnumerable<T>`. На первый взгляд, это кажется настолько очевидным, что даже не стоит о нем говорить — разве может существовать альтернатива? Хорошо, тогда как насчет *активной доставки* данных взамен пассивной? Вместо удержания контроля над потребителем данных поставщик может взять управление на себя, позволяя потребителю данных реагировать, когда становятся доступными новые данные. Не переживайте, если большинство из этого звучит пугающе чуждо; в действительности вы уже знакомы с этой фундаментальной концепцией в форме событий. Если вам комфортно с идеей подписки на событие, реакцией на него и впоследствии отмены подписки, то это послужит хорошей отправной точкой.

Reactive Extensions for .NET является проектом Microsoft (`http://mng.bz/R7ip`); доступно несколько версий, включая версию, нацеленную на JavaScript. В наши дни актуальную версию проще всего получать через NuGet. Вы могли сталкиваться с Reactive Extensions под разными названиями, но наиболее распространены аббревиатуры *Rx* и LINQ *to Rx*, поэтому они и будут использоваться здесь. Данная тема будет описана даже более поверхностно, чем другие технологий, рассматриваемые в этой главе. Мало того, что есть много материала, который необходимо изучить о самой библиотеке, так она еще и требует совершенно иного образа мышления. На канале Channel 9 загружено несколько видеороликов (`http://channel9.msdn.com/tags/Rx/`) — некоторые из них основаны на математических аспектах, тогда как другие больше ориентированы на практику. В этом разделе внимание будет акцентироваться на способе, которым концепции LINQ могут быть применены в такой модели с активным источником для потока данных.

Для введения сведений достаточно, поэтому давайте ознакомимся с двумя интерфейсами, которые формируют основу LINQ to Rx.

12.5.1. IObservable<T> и IObserver<T>

Модель данных LINQ to Rx *математически двойственна* обычной модели `IEnumerable<T>`[8]. Когда вы проходите по пассивной коллекции, то на самом деле начинаете с указания "Предоставь мне итератор" (вызов метода `GetEnumerator()`) и затем многократно запрашиваете "Если ли еще один элемент? Если да, он мне сейчас нужен" (через вызовы методов `MoveNext()` и `Current()`). В LINQ to Rx это перевернуто. Вместо запрашивания итератора вы предоставляете наблюдателя. Затем вместо запрашивания очередного элемента ваш код сообщает, когда он готов — либо когда возникает ошибка или достигнут конец данных.

Ниже представлены объявления двух задействованных интерфейсов:

```
public interface IObservable<T>
{
  IDisposable Subscribe(IObserver<T> observer);
}
```

[8] За более подробными исследованиями этой двойственности — и сущности самого LINQ — рекомендую статью под названием "The Essence of LINQ — MinLINQ" ("Сущность LINQ — MinLINQ"), опубликованную Бартом де Сметом в своем блоге по адресу `http://mng.bz/96Wh`.

```
public interface IObserver<T>
{
  void OnNext(T value);
  void OnCompleted();
  void OnException(Exception error);
}
```

Эти интерфейсы в действительности являются частью .NET 4 (и находятся в пространстве имен `System`), хотя остальная библиотека LINQ to Rx организована как отдельная загрузка. На самом деле в .NET 4 они выглядят как `IObservable<out T>` и `IObserver<in T>`, выражая ковариантность `IObservable` и контравариантность `IObserver`. Дополнительные сведения об обобщенной вариантности будут даны в следующей главе, но для простоты я представляют эти интерфейсы здесь так, как если бы они были инвариантными. Помните: по одной концепции за раз!

На рис. 12.8 двойственность показана в терминах перетекания данных в каждой модели.

Я предполагаю, что не одинок в своем мнении, что модель с активным источником данных труднее для восприятия, т.к. она обладает естественной возможностью работать асинхронно. Но посмотрите, насколько она проще модели с пассивным источником, в терминах потоковой диаграммы. Отчасти это объясняется подходом с множеством методов, используемым в модели с пассивным источником; если бы только `IEnumerator<T>` имел метод с сигнатурой `bool TryGetNext(out T item)`, все было бы до некоторой степени проще.

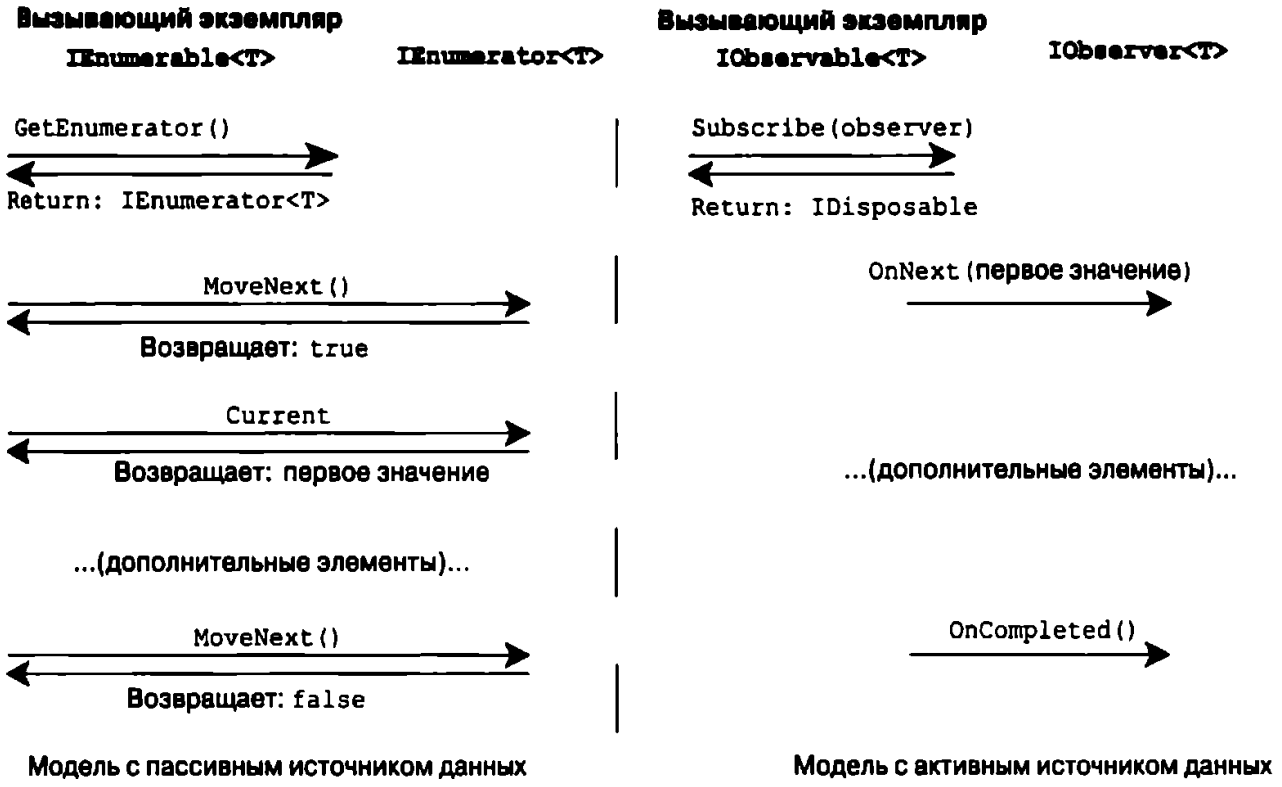

Рис. 12.8. Диаграмма последовательностей, отражающая двойственность интерфейсов `IEnumerable<T>` и `IObservable<T>`

Ранее я упоминал, что технология LINQ to Rx похожа на события, с которыми вы уже знакомы. Вызов `Subscribe()` на наблюдаемом объекте подобен применению операции `+=` к событию для регистрации обработчика. Освобождаемое значение, возвращаемое методом `Subscribe()`, запоминает переданного наблюдателя; его освобождение похоже на использование операции `-=` с тем же самым обработчиком. Во многих случаях отменять подписку для наблюдаемого объекта вовсе не обязательно; это доступно просто на случай, когда подписку необходимо отменить где-то на полпути последовательности — своего рода эквивалент раннему прекращению цикла `foreach`.

Отсутствие освобождения значения `IDisposable` может вам казаться грубой ошибкой, но в LINQ to Rx это часто безопасно. Ни в одном из примеров данной главы возвращаемое значение метода `Subscribe` не применяется.

Об интерфейсе `IObservable<T>` сказано все, но как насчет самого наблюдателя? Почему он имеет три метода? Представим обычную модель с пассивным источником, в которой для любой пары вызовов `MoveNext()/Current()` могут возникать три ситуации.

- Вы можете находиться в конце последовательности, в случае чего метод `MoveNext()` возвращает `false`.
- Вы пока еще не достигли конца последовательности, в случае чего метод `MoveNext()` возвращает `true`, а свойство `Current` — новое значение.
- Может возникнуть ошибка — например, не удается прочитать очередную строку из сетевого подключения. В этом случае сгенерируется исключение.

Интерфейс `IObserver<T>` представляет каждую из указанных возможностей в виде отдельного метода. Как правило, наблюдатель будет многократно вызывать свой метод `OnNext()` и в конце метод `OnCompleted()`, если только не возникла какая-то ошибка, в случае чего вместо `OnCompleted()` будет вызван метод `OnError()`. После того, как последовательность завершена или произошла ошибка, никаких дальнейших вызовов методов не делается. Тем не менее, потребность в прямой реализации интерфейса `IObserver<T>` возникает редко. Для `IObservable<T>` доступно множество расширяющих методов, в том числе перегруженные версии `Subscribe()`, и они позволяют подписываться на наблюдаемый объект, просто указывая подходящие делегаты. Обычно предоставляется делегат, подлежащий выполнению для каждого элемента, и необязательный делегат для выполнения по завершении либо делегат для случая ошибки или оба.

Памятуя об описанных выше теоретических сведениях, давайте рассмотрим действительный код, в котором используется LINQ to Rx.

12.5.2. Простое начало (снова)

Мы будем демонстрировать LINQ to Rx таким же способом, как это делалось при представлении LINQ to Objects — начнем с применения диапазона.

Вместо `Enumerable.Range()` мы будем использовать метод `Observable.Range()`, который создает наблюдаемый диапазон. Каждый раз, когда какой-то наблюдатель подписывается на диапазон, этому наблюдателю будут выдаваться числа при вызове им метода `OnNext()`, за которым следует `OnCompleted()`. Мы начнем с наиболее простой задачи, всего лишь выводя на консоль значения по мере их получения и сообщение при наступлении конца или при возникновении ошибки.

В листинге 12.13 показано, что это требует меньше кода, чем в случае применения модели с пассивным источником данных.

Листинг 12.13. Первое знакомство с IObservable<T>

```
var observable = Observable.Range(0, 10);
observable.Subscribe(x => Console.WriteLine("Received {0}", x),
                     e => Console.WriteLine("Error: {0}", e),
                    () => Console.WriteLine("Finished"));
```

В данной ситуации трудно представить себе, как можно было бы получить ошибку, однако ради полноты делегат для уведомления об ошибке все же включен. Результаты вполне ожидаемы:

```
Received 0
Received 1
...
Received 9
Finished
```

Наблюдаемый объект, возвращаемый методом `Range()`, известен как *"холодный" наблюдаемый объект*: он остается бездействующим до тех пор, пока какой-либо наблюдатель не подпишется на него, после чего будет выдавать значения этому отдельному наблюдателю. Если вдобавок подпишется еще один наблюдатель, он будет видеть другую копию диапазона. Это совсем не похоже на обычное событие вроде щелчка на кнопке, когда множество наблюдателей может быть подписано на одну и ту же действительную последовательность значений, и значения могут выдаваться независимо от того, существуют какие-то наблюдатели или нет. (В конце концов, щелкать на кнопке можно даже при отсутствии присоединенных к ней обработчиков событий.) Последовательности подобного рода носят название *"горячих" наблюдаемых объектов*. Важно знать, с каким типом приходится иметь дело, даже при условии, что к обоим видам применим один и тот же набор операций. Теперь, когда выполнено простейшее действие, давайте попробуем воспользоваться рядом знакомых операций LINQ.

12.5.3. Запрашивание наблюдаемых объектов

К этому времени вы уже должны хорошо представлять себе шаблон — в статическом классе (с довольно предсказуемым именем `Observable`) определены разнообразные расширяющие методы, которые выполняют подходящие трансформации. Мы рассмотрим лишь несколько доступных операций и поразмышляем о том, что *не* доступно и по какой причине.

Фильтрация и проецирование

Начнем прямо с выражения запроса, которое принимает последовательность чисел, отфильтровывает нечетные числа и возводит в квадрат оставшиеся. Затем мы подпишем метод `Console.WriteLine()` на финальный результат запроса, чтобы любые генерируемые элементы были выведены на консоль. В листинге 12.14 приведен необходимый код — посмотрите, насколько легко превратить выражение запроса в запрос LINQ to Objects.

Листинг 12.14. Фильтрация и проецирование в LINQ to Rx

```
var numbers = Observable.Range(0, 10);
var query = from number in numbers
            where number % 2 == 0
            select number * number;
query.Subscribe(Console.WriteLine);
```

В целях простоты здесь не добавлялись обработчики для завершения или ошибки, а применение преобразования из группы методов `Console.WriteLine()` в `Action<int>` позволило сохранить код изящным и кратким. Запрос выдает те же результаты, как если это было в LINQ to Objects: `0`, `4`, `16` и т.д. А теперь перейдем к группированию.

Группирование

Конструкция `group by` выражения запроса в LINQ to Rx генерирует новый объект `IGroupedObservable<T>` для каждой группы, несмотря на то, что дальнейшее *действие* с группами не всегда очевидно. Например, не такой уж редкостью является ситуация с наличием вложенной подписки, когда наблюдатель подписывается на каждую вновь создаваемую группу. Результаты *внутри* каждой группы генерируются по мере их получения конструкцией группирования — в сущности это действует как своего рода перенаправление, подобное контролеру в театре, который проверяет у вошедшего человека билет и направляет его в соответствующий сектор. В противоположность этому технология LINQ to Objects собирает всю группу полностью, прежде чем возвращать ее, а это значит, что она должна дочитать последовательность до конца, буферизируя результаты.

В листинге 12.15 приведен пример такой вложенной подписки и также продемонстрирован способ выдачи результатов группы.

Листинг 12.15. Группирование чисел с делением по модулю 3

```
var numbers = Observable.Range(0, 10);
var query = from number in numbers
            group number by number % 3;
query.Subscribe(group => group.Subscribe
   (x => Console.WriteLine("Value: {0}; Group: {1}", x, group.Key)));
```

Пожалуй, понять этот запрос будет проще, если вспомнить, что работа с группами в LINQ to Objects часто предусматривает наличие вложенного цикла `foreach` — вот так и LINQ to Rx присутствуют вложенные подписки.

Когда вы сомневаетесь, попробуйте найти двойственность между двумя моделями данных. В LINQ to Objects вы обычно обрабатываете целые группы по очереди, тогда как порядок в LINQ to Rx означает, что вывод запроса из листинга 12.15 будет выглядеть следующим образом:

```
Value: 0; Group: 0
Value: 1; Group: 1
Value: 2; Group: 2
Value: 3; Group: 0
Value: 4; Group: 1
Value: 5; Group: 2
Value: 6; Group: 0
Value: 7; Group: 1
Value: 8; Group: 2
Value: 9; Group: 0
```

Это обретает особый смысл при обдумывании в терминах модели с активным источником, а в ряде случаев означает, что операции, которые требовали бы буферизации значительного объема данных в LINQ to Objects, могут быть реализованы в LINQ to Rx намного более эффективно.

В качестве последнего примера рассмотрим еще одну операцию, имеющую дело с несколькими последовательностями.

Выравнивание

Библиотека LINQ to Rx содержит несколько перегруженных версий `SelectMany()`, при этом лежащая в основе идея по-прежнему такая же, как в LINQ to Objects: каждый элемент в исходной последовательности создает новую последовательность, а результатом является объединение всех этих новых последовательностей и его выравнивание. В листинге 12.16 сказанное демонстрируется в действии — запрос в нем немного напоминает запрос из листинга 11.16, приведенного во время начального обсуждения `SelectMany()` в LINQ to Objects.

Листинг 12.16. Генерация множества диапазонов с помощью `SelectMany()`

```
var query = from x in Observable.Range(1, 3)
            from y in Observable.Range(1, x)
            select new { x, y };
query.Subscribe(Console.WriteLine);
```

Ниже показаны результаты, которые должны быть довольно предсказуемыми:

```
{ x = 1, y = 1 }
{ x = 2, y = 1 }
{ x = 2, y = 2 }
{ x = 3, y = 1 }
{ x = 3, y = 2 }
{ x = 3, y = 3 }
```

В этом случае результаты детерминированы, но только потому, что по умолчанию `Observable.Range()` выдает элементы в текущем потоке. Вполне возможно иметь дело с несколькими последовательностями, которые генерируются во множестве разных потоков.

Ради интереса можете изменить второй вызов `Observable.Range()`, указав `Scheduler.ThreadPool` в качестве третьего аргумента. В результате каждая внутренняя последовательность соблюдает свой порядок, но отдельные последовательности могут перемешиваться друг с другом. Вообразите себе стадион с одним судьей со стартовым пистолетом, который обслуживает множество разных забегов, идущих подряд — даже если известен победитель каждого забега, вы совершенно не знаете, какой забег будет финиширован первым.

Приношу свои извинения, если все это вас утомило. В утешение отмечу, что испытываю похожие чувства. Однако одновременно я нахожу это увлекательным.

Что доступно, а что нет?

Вы уже знаете, что конструкция `let` работает, просто вызывая метод `Select()`, что естественным образом вписывается в LINQ to Rx, но не все операции LINQ to Objects реализованы в LINQ to Rx. Как правило, отсутствуют те операции, которые приводили бы к буферизации своего вывода и возвращению нового наблюдаемого объекта. Например, не существует методов `Reverse()` и `OrderBy()`. Для языка C# это вполне нормально — он всего лишь не позволяет использовать конструкцию `orderby` в выражении запроса, основанном на наблюдаемых объектах. Доступен метод `Join()`, однако он не имеет дело с наблюдаемыми объектами напрямую — он обрабатывает *планы соединений*.

Это часть реализации Rx исчисления соединений, и она выходит далеко за рамки материала настоящей книги. Подобным же образом отсутствует метод `GroupJoin()`, поэтому операция `join...into` не поддерживается.

Описание различных стандартных операций запросов LINQ, которые не покрыты синтаксисом выражений запросов, а также широкого спектра дополнительных методов, делающих их доступными, ищите в документации по пространству имен `System.Reactive`. Хотя вас может несколько разочаровать отсутствие в LINQ to Rx знакомой функциональности из LINQ to Objects (в большинстве случаев из-за того, что она не имеет здесь смысла), вы удивитесь, насколько в действительности богат набор доступных методов. Многие новые методы впоследствии были перенесены в LINQ to Objects и находятся в сборке `System.Interactive`.

12.5.4. Какой в этом смысл?

Я полностью осознаю, что пока еще не привел веских доводов в пользу применения LINQ to Rx. Это неслучайно, т.к. я не намерен приводить полноценный пример – в рамках этой главы он второстепенен и занял бы слишком много места. Тем не менее, библиотека Rx предлагает элегантный подход к размышлениям обо всех видах асинхронных процессов, таких как обычные события .NET (которые можно рассматривать как наблюдаемые объекты, используя метод `Observable.FromEvent()`), асинхронный ввод-вывод и обращения к веб-службам. Она предоставляет эффективный способ управления сложностью и параллелизмом. Несомненно, она труднее в освоении, чем LINQ to Objects, но если вы находитесь в ситуации, когда библиотека Rx оказывается полезной, значит, вы уже столкнулись с непомерной сложностью.

Причина, по которой я решил раскрыть Rx в этой книге, несмотря на невозможность уделить ей столько внимания, сколько она заслуживает, связана с тем, что эта библиотека позволяет понять, почему LINQ был спроектирован именно так, а не по-другому. Хотя доступны методы преобразований между `IEnumerable<T>` и `IObservable<T>`, отношение наследования отсутствует. Если бы в языке присутствовало требование, что типы, задействованные в LINQ, должны быть пассивными последовательностями, то никакой поддержки Rx в выражениях запросов не было бы вообще. Последствия оказались бы еще более катастрофическими, если бы расширяющие методы были каким-то образом ограниченными только интерфейсом `IEnumerable<T>`. Кроме того, вы видели, что не все нормальные операции LINQ применимы к Rx. Именно поэтому так важно, что язык указывает трансляции запросов в терминах шаблона, который должен поддерживаться настолько, насколько это имеет смысл для заданного поставщика. Надеюсь, вы поняли, что хотя работа с моделями с активным и пассивным источниками данных совершенно отличается, LINQ действует в качестве объединяющей силы там, где это возможно.

Возможно, вас обрадует, что последняя тема в данной главе будет намного проще – мы возвратимся обратно к LINQ to Objects, но на этот раз займемся написанием собственных расширяющих методов.

12.6. Расширение LINQ to Objects

Одной из замечательных характеристик LINQ является его расширяемость. Вы можете не только предложить собственные поставщики запросов и модели данных, но также дополнить существующие. По моему опыту, самая распространенная ситуация, когда это оказывается полезным, возникает с LINQ to Objects. Если необходим опреде-

ленный тип запроса, который напрямую не поддерживается (либо реализуется с помощью стандартных операций запросов неуклюже или неэффективно), можно написать собственный такой тип. Разумеется, построение универсального обобщенного метода может быть более сложным, чем просто решение непосредственной задачи, но если вы обнаружите, что пишете похожий код несколько раз, полезно рассмотреть возможность его рефакторинга в виде новой операции.

Лично мне нравится создавать операции запросов. При этом возникают интересные технические проблемы, однако редко требуется большой объем кода, а результаты могут оказаться весьма элегантными. В данном разделе мы взглянем на ряд путей, позволяющих обеспечить эффективное и предсказуемое поведение специальных операций, а затем рассмотрим полноценный пример выборки случайного элемента из последовательности.

12.6.1. Руководство по проектированию и реализации

Большинство руководящих принципов могут показаться вполне очевидными, но этот раздел помогает сформировать удобный контрольный список, полезный при написании операции.

Модульные тесты

Как правило, написать для операций качественный набор модульных тестов довольно легко, хотя вы можете быть удивлены тем, сколько их понадобится для такого, на первый взгляд, простого кода. Не забывайте тестировать краевые случаи, такие как пустые последовательности и недопустимые аргументы. В проекте модульного тестирования в рамках MoreLINQ (`http://code.google.com/p/morelinq/`) имеются вспомогательные методы, которые, возможно, вы решите задействовать в своих тестах.

Проверка аргументов

Хорошие методы проверяют свои аргументы, но возникает проблема, когда дело доходит до операций LINQ. Как вы уже видели, многие операции возвращают еще одну последовательность, и простейшим способом реализовать такую функциональность являются итераторные блоки. Но в действительности проверка аргументов должна выполняться при вызове метода, не дожидаясь, пока вызывающий код решит начать проход по результатам. Если вы собираетесь использовать итераторный блок, разбейте свой метод на два: проводите проверку аргументов в открытом методе, а затем вызывайте закрытый метод для выполнения итерации.

Оптимизация

Сам по себе интерфейс `IEnumerable<T>` довольно слаб в плане поддерживаемых операций, но тип времени выполнения последовательности, с которой вы работаете, может обладать значительно большей функциональностью. Например, операция `Count()` будет всегда работать, но обычно это будет операция со сложностью O(n). Тем не менее, если вы вызовете ее на реализации `ICollection<T>`, она может воспользоваться свойством `Count` напрямую, что в общем случае даст сложность O(1). В .NET 4 такая оптимизация расширена с целью охвата также и `ICollection`. Аналогично, извлечение индивидуального элемента по индексу является медленным в общем случае, но может быть эффективным, если последовательность реализует `IList<T>`.

Если ваша операция может выиграть от таких оптимизаций, предусмотрите разные пути выполнения в зависимости от типа времени выполнения. Для тестирования мед-

ленного пути всегда можно вызвать `Select(x => x)` на `List<T>`, чтобы извлечь последовательность, отличную от списка. Тип `LinkedList<T>` может тестировать случай, когда нужен объект `ICollection<T>`, который не реализует `IList<T>`.

Документация

Важно документировать то, что ваш код делает с входными данными, и также ожидаемую производительность операции. Это особенно важно, если метод должен работать с несколькими последовательностями: какая из них будет обработана первой и насколько? Должен ли код организовать поток для данных, буферизировать их или совмещать то и другое? Какое выполнение применяется — отложенное или немедленное? Могут ли какие-то параметры принимать значение `null`, и если так, то имеет ли это особый смысл?

Выполнение однократного прохода, когда это возможно

На уровне интерфейса в `IEnumerable<T>` разрешено проходить по одной и той же последовательности множество раз — потенциально активных итераторов одновременно может быть несколько. Однако это редко оказывается удачной идеей. Когда только возможно, благоразумно проходить по входным последовательностям только один раз. Это означает, что код будет работать даже для невоспроизводимых последовательностей, таких как строки, читаемые из сетевого потока. Если читать последовательность несколько раз действительно необходимо (и нежелательно буферизировать всю последовательность целиком, как это делает `Reverse()`), в документации вы должны привлечь к этому особое внимание.

Обеспечение освобождения итераторов

В большинстве случаев для прохождения по источнику данных можно использовать оператор `foreach`. Но временами удобно трактовать первый элемент по-другому, и тогда применение итератора напрямую может привести к более простому коду. В такой ситуации не забудьте предусмотреть блок `using` для итератора. Возможно, вы не привыкли освобождать итераторы самостоятельно, поскольку обычно это делает оператор `foreach`, что может затруднить выявление ошибки.

Поддержка специальных сравнений

Многие операции LINQ имеют перегруженные версии, которые позволяют указывать подходящую реализацию `IEqualityComparer<T>` или `IComparer<T>`. Если вы строите библиотеку универсального назначения для других (потенциально для контактирующих с вами разработчиков), может оказаться полезным предоставление аналогичных перегруженных версий. С другой стороны, если вы являетесь единственным пользователем данной библиотеки или ее использование планируется в рамках команды, членом которой вы являетесь, то это можно делать в стиле "реализовать при необходимости". Тем не менее, это легко: обычно более простые перегруженные версии просто вызывают более сложные версии, передавая `EqualityComparer<T>.Default` или `Comparer<T>.Default` в качестве сравнения.

А теперь давайте проверим, не расходятся ли слова с делом.

12.6.2. Пример расширения: выборка случайного элемента

Цель рассматриваемого здесь метода проста: для заданной последовательности и экземпляра класса `Random` возвратить случайный элемент из этой последовательности.

Можно было бы добавить перегруженную версию, не требующую экземпляра Random, но я предпочитаю делать зависимость он генератора случайных чисел явной. По разным причинам случайность — тема сложная, и вместо обсуждения ее здесь я предлагаю ознакомиться с моей статьей (на английском языке), размещенной на веб-сайте книги (http://mng.bz/h483). Кроме того, чтобы сэкономить место, я не включил XML-документацию и модульные тесты в листинг 12.17, но все это доступно в загружаемом коде.

Листинг 12.17. Расширяющий метод для выбора случайного элемента из последовательности

```
public static T RandomElement<T>(this IEnumerable<T> source,
                                 Random random)
{
  if (source == null)                                    ←❶ Проверка достоверности аргументов
  {
    throw new ArgumentNullException("source");
  }
  if (random == null)
  {
    throw new ArgumentNullException("random");
  }
  ICollection collection = source as ICollection;        ←❷ Оптимизация для коллекций
  if (collection != null)
  {
    int count = collection.Count;
    if (count == 0)
    {
      throw new InvalidOperationException("Sequence was empty.");
    }
    int index = random.Next(count);
    return source.ElementAt(index);     ← Дальнейшая оптимизация ElementAt()
  }
  using (IEnumerator<T> iterator = source.GetEnumerator()) ←❸ Обработка медленного случая
  {
    if (!iterator.MoveNext())
    {
      throw new InvalidOperationException("Sequence was empty.");
    }
    int countSoFar = 1;
    T current = iterator.Current;
    while (iterator.MoveNext())
    {
      countSoFar++;
      if (random.Next(countSoFar) == 0)      ←❹ Замена текущего предположения с подходящей вероятностью
      {
        current = iterator.Current;
      }
    }
    return current;
  }
```

В листинге 12.17 не демонстрируется прием разбиения расширяющего метода на проверку достоверности аргументов и реализацию, поскольку итераторный блок в нем отсутствует. Чтобы увидеть пример этого, взгляните еще раз на реализацию операции `Where()` в разделе 10.3.3. Никакие специальные сравнения не требуются, не говоря уже о том, что соблюдаются все пункты из приведенного выше контрольного списка.

Сначала аргументы проверяются на предмет достоверности очевидным путем ❶. Все становится более интересным, когда исходная последовательность реализует `ICollection` ❷[9]. Это позволяет получить счетчик с минимальными затратами и затем сгенерировать одиночное случайное число для выяснения, какой элемент должен быть выбран. Случай, когда исходная последовательность реализует `IList<T>`, *явно* не обрабатывается; взамен это поручается `ElementAt()` (согласно его документации).

Имея дело с последовательностью, отличной от коллекции (такой как результат выполнения другой операции запроса), хотелось бы избежать подсчета и затем выбора элемента; это потребовало бы либо буферизации содержимого последовательности, либо прохода по ней два раза. Вместо этого она проходится один раз за счет явной выборки итератора ❸, так что можно легко проверить, не пуста ли последовательность. Умелый прием[10] предпринят в строке ❹ — здесь текущее предположение о случайном элементе заменяется элементом из итератора с вероятностью $1/n$, где n представляет собой количество элементов, встреченных до сих пор. В результате вероятность замены первого элемента вторым составляет $^1/_2$, вероятность замены результата после двух элементов третьим элементом — $^1/_3$ и т.д. Финальный результат заключается в том, что все элементы последовательности имеют одинаковые шансы быть выбранными, и удалось обеспечить проход только один раз.

Конечно, важным моментом является отнюдь не то, что делает этот отдельный метод — такие проблемы должны были рассматриваться при его реализации. Зная, на что обращать внимание, реализация надежного метода наподобие этого на самом деле не потребует больших усилий, и ваш личный набор инструментов со временем будет расти.

12.7. Резюме

Уф! Эта глава оказалась полной противоположностью большинству других глав книги. Вместо подробного рассмотрения какой-то одной темы был раскрыт целый ряд технологий LINQ, но на поверхностном уровне.

Я изначально не предполагал, что вы сумеете стать специалистом по какой-то из затронутых здесь технологий, но надеюсь, что вы обрели более глубокое понимание причин важности LINQ. Речь идет не о языке XML, запросах в памяти, запросах SQL, наблюдаемых объектах или перечислителях, а о согласованности выражения и предоставлении компилятору C# возможности проверки достоверности запросов, по крайней мере, до некоторой степени, независимо от их целевой платформы выполнения.

Теперь вы должны понимать, почему деревья выражений настолько важны, ведь они входят в то небольшое число элементов *инфраструктуры*, о которых компилятор C# осведомлен непосредственно (наряду со строками, `IDisposable`, `IEnumerable<T>` и `Nullable<T>`, например). Они действуют в качестве паспортов, разрешая поведению

---

[9] Загружаемый код содержит одну и ту же проверку для реализаций `ICollection<T>`, как это делает метод `Count()` в .NET 4. Это в точности тот же самый блок кода, просто с другим типом и именем переменной.

[10] Я спокойно заявляю, что это умелый прием, поскольку это не моя идея, хотя и моя реализация.

выходить за границы локальной машины и выражая логику на иностранном языке, понятном для поставщика LINQ.

Важны не только деревья выражений — мы также полагаемся на трансляцию выражений запросов, выполняемую компилятором, и способ, которым лямбда-выражения могут быть преобразованы в делегаты и деревья выражений. Кроме того, важны и расширяющие методы, т.к. без них каждый поставщик вынужден был бы предоставлять реализации всех значимых методов. Взглянув снова на все новые средства языка C#, вы обнаружите, что лишь немногие из них в той или иной степени не ориентированы на значительную поддержку LINQ. Это одна из причин написания данной главы: нужно было показать связи между всеми такими средствами.

Тем не менее, не следует говорить о LINQ слишком восторженно. Помимо положительных качеств LINQ вы видели и затруднения. Язык LINQ не всегда позволяет выражать все необходимое в запросе, равно как и не скрывает *все* детали лежащего в основе источника данных. Когда дело доходит до поставщиков LINQ для баз данных, несоответствия принципов, которые создавали разработчикам проблемы в прошлом, по-прежнему никуда не делись: их влияние можно уменьшить с помощью систем ORM и тому подобного, но без надлежащего понимания выполняемого запроса, скорее всего, возникнут крупные проблемы. В частности, не рассматривайте язык LINQ как возможность устранения необходимости в понимании SQL — думайте о нем как о способе сокрытия кода SQL, когда вас не интересуют детали. Подобным же образом для планирования эффективного параллельного запроса вы должны знать, где упорядочение имеет значение, а где нет, и по возможности немного помочь инфраструктуре, предоставляя дополнительную информацию.

С момента выхода .NET 3.5 я с удовольствием наблюдал, насколько искренне сообщество приняло LINQ. Кроме того, есть много интересных случаев использования средств C# 4, которые вы увидите в следующей части книги.

Часть IV

C# 4: изящная игра с другими

Язык C# 4 — забавное животное. Его нельзя охарактеризовать как "имеющего несколько в основном не связанных крупных новых средств" подобно C# 2, ни как "все ради LINQ" подобно C# 3. Вместо этого новые средства C# 4 как бы попадают между указанными двумя характеристиками. Главной темой является способность к взаимодействию, но многие средства в равной степени полезны, даже если вам никогда не придется работать с другими средами.

Лично мне больше всего нравятся два средства из C# 4 — необязательные параметры и именованные аргументы. Они относительно просты, но могут найти применение во многих местах, улучшая читабельность кода и в целом делая жизнь более приятной. Тратите понапрасну много времени на выяснение, что конкретно означает тот или иной аргумент? Назначьте им имена. Устали от написания бесконечного числа перегруженных версий, чтобы не заставлять указывать в вызывающем коде абсолютно все аргументы? Сделайте некоторые параметры необязательными.

Если вы имеете дело с COM, то C# 4 станет для вас буквально глотком чистого воздуха. Начнем с того, что описанные выше средства намного упрощают работу с рядом API-интерфейсов, в которых проектировщики компонентов предполагали, что вы будете пользоваться языком, поддерживающим необязательные параметры и именованные аргументы. Помимо этого имеется улучшенное развертывание, поддержка именованных индексаторов и полезное сокращение, позволяющее избежать повсеместной передачи аргументов по ссылке. Крупнейшее средство C# 4 — динамическая типизация — также способствует упрощению интеграции с COM.

Все эти темы мы рассмотрим в главе 13 наряду с заумной темой обобщенной вариантности применительно к интерфейсам и делегатам. Не волнуйтесь: мы не будем торопиться, к тому же большую часть времени знание деталей не понадобится — все это обеспечивает работоспособность кода, которую в любом случае можно было ожидать и от версии C# 3.

В главе 14 раскрывается динамическая типизация и исполняющая среда динамического языка (Dynamic Language Runtime — DLR). Это громадная тема. Основное внимание будет сосредоточено на том, как динамическая типизация реализована в языке C#, но мы также рассмотрим несколько примеров взаимодействия с динамическими языками наподобие IronPython, а также примеры того, каким образом тип может динамически реагировать на вызовы методов, доступ к свойствам и т.д. Здесь уместно сделать небольшое замечание: тот факт, что это является крупным средством, вовсе не означает, что вы начнете неожиданно обнаруживать динамические выражения повсюду в своем коде. Данное средство не настолько распространено, как, например, LINQ, но когда динамическая типизация действительно нужна, вы сможете удостовериться в том, насколько она удачно реализована в C# 4.

Глава 13

Небольшие изменения, направленные на упрощение кода

В этой главе...

- Необязательные параметры
- Именованные аргументы
- Модернизация параметров `ref` в COM
- Встраивание основных сборок взаимодействия с COM
- Вызов именованных индексаторов, объявленных в COM
- Обобщенная вариантность для интерфейсов и делегатов
- Изменения в блокировке и события, подобные полям

Как и в предшествующих версиях, в C# 4 имеется несколько мелких средств, которые не дотягивают до того, чтобы им были посвящены отдельные главы. На самом деле, в C# 4 существует только одно действительно *крупное* средство — динамическая типизация, которое будет раскрыто в следующей главе. Изменения, рассматриваемые в настоящей главе, просто делают язык C# немного более приятным в работе, особенно если вам приходится регулярно иметь дело с COM. Эти средства в целом позволяют писать более ясный код, устраняют монотонную работу, связанную с вызовами COM, или упрощают развертывание.

Заставили ли перечисленные средства ваше сердце биться чаще от волнения? Вряд ли. Тем не менее, эти средства хороши, а некоторые из них могут оказаться широко применимыми. Давайте начнем с того, что посмотрим, как мы вызываем методы.

13.1. Необязательные параметры и именованные аргументы

Необязательные параметры и именованные аргументы, пожалуй, можно назвать Бэтменом и Робином[1] в C# 4. Они отдельны друг от друга, но обычно встречаются вместе. Я пока буду держать их отдельно, чтобы с ними можно было ознакомиться по очереди, а затем эти средства будут использоваться вместе в более интересных примерах.

Параметры и аргументы

В этом разделе вполне очевидно много раз будут упоминаться параметры и аргументы. В неофициальной беседе эти два термина часто применяются взаимозаменяемо, но я буду использовать их согласно формальным определениям. Просто в качестве напоминания: *параметр* (также известный как *формальный параметр*) — это переменная, которая является частью объявления метода или индексатора. *Аргумент* — это выражение, применяемое при вызове метода или индексатора. Для примера взгляните на следующий фрагмент кода:

```
void Foo(int x, int y)
{
  // Делать что-то с x и y
}
...
int a = 10;
Foo(a, 20);
```

Параметрами здесь являются `x` и `y`, а аргументами — `a` и `20`.

Сначала мы рассмотрим необязательные параметры.

13.1.1. Необязательные параметры

Необязательные параметры существовали в Visual Basic на протяжении многих лет, и они присутствовали в среде CLR, начиная с .NET 1.0. Концепция здесь проста: некоторые параметры являются необязательными, поэтому их значения не требуется явно указывать в вызывающем коде. Любой параметр, для которого в вызывающем коде не предусмотрен аргумент, получает свое стандартное значение.

Мотивация

Необязательные параметры обычно используются, когда для операции должно быть указано множество значений, и на протяжении длительного времени применяются одни и те же значения. Например, предположим, что необходимо прочитать текстовый файл; может понадобиться предоставить метод, который позволяет вызывающему коду задавать имя файла и используемую кодировку. Однако кодировкой почти всегда является UTF-8, так что было бы неплохо иметь возможность применять ее автоматически, если именно она и нужна. Исторически сложилось так, что идиоматическим подходом для достижения этой цели в C# была перегрузка метода: объявление одного метода со всеми возможными параметрами и других методов, которые вызывают первый метод, передавая ему подходящие стандартные значения.

[1] Или если вы — знаток высокой культуры, то короткие оперы "Сельская честь" и "Паяцы" (которые во многих театрах играются вместе, в один вечер).

Например, методы могут быть созданы следующим образом:

```
public IList<Customer> LoadCustomers(string filename,
                                     Encoding encoding)
{
  ...                                          ← Выполнение реальной работы
}
public IList<Customer> LoadCustomers(string filename)
{
  return LoadCustomers(filename, Encoding.UTF8); ← Использование UTF-8 в качестве стандартного значения
}
```

Это хорошо работает для одного параметра, но становится сложным, когда вариантов много, т.к. дополнительный вариант удваивает количество возможных перегруженных версий. Если два параметра имеют один и тот же тип, такой подход естественным образом приводит к нескольким методам с одинаковыми сигнатурами, что недопустимо. Часто один набор перегруженных версий также требуется для нескольких типов параметров. Например, метод `XmlReader.Create()` может создавать объект `XmlReader` из экземпляра `Stream`, `TextReader` или `string`, но он также предоставляет возможность указания `XmlReaderSettings` и других аргументов. Из-за такого дублирования для метода предусмотрено 12 перегруженных версий.

За счет использования необязательных параметров их количество можно было бы значительно сократить. Давайте посмотрим, как это сделать.

Объявление необязательных параметров и пропуск их при указании аргументов

Чтобы сделать параметр необязательным, нужно просто предоставить для него стандартное значение, что похоже на инициализатор переменной. На рис. 13.1 показан метод с тремя параметрами: два необязательных и один обязательный.

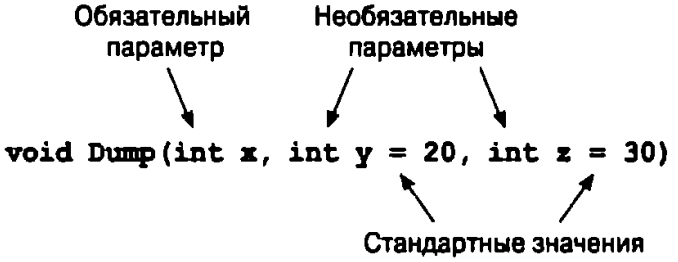

Рис. 13.1. Объявление необязательных параметров

Этот метод всего лишь выводит значения аргументов на консоль, но этого вполне достаточно для оценки происходящего. В листинге 13.1 приведен код с объявлением метода и его тремя вызовами, в каждом из которых указывается разное количество аргументов.

Листинг 13.1. Объявление и вызов метода с необязательными параметрами

```
static void Dump(int x, int y = 20, int z = 30)   ←❶ Объявление метода с необязательными параметрами
{
  Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
...
Dump(1, 2, 3);          ←❷ Вызов метода со всеми аргументами
Dump(1, 2);             ←❸ Пропуск одного аргумента
Dump(1);                ←❹ Пропуск двух аргументов
```

Необязательными параметрами являются те, для которых заданы стандартные значения ❶. Если в вызывающем коде не указан параметр y, его начальным значением будет 20, и подобным же образом параметр z получит начальное значение 30. В первом вызове ❷ все аргументы явно указаны; в последующих вызовах (❸ и ❹) пропущен один или два аргумента, поэтому для них применяются стандартные значения. При отсутствии одного аргумента компилятор предполагает, что был опущен последний параметр, затем предпоследний и т.д. Вывод выглядит следующим образом:

```
x=1 y=2 z=3
x=1 y=2 z=30
x=1 y=20 z=30
```

Обратите внимание, что хотя компилятор *мог бы* использовать какой-то более интеллектуальный анализ типов необязательных параметров и аргументов, чтобы выяснить, какие из них пропущены, он этого не делает, а предполагает, что аргументы указываются в том же порядке, что и параметры[2]. Это означает, что следующий код будет недопустимым:

```
static void TwoOptionalParameters(int x = 10, string y = "default")
{
  Console.WriteLine("x={0} y={1}", x, y);
}
...
TwoOptionalParameters("second parameter"); ◄— Ошибка!
```

В этом коде предпринимается попытка вызвать метод `TwoOptionalParameters()` с указанием строки для *первого* аргумента. Перегруженная версия, в которой первый параметр поддерживает преобразование из строки, отсутствует, поэтому компилятор сообщит об ошибке. И это хорошо — распознавание перегруженной версии достаточно сложно (особенно когда задействовано выведение обобщенных типов) и без опробования всех перестановок в попытках нахождения той, которая *возможно* будет вызвана. Если вы хотите опустить значение для одного необязательного параметра, но указать значение для параметра, который находится дальше в списке, придется прибегнуть к именованным аргументам.

Ограничения на необязательных параметрах

Для необязательных параметров предусмотрено несколько правил. Все необязательные параметры должны располагаться после обязательных параметров. Исключением является *массив параметров* (объявленный с помощью модификатора `params`), который по-прежнему следует помещать в конец списка параметров, но он может находиться после необязательных параметров. Массив параметров нельзя объявлять как необязательный параметр — если в вызывающем коде значения для него не указаны, будет применяться пустой массив. Необязательные параметры не могут иметь модификаторы `ref` или `out`.

Необязательные параметры могут быть любого типа, но существуют ограничения относительно указываемых стандартных значений. Всегда можно использовать константы: числовые и строковые литералы, `null`, члены `const`, члены перечислений и операцию `default(T)`. Кроме того, для типов значений можно вызывать конструктор без параметров, хотя в любом случае это эквивалентно применению операции `default(...)`.

---

[2] Конечно, если не применяются именованные аргументы, о которых вы вскоре узнаете.

Должно быть доступным неявное преобразование типа указанного значения в тип параметра, но оно *не* может быть преобразованием, определяемым пользователем.

В табл. 13.1 приведены некоторые примеры допустимых списков параметров.

Таблица. 13.1. Допустимые списки параметров метода, содержащие необязательные параметры

Объявление	Примечания
`Foo(int x, int y = 10)`	Для стандартного значения используется числовой литерал
`Foo(decimal x = 10)`	Неявное встроенное преобразование из `int` в `decimal`
`Foo(string name = "default")`	Для стандартного значения используется строковый литерал
`Foo(DateTime dt = new DateTime())`	Нулевое значение типа `DateTime`
`Foo(DateTime dt = default(DateTime))`	Альтернативный синтаксис для нулевого значения
`Foo<T>(T value = default(T))`	Операция получения стандартного значения работает с параметрами типов
`Foo(int? x = null)`	Преобразование, допускающее `null`
`Foo(int x, int y = 10, params int[] z)`	Массив параметров после необязательных параметров

Для противовеса в табл. 13.2 показано несколько недопустимых списков параметров с объяснениями, почему они не разрешены.

Таблица. 13.2. Недопустимые списки параметров метода, содержащие необязательные параметры

Объявление (ошибочное)	Примечания
`Foo(int x = 0, int y)`	Обязательный параметр, отличный от `params`, находится после необязательного параметра
`Foo(DateTime dt = DateTime.Now)`	Стандартные значения должны быть константами
`Foo(XName name = "default")`	Преобразование из `string` в `XName` определено пользователем
`Foo(params string[] names = null)`	Массивы параметров не могут быть необязательными
`Foo(ref string name = "default")`	Параметры `ref/out` не могут быть необязательными

Тот факт, что стандартное значение должно быть константой, является недостатком в двух разных ситуациях. Одна из них знакома в несколько ином контексте, который мы сейчас и рассмотрим.

Разные версии и необязательные параметры

Ограничения, налагаемые на стандартные значения для необязательных параметров, могут напоминать ограничения на полях `const` или значениях атрибутов, и они действительно ведут себя очень похоже. В обоих случаях, когда компилятор ссылается на значение, он копирует его напрямую в вывод. Сгенерированный код IL действует в точности, как если бы первоначальный исходный код содержал стандартное значение. Это означает, что если вы когда-либо *измените* стандартное значение без перекомпиляции всего кода, в котором имеются ссылки на него, то старый вызывающий код будет по-прежнему использовать старое стандартное значение.

Для ясности рассмотрим следующую последовательность шагов.

1. Создайте библиотеку классов (`Library.dll`) с показанным ниже классом:

```
public class LibraryDemo
{
   public static void PrintValue(int value = 10)
   {
      System.Console.WriteLine(value);
   }
}
```

2. Создайте консольное приложение (`Application.exe`), которое ссылается на эту библиотеку классов:

```
public class Program
{
   static void Main()
   {
      LibraryDemo.PrintValue();
   }
}
```

3. Запустите приложение — оно вполне предсказуемо выведет на консоль значение 10.

4. Измените объявление метода `PrintValue()` следующим образом и перекомпилируйте ***только*** библиотеку классов:

```
public static void PrintValue(int value = 20)
```

5. Снова запустите приложение — оно по-прежнему выведет на консоль значение 10. Дело в том, что это значение было скомпилировано прямо в исполняемый файл.

6. Перекомпилируйте приложение и запустите его — на этот раз оно выведет на консоль значение 20.

Такая проблема с версиями может привести к ошибкам, которые трудно отслеживать, поскольку весь код *выглядит* корректным. По существу вы ограничены применением подлинных констант, которые никогда не должны изменяться, выступая в качестве стандартных значений для необязательных параметров[3]. Эта система обладает одним преимуществом: она дает вызывающему коду гарантию того, что значение, которое ему известно на этапе компиляции, и является тем значением, которое будет использоваться

[3] Или можно было бы просто согласиться, что при изменении значения необходимо перекомпилировать весь имеющийся код. Во многих контекстах это разумный компромисс.

во время выполнения. Разработчики могут чувствовать себя более комфортно с этим, чем с динамически вычисляемым значением либо тем, которое зависит от версии библиотеки, применяемой во время выполнения.

Конечно, это также означает невозможность использовать значения, которые нельзя выразить в виде констант. Например, вы не можете создать метод со стандартным значением вроде "текущего времени".

Придание стандартным значениям большей гибкости за счет допустимости null

К счастью, существует способ обойти ограничение, касающееся того, что стандартные значения должны быть константами. По сути, для представления стандартного значения вы вводите "магическое" значение и затем внутри самого метода заменяете его *действительным* стандартным значением. Если понятие *"магическое" значение* вызывает у вас беспокойство, я не удивлен, но для этого "магического" значения мы будем применять null, которое уже представляет отсутствие нормального значения. Если типом параметра обычно был бы тип значения, мы просто назначаем ему соответствующий тип значения, допускающий null, после чего можно будет по-прежнему указывать, что стандартным значением является null.

В качестве примера давайте рассмотрим ситуацию, похожую на ту, что я использовал при введении во всю тему: позволим вызывающему коду передавать методу нужную кодировку текста, но установим для нее стандартное значение UTF-8. Вы не можете указать стандартную кодировку как Encoding.UTF8, потому что это не константное значение, но можете трактовать значение null параметра как "применить стандартное значение". Для демонстрации обработки типов значений создадим метод, добавляющий метку времени к текстовому файлу с сообщением. Мы установим для кодировки стандартное значение UTF-8, а для метки времени — текущее время. В листинге 13.2 приведен полный исходный код и несколько примеров использования метода.

Листинг 13.2. Применение стандартных значений null для обработки неконстантных ситуаций

```
static void AppendTimestamp(string filename,          <-- Два обязательных параметра
                            string message,
                            Encoding encoding = null  <--❶ Два необязательных параметра
                            DateTime? timestamp = null)
{
  Encoding realEncoding = encoding ?? Encoding.UTF8;  <--❷ Применение операции объединения с null для удобства

  DateTime realTimestamp = timestamp ?? DateTime.Now;
  using (TextWriter writer = new StreamWriter(filename,
                                              true,
                                              realEncoding))
  {
    writer.WriteLine("{0:s}: {1}", realTimestamp, message);
  }
}
...
AppendTimestamp("utf8.txt", "First message");
AppendTimestamp("ascii.txt", "ASCII", Encoding.ASCII);
AppendTimestamp("utf8.txt", "Message in the future", null,  <--❸ Явное использование null
                new DateTime(2030, 1, 1));
```

В листинге 13.2 иллюстрируется несколько изящных возможностей этого подхода. Прежде всего, он решает проблему версий. Стандартными значениями для необязательных параметров являются `null` ❶, но *действительными* — "кодировка UTF-8" и "текущая дата и время". Ни один из них не может быть выражен в виде константы, и если когда-нибудь вы решите изменить действующее стандартное значение (например, с целью использования текущего времени UTC взамен местного), то сможете сделать это без перекомпиляции всего кода, в котором вызывается метод `AppendTimestamp()`. Разумеется, изменение действующего стандартного значения поменяет поведение метода; этому должно уделяться такое же внимание, как и любым другим изменениям в коде. На данной стадии версиями управляете вы сами (как автор библиотеки) — в сущности, вы берете на себя ответственность за то, чтобы код у клиентов не нарушил свою работу. Во всяком случае, это более знакомая территория; вам известно, что весь вызывающий код получит одно и то же поведение, невзирая на перекомпиляцию.

В листинге 13.2 также вводится дополнительный уровень гибкости. Мало того, что необязательные параметры подразумевают возможность сделать вызовы более короткими, но наличие специального значения типа "применить стандартное значение" означает, что при желании можно *явно* делать вызов, позволяя методу выбрать подходящее значение. В данный момент это единственный известный вам способ явного указания метки времени без предоставления также и кодировки ❷, но ситуация изменится, когда мы приступим к рассмотрению именованных аргументов.

Благодаря операции объединения с `null`, со значениями необязательных параметров работать легко ❸. Из-за оформления печатной страницы в этом примере применялись отдельные переменные, но в реальном коде, скорее всего, те же самые выражения будут использоваться прямо в вызовах конструктора типа `StreamWriter` и метода `WriteLine()`.

С данным подходом связаны два недостатка. Прежде всего, если вызывающий код *непреднамеренно* передаст значение `null` по причине ошибки, вместо генерации исключения будет просто выбрано стандартное значение. В случаях, когда вы применяете тип значения, допускающий `null`, а в вызывающем коде будет либо явно использоваться `null`, либо присутствовать аргумент типа, не допускающего `null`, это не является значительной проблемой, но для ссылочных типов проблема может возникнуть.

Вдобавок это требует отказа от применения `null` как "настоящего" значения[4]. Иногда необходимо, чтобы *`null`* считалось именно значением *`null`*, и если вы не желаете использовать его в качестве стандартного значения, то должны будете найти другую константу или просто оставить параметр обязательным. Однако в прочих ситуациях, когда отсутствует очевидное константное значение, которое четко и *всегда* будет правильным стандартным значением, я рекомендую данный подход к объявлению необязательных параметров как такой, которому легко следовать согласованным образом и избавиться от ряда обычных трудностей.

Мы еще должны посмотреть, как необязательные параметры влияют на распознавание перегруженных версий, но имеет смысл отложить это до того, как мы разберем работу именованных аргументов.

---

[4] Нам почти наверняка понадобится второе специальное значение, подобное `null`, которое имеет следующий смысл: "использовать для этого параметра стандартное значение". Затем можно было бы позволить этому специальному значению либо передаваться автоматически для пропущенных аргументов, либо явно указываться в списке аргументов. Уверен, что это привело бы к десяткам проблем, но поэкспериментировать было бы интересно.

13.1.2. Именованные аргументы

Базовая идея именованных аргументов заключается в том, что при передаче значения аргумента можно также указать имя параметра, для которого предназначено это значение. Затем компилятор проверяет, *есть* ли параметр с таким именем, и применяет для него заданное значение. Это средство даже само по себе может улучшить читабельность в ряде случаев. На практике именованные аргументы наиболее удобны в ситуациях, когда также используются и необязательные параметры, но мы рассмотрим сначала простой случай.

Индексаторы, необязательные параметры и именованные аргументы

Необязательные параметры и именованные аргументы *можно* применять с индексаторами, как и с методами. Однако это полезно только для индексаторов с более чем одним параметром: как бы то ни было, нельзя получить доступ к индексатору, не указав хотя бы один аргумент. Учитывая такое ограничение, я не ожидаю встретить много случаев использования этого средства с индексаторами и не демонстрирую их в главе. Тем не менее, все работает вполне предсказуемо.

Я не сомневаюсь, что вам приходилось видеть код, который выглядит примерно так:

```
MessageBox.Show("Please do not press this button again", // текст
                "Ouch!"); // заголовок
```

Выбранный пример довольно скучный, но все может стать намного хуже при большем количестве аргументов, особенно если многие из них имеют тот же самый тип. Тем не менее, он по-прежнему реалистичен; даже при наличии всего лишь двух параметров мне было бы трудно понять смысл каждого аргумента, если бы не присутствующие комментарии. Тем не менее, проблема остается: комментарии могут сообщать некорректные сведения о коде, который они описывают. Не существует средства для их проверки. В противоположность этому, именованные аргументы запрашивают помощь у компилятора.

Синтаксис

Все, что понадобится сделать для прояснения кода в предыдущем примере – снабдить каждый аргумент префиксом в форме имени соответствующего параметра и двоеточием:

```
MessageBox.Show(text: "Please do not press this button again",
                caption: "Ouch!");
```

Нельзя сказать, что выбранные имена являются наиболее осмысленными (я предпочел бы `title`, а не `caption`), но, во всяком случае, теперь вы будете знать, когда что-то пойдет не так.

Конечно, самым распространенным способом заставить что-то пойти не так будет указание аргументов в неправильном порядке. Не будь именованных аргументов, это стало бы проблемой: порции текста поменялись бы местами в диалоговом окне сообщения. Благодаря именованным аргументам, порядок их следования становится неважным. Показанный выше код можно переписать следующим образом:

```
MessageBox.Show(caption: "Ouch!",
                text: "Please do not press this button again");
```

Текст по-прежнему будет находиться в правильных местах диалогового окна, т.к. компилятор выяснит, что вы имели в виду, на основе имен.

В качестве еще одного примера взгляните на вызов конструктора типа `StreamWriter` в листинге 13.2. Во втором аргументе задано просто `true` — что оно означает? Поток должен принудительно сбрасываться после каждой записи? Включить маркер порядка следования байтов? Дополнять существующий файл вместо создания нового? Ниже представлен эквивалентный вызов, в котором применяются именованные аргументы:

```
new StreamWriter(path: filename,
                 append: true,
                 encoding: realEncoding)
```

В обоих примерах вы видели, что именованные аргументы фактически придают значениям семантический *смысл*. В нескончаемом поиске способов сделать код воспринимаемым людьми так же хорошо, как компьютерами, это несомненный шаг вперед.

Разумеется, я не предлагаю использовать именованные аргументы там, где смысл очевиден и без них. Подобно любому средству, они должны применяться осмотрительно и взвешенно.

Именованные аргументы с модификаторами `out` и `ref`

Если необходимо указать имя аргумента для параметра `ref` или `out`, модификатор `ref` или `out` помещается после имени и перед аргументом. Используя метод `int.TryParse()` в качестве примера, можно написать следующий код:

```
int number;
bool success = int.TryParse("10", result: out number);
```

Для исследования ряда других аспектов синтаксиса в листинге 13.3 показан метод с тремя целочисленными параметрами, подобный тому, что применялся в начале описания необязательных параметров.

Листинг 13.3. Простые примеры использования именованных аргументов

```
static void Dump(int x, int y, int z)  <-- ❶ Объявление метода как обычно
{
  Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
...
Dump(1, 2, 3);                          <-- ❷ Вызов метода как обычно
Dump(x: 1, y: 2, z: 3);                 <-- ❸ Указание имен для всех аргументов
Dump(z: 3, y: 2, x: 1);
Dump(1, y: 2, z: 3);                    <-- ❹ Указание имен для некоторых аргументов
Dump(1, z: 3, y: 2);
```

Все вызовы в листинге 13.3 дают один и тот же вывод: `x=1, y=2, z=3`. Фактически в коде осуществляется обращение к одному методу пятью разными способами. Полезно отметить, что никаких трюков при объявлении метода не предпринималось ❶; именованные аргументы можно применять с любым методом, принимающим параметры. Сначала метод вызывается как обычно, без использования новых средств ❷. Это своего рода контрольная точка, позволяющая удостовериться, что все остальные вызовы на самом деле эквивалентны. Затем делаются два обращения к методу с применением только

именованных аргументов ❶. Во втором вызове порядок следования аргументов изменен на обратный, но результат по-прежнему тот же самый, поскольку аргументы сопоставляются с параметрами по именам, а не позициям. Наконец, в последних двух вызовах используется смесь из именованных и *позиционных аргументов* ❷. Позиционный аргумент — это такой, для которого не указано имя, поэтому любой аргумент в допустимом коде C# 3 с точки зрения C# 4 является позиционным.

На рис. 13.2 показано, как работает последняя строка кода.

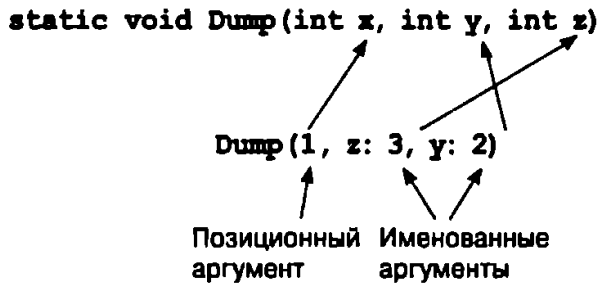

Рис. 13.2. Позиционные и именованные аргументы в одном и том же вызове

Все именованные аргументы должны располагаться после позиционных аргументов — произвольно переключаться между стилями нельзя. Позиционные аргументы *всегда* ссылаются на соответствующие параметры в объявлении метода — невозможно пропустить параметр, не указав для него позиционный аргумент, но задать его позже с помощью именованного аргумента. Это означает, что следующие вызовы метода будут недопустимыми.

- `Dump(z: 3, 1, y: 2)`. Позиционные аргументы должны находиться перед именованными.
- `Dump(2, x: 1, z: 3)`. Поскольку значение для `x` уже указано в первом позиционном аргументе, задавать его снова посредством именованного аргумента не разрешено.

Хотя в *конкретном случае*, показанном в листинге 13.3, вызовы метода эквивалентны, так происходит не *всегда*. Давайте посмотрим, почему переупорядочение аргументов может изменить поведение.

Порядок оценки аргументов

Вы привыкли к тому, что в C# аргументы оцениваются в порядке их указания, и до выхода версии C# 4 это всегда был порядок, в котором определены параметры. В C# 4 справедлива только первая часть утверждения: аргументы по-прежнему оцениваются в порядке их написания, даже если он не совпадает с порядком объявления параметров. Это имеет значение, если оцениваемые аргументы имеют побочные эффекты.

Как правило, необходимо по возможности избегать наличия побочных эффектов в аргументах, но есть случаи, когда это может сделать код яснее. Более реалистичное правило заключается в том, чтобы пытаться избегать побочных эффектов, которые могут пересекаться друг с другом. Для демонстрации порядка выполнения мы нарушим оба эти правила. Ни в коем случае не считайте это рекомендацией поступать аналогичным образом.

Первым делом, мы создадим относительно безобидный пример, введя метод, который выводит на консоль свои входные данные и возвращает их — своего рода регистрирующую эхо-службу. Мы будем применять значения, возвращаемые тремя вызовами, при обращении к методу `Dump()` (который здесь не показан, т.к. он не менялся). В листинге 13.4 представлены два вызова `Dump()`, которые дают в результате отличающийся вывод.

Листинг 13.4. Оценка аргументов

```
static int Log(int value)
{
  Console.WriteLine("Log: {0}", value);
  return value;
}
...
Dump(x: Log(1), y: Log(2), z: Log(3));
Dump(z: Log(3), x: Log(1), y: Log(2));
```

Результаты выполнения кода из листинга 13.4 показывают, что случилось:

```
Log: 1
Log: 2
Log: 3
x=1 y=2 z=3
Log: 3
Log: 1
Log: 2
x=1 y=2 z=3
```

В обоих случаях параметры x, y и z в методе `Dump()` имеют значения 1, 2 и 3, соответственно. Но можно заметить, что хотя в первом вызове они были оценены именно в таком порядке (который был эквивалентным использованию позиционных аргументов), во втором вызове сначала оценивалось значение для параметра z.

Воздействие можно сделать даже более значительным за счет применения побочных эффектов, которые изменяют результат оценки аргументов, как показано в листинге 13.5, где снова используется тот же самый метод `Dump()`.

Листинг 13.5. Неправильный порядок оценки аргументов

```
int i = 0;
Dump(x: ++i, y: ++i, z: ++i);
i = 0;
Dump(z: ++i, x: ++i, y: ++i);
```

Результатом кода из листинга 13.5 может быть кровавая сцена убийства, когда кто-то, кому доведется сопровождать такой код, будет испытывать желание последовать за его автором с топором. Да, говоря формально, последняя строка приводит к получению x=2 y=3 z=1, но я уверен, вы понимаете, что я имел в виду. Просто скажем "нет" коду подобного рода. Разумеется, необходимо упорядочивать свои аргументы для достижения лучшей читабельности. Например, вы можете решить, что оформление вызова `MessageBox.Show()` с расположением заголовка над текстом в самом коде более точно отражает компоновку на экране. Тем не менее, если вы хотите опираться на специфичный порядок оценки аргументов, предусмотрите несколько локальных переменных, чтобы подходящий код выполнялся в отдельных операторах. Компилятор на это не обратит внимания, поскольку он будет следовать правилам спецификации, но такой прием позволяет снизить риск "безобидного рефакторинга", который неумышленно вводит тонкую ошибку.

Давайте займемся более приятными делами — объединим эти два средства (необязательные параметры и именованные аргументы) и посмотрим, насколько аккуратнее может стать код.

13.1.3. Объединение двух средств

Необязательные параметры и именованные аргументы могут работать в тандеме безо всяких дополнительных усилий с вашей стороны. Нередки случаи, когда есть группа параметров, для которых имеются очевидные стандартные значения, но трудно предугадать, какие из них будут указаны явно в вызывающем коде. На рис. 13.3 показаны почти все комбинации: обязательный параметр, два обязательных параметра, позиционный аргумент, именованный аргумент и пропуск аргумента для необязательного параметра.

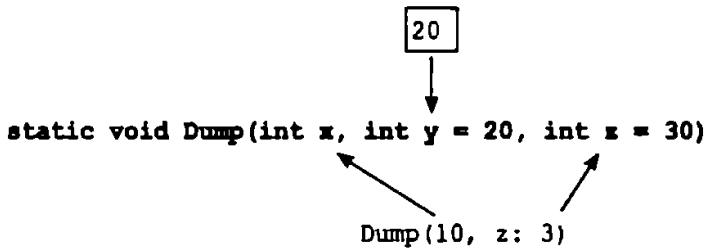

Рис. 13.3. Смешивание именованных аргументов и необязательных параметров

Возвращаясь к более раннему примеру, в листинге 13.2 было необходимо добавить метку времени в файл с применением стандартной кодировки UTF-8, однако конкретной метки времени. В этом коде для аргумента кодировки использовалось значение `null`, но теперь мы можем записать тот же код более просто, как демонстрируется в листинге 13.6.

Листинг 13.6. Объединение именованных аргументов и необязательных параметров

```
static void AppendTimestamp(string filename,
                            string message,
                            Encoding encoding = null,
                            DateTime? timestamp = null)
{
    ← Та же реализация, что и ранее
}
...
AppendTimestamp("utf8.txt", "Message in the future", ← Аргумент кодировки не указан
            timestamp: new DateTime(2030, 1, 1)); ← Именованный аргумент для метки времени
```

В этой довольно простой ситуации выгода не особенно велика, но в случаях, когда нужно пропустить три или четыре аргумента, однако указать один последний, это действительно бесценно.

Вы видели, что необязательные параметры сокращают потребность в очень длинных списках перегруженных версий, но есть один специфичный пример, о котором стоит упомянуть, и связан он с неизменяемостью.

Неизменяемость и инициализация объектов

Один аспект версии C# 4 меня несколько разочаровывает — не было предпринято особо много *явных* усилий, чтобы сделать неизменяемость более простой. Неизменяемые типы являются основной частью функционального программирования, и в C# постепенно появлялась все большая и большая поддержка функционального стиля, но исключая неизменяемость.

Инициализаторы объектов и коллекций упрощают работу с *изменяемыми* типами, а неизменяемые типы остались с носом. (Автоматически реализуемые свойства также попадают в эту категорию.) К счастью, хотя они не проектировались специально для поддержания неизменяемости, именованные аргументы и необязательные параметры позволяют писать код, подобный инициализатору объекта, который обращается к конструктору или другому фабричному методу.

Например, предположим, что вы создали класс Message, который требует указания адреса *отправителя*, адреса *получателя* и *тела* сообщения, а также необязательной темы и вложения. (Для простоты будем придерживаться варианта с единственным получателем.) Вы *могли бы* создать изменяемый тип с подходящими записываемыми свойствами и конструировать экземпляры приблизительно так:

```
Message message = new Message {
  From = "skeet@pobox.com",
  To = "csharp-in-depth-readers@everywhere.com",
  Body = "Hope you like the third edition",
  Subject = "A quick message"
};
```

Здесь присутствуют две проблемы. Во-первых, такой прием никак не принуждает предоставлять обязательные данные. Вы могли бы заставить указывать обязательные данные в конструкторе, но тогда (до выхода C# 4) смысл аргументов не был бы очевидным:

```
Message message = new Message(
  "skeet@pobox.com",
  "csharp-in-depth-readers@everywhere.com",
  "Hope you like the third edition")
{
  Subject = "A quick message"
};
```

Во-вторых, такой шаблон инициализации просто не будет работать с неизменяемыми типами. Компилятор должен вызвать средство установки свойства *после* того, как он инициализировал объект.

Но необязательные параметры и именованные аргументы можно применять для того, чтобы приблизиться к чему-то, что обладает полезными характеристиками в первой форме (указание только интересующих данных и предоставление имен), не теряя проверку, какие аспекты сообщения являются обязательными, или преимущества неизменяемости. В листинге 13.7 показана возможная сигнатура конструктора и шаг конструирования для представленного ранее сообщения.

Листинг 13.7. Использование необязательных параметров и именованных аргументов для неизменяемости

```
public Message(string from, string to,
               string body, string subject = null,
               byte[] attachment = null)
{
    <-- Здесь находится обычный код инициализации
}
...
```

```
Message message = new Message(
   from: "skeet@pobox.com",
   to: "csharp-in-depth-readers@everywhere.com",
   body: "I hope you like the third edition",
   subject: "A quick message"
);
```

Мне действительно нравится такой подход в плане читабельности и полной аккуратности. Не требуются сотни перегруженных версий конструктора — нужна лишь одна, в которой определенные параметры сделаны необязательными. Тот же самый синтаксис будет также работать со статическими методами создания, отличными от инициализаторов объектов. Единственный недостаток в том, что подход основан на том, что ваш код написан на языке, который поддерживает необязательные параметры и именованные аргументы; иначе в вызывающем коде придется предпринимать неуклюжие действия с целью указания значений для всех необязательных параметров. Очевидно, что неизменяемость не ограничивается получением значений от кода инициализации, однако это долгожданный шаг в правильном направлении.

Прежде чем переходить к вопросам, связанным с COM, осталось рассмотреть пару аспектов, касающихся деталей того, как компилятор обрабатывает ваш код, и трудности качественного проектирования API-интерфейсов.

Распознавание перегруженных версий

Вполне понятно, что именованные аргументы и необязательные параметры влияют на то, как компилятор распознает перегруженные версии — если доступно сразу несколько сигнатур метода с одним и тем же именем, то какая из них должна быть выбрана? Необязательные параметры могут увеличить количество подходящих методов (если некоторые методы имеют большее число параметров, чем указано аргументов), а именованные аргументы — уменьшить количество подходящих методов (за счет исключения методов, не имеющих параметров с соответствующими именами).

Большей частью изменения интуитивно понятны: чтобы проверить, является ли отдельный метод подходящим, компилятор пытается построить список аргументов, которые *могли бы* передаваться, используя позиционные аргументы по порядку, а затем сопоставляя именованные аргументы с оставшимися параметрами. Если обязательный параметр не был указан или именованный аргумент не соответствует ни одному из оставшихся параметров, такой метод считается неподходящим. Дополнительные сведения об этом можно найти в разделе 7.5.3 спецификации, но есть две ситуации, которые заслуживают особого внимания.

Первая ситуация возникает, когда два метода оказываются подходящими, но в одном предоставлены *все* аргументы явным образом, тогда как в другом применяется необязательный параметр, заполняемый стандартным значением. Преимущество получает метод, в котором не используются какие-либо стандартные значения. Однако это не сводится просто к сравнению количества применяемых стандартных значений — это строгое разделение в форме "используются стандартные значения или нет". Например, взгляните на следующий код:

```
static void Foo(int x = 10) {}
static void Foo(int x = 10, int y = 20) {}
...
Foo();          <-- 1 Ошибка: неоднозначность
Foo(1);         <-- 2 Вызов первой перегруженной версии
Foo(y: 2);      <-- 3 Вызов второй перегруженной версии
Foo(1, 2);      <-- 4 Вызов второй перегруженной версии
```

В первом вызове ❶ оба метода оказываются подходящими из-за своих необязательных параметров. Но компилятор не может выяснить, какой из них вы намеревались вызвать, и сообщит об ошибке. Во втором вызове ❷ оба метода по-прежнему подходят, но используется первая перегруженная версия, поскольку она может быть применена без использования стандартных значений, в то время как во второй перегруженной версии применяется стандартное значение для y. В случае третьего и четвертого вызовов подходящей будет только вторая перегруженная версия. В третьем вызове ❸ присутствует именованный аргумент y, а в четвертом ❹ указаны два аргумента; все это означает, что первая перегруженная версия не может использоваться.

Перегруженные версии и наследование не всегда хорошо сочетаются

Все это предполагает, что компилятор найдет множество перегруженных версий для выбора среди них подходящей. Если некоторые методы объявлены в базовом типе, но в каком-то из производных типов имеются подходящие методы, то предпочтение будет отдано последним. Так было всегда, и это могло приводить к получению неожиданных результатов (дополнительные подробности (на английском языке) и примеры ищите на веб-сайте книги: `http://mng.bz/aEmE`), но теперь необязательные параметры означают возможность наличия большего числа подходящих методов, чем можно было ожидать. Я советую избегать перегрузки метода базового класса в производном классе, если только это не сулит крупной выгоды.

Вторая ситуация связана с тем, что именованные аргументы иногда могут быть альтернативой приведению для оказания компилятору помощи в распознавании перегруженных версий. Временами вызов может быть неоднозначным из-за того, что аргументы могут быть преобразованы в типы параметров в двух разных методах, но во всех остальных отношениях ни один из методов не является лучше другого. Например, рассмотрим следующие сигнатуры методов и вызов:

```
void Method(int x, object y) { ... }
void Method(object a, int b) { ... }
...
Method(10, 10);          ◄— Неоднозначный вызов
```

Подходят оба метода и ни один из них не лучше другого. Есть два способа решения проблемы при условии, что вы не можете изменить имена методов, устранив подобным образом неоднозначность. (Это мой предпочтительный подход. Сделайте имя каждого метода более информативным и специфичным, что в целом улучшит читабельность кода.) Для снятия неоднозначности можно либо явно привести один из аргументов, либо воспользоваться именованными аргументами:

```
void Method(int x, object y) { ... }
void Method(object a, int b) { ... }
...
Method(10, (object) 10); ◄— Приведение с целью устранения неоднозначности
Method(x: 10, y: 10);    ◄— Именование с целью устранения неоднозначности
```

Разумеется, это работает только в случае, если параметры имеют разные имена в разных методах, но такой удобный трюк полезно знать. Иногда приведение дает более читабельный код, а иногда именование. Это просто дополнительное оружие в борьбе за ясность кода.

К сожалению, описанному подходу присущ и недостаток, связанный с именованными аргументами в целом: появляется еще одна вещь, о которой придется заботиться в случае изменения имен параметров.

Тихий ужас изменения имен

В прошлом имена параметров не были особо существенными, если вы имели дело исключительно с языком C#. В других языках, возможно, это имело значение, но в C# параметры были важны только во время просмотра их в IntelliSense и анализа кода интересующего метода. Теперь имена параметров метода фактически являются частью API-интерфейса, даже если вы пользуетесь только C#. Если когда-нибудь позже вы решите изменить их, работа кода может нарушиться — любой вызов, в котором применялся именованный аргумент для ссылки на один из ваших параметров, перестанет компилироваться. Это может не быть большой проблемой, если ваш код предназначен для внутреннего потребления, но в случае написания открытого API-интерфейса имейте в виду, что изменения имени параметра — важное событие. В действительности так было всегда, но если весь вызывающий код написан на C#, была возможность не обращать на это внимания вплоть до настоящего момента.

Переименование параметров — плохая затея, но перестановка имен еще хуже. Вызывающий код может по-прежнему компилироваться, но получить другой смысл. Особенно зловредной формой этого является переопределение метода с перестановкой мест имен параметров в переопределенной версии. Компилятор всегда будет просматривать самую глубокую переопределенную версию, о которой ему известно, на основе статического типа выражения, используемого в качестве цели вызова метода. Вряд ли вы захотите попасть в ситуацию, когда вызов одной и той же реализации метода с тем же самым списком аргументов ведет себя по-разному в зависимости от статического типа переменной.

Заключение

Именованные аргументы и необязательные параметры являются, на первый взгляд, двумя простейшими из всех средств C# 4, тем не менее, как вы видели, с ними связано немало сложностей. Базовые идеи легко выразить и понять, но важнее всего то, что большую часть времени ни о чем другом беспокоиться не придется. Вы можете пользоваться преимуществом необязательных параметров по сокращению количества перегруженных версий, которые пришлось бы писать, а именованные аргументы могут сделать код намного более читабельным, когда присутствует несколько легко сбивающих с толку аргументов.

Вероятно, самым сложным моментом будет решение относительно того, какие стандартные значения применить, учитывая потенциальные проблемы, связанные с версиями. Кроме того, вопрос имен параметров теперь более очевиден, чем ранее, и вы должны проявлять осмотрительность при переопределении существующих методов, чтобы избежать нанесения ущерба вызывающему коду.

А теперь давайте перейдем к рассмотрению новых средств, связанных с COM.

13.2. Модернизация взаимодействия с COM

Я охотно соглашусь с тем, что являюсь далеко не экспертом в COM. Когда я пробовал работать с COM до выхода платформы .NET, то всегда сталкивался с проблемами, которые, вне всякого сомнения, частично были связаны с нехваткой у меня знаний, а частично объяснялись тем, что используемые мною компоненты были неудачно спроек-

тированы или реализованы. Хотя общее впечатление о COM, как о своего рода "черной магии", все же осталось. Я хорошо осведомлен о том, что в COM есть много замечательного, но, к сожалению, мне не удалось найти основания вернуться и изучить эту технологию во всех деталях — и похоже деталей, которые следовало бы изучить, должно быть *много*.

Материал этого раздела специфичен для Microsoft

Изменения во взаимодействии с COM имеют смысл не для всех компиляторов C#, и компилятор, не реализующий эти средства, по-прежнему считается совместимым со спецификацией.

В целом платформа .NET сделала технологию COM несколько более дружественной, но до сих пор были отдельные преимущества в случае применения ее в коде Visual Basic, а не C#. Как вы увидите в этом разделе, в версии C# 4 ситуация значительно выровнялась. Из-за высокой популярности в примерах этой главы будет использоваться Word, а в примерах следующей главы — Excel. Тем не менее, с новыми средствами не связано ничего такого, что было бы специфичным для пакета Office; вы должны удостовериться в том, что практика работы с COM стала лучше в C# 4, чем бы вы ни занимались.

13.2.1. Ужасы автоматизации Word до выхода C# 4

Пример будет прост — мы собираемся запустить Word, создать документ с одним абзацем текста, сохранить его и выйти из Word. Звучит совсем несложно, не так ли? Если бы это было так. В листинге 13.8 приведен код, который необходимо было писать до выхода версии C# 4.

Листинг 13.8. Создание и сохранение документа в C# 3

```
object missing = Type.Missing;

Application app = new Application { Visible = true }; <-- ❶ Запуск Word
app.Documents.Add(ref missing, ref missing,           <-- ❷ Создание нового документа
                  ref missing, ref missing);
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add(ref missing);
para.Range.Text = "Thank goodness for C# 4";

object filename = "demo.doc";                         <-- ❸ Сохранение документа
object format = WdSaveFormat.wdFormatDocument97;
doc.SaveAs(ref filename, ref format,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing);

doc.Close(ref missing, ref missing, ref missing);     <-- ❹ Завершение Word
app.Application.Quit(ref missing, ref missing, ref missing);
```

Каждый шаг в коде звучит просто: сначала создается экземпляр типа COM ❶ и делается видимым с помощью выражения инициализатора объекта; затем создается и заполняется новый документ ❷.

Механизм вставки текста в документ не настолько прямолинеен, как вы могли ожидать, но стоит вспомнить, что документ Word может иметь довольно сложную структуру; все еще не так плохо, как могло бы быть. Пара вызовов методов принимают необязательные параметры по ссылке; они не нужны, поэтому для них передается по ссылке локальная переменная со значением `Type.Missing`. Если вам когда-либо приходилось взаимодействовать с COM ранее, то, скорее всего, такой шаблон будет выглядеть хорошо знакомым.

Дальше идет по-настоящему сложный фрагмент: сохранение документа ❶. Да, метод `SaveAs()` действительно имеет 16 параметров, из которых здесь задействуются только 2. И даже эти 2 параметра должны передаваться по ссылке, что означает необходимость в создании для них локальных переменных. В плане читабельности получается полный кошмар. Не переживайте — скоро мы во всем разберемся.

Наконец, документ и приложение закрываются ❷. За исключением того факта, что оба вызова имеют по три необязательных параметра, больше нет ничего интересного.

Начнем с применения средств, которые уже были представлены в главе — даже их одних будет достаточно, чтобы значительно сократить пример.

13.2.2. Реванш необязательных параметров и именованных аргументов

На первых порах давайте избавимся от всех аргументов, соответствующих необязательным параметрам, в которых вы не заинтересованы. Это также означает, что переменная `missing` больше не нужна.

По-прежнему остались 2 параметра из возможных 16 для метода `SaveAs()`. В данный момент их смысл несложно понять по именам локальных переменных, но если они случайно перепутаны? Все параметры являются слабо типизированными, так что приходится действительно строить догадки. Для прояснения вызова можно легко назначить аргументам имена. Если вы хотите использовать один из дальнейших параметров, то могли бы в любом случае указать его имя, просто чтобы пропустить ненужные параметры.

В листинге 13.9 показан код, который уже выглядит намного яснее.

Листинг 13.9. Автоматизация Word с применением обычных средств C# 4

```
Application app = new Application { Visible = true };
app.Documents.Add();
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add();
para.Range.Text = "Thank goodness for C# 4";

object filename = "demo.doc";
object format = WdSaveFormat.wdFormatDocument97;
doc.SaveAs(FileName: ref filename, FileFormat: ref format);

doc.Close();
app.Application.Quit();
```

Это намного лучше, хотя все еще приходится создавать локальные переменные для аргументов метода `SaveAs()`, которые вы *указываете*. Кроме того, при внимательном чтении может возникнуть интерес к удаленным дополнительным параметрам. Они являются параметрами `ref`, однако необязательными — сочетание, которое в C# обычно не поддерживаются. Что происходит?

13.2.3. Ситуации, когда параметр `ref` в действительности таковым не является

В отношении параметров `ref` в языке C# обычно принимается довольно жесткая линия поведения. Помечаться модификатором `ref` должен не только параметр, но и аргумент. Тем самым демонстрируется тот факт, что вы осознаете происходящее — в вызываемом методе значение вашей переменной может быть изменено. Все это хорошо в нормальном коде, но в API-интерфейсах COM параметры `ref` используются почти *везде* по причинам, связанным с производительностью, и обычно не модифицируют передаваемые переменные. Передача аргументов по ссылке в C# несколько затруднена. Требуется не только указать модификатор `ref`, но также предусмотреть переменную. Передавать по ссылке *значения* нельзя.

В C# 4 компилятор существенно упрощает дело, позволяя передавать аргумент методу COM по значению, даже если он соответствует параметру `ref`. Взгляните на следующий вызов, в котором `argument` может быть переменной типа `string`, но параметр объявлен как `ref object`:

```
comObject.SomeMethod(argument);
```

Компилятор выдает код, эквивалентный приведенному ниже:

```
object tmp = argument;
comObject.SomeMethod(ref tmp);
```

Обратите внимание, что любые изменения, сделанные в методе `SomeMethod()`, отбрасываются, поэтому вызов на самом деле ведет себя так, как если бы переменная `argument` передавалась по значению. Тот же самый процесс применяется для необязательных параметров `ref`; для каждого из них задействуется локальная переменная, инициализированная `Type.Missing`, которая передается по ссылке методу COM. Если вы декомпилируете готовый код C#, то увидите, что сгенерированный код IL довольно-таки объемен из-за наличия всех этих дополнительных переменных. Теперь можно внести финальные штрихи в пример с документом Word (листинг 13.10).

Листинг 13.10. Передача аргументов по значению методам COM

```
Application app = new Application { Visible = true };
app.Documents.Add();
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add();
para.Range.Text = "Thank goodness for C# 4";
doc.SaveAs(FileName: "test.doc",                    <— Аргументы, переданные по значению
           FileFormat: WdSaveFormat.wdFormatDocument97);
doc.Close();
app.Application.Quit();
```

Итак, окончательный код стал намного яснее того, с которого все начиналось. В API-интерфейсах вроде Word по-прежнему приходится иметь дело со сбивающим с толку набором методов, свойств и событий в основных типах, таких как `Application` и `Document`, но, во всяком случае, ваш код будет намного проще читать.

В терминах изменений исходного кода есть один последний аспект поддержки COM, который необходимо рассмотреть, прежде чем переходить к улучшениям развертывания, доступным в C# 4.

13.2.4. Вызов именованных индексаторов

Некоторые аспекты C# 4 предоставляют поддержку для средств, доступных в Visual Basic на протяжении долгого времени, и это одно из таких средств. Среда CLR, технология COM и язык Visual Basic допускают нестандартные свойства с параметрами — *именованные индексаторы* в терминах C#. До выхода версии C# 4 в языке C# не только было запрещено напрямую объявлять собственные именованные индексаторы[5], но даже не предлагался способ доступа к ним с использованием синтаксиса свойств. Единственным индексатором, которым можно было пользоваться в C#, являлся такой, который был объявлен как *стандартное свойство* для типа. Это не было особой проблемой для компонентов .NET, реализованных на Visual Basic, т.к. именованные индексаторы в основном не поощрялись. Однако в компонентах COM, подобных тем, что применяются для Office, они использовались гораздо интенсивнее. Версия C# 4 позволяет вызывать именованные индексаторы в более естественной манере, но их по-прежнему нельзя объявлять для собственных типов C#.

Снова конфликты терминологии!

Термин *индексатор* используется повсеместно в этом разделе для описания того, что в VB известно как *параметризованное свойство*. В спецификации CLI оно называется *индексированным свойством*. Какая бы терминология не применялась, в IL это объявляется как свойство, принимающее параметры. Нормальный индексатор (насколько это касается C#) определяется с помощью *стандартного члена* (или *стандартного свойства*) для типа — например, стандартным членом типа `StringBuilder` является свойство `Chars` (которое имеет параметр `Int32`). Когда здесь речь идет об *именованных индексаторах*, имеются в виду те, которые *не являются* стандартными для типа, поэтому на них можно ссылаться по имени.

В примере снова будет использоваться Word, и на этот раз отображаются разные смысловые трактовки для слов. В типе `_Application` внутри API-интерфейса Word определен индексатор `SynonymInfo` с приблизительно таким объявлением:

```
SynonymInfo SynonymInfo[string Word,
                        ref object LanguageId = Type.Missing]
```

Это не является допустимым синтаксисом C#, поскольку именованный индексатор объявлять нельзя, но надо надеяться, что его назначение вполне понятно (он выдает сведения о синонимах для заданного слова). Имя индексатора — `SynonymInfo`. Он *возвращает* ссылку на объект `SynonymInfo` и принимает два параметра, один из которых является необязательным. (Тот факт, что имена индексатора и возвращаемого типа совпадают, в данной ситуации совершенно случаен.)

Индексатор `SynonymInfo` может применяться с целью нахождения значений слова и синонимов для каждого значения. В листинге 13.11 продемонстрированы три разных способа использования индексатора для отображения количества значений трех разных слов.

[5] Во всяком случае, непосредственно. Можно вручную применить атрибут `System.Runtime.CompilerServices.IndexerNameAttribute`, но это не то, что в C# считается частью языка.

Листинг 13.11. Применение индексатора SynonymInfo для подсчета значений слов

```
static void ShowInfo(SynonymInfo info)
{
  Console.WriteLine("{0} has {1} meanings", info.Word, info.MeaningCount);
}
...
Application app = new Application { Visible = false };

object missing = Type.Missing;
ShowInfo(app.get_SynonymInfo("painful", ref missing));   <-- ❶ Использование раннего синтаксиса C#

ShowInfo(app.SynonymInfo["nice", WdLanguageID.wdEnglishUS]);   <-- ❷ Использование индексатора с двумя параметрами

ShowInfo(app.SynonymInfo[Word: "features"]);   <-- ❸ Получение преимущества от необязательного параметра

app.Application.Quit();
```

Даже без именованных индексаторов показанные ранее средства помогают облегчить проблемы, присущие прежнему синтаксису C# ❶; к примеру, можно было бы вызвать `app.get_SynonymInfo("better")` и получить преимущество средства необязательных параметров. Но благодаря второму и третьему вызовам `ShowInfo` (❷ и ❸), легко заметить, что синтаксис индексатора выглядит менее неуклюжим, чем обращение к `get_`. Можно было бы утверждать, что это должно быть вызовом метода или же свойством `SynonymInfo` без параметров, которое возвращает коллекцию с подходящим стандартным индексатором. Это один из общих аргументов, приводимых проектировщиками C# относительно отсутствия реализации *полной* поддержки для именованных индексаторов, включая их объявление в рамках C#. Но дело в том, что это уже индексатор в Word, так что было бы неплохо им воспользоваться как таковым[6]. Во втором вызове `ShowInfo()` ❷ применяется средство неявных параметров `ref`, описанное в разделе 13.2.3, а в третьем вызове ❸ опущен необязательный параметр и именован оставшийся аргумент (просто ради интереса).

С необязательными параметрами и индексаторами связана одна характерная особенность: если *все* параметры являются необязательными, и вы не хотите указывать аргументы вообще, то должны опустить квадратные скобки. Вместо `foo.Indexer[]` необходимо указывать `foo.Indexer`. Все это применимо как к получению значения из индексатора, так и к его установке.

Пока все идет хорошо, но написание кода — это только часть схватки. Как правило, должна быть возможность развернуть его также на других машинах. И снова C# 4 упрощает решение этой задачи.

13.2.5. Связывание основных сборок взаимодействия

При компиляции с участием некоторого типа COM используется сборка, сгенерированная для библиотеки компонентов. Обычно применяется *основная сборка взаимодействия* (Primary Interop Assembly — PIA), которая является канонической сборкой взаимодействия для библиотеки COM, подписанной издателем. Сгенерировать такие

[6] Возможно, это было бы более интересно для раскрытия реального смысла, но в результате возникли бы проблемы, связанные с взаимодействием, которые не имеют отношения к материалу настоящей главы.

сборки для собственных библиотек COM можно с помощью инструмента Type Library Importer (`tlbimp`). Сборки PIA упрощают жизнь в плане наличия одного достоверного способа доступа к типам COM, но усложняют ее в других аспектах. Они могут быть довольно большими, и полная сборка PIA должна присутствовать, даже если используется небольшое подмножество функциональности. Вдобавок на машине, где производится развертывание, должна находиться сборка PIA той же самой версии, которая участвовала в компиляции. Это может создать сложности в ситуациях, когда условия лицензирования запрещают распространение самой сборки PIA, и приходится полагаться на то, что корректная версия уже установлена. При наличии нескольких версий, которые предоставляют необходимую функциональность, может понадобиться поставлять разные версии *вашего* кода, чтобы обеспечить работоспособность ссылок.

В C# 4 предлагается совершенно иной подход. Вместо *ссылки* на сборку PIA подобно любой другой сборке можно *связать* ее. В Visual Studio 2010 и последующих версиях это делается в окне Properties (Свойства) для ссылки на сборку, как показано на рис. 13.4.

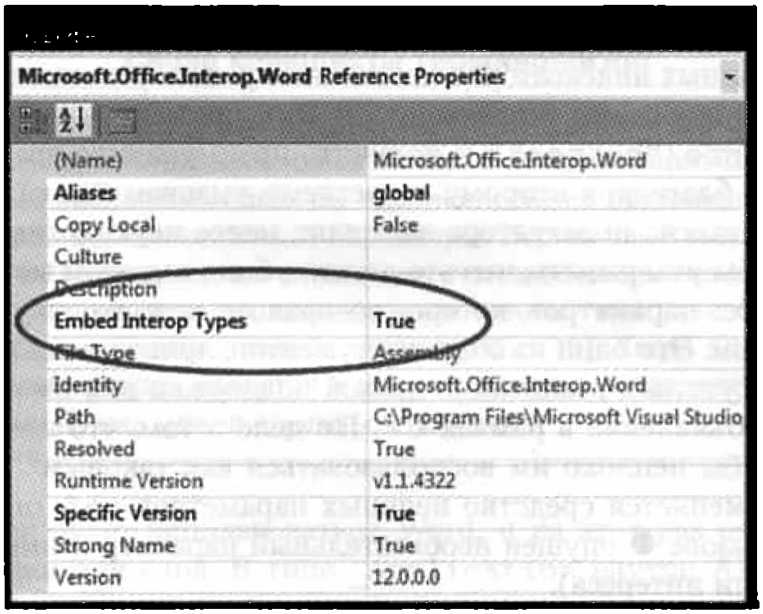

Рис. 13.4. Связывание сборок PIA в Visual Studio 2010

Приверженцы командной строки для связывания вместо ссылки могут указать ключ `/l` (или `/link`) взамен `/r` (или `/reference`):

```
csc /l:Path\To\PIA.dll MyCode.cs
```

Когда сборка PIA связывается, компилятор внедряет лишь необходимые части PIA прямо в вашу сборку. Берутся только нужные типы, и только члены внутри этих типов. Например, для показанного ранее кода компилятор создает следующие типы:

```
namespace Microsoft.Office.Interop.Word
{
  [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
  public interface _Application
  [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
  public interface _Document
  [ComImport, CompilerGenerated, TypeIdentifier, Guid("...")]
  public interface Application : _Application
  [ComImport, Guid("..."), TypeIdentifier, CompilerGenerated]
  public interface Document : _Document
```

```
  [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
  public interface Documents : IEnumerable
  [TypeIdentifier("...", "WdSaveFormat"), CompilerGenerated]
  public enum WdSaveFormat
}
```

Интерфейс `_Application` выглядит приблизительно так:

```
[ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
public interface _Application
{
  void_VtblGap 1_4();
  Documents Documents { [...] get; }
  void_VtblGap2_1();
  Document ActiveDocument { [...] get; }
}
```

Для экономии пространства идентификаторы GUID и атрибуты свойств здесь не указаны, но вы всегда можете просмотреть внедренные типы с помощью Reflector. Это только интерфейсы и перечисления; никакой реализации не присутствует. С учетом того, что обычная сборка PIA имеет тип `CoClass`, представляющий действительную реализацию (но передающий всю работу реальному типу COM, конечно же), когда компилятор нуждается в создании экземпляра какого-то типа COM через связанную сборку PIA, он делает это с использованием идентификатора GUID, ассоциированного с типом. Например, строка кода в примере с документом Word, в которой создается экземпляр типа `Application`, транслируется в следующий код, если связывание включено[7]:

```
Application application = (Application) Activator.CreateInstance(
   Type.GetTypeFromCLSID (new Guid("...")));
```

На рис. 13.5 демонстрируется разница между ссылкой и связыванием во время выполнения.

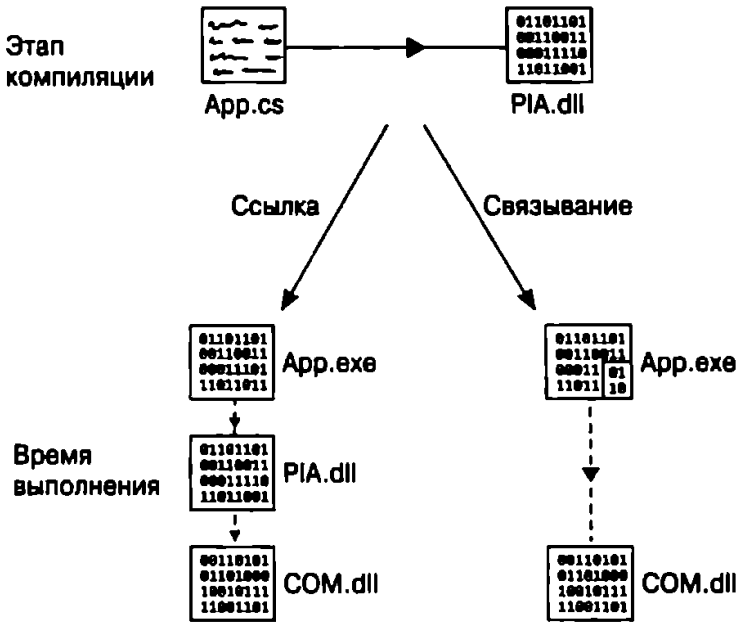

Рис. 13.5. Сравнение ссылки и связывания

[7] Во всяком случае, в близкий к показанному код. Инициализатор объекта делает его чуть более сложным, т.к. компилятор применяет дополнительную временную переменную.

Библиотеки внедренных типов обладают различными преимуществами.

- Упрощается развертывание: исходная сборка PIA не нужна, поэтому не придется полагаться на то, что правильная ее версия уже присутствует, или поставлять сборку PIA самостоятельно.
- Упрощается управление версиями: до тех пор, пока вы используете только члены из той версии библиотеки COM, которая действительно установлена, не имеет значения, с какой версией сборки PIA осуществляется компиляция — с более ранней или более поздней.
- Вариантные типы трактуются как динамические типы, сокращая объем необходимых приведений.

Пока не стоит переживать по поводу неясности последнего пункта — сначала нужно ознакомиться с динамической типизацией. Все детали будут раскрыты в следующей главе.

Как видите, в Microsoft серьезно занялись взаимодействием с COM в версии C# 4, сделав весь процесс разработки менее болезненным. Разумеется, уровень болезненности всегда варьировался в зависимости от библиотеки COM, задействованной при разработке, поэтому от введения новых средств одни выиграли больше, а другие — меньше.

Следующее средство совершенно не связано с COM, именованными аргументами и необязательными параметрами, но оно снова направлено на некоторое облегчение процесса разработки.

13.3. Обобщенная вариантность для интерфейсов и делегатов

Вы можете помнить, что в главе 3 я упоминал о наличии в среде CLR определенной поддержки для вариантности в обобщенных типах, но на уровне языка C# она пока не была доступна. В C# 4 ситуация изменилась. Язык C# получил синтаксис, требуемый для объявления обобщенной вариантности, и теперь компилятору известно о возможных преобразованиях для интерфейсов и делегатов.

Это не в корне изменяющее жизнь средство — в большей степени это способ сглаживания ряда шероховатостей, с которыми вы иногда сталкивались. Оно даже не устраняет все шероховатости; существуют разнообразные ограничения, главным образом призванные сохранить обобщения полностью безопасными к типам. Тем не менее, это хорошее средство, которое полезно иметь в своем арсенале.

На всякий случай необходимо напомнить вам о том, что собой представляет вариантность, поэтому давайте начнем с краткого повторения двух базовых форм, в которых она проявляется.

13.3.1. Типы вариантности: ковариантность и контравариантность

По существу вариантность — это возможность использования объекта одного типа, как если бы он имел другой тип, в безопасной к типам манере. Вы привыкли к вариантности в терминах обычного наследования: например, если метод имеет объявленный возвращаемый тип `Stream`, то из реализации можно возвращать и `MemoryStream`. Обобщенная вариантность представляет собой ту же концепцию, но применительно к обобщениям, где она становится несколько более сложной. Вариантность применяется к параметрам типов внутри интерфейсов и типов делегатов. Именно на этом необходимо сосредоточить внимание.

В конечном счете, не особенно важно, запомните ли вы терминологию, используемую в этом разделе. Она будет полезна при чтении главы, но вряд ли вы будете ее употреблять в разговоре. Намного важнее понять сами концепции.

Существуют два типа вариантности: *ковариантность* и *контравариантность*. По сути, они представляют одну и ту же идею, но применяются в контексте значений, перемещающихся в разных направлениях. Мы начнем с ковариантности, которая, как правило, проще для понимания.

Ковариантность: значения, поступающие из API-интерфейса

Ковариантность связана со значениями, возвращаемыми из операции обратно в вызывающий код. Давайте вообразим себе очень простой обобщенный интерфейс, представляющий шаблон фабрики. Он имеет единственный метод `CreateInstance()`, который будет возвращать экземпляр подходящего типа. Ниже показан код:

```
interface IFactory<T>
{
  T CreateInstance();
}
```

В настоящий момент тип `T` в интерфейсе встречается только однажды (не считая имени интерфейса). Он используется только в качестве *возвращаемого значения* — это *выход* метода. В результате обретает смысл наличие возможности трактовать фабрику специализированного типа как фабрику более общего типа. Пользуясь терминами из реальной жизни, фабрику по производству пиццы можно считать пищевой фабрикой.

Контравариантность: значения, поступающие в API-интерфейс

Контравариантность является полной противоположностью. Она связана со значениями, передаваемыми *в* API-интерфейс вызывающим кодом: вместо вырабатывания API-интерфейс потребляет значения. Представим еще один простой интерфейс, который может красиво выводить на консоль документ определенного типа. И снова он имеет только один метод, на этот раз называемый `Print()`:

```
interface IPrettyPrinter<T>
{
  void Print(T document);
}
```

Теперь тип `T` встречается только во *входных* позициях интерфейса, выступая в качестве параметра. Чтобы опять поместить это в рамки конкретных терминов, можно сказать, что при наличии реализации `IPrettyPrinter<SourceCode>` должна быть возможность ее применения как `IPrettyPrinter<CSharpCode>`.

Инвариантность: значения перемещаются в обоих направлениях

Если ковариантность применяется, когда значения только поступают *из* API-интерфейса, а контравариантность — когда значения только передаются *в* API-интерфейс, тогда что происходит, когда значение перемещается в обоих направлениях? Если одним словом, то ничего. Такой тип был бы *инвариантным*.

Ниже показан интерфейс, представляющий тип, который может сериализировать и десериализировать какой-то тип данных:

```
interface IStorage<T>
{
  byte[] Serialize(T value);
  T Deserialize(byte[] data);
}
```

На этот раз экземпляр IStorage<T> для конкретного типа T нельзя трактовать как реализацию этого интерфейса для более или менее специализированного типа. Если вы попытаетесь использовать его ковариантным путем (например, считать IStorage<Customer> как IStorage<Person>), может получиться вызов метода Serialize() с объектом, который не удастся обработать. Подобным же образом, если вы попробуете применять его контравариантным способом, при десериализации определенных данных может быть получен непредвиденный тип.

Если это поможет, то думайте об инвариантности как о чем-то, похожем на параметры ref; для передачи по ссылке переменная должна быть *точно* такого же типа, как сам параметр, поскольку значение поступает в метод и затем благополучно получается из него обратно.

13.3.2. Использование вариантности в интерфейсах

Язык C# 4 позволяет указывать в объявлении обобщенного интерфейса или делегата, что параметр типа может использоваться ковариантно, с помощью модификатора out, или контравариантно посредством модификатора in. После того как тип объявлен, неявно становятся доступными связанные виды преобразований. Это работает одинаково как в интерфейсах, так и в делегатах, но для ясности я продемонстрирую их отдельно. Давайте начнем с интерфейсов, т.к. они могут быть чуть более знакомы и мы уже применяли их для описания вариантности.

Вариантные преобразования являются ссылочными

Любое преобразование, использующее вариантность или ковариантность — это *ссылочное преобразование*, т.е. после преобразования возвращается та же самая ссылка. Преобразование не создает новый объект; оно просто трактует существующую ссылку, как если бы она соответствовала целевому типу. Это точно совпадает с приведением между ссылочными типами в иерархии: если привести Stream к MemoryStream (или применить неявное преобразование другим способом), по-прежнему останется только один объект. Как вы увидите далее, сама природа таких преобразований вносит ряд ограничений, но это означает, что они эффективны, и делает поведение проще для понимания в терминах объектной идентичности.

В этот раз для демонстрации идей будут использоваться знакомые интерфейсы с несколькими простыми типами, определенными пользователем, для аргументов типов.

Выражение вариантности с помощью модификаторов in и out

Существуют два интерфейса, которые демонстрируют вариантность особенно эффективно: IEnumerable<T> ковариантен в T, а IComparer<T> контравариантен в T. Вот их новые объявления типов в .NET 4:

```
public interface IEnumerable<out T>
public interface IComparer<in T>
```

Это довольно легко запомнить: если параметр типа применяется только для выхода, можно использовать модификатор out; если же он применяется для ввода, можно использовать in. Компилятору совершенно не известно, можете ли вы запомнить, какая форма называется ковариантностью, а какая — контравариантностью!

К сожалению, инфраструктура не содержит многих иерархий наследования, которые помогли бы продемонстрировать вариантность максимально ясно, поэтому я вернусь к стандартному объектно-ориентированному примеру с фигурами. В состав загружаемого исходного кода входят определения для IShape, Circle и Square, которые довольно очевидны. Интерфейс открывает свойства для ограничивающего прямоугольника фигуры и ее площади. Во многих последующих примерах я буду применять два списка, код конструирования которых в справочных целях приведен ниже:

```
List<Circle> circles = new List<Circle>
{
  new Circle(new Point(0, 0), 15),
  new Circle(new Point(10, 5), 20),
};
List<Square> squares = new List<Square>
{
  new Square(new Point(5, 10), 5),
  new Square(new Point(-10, 0), 2)
};
```

Единственный важный момент здесь связан с типами переменных — они объявлены как List<Circle> и List<Square>, а не List<IShape>. Часто это может быть полезным — например, если где-то осуществлялся доступ к списку кружочков, могла возникнуть необходимость в получении специфичных для членов без приведения. Действительные значения, задействованные в коде конструирования, совершенно не важны; для ссылки на те же самые списки я буду везде использовать имена circles и squares, но без дублирования кода[8].

Использование ковариантности интерфейсов

Для демонстрации ковариантности мы попробуем построить список фигур из списка кружочков и списка прямоугольников. В листинге 13.12 показаны два разных подхода, ни один из которых не работал бы в версии C# 3.

Листинг 13.12. Использование вариантности для построения списка общих фигур из специализированных списков

```
List<IShape> shapesByAdding = new List<IShape>();   ←❶ Добавление списков
shapesByAdding.AddRange(circles);
shapesByAdding.AddRange(squares);

List<IShape> shapesByConcat =
        circles.Concat<IShape>(squares).ToList();   ←❷ Использование LINQ для конкатенации
```

Фактически код в листинге 13.12 иллюстрирует ковариантность в четырех местах, каждый раз преобразуя последовательность кружочков и прямоугольников в последовательность общих фигур, насколько в этом заинтересована система типов. Сначала создается новый экземпляр List<IShape> и вызывается метод AddRange() для добавления к нему списков кружочков и прямоугольников ❶. (Можно было бы вместо этого передать один из них конструктору и затем только раз вызвать AddRange().)

[8] В полном исходном коде решения они доступны в виде свойств статического класса Shapes, но во фрагментах кода я включаю код конструирования везде, где он необходим, что позволит вам при желании легко его подстроить.

Параметр для метода `List<T>.AddRange()` имеет тип `IEnumerable<T>`, поэтому в данном случае каждый список трактуется как `IEnumerable<IShape>` — то, что ранее было невозможным. Метод `AddRange()` *мог бы* быть реализован как обобщенный метод с собственным параметром типа, но этого не произошло — в таком случае некоторые оптимизации оказались бы трудно достижимыми или невозможными.

Еще один способ создания списка, который содержит данные из двух существующих последовательностей, предусматривает применение LINQ ❷. Нельзя напрямую вызвать `circles.Concat(squares)`, т.к. это запутало бы механизм выведения типов, но можно явно указать аргумент типа. Переменные `circles` и `squares` неявно преобразуются в `IEnumerable<IShape>` посредством ковариантности. Такое преобразование на самом деле не изменяет значение, а только устанавливает то, каким образом компилятор *трактует* это значение. Важно то, что отдельная копия не создается. Ковариантность особенно существенна в LINQ to Objects, т.к. большая часть API-интерфейса выражена в терминах `IEnumerable<T>`. С другой стороны, контравариантность не так важна, поскольку контравариантных типов относительно немного.

Безусловно, в C# 3 были доступны другие способы решения той же проблемы. Для исходных фигур можно было строить экземпляры `List<IShape>` вместо `List<Circle>` и `List<Square>`; можно было использовать операцию `Cast()` из LINQ для преобразования специализированных списков в более общие списки; можно было написать собственный класс списка с обобщенным методом `AddRange()`. Однако ни один из перечисленных способов не обладает такой степенью удобства или эффективности, как предлагаемые здесь альтернативы.

Использование контравариантности интерфейсов

Для демонстрации контравариантности будут применяться те же самые типы фигур. На этот раз мы воспользуемся списком кружочков и компаратором, который способен сравнивать *любые* две фигуры, просто сопоставляя их площади. Это не удалось бы сделать в версиях, предшествующих C# 4, т.к. `IComparer<IShape>` нельзя трактовать как `IComparer<Circle>`, но в листинге 13.13 показано, как на выручку приходит контравариантность.

Листинг 13.13. Сортировка кружочков с использованием универсального компаратора и контравариантности

```
class AreaComparer : IComparer<IShape> ←❶ Сравнение фигур по площади
{
   public int Compare(IShape x, IShape y)
   {
      return x.Area.CompareTo(y.Area);
   }
}
...
IComparer<IShape> areaComparer = new AreaComparer();
circles.Sort(areaComparer);          ←❷ Сортировка с применением контравариантности
```

Здесь нет ничего сложного. Класс `AreaComparer` ❶ настолько прост, насколько может быть проста реализация `IComparer<T>`; к примеру, он не нуждается в поддержке состояния. Обычно в методе `Compare()` присутствует некоторая обработка значений `null`, но для целей демонстрации она необязательна.

Полученный экземпляр IComparer<IShape> используется для сортировки списка кружочков ❷. Аргументом метода circles.Sort() должен быть IComparer<Circle>, но контравариантность допускает неявное преобразование компаратора. Проще простого.

Вот так сюрприз!

Если бы кто-то предъявил вам этот код, как будто он написан на C# 3, вы могли бы посмотреть на него и решить, что он заработает. Кажется вполне очевидным, что он *должен* работать, и это распространенное впечатление; инвариантность в C# 2 и C# 3 часто оказывается неприятным сюрпризом. Новые возможности C# 4 в этой области не вводят какие-то новые концепции, о которых вы не знали ранее; они просто обеспечивают большую гибкость.

Это были два простых примера, в которых использовались интерфейсы с единственным методом, но те же самые принципы применимы и к более сложным API-интерфейсам. Разумеется, чем сложнее интерфейс, тем более вероятно, что параметр типа будет использоваться и для входа, и для выхода, что делает его инвариантным. Позже мы рассмотрим несколько более запутанных примеров, но сначала взглянем на делегаты.

13.3.3. Использование вариантности в делегатах

Теперь, когда вы видели, как применять вариантность в интерфейсах, эти знания легко распространить на делегаты. Мы снова будем использовать знакомые типы:

```
delegate T Func<out T>()
delegate void Action<in T>(T obj)
```

На самом деле они эквивалентны интерфейсам IFactory<T> и IPrettyPrinter<T>, с которых мы начинали. С помощью лямбда-выражений работать с ними несложно, даже когда они выстраиваются в цепочку. В листинге 13.14 приведен пример применения типов, представляющих формы.

Листинг 13.14. Использование вариантности с простыми делегатами Func<T> и Action<T>

```
Func<Square> squareFactory = () => new Square(new Point(5, 5), 10);
Func<IShape> shapeFactory = squareFactory;      ←❶ Преобразование Func<T> с использованием ковариантности

Action<IShape> shapePrinter = shape => Console.WriteLine(shape.Area);
Action<Square> squarePrinter = shapePrinter;    ←❷ Преобразование Action<T> с использованием контравариантности

squarePrinter(squareFactory());                 ← Проверка работоспособности...
shapePrinter(shapeFactory());
```

Надеюсь, что к этому времени код требует лишь небольших пояснений. Фабрика прямоугольников всегда производит прямоугольник в той же самой позиции со сторонами длиной 10 единиц. Ковариантность без проблем позволяет трактовать фабрику прямоугольников как фабрику общих фигур ❶. Затем создается универсальное действие, которое выводит на консоль площадь любой предоставленной фигуры. На этот

раз используется контравариантное преобразование для трактовки действия как такого, которое может быть применено к любому прямоугольнику ❸. Наконец, действию с прямоугольниками передается результат вызова фабрики прямоугольников, а действию с формами — результат вызова фабрики форм. Как и можно было ожидать, оба действия выводят на консоль значение `100`.

Конечно, здесь использовались только делегаты с единственным параметром типа. Что произойдет в случае применения делегатов или интерфейсов с несколькими параметрами типов? Что можно сказать об аргументах типов, которые сами являются обобщенными делегатами? Понятно, что все может стать достаточно сложным.

13.3.4. Сложные ситуации

Прежде чем я попытаюсь вызвать у вас головокружение, я должен вас немного успокоить. Несмотря на то что в этом разделе мы будем делать несколько странные и удивительные вещи, компилятор не позволит допустить ошибки. Вас по-прежнему могут запутывать сообщения об ошибках, если вы используете множество параметров типов экстравагантными способами, но как только все скомпилируется, можно больше не переживать. Сложность возможна как при вариантности в форме делегатов, так и при вариантности в форме интерфейсов, хотя версия с делегатами обычно более лаконична. Давайте начнем с относительно простого примера.

Одновременная ковариантность и контравариантность с помощью типа `Converter<TInput, TOutput>`

Тип делегата `Converter<TInput, TOutput>` существовал со времен .NET 2.0. Он фактически является `Func<T, TResult>`, но с более ясно выраженным назначением. В .NET 4 он превратился в `Converter<in TInput, out TOutput>`, где хорошо видно, какой вид вариантности имеет каждый параметр типа.

В листинге 13.15 демонстрируется несколько комбинаций вариантности с применением простого конвертера.

Листинг 13.15. Демонстрация ковариантности и контравариантности посредством одного типа

```
Converter<object, string> converter = x => x.ToString(); ❶
Converter<string, string> contravariance = converter;
Converter<object, object> covariance = converter;
Converter<string, object> both = converter;               ❷
```

❶ Преобразование объектов в строки

❷ Преобразование строк в объекты

В листинге 13.15 показаны вариантные преобразования, доступные для делегата типа `Converter<object, string>` — делегата, который принимает любой объект и производит строку. Сначала реализуется делегат с использованием простого лямбда-выражения, в котором вызывается метод `ToString()` ❶. Как только это случилось, мы больше никогда в действительности не *вызываем* делегат, так что вполне можно было бы указать ссылку `null`. Однако я считаю, что думать о вариантности проще, если привязываться к конкретному действию, которое *произошло* бы в случае его вызова.

Следующие две строки кода относительно прямолинейны при условии, что вы сосредоточите внимание только на одном параметре типа за раз. Параметр типа `TInput` встречается только во входной позиции, поэтому имеет смысл применять его контравариантно, используя `Converter<object, string>` как `Converter<Button, string>`. Другими словами, если можно передавать в конвертер *любую* объектную ссылку, то определенно можно передавать ссылку `Button`. Подобным же образом, параметр типа `TOutput` обнаруживается только в выходной позиции (возвращаемый тип), так что имеет смысл применять его ковариантно; если конвертер всегда возвращает ссылку на строку, то его можно безопасно использовать там, где нужно только гарантировать возвращение объектной ссылки.

Последняя строка кода — это просто логическое расширение описанной идеи ❷. В ней применяются контравариантность и ковариантность в рамках одного преобразования, чтобы получить конвертер, который принимает только экземпляры `Button` и возвращает только объектную ссылку. Обратите внимание, что обратное преобразование *невозможно* без приведения — по существу вы ослабили гарантии по всем пунктам и не можете их снова усилить неявным образом.

Давайте немного повысим ставки и посмотрим, насколько сложные вещи можно получить, если как следует постараться.

Безумие функций высшего порядка

Действительно странное поведение начинается при комбинировании вариантных типов друг с другом. Я не собираюсь здесь вдаваться в особые детали, а просто хочу, чтобы вы оценили потенциал сложности. Взгляните на следующие четыре объявления делегатов:

```
delegate Func<T> FuncFunc<out T>();
delegate void ActionAction<out T>(Action<T> action);
delegate void ActionFunc<in T>(Func<T> function);
delegate Action<T> FuncAction<in T>();
```

Каждое объявление эквивалентно вложению одного из стандартных делегатов внутрь другого. Например, делегат `FuncAction<T>` эквивалентен `Func<Action<T>>`. Оба они представляют функцию, возвращающую экземпляр `Action`, которому может быть передан экземпляр `T`. Но должно это быть ковариантным или контравариантным? Хорошо, функция будет *возвращать* действие для выполнения над `T`, так что это кажется ковариантным, но это действие затем *получает* `T`, что выглядит контравариантным. Ответ заключается в том, что делегат *является* контравариантным в `T`, именно поэтому он объявлен с модификатором `in`.

В качестве эмпирического правила: можете рассматривать вложенную контравариантность как инверсию предыдущей вариантности, а ковариантность — нет, поэтому, несмотря на то, что делегат `Action<Action<T>>` ковариантен в `T`, делегат `Action<Action<Action<T>>>` контравариантен. Сравните это с вариантностью `Func<T>`, где можно записывать `Func<Func<Func<...Func<T>...>>>` с произвольной глубиной вложения и по-прежнему иметь ковариантность.

Просто чтобы привести похожий пример с использованием интерфейсов, вообразите себе некоторый компаратор, позволяющий сравнивать последовательности. Если он может сравнивать две последовательности произвольных объектов, то определенно может сравнивать две последовательности строк, но не наоборот. Представив сказанное в виде кода (без реализации интерфейса!), можно трактовать это следующим образом:

```
IComparer<IEnumerable<object>> objectsComparer = ...;
IComparer<IEnumerable<string>> stringsComparer = objectsComparer;
```

Такое преобразование законно: `IEnumerable<string>` — это "меньший" тип, чем `IEnumerable<object>` из-за ковариантности `IEnumerable<T>`. Затем контравариантность `IComparer<T>` разрешает преобразование из компаратора "большего" типа в компаратор "меньшего" типа.

Конечно, в этом разделе рассматривались только делегаты и интерфейсы с единственным параметром типа — все это также применимо и в случаях с множеством параметров типов. Однако не переживайте: вряд ли вам особенно часто будет требоваться такой вид заумной вариантности, и когда это понадобится, вы должны запросить помощь у компилятора. На самом деле я просто хотел, чтобы вы знали о таких возможностях.

С другой стороны, есть вещи, которые могут показаться доступными для выполнения, но в действительности они не поддерживаются.

13.3.5. Ограничения и замечания

Поддержка вариантности, предлагаемая C# 4, ограничивается главным образом тем, что предоставляется средой CLR. Языку было бы затруднительно поддерживать преобразования, которые запрещены лежащей в основе платформой. Это может привести к нескольким сюрпризам.

Отсутствие вариантности для параметров типов в классах

Вариантные параметры типов могут иметь только интерфейсы и делегаты. Даже при наличии класса, который использует параметр типа только для входа (или только для выхода), модификаторы `in` или `out` указывать нельзя. Например, класс `Comparer<T>`, распространенная реализация `IComparer<T>`, инвариантен — не существует каких-либо преобразований из `Comparer<IShape>` в `Comparer<Circle>`.

Не принимая во внимание трудности реализации, которые могут возникнуть в связи с этим, я бы сказал, что концептуально это имеет определенный смысл. Интерфейсы представляют способ взгляда на объект со специфичного ракурса, тогда как классы больше направлены на *действительный* тип объекта. Правда, данный аргумент несколько ослабляется тем фактом, что наследование позволяет трактовать объект как экземпляр любого класса в его иерархии наследования. Так или иначе, среда CLR не разрешает это.

Вариантность поддерживает только ссылочные преобразования

Вариантность нельзя применять между двумя произвольными аргументами типов лишь потому, что между ними существует преобразование. Оно должно быть *ссылочным преобразованием*. По существу это условие ограничивает до преобразований, которые оперируют на ссылочных типах и не влияют на двоичное представление ссылки. Это сделано для того, чтобы среда CLR могла знать, что операции будут безопасными к типам, без необходимости во внедрении повсюду кода для действительного преобразования. Как упоминалось в разделе 13.3.2, вариантные преобразования сами по себе являются ссылочными, так что в любом случае дополнительный код будет отсутствовать.

В частности, это ограничение запрещает любые преобразования типов значений и преобразования, определяемые пользователем. Например, все перечисленные ниже преобразования недопустимы.

- `IEnumerable<int>` в `IEnumerable<object>` — упаковывающее преобразование;
- `IEnumerable<short>` в `IEnumerable<int>` — преобразование типа значения;
- `IEnumerable<string>` в `IEnumerable<XName>` — преобразование, определяемое пользователем.

Преобразования, определяемые пользователем, скорее всего, не будут проблемой, т.к. они относительно редки, но ограничение, касающееся типов значений, вы можете счесть досадным.

Параметры out не являются выходными позициями

Это стало для меня неожиданностью, хотя в ретроспективе оно имеет смысл. Рассмотрим делегат со следующим определением:

```
delegate bool TryParser<T>(string input, out T value)
```

Могло показаться, что тип T получилось бы сделать ковариантным — в конце концов, он используется только в выходной позиции... или нет?

На самом деле среде CLR ничего не известно о параметрах out. Что касается данного вопроса, они представляют собой всего лишь параметры ref с примененным к ним атрибутом [Out]. В C# этому атрибуту придается специальный смысл в терминах ясного присваивания, но в CLR это не делается. Параметры ref означают, что данные перемещаются в обоих направлениях, поэтому если есть параметр ref типа T, то тип T инвариантен.

На самом деле, даже если бы среда CLR поддерживала параметры out естественным образом, они все равно не были бы безопасными, поскольку могут применяться во входных позициях в рамках метода; после записи в переменную можно также читать из нее. Было бы неплохо воспринимать параметры out как "копирование значения во время возврата", но это, по сути, смешивает понятия аргумента и параметра, что может привести к проблемам, если они относятся не к одному и тому же типу. Демонстрировать это несколько кропотливо, но на сайте книги предлагается пример.

Делегаты и интерфейсы, в которых используются параметры out, встречаются редко, так что это вас может никогда и не коснуться, но на всякий случай об этом полезно знать.

Вариантность должна быть явной

Когда был представлен синтаксис для выражения вариантности — использование модификатора in или out к параметрам типов — вас могло интересовать, зачем вообще об этом беспокоиться. Компилятор способен проверить любую вариантность, которую вы пытаетесь применить, на предмет допустимости, так почему бы просто не использовать ее автоматически?

Это *могли бы* сделать (по крайней мере, во многих случаях), но я рад, что не сделали. В нормальных обстоятельствах к интерфейсу можно добавлять методы и влиять только на реализации, а не на вызывающий код. Но если объявить, что какой-то параметр типа является вариантным, и затем добавить метод, нарушающий данную вариантность, то это окажет воздействие также и на весь вызывающий код. Предвижу, что такое положение дел вызовет немало путаницы. Вариантность требует определенных размышлений о том, что может понадобиться делать в будущем, и вынуждает разработчиков явно включать модификатор, способствуя тщательному планированию перед принятием решения по поводу вариантности.

Когда дело доходит до делегатов, доводов в пользу такой явной природы меньше; любое изменение сигнатуры, которое могло бы повлиять на вариантность, скорее всего, нарушит работу существующего кода, в котором используется делегат. Но здесь можно многое сказать о согласованности — было бы странно, если требовалось бы задавать вариантность в объявлениях интерфейсов, но не делегатов.

Остерегайтесь нарушающих изменений

Всякий раз, когда становятся доступными новые изменения, возникает риск нарушения работы текущего кода. Например, если вы опираетесь на то, что результаты операций `is` или `as` *не* допускают вариантность, то ваш код будет вести себя по-другому при выполнении в среде .NET 4. Более того, есть случаи, когда инструмент распознавания перегруженных версий будет выбирать другой метод из-за того, что теперь он стал более подходящим. Это еще одна причина для указания вариантности явным образом: она сокращает риск нарушения работы кода.

Такие ситуации должны возникать довольно редко, и польза от вариантности значительно превышает потенциальный ущерб, вызываемый ее недостатками. Ведь для выявления тонких изменений есть модульные тесты, не так ли? На самом деле команда проектировщиков языка C# очень серьезно относится к нарушениям работы кода, но иногда введение нового средства без такого нарушения попросту невозможно.

Групповые делегаты и вариантность не должны смешиваться

Обычно обобщения гарантируют, что во время выполнения проблемы, связанные с безопасностью типов, возникать не будут при условии, что не задействованы приведения. К сожалению, при объединении вариантных типов делегатов возникает опасная ситуация. Лучше всего ее продемонстрировать в коде:

```
Func<string> stringFunc = () => "";
Func<object> objectFunc = () => new object();
Func<object> combined = objectFunc + stringFunc;
```

Этот код компилируется без проблем, поскольку существует ковариантное ссылочное преобразование из выражения типа `Func<string>` в `Func<object>`. Но сам объект по-прежнему имеет тип `Func<string>`, а метод `Delegate.Combine()`, выполняющий фактическую работу, требует, чтобы его аргументы имели одинаковый тип — иначе он не будет знать, какой тип делегата необходимо создать. Во время выполнения показанный выше код приведет к генерации исключения `ArgumentException`.

Эта проблема была обнаружена на относительно позднем этапе цикла выпуска .NET 4, и проектировщики в Microsoft осведомлены о ней. Есть надежда, что проблема будет устранена для большинства случаев в будущем выпуске (в .NET 4.5 она осталась). А до тех пор придется пользоваться обходным путем: можно создать новый объект делегата корректного типа на основе вариантного типа делегата и объединить его с другим делегатом того же самого типа. Например, вот как модифицировать предыдущий код, чтобы сделать его работоспособным:

```
Func<string> stringFunc = () => "";
Func<object> defensiveCopy = new Func<object>(stringFunc);
Func<object> objectFunc = () => new object();
Func<object> combined = objectFunc + defensiveCopy;
```

К счастью, согласно моему опыту, такая проблема возникает редко.

Отсутствие вариантности, задаваемой вызывающим кодом, или частичной вариантности

Хотя по сравнению со всем остальным эта тема затрагивается исключительно ради интереса, полезно отметить, что вариантность C# *совершенно* отличается от вариантности в системе Java. Обобщенной вариантности Java удается оставаться исключительно гибкой за счет обращения к ней с другой стороны: вместо того, чтобы вариантность объявлял сам тип, необходимую вариантность может выражать код, *использующий* этот тип.

Нужна дополнительная информация?

Эта книга посвящена отнюдь не обобщениям Java, но если вы заинтересовались, почитайте статью (на английском языке) Анжелики Лангер "Java Generics FAQs" ("Часто задаваемые вопросы по обобщениям Java"), находящуюся по адресу `http://mng.bz/3qgO`. Будьте осторожны: это огромная и сложная тема!

Например, интерфейс `List<T>` в Java является грубым эквивалентом `IList<T>` в C#. Он содержит методы как для добавления элементов, так и для их извлечения, поэтому в C# он четко инвариантен, но в Java этот тип можно декорировать в вызывающем коде, чтобы выразить, какая вариантность необходима. После этого компилятор предотвратит использование членов, которые противоречат заданной вариантности. Показанный ниже код оказался бы совершенно допустимым:

```
List<Shape> shapes1 = new ArrayList<Shape>();
List<? super Square> squares = shapes1; ← Объявление, использующее контравариантность
squares.add(new Square(10, 10, 20, 20));

List<Circle> circles = new ArrayList<Circle>();
circles.add(new Circle(10, 10, 20));
List<? extends Shape> shapes2 = circles; ← Объявление, использующее ковариантность
Shape shape = shapes2.get(0);
```

В целом я отдаю предпочтение обобщениям C# перед обобщениями Java, к тому же во многих случаях стирание типов может стать особой проблемой. Тем не менее, я нахожу такую трактовку вариантности крайне интересной. Я не ожидаю увидеть что-то подобное в будущих версиях C#, поэтому тщательно обдумывайте, как разделить свои интерфейсы, чтобы обеспечить гибкость, не привнося больше сложности, чем действительно необходимо.

Прежде чем завершить главу, осталось рассмотреть еще два почти тривиальных изменения — то, как компилятор C# обрабатывает операторы `lock`, и события, подобные полям.

13.4. Мелкие изменения в блокировке и событиях, подобных полям

Я не хочу уделять слишком много внимания этим изменениям; возможно, они никогда вас не затронут. Но если вам придется анализировать скомпилированный код в попытках понять, почему он выглядит именно так, а не иначе, то полезно знать, что происходит.

13.4.1. Надежная блокировка

Давайте взглянем на небольшой фрагмент кода C#, в котором применяется блокировка. Детали того, что происходит внутри блокировки, не особенно важны, но для ясности предусмотрен только один оператор:

```
lock (listLock)
{
  list.Add("item");
}
```

До появления C# 4 — и включая C# 4, если вы используете в качестве целевой версию платформы, предшествующую .NET 4, — этот код был бы скомпилирован следующим образом:

```
object tmp = listLock;          <-- ❶ Копирование ссылки для блокировки
Monitor.Enter(tmp);             <-- Запрос блокировки перед блоком try
try
{
  list.Add("item");
}

finally
{
  Monitor.Exit(tmp);            <-- Освобождение блокировки, что бы ни делал метод Add()
}
```

Это *почти* нормально — в частности, удается избежать пары проблем. Требуется гарантия того, что освобождается тот же самый монитор, который был первоначально запрошен, поэтому ссылка для блокировки копируется во временную локальную переменную ❶. Кроме того, это также означает, что выражение блокировки оценивается только один раз. Затем *перед* блоком `try` блокировка запрашивается. Вот почему мы не пытаемся освободить блокировку в блоке `finally` в случае, если поток прекращает работу без успешного ее запрашивания в первую очередь. Это приводит к другой проблеме: если поток прекратится *после* того, как блокировка запрошена, но *перед* входом в блок `try`, то эта блокировка не будет освобождена. В результате вполне может возникнуть взаимоблокировка — другой поток может бесконечно ожидать освобождения блокировки данным потоком. Хотя на протяжении длительного времени среда CLR всячески препятствовала этому, возможность все же исключена не полностью.

Необходим какой-то способ атомарного запроса блокировки и получения свидетельства о том, что она была успешно запрошена. К счастью, в .NET 4 это стало возможным благодаря новой перегруженной версии метода `Monitor.Enter()`, которую компилятор C# 4 применяет следующим образом:

```
bool acquired = false;
object tmp = listLock;
try
{
  Monitor.Enter(tmp, ref acquired); <-- Запрос блокировки внутри блока try
  list.Add("item");
}

finally
{
  if (acquired)                     <-- ❷ Условное освобождение блокировки
  {
    Monitor.Exit(tmp);
  }
}
```

Теперь блокировка будет освобождаться согласованным образом тогда и только тогда, когда она в первую очередь успешно получена.

Надо заметить, что в некоторых случаях взаимоблокировка — не самый худший результат; иногда приложению опаснее продолжить работу, чем просто остановиться[9]. Однако *полагаться* на такое условие взаимоблокировки нелепо; лучше по возможности вообще избегать прекращения потоков. (Прекращение текущего выполняющегося потока несколько лучше, т.к. вы имеете больший контроль — это то, что делает метод `Response.Redirect()` в ASP.NET, например, но я по-прежнему рекомендую поискать более удачные формы управления потоком.)

Осталось раскрыть последнюю тему, прежде чем мы перейдем к рассмотрению действительно крупного средства C# 4.

13.4.2. Изменения в событиях, подобных полям

В способ реализации *событий, подобных полям*, в C# 4 были внесены два изменения, которые заслуживают краткого упоминания. Вряд ли они коснутся вас, хотя они являются *потенциально* нарушающими изменениями.

Просто ради повторения: события, подобные полям — это события, которые объявляются так, как если бы они были полями, без блоков добавления/удаления, примерно так:

```
public event EventHandler Click;
```

Во-первых, изменился способ достижения безопасности потоков. До выхода C# 4 события, подобные полям, давали в результате код, который применяет блокировку либо на `this` (для событий экземпляра), либо на объявленном типе (для статических событий). Начиная с C# 4, компилятор реализует безопасную к потокам атомарную подписку и отмену подписки с использованием метода `Interlocked.CompareExchange<T>()`. В отличие от предыдущего изменения, внесенного в оператор `lock`, это применимо даже в случае использования в качестве целевой платформы более ранних версий .NET Framework.

Во-вторых, изменился смысл имени события *внутри объявленного класса*. В прошлом, если вы подписывались на событие (или отменяли подписку) внутри класса, который содержал объявление этого события (например, `Click += DefaultClickHandler;`), это вело непосредственно к поддерживающему полю, полностью пропуская реализацию блоков добавления/удаления. Теперь это не так; когда вы применяете операцию += или -=, имя события относится к самому событию, а не к поддерживающему полю. Когда это имя используется для любых других целей (обычно для присваивания или вызова), оно по-прежнему ссылается на поддерживающее поле.

Оба изменения разумны и делают код более аккуратным, хотя, возможно, вы и не замечали их при ежедневном употреблении. Если вас интересует дополнительная информация, то Крис Барроуз детально разбирает эту тему в своем блоге по адресу `http://mng.bz/Kyr4`.

---

[9] На эту тему у Эрика Липперта есть великолепная запись в блоге, озаглавленная "Locks and exceptions do not mix" ("Блокировки и исключения не смешиваются"): `http://mng.bz/Qy7p`.

13.5. Резюме

Эта глава немного напоминала смесь несвязанных фрагментов, относящихся к разнообразным индивидуальным областям. С другой стороны, технология COM значительно выигрывает от именованных аргументов и необязательных параметров, поэтому некоторое перекрытие между областями все же было.

Я подозреваю, что разработчикам на C# понадобится определенное время, чтобы наловчиться эффективно пользоваться новыми возможностями для параметров и аргументов. Перегрузка по-прежнему обеспечивает дополнительную переносимость для языков, не поддерживающих необязательные параметры, а именованные аргументы в некоторых ситуациях могут выглядеть странными, пока вы не привыкнете к ним. Хотя преимущества могут быть значительными, как демонстрировалось в примере с построением экземпляров неизменяемых типов. Вам придется проявлять определенную осторожность во время присваивания стандартных значений необязательным параметрам, но я надеюсь, что вы сочтете совет относительно применения `null` как "стандартного значения по умолчанию" полезным и гибким, поскольку этот прием позволяет эффективно обойти ряд ограничений и ловушек, которые могли бы возникнуть в противном случае.

Работа с технологией COM прошла *очень долгий* путь до C# 4. Я по-прежнему предпочитаю использовать чистые управляемые решения, где они доступны, но, по крайней мере, взаимодействующий с COM код теперь стал более читабельным и улучшена процедура его развертывания. Мы пока еще рассмотрели не все усовершенствования во взаимодействии с COM, т.к. средства динамической типизации, обсуждаемые в следующей главе, также оказывают влияние на COM. Но, даже не принимая их во внимание, вы видели, как с применением нескольких простых шагов краткий пример стал намного более симпатичным.

Последней важной темой в этой главе была обобщенная вариантность, теперь доступная для интерфейсов и делегатов. Иногда вы можете использовать вариантность, даже не подозревая об этом, и я думаю, что большинство разработчиков более охотно будут применять вариантность, объявленную в интерфейсах и делегатах инфраструктуры, чем создавать что-то свое. Мне жаль, если временами материал становился сложным, но полезно знать, что именно там происходит. Если это как-то вас утешит, то Эрик Липперт, бывший член команды проектировщиков языка C#, публично признался в своем блоге (`http://mng.bz/79d8`), что функции высшего порядка *даже у него* вызывают головную боль, так что вы в неплохой компании. Эта запись в блоге Эрика является одной из серии записей, посвященных вариантности (`http://mng.bz/94H3`), которые, как и все остальные, представляют собой обмен мнениями относительно задействованных проектных решений. Если знаний вариантности пока еще недостаточно, исключительно полезно почитать их.

Ради полноты мы также взглянули на изменения, внесенные в способ обработки компилятором C# блокировки и событий, подобным полям.

В этой главе речь шла об *относительно* небольших изменениях в языке C#. В главе 14 будет рассматриваться нечто намного более фундаментальное: возможность использования C# в динамической манере.

Глава 14

Динамическое связывание в статическом языке

В этой главе...

- Что означает быть динамическим
- Как использовать динамическую типизацию в C# 4
- Примеры с COM, Python и рефлексией
- Как реализуется динамическая типизация
- Динамическое реагирование

Язык C# всегда был статически типизированным безо всяких исключений. Существовало несколько областей, где компилятор искал определенные имена, а не интерфейсы, такие как нахождение подходящих методов `Add()` для инициализаторов коллекций, но в языке не было ничего по-настоящему динамического помимо обычного полиморфизма. В C# 4 это изменилось — во всяком случае, частично. Простейший способ объяснения состоит в следующем: появился новый статический тип по имени `dynamic`, с помощью которого можно пробовать делать почти все на этапе компиляции и позволить инфраструктуре разобраться со всем этим во время выполнения. Конечно, его характеристики сказанным не исчерпываются, но это основные положения.

С учетом того, что язык C# по-прежнему остается статически типизированным везде, где *не* используется `dynamic`, я не ожидаю внезапного перехода поклонников динамического программирования в лагерь сторонников C#. Это не является причиной добавления данного средства: оно предназначено главным образом для поддержки взаимодействия. Когда в экосистему .NET влились динамические языки, такие как IronRuby и IronPython, было бы нелепо не иметь возможности вызывать код C# из кода IronPython и наоборот. Подобным же образом, разработка с применением API-интерфейсов COM

раньше была неудобной в C# и предполагала обилие приведений, разбросанных в коде. Динамическая типизация решает все эти проблемы. С другой стороны, существует множество проектов, в которых динамическая типизация применяется в рамках C# для упрощения доступа к данным.

В этой главе часто будет повторяться одно предупреждение: в отношении динамической типизации следует проявлять осторожность. Динамическую типизацию интересно исследовать, и она хорошо реализована, но я по-прежнему рекомендую тщательно подумать, прежде чем приступать к ее интенсивному использованию. Как и в случае любого другого нового средства, взвесьте все "за" и "против" вместо того, чтобы необдуманно заняться им только из-за лаконичности (которая, несомненно, присуща средству динамической типизации). Инфраструктура выполняет немалую работу по оптимизации динамического кода, но в большинстве случаев он будет медленнее, чем статический код. Более важен факт утраты значительной части безопасности на этапе компиляции. Несмотря на то что модульное тестирование помогает найти множество ошибок, которые могут неожиданно обнаружиться, когда компилятор не способен оказать значимую помощь, я все же предпочитаю получать от компилятора немедленный отклик при попытке вызова метода, который не существует или не может быть вызван с заданным набором аргументов.

Тем не менее, есть ситуации, при которых уровень безопасности, обеспечиваемый компилятором, изначально не сильно строг. Например, с кодом, применяющим рефлексию, многое может пойти не так, и далеко не все ошибки компилятор сумеет выявить. Действительно ли существует метод, который вы пытаетесь вызвать, указывая его имя? Доступен ли он вашему коду? Предоставлены ли подходящие аргументы? Компилятор не сможет помочь ни с одной из перечисленных проблем. Эквивалентный динамический код по-прежнему не позволяет выявить такие ошибки на этапе компиляции, но он хотя бы проще в чтении и понимании. Все сводится к использованию наиболее приемлемого подхода для решения конкретной проблемы, над которой производится работа.

Динамическое поведение может быть удобно в ситуациях, в которых естественным образом приходится иметь дело с динамическими средами или данными, но если вы на самом деле собираетесь писать крупные порции кода динамически, то я советую обратиться к языку, где такой стиль является *нормальным*, а не исключительным. Все же язык C# был *спроектирован* для статической типизации; языки, которые были динамическими с самого начала, часто располагают разнообразными средствами, помогающими более продуктивно работать с динамическим поведением. Сейчас, когда из C# легко вызывать код на таких языках, реализуемый код можно разделить на части: те, которые извлекают преимущество от почти полностью динамического стиля, и те, которые лучше работают со статической типизацией.

Я не хотел бы преуменьшать важность динамического поведения. Там, где динамическая типизация *удобна*, она может быть намного проще альтернативных решений. В этой главе мы рассмотрим базовые правила динамической типизации в C# 4 и затем ознакомимся с несколькими примерами: взаимодействие с COM динамическим образом, обращение к коду на IronPython и значительное упрощение рефлексии. Все это можно делать, не вникая в детали, но после того, как вы получите представление о динамической типизации, мы посмотрим, что происходит "за кулисами". В частности, мы обсудим исполняющую среду динамического языка (Dynamic Language Runtime – DLR) и действия, которые компилятор C# предпринимает, когда встречает динамический код. Наконец, вы увидите, как реализовать для собственных типов динамическое реагирование на вызовы методов, доступ к свойствам и тому подобное. Но сначала нужно сделать шаг назад.

14.1. Что? Когда? Почему? Как?

Перед тем, как перейти к написанию кода, демонстрирующего это новое средство C# 4, полезно разобраться в причине его появления. Я не знаю ни одного другого языка, который бы перешел из полностью статического в частично динамический; это значительный шаг в эволюции C# независимо от того, часто вы применяете динамическую типизацию либо только изредка.

Мы начнем с того, что по-новому взглянем на смысл понятий *динамическая* и *статическая*, принимая во внимание ряд основных сценариев использования динамической типизации в C#, и затем погрузимся в ее реализацию в рамках C# 4.

14.1.1. Что такое динамическая типизация?

В главе 2 были приведены характеристики системы типов и объяснения того, что C# ранее был статически типизированным языком. Компилятору известны типы выражений в коде, а также члены, доступные в любом типе. Он применяет довольно сложный набор правил для точного определения члена, который должен использоваться в той или иной ситуации. Это включает распознавание перегруженных версий; единственное, что откладывается на более поздний период — выбор реализаций виртуальных методов, который зависит от типа объекта во время выполнения. Процесс выяснения, какой член подлежит применению, называется *связыванием*, и в статически типизированном языке оно происходит на этапе компиляции.

В динамически типизированном языке все связывание осуществляется во время выполнения. Компилятор или анализатор может проверить код на предмет *синтаксической* правильности, но не способен удостовериться в том, что методы, которые вызываются, или свойства, к которым производится доступ, действительно существуют. Ситуация немного напоминает текстовый процессор без словаря: он может проверить пунктуацию, но не правописание, поэтому если нужна какая-то уверенность в корректной работе кода, крайне необходим качественный набор модульных тестов. Некоторые динамические языки являются только интерпретируемыми, и компилятор в них вообще не задействован. Другие предоставляют как интерпретатор, так и компилятор, чтобы сделать возможным быструю разработку с применением среды REPL (read, evaluate, print loop — бесконечный цикл из чтения, вычисления, вывода).

REPL и C#

Строго говоря, среда REPL связана не только с динамическими языками. Некоторые статически типизированные языки располагают *интерпретаторами*, которые компилируют на лету. Особенно в этом плане заметен язык F#, сопровождаемый инструментом под названием *F# Interactive*, который делает в точности сказанное. Однако интерпретаторы намного более распространены для динамических языков, чем для статических.

В C# имеются похожие инструменты: средство вычисления выражений, лежащее в основе окон Watch (Контрольное значение) и Immediate (Интерпретация) среды Visual Studio, можно считать формой REPL, а в проекте Mono есть C# Shell (`www.mono-project.com/CsharpRepl`).

Следует отметить, что новые динамические средства C# 4 *не* предусматривают интерпретацию исходного кода C# во время выполнения; к примеру, прямой эквивалент JavaScript-функции `eval()` отсутствует. Для выполнения кода, основанного на строковых данных, должен использоваться либо API-интерфейс CodeDOM (в частности, тип `CSharpCodeProvider`), либо простая рефлексия с целью вызова отдельных членов. Еще одним вариантом является проект Roslyn, хотя на время написания он был доступен лишь в форме Community Technology Preview.

Разумеется, тот же вид работы должен быть сделан в *определенный* момент времени независимо от того, какой подход взят на вооружение. За счет запрашивания у компилятора большего объема действий перед выполнением статические системы обычно работают лучше динамических. Учитывая упомянутые до сих пор недостатки, может удивить, почему вообще у кого-либо возникает желание возиться с динамической типизацией.

14.1.2. Когда динамическая типизация удобна и почему?

В пользу динамической типизации можно привести два важных аргумента. Прежде всего, если вам известно имя вызываемого члена, необходимые аргументы и объект, на котором он должен быть вызван, то это все, что требуется. Это может выглядеть как вся информация, которую нужно было бы иметь в любом случае, но обычно компилятору C# понадобятся дополнительные сведения. Для точной идентификации члена критически важно знать тип объекта, на котором производится вызов, и типы аргументов. Иногда эти типы на этапе компиляции не известны, несмотря на то, что знаний *вполне* достаточно для того, чтобы иметь уверенность в наличии подходящего члена во время выполнения кода.

Например, если вы знаете, что используемый объект имеет интересующее вас свойство `Length`, то не играет никакой роли, к какому типу относится этот объект — `String`, `StringBuilder`, `Array`, `Stream` или любому другому типу, содержащему указанное свойство. Вы не нуждаетесь в том, чтобы это свойство определялось каким-то общим базовым классом или интерфейсом, что было бы полезно в случае отсутствия такого типа. Это называется *утиной типизацией* и происходит из следующего утверждения: "Если нечто выглядит как утка, плавает как утка и крякает как утка, то это, наверное, и есть утка"[1]. Даже когда *имеется* тип, который предлагает все необходимое, временами раздражает потребность в сообщении компилятору, о каком точно типе идет речь. Особенно это заметно во время применения API-интерфейсов Microsoft Office через COM. Многие методы и свойства объявлены как возвращающие тип `VARIANT`, а это означает, что код C#, использующий такие вызовы, часто обильно сдобрен приведениями. Утиная типизация позволяет опустить все эти приведения при условии, что вы уверены в том, что делаете.

Второй важной характеристикой динамической типизации является возможность объекта отвечать на вызов, анализируя предоставленные ему имя и аргументы. Она может себя вести так, как если бы член был объявлен типом обычным образом, даже хотя имена членов невозможно узнать вплоть до времени выполнения. Например, взгляните на следующий вызов:

```
books.FindByAuthor("Joshua Bloch")
```

В нормальных обстоятельствах это требовало бы объявления члена `FindByAuthor()` проектировщиком задействованного типа. На уровне динамических данных может быть

---

[1] Статья в Википедии об утиной типизации содержит больше сведений об истории происхождения этого термина: `http://ru.wikipedia.org/wiki/Утиная_типизация`.

предусмотрена небольшая порция кода для анализа вызовов подобного рода. Она могла бы обнаружить, что в связанных данных (будь то данные из базы данных, XML-документ, жестко закодированные данные или что-нибудь еще) имеется свойство `Author`, и действовать соответствующим образом.

В таком случае было бы принято решение, что вы хотите выполнить запрос с применением указанного аргумента в качестве имени автора. В некотором смысле это всего лишь более сложный способ записи кода, подобного показанному ниже:

```
books.Find("Author", "Joshua Bloch")
```

Однако первый фрагмент кода выглядит более уместным; вызывающему коду известна часть `Author` статически, несмотря на то, что получающий код о ней не знает. В некоторых ситуациях этот подход может использоваться для имитации предметно-ориентированных языков (domain-specific language — DSL). Он также может применяться при создании естественного API-интерфейса для исследования структур данных, таких как деревья XML.

Еще одна особенность программирования на динамических языках *способствует* экспериментальному стилю программирования с использованием подходящего интерпретатора, как упоминалось ранее. Это не относится *напрямую* к C# 4, но тот факт, что C# 4 может широко взаимодействовать с динамическими языками, функционирующими под управлением среды DLR, означает возможность решения задачи на одном из динамических языков и потреблению результатов непосредственно в коде C#, без необходимости в последующем переносе решения в C#.

Мы рассмотрим описанные сценарии более глубоко на конкретных примерах, когда обсудим основы динамических возможностей C# 4. Полезно кратко отметить, что если эти преимущества к вам *не* применимы, то динамическая типизация, скорее всего, будет помехой, нежели помощью. Многие разработчики не нуждаются в использовании динамической типизации при ежедневном кодировании, и даже когда она требуется, это может касаться только небольшой части кода. Как и любым средством, им можно злоупотребить. На мой взгляд, обычно лучше хорошо подумать, не позволят ли какие-то альтернативные проектные решения справиться с той же самой задачей более элегантно с применением статической типизации. Но мое мнение предвзято из-за наличия опыта работы со статически типизированными языками — чтобы увидеть широкое многообразие преимуществ кроме тех, что представлены здесь, полезно почитать книги по динамически типизированным языкам, таким как Python и Ruby.

Вероятно, вы уже горите желанием увидеть какой-то реальный код прямо сейчас, поэтому далее будет представлен краткий обзор того, что происходит, и затем приведено несколько примеров.

14.1.3. Как в C# 4 поддерживается динамическая типизация?

В C# 4 введен новый тип по имени `dynamic`. Компилятор трактует этот тип не так, как любой нормальный тип CLR[2]. Выражение, в котором используется динамическое значение, приводит к радикальному изменению поведения компилятора. Вместо попытки выяснения, для чего *точно* предназначен код, подходящего связывания каждого обращения к членам, распознавания перегруженных версий и выполнения тому подобных действий, компилятор просто анализирует исходный код, чтобы определить *вид* опера-

[2] На самом деле `dynamic` не представляет отдельный тип CLR. В действительности это тип `System.Object` в сочетании с атрибутом `System.Dynamic.DynamicAttribute`. Мы рассмотрим данный аспект более подробно в разделе 14.4, но пока можно делать вид, что это реальный тип.

ции, которую вы пытаетесь запустить, ее имени, задействованных аргументов и любой другой существенной информации. Вместо выдачи инструкций IL для выполнения кода напрямую компилятор генерирует код, который обращается к среде DLR со всей требуемой информацией. Остальная работа производится во время выполнения.

Во многих отношениях это похоже на разные виды кода, генерируемого в результате преобразования лямбда-выражений. Может быть получен либо код для выполнения требуемых действий (в случае преобразования в тип делегата), либо код, который строит *описание* требуемых действий (в случае преобразования в дерево выражения). Позже вы увидите, что деревья выражений исключительно важны в среде DLR, и компилятор C# часто будет использовать деревья выражений для описания кода. (В простейших случаях, когда нет ничего кроме обращения к члену, необходимость в наличии дерева выражения отсутствует.) Когда среда DLR добирается до связывания подходящего вызова во время выполнения, она проходит через сложный процесс, призванный определить, что должно произойти. Этот процесс не только должен принимать во внимание обычные правила C# для распознавания перегруженных версий методов и т.д., но также учитывать возможность того, что сам объект может стремиться стать частью решения, как было ранее показано в примере с `FindByAuthor()`.

Большинство действий происходит "за кулисами" — исходный код, который вы пишете для применения динамической типизации, может быть действительно простым.

14.2. Пятиминутное руководство по dynamic

Помните ли вы, сколько новых элементов синтаксиса было задействовано при изучении LINQ? Динамическая типизация в этом отношении является полной противоположностью: есть одно контекстное ключевое слово `dynamic`, которое можно использовать в большинстве мест, где применялось бы имя типа. Это и весь необходимый новый синтаксис. Главные правила, касающиеся `dynamic`, выразить легко.

- Существует неявное преобразование из почти любого типа CLR в `dynamic`.
- Существует неявное преобразование из любого выражения типа `dynamic` в почти любой тип CLR.
- Выражения, в которых используется значение типа `dynamic`, обычно вычисляются динамически.
- Статическим типом динамически вычисляемого выражения, как правило, считается `dynamic`.

Как вы увидите в разделе 14.4, подробные правила сложнее, но на данный момент давайте придерживаться упрощенной версии.

В листинге 14.1 приведен полный пример, демонстрирующий все перечисленные выше правила.

Листинг 14.1. Применение `dynamic` для прохода по списку с конкатенацией строк

```
dynamic items = new List<string> { "First", "Second", "Third" };
dynamic valueToAdd = "!";
foreach (dynamic item in items)
{
  string result = item + valueToAdd;
  Console.WriteLine(result);
}
```

Результат выполнения кода из листинга 14.1 не должен стать неожиданностью: на консоль выводятся строки `First!`, `Second!` и `Third!`. В этом случае можно было бы легко указать типы переменных `items` и `valueToAdd` явно, и все работало бы нормально, но представьте себе, что вместо жесткого кодирования эти переменные получают свои значения из других источников данных. Что произойдет, если понадобится добавить целое число, а не строку?

В листинге 14.2 содержится небольшая вариация. *Объявление* переменной `valueToAdd` не изменилось, а модифицировано только выражение присваивания.

Листинг 14.2. Динамическое сложение целых числе и строк

```
dynamic items = new List<string> { "First", "Second", "Third" };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
  string result = item + valueToAdd;   <-- Конкатенация string и int
  Console.WriteLine(result);
}
```

На этот раз первым результатом является `First2`, который, надо надеяться, вы ожидали. Используя статическую типизацию, пришлось бы явно изменить объявление `valueToAdd` с `string` на `int`. Тем не менее, операция сложения по-прежнему строит строку. А что если вы также измените тип элементов на целочисленный? Давайте опробуем одну простую модификацию, показанную в листинге 14.3.

Листинг 14.3. Сложение целочисленных значений друг с другом

```
dynamic items = new List<int> { 1, 2, 3 };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
  string result = item + valueToAdd;   <-- Сложение int и int
  Console.WriteLine(result);
}
```

Вот беда! По-прежнему предпринимается попытка преобразовать результат сложения в строку. Единственными разрешенными преобразованиями являются те, которые обычно существуют в языке C#, поэтому никаких преобразований из `int` в `string` не доступно. В результате генерируется исключение (во время выполнения, конечно же):

```
Unhandled Exception:
  Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
Cannot implicitly convert type 'int' to 'string'
  at CallSite.Target(Closure , CallSite , Object )
  at System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet]
       (CallSite site, T0 arg0)
  ...
Необработанное исключение:
  Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
Не удалось неявно преобразовать тип int в string
  в CallSite.Target(Closure , CallSite , Object )
  в System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet]
       (CallSite site, T0 arg0)
  ...
```

Если только вы не являетесь полным совершенством, то вполне возможно будете много раз сталкиваться с исключением RuntimeBinderException, когда приступите к применению динамической типизации. В некоторой степени оно представляет собой новое исключение NullReferenceException; вы обязательно столкнетесь с ним, но при определенной доле везения это случится в контексте модульных тестов, а не в отчетах об ошибках, поступающих от клиентов. Как бы то ни было, ситуацию можно исправить, изменив тип переменной result на dynamic, так что преобразование не потребуется.

Если вдуматься, то зачем вообще возиться с переменной result? Можно было бы просто вызвать метод Console.WriteLine() непосредственно. В листинге 14.4 представлены изменения.

Листинг 14.4. Сложение целых чисел с целым числом, но без генерации исключения

```
dynamic items = new List<int> { 1, 2, 3 };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
  Console.WriteLine(item + valueToAdd); <-- Вызов перегруженной версии с аргументом int
}
```

В результате на консоль вполне ожидаемо выводятся числа 3, 4 и 5. Изменение входных данных теперь изменяет не только операцию, которая выбирается во время выполнения, но также и вызываемую перегруженную версию метода Console.WriteLine(). При первоначальных данных вызывалась бы версия Console.WriteLine(string), а после обновления переменных — версия Console.WriteLine(int). Данные могли бы даже содержать смесь значений, изменяя вызываемую версию метода на каждой итерации!

Использовать dynamic в качестве объявленного типа можно также для полей, параметров и возвращаемых типов. Это резко контрастирует с применением ключевого слова var, которое ограничено локальными переменными.

Отличия между var и dynamic

Во многих примерах, показанных до сих пор, когда типы были действительно известны на этапе компиляции, для объявления переменных можно было бы использовать var. На первый взгляд, эти два средства выглядят очень похожими. Оба случая подобны объявлению переменной без указания ее типа, но с помощью dynamic тип переменной явно устанавливается в динамический. Применять var можно только в случае, если компилятор способен вывести тип переменной *статически*, и система типов действительно остается полностью статической. Разумеется, если вы используете var для переменной, которая инициализирована выражением типа dynamic, то переменная в итоге также будет типизирована (статически) как dynamic. Учитывая путаницу, которую это может вызвать, я настоятельно рекомендую не поступать так.

Компилятор осведомлен о записываемой им информации, а код, который затем *использует* это информацию во время выполнения, не менее искусен: его по праву можно считать мини-компилятором C#. Он применяет любую статическую информацию о типах, которая была известна на этапе компиляции, чтобы обеспечить по возможности максимально интуитивно понятное поведение кода.

За исключением нескольких деталей о том, чего *нельзя* сделать с помощью динамической типизации, это все сведения, которые действительно необходимо знать, чтобы приступить к использованию динамической типизации в своем коде. Позже мы еще вернемся к ограничениям, а также к подробной информации о том, что компилятор на самом деле делает, но сначала давайте посмотрим, где динамическая типизация по-настоящему *удобна*.

14.3. Примеры применения динамической типизации

Динамическая типизация немного похожа на небезопасный код, или взаимодействие с машинным кодом с использованием P/Invoke. Многие разработчики или не будут в ней нуждаться, или будут применять раз в сто лет. Для других разработчиков — особенно для тех, кто имеет дело с Microsoft Office — динамическая типизация даст большой скачок в продуктивности, либо упрощая существующий код, либо делая возможными радикально отличающиеся подходы к решению задач.

Этот раздел ни в коем случае не претендует на то, чтобы быть исчерпывающим. После выхода в свет второго издания этой книги в нескольких проектах с открытым кодом была успешно задействована динамическая типизация; в их число входят Massive (`https://github.com/robconery/massive`), Dapper (`http://code.google.com/p/dapper-dot-net/`) и Json.NET (`http://json.codeplex.com`). Все названные примеры работают с данными — будь то взаимодействие с базой данных, сериализация или десериализация JSON. Конечно, это не говорит о том, что динамическая типизация полезна только в области данных, и мне не хотелось бы предсказывать, какие новаторские случаи использования может предъявить сообщество в будущем.

Мы рассмотрим здесь три примера: работу с Excel, вызов кода Python и применение обычных управляемых типов .NET более гибким образом.

14.3.1. COM в общем и Microsoft Office в частности

Вы уже видели большую часть новых средств C# 4, касающихся взаимодействия с COM, но в главе 13 не раскрывалось одно средство из-за того, что на тот момент не была описана динамическая типизация.

Если вы решите встраивать используемые типы взаимодействия в сборку (указав компилятору ключ командной строки `/l` или установив свойство Embed Interop Types (Встраивать типы взаимодействия) в `true`), то все в API-интерфейсе, что иначе было бы объявлено как `object`, изменится на `dynamic`. Это намного упростит работу с относительно слабо типизированными API-интерфейсами вроде тех, которые предлагает пакет Office. (Хотя объектная модель в Office умеренно строга сама по себе, многие свойства открыты как имеющие *вариантные типы*, поскольку они могут работать с числами, строками, датами и т.д.)

Здесь снова будет представлен краткий пример — даже короче примера с Word из главы 13. Динамический аспект легко понять именно в таком сценарии. Мы установим первые 20 ячеек в верхней строке нового листа Excel в числа от 1 до 20. В листинге 14.5 показана начальная статически типизированная порция кода, позволяющая решить эту задачу.

Листинг 14.5. Установка диапазона значений с помощью статической типизации

```
var app = new Application { Visible = true };  ❶ Открыть Excel с активным рабочим листом
app.Workbooks.Add();
Worksheet worksheet = (Worksheet) app.ActiveSheet;
Range start = (Range) worksheet.Cells[1, 1];  ❷ Определить начальную и конечную ячейки
Range end = (Range) worksheet.Cells[1, 20];
worksheet.Range[start, end].Value
     = Enumerable.Range(1, 20).ToArray();  ❸ Заполнить диапазон значений [1, 20]
```

Этот код полагается на наличие директивы `using` для пространства имен `Microsoft.Office.Interop.Excel` (здесь не показанной), так что на этот раз тип `Application` относится к Excel, а не к Word. Кроме того, здесь применяются новые возможности C# 4, позволяющие не указывать аргумент для необязательного параметра в вызове `Workbooks.Add()` при настройке среды ❶, а также использовать именованный индексатор ❷.

Когда приложение Excel запущено, выясняются начальная и конечная ячейки всего диапазона. В этом случае они находятся в одной и той же строке, но взамен можно было бы создать прямоугольный диапазон, выбрав два противоположных угла. Создать диапазон можно было бы и с помощью единственного вызова метода `Range["A1:T1"]`, но лично я нахожу работу только с числами более простой. Имена ячеек наподобие B3 хороши для людей, но труднее для применения в программах.

Имея диапазон, можно установить все значения в нем, присваивая свойству `Value` массив целых чисел ❸. Поскольку устанавливается одиночная строка, применяется одномерный массив; для установки диапазона, охватывающего несколько строк, понадобится прямоугольный массив.

Все это работает, но в шести строках кода пришлось использовать три приведения. Индексатор, вызываемый через `Cells`, и свойство `ActiveSheet` обычно объявлены как возвращающие тип `object`. (Разнообразные параметры *также* объявлены с типом `object`, но это не играет особой роли, т.к. существует неявное преобразование из любого типа, отличного от указателя, в тип `object` — приведение требуется только для обратного преобразования.) Приложение Excel в конце кода не закрывается, поэтому вы можете наблюдать на экране открытый рабочий лист.

В случае встраивания основной сборкой взаимодействия требуемых типов в ваш двоичный файл типы во всех примерах становятся `dynamic`. Благодаря неявному преобразованию из `dynamic` в другие типы, можно устранить все приведения, как показано в листинге 14.6.

Листинг 14.6. Использование неявных преобразований из `dynamic` при работе с Excel

```
var app = new Application { Visible = true };
app.Workbooks.Add();
Worksheet worksheet = app.ActiveSheet;
Range start = worksheet.Cells[1, 1];
Range end = worksheet.Cells[1, 20];
worksheet.Range[start, end].Value = Enumerable.Range(1, 20)
                                                .ToArray();
```

Это тот же самый код, как в листинге 14.5, но без приведений.

Полезно отметить, что преобразования по-прежнему проверяются во время выполнения. Если вы измените тип в объявлении `start` на `Worksheet`, то преобразование потерпит неудачу и сгенерируется исключение. Разумеется, вы вовсе не обязаны выполнять преобразование. *Можно было бы* оставить все с типом `dynamic`, как показано в листинге 14.7.

Листинг 14.7. Использование во всех местах типа `dynamic`

```
var app = new Application { Visible = true };
app.Workbooks.Add();
dynamic worksheet = app.ActiveSheet;
dynamic start = worksheet.Cells[1, 1];
dynamic end = worksheet.Cells[1, 20];
worksheet.Range[start, end].Value = Enumerable.Range(1, 20)
                                                .ToArray();
```

Какой код яснее? Я сторонник старомодной статической типизации, поэтому отдаю предпочтение версии из листинга 14.6. В каждой строке заявлены ожидаемые типы, поэтому если возникают какие-то вопросы, я могу выяснить их немедленно, не дожидаясь до момента, когда буду пытаться применить значение способом, который оно не поддерживает.

В терминах продуктивности на начальных этапах разработки оба подхода обладают как достоинствами, так и недостатками. В случае использования `dynamic` не приходится выяснять, какой конкретный тип ожидается; можно просто работать со значением, и до тех пор, пока все необходимые операции поддерживаются, все в порядке.

С другой стороны, за счет применения статической типизации на каждой стадии можно просматривать, что именно доступно, с помощью средства IntelliSense. Динамическая типизация по-прежнему используется для предоставления неявного преобразования в `Worksheet` и `Range` — только она применяется пошагово, а не одновременно. Переход от статической типизации к динамической поначалу может не выглядеть чем-то особенным, т.к. этот пример относительно прост, но по мере увеличения сложности кода будут появляться преимущества лучшей читабельности из-за устранения всех ранее нужных приведений.

В некоторой степени все это было оттолоском из прошлого — COM является довольно старой технологией. А теперь давайте займемся взаимодействием с чем-то более новым: IronPython.

14.3.2. Динамические языки, подобные IronPython

В этом разделе в качестве примера используется только IronPython, однако он определенно не является единственным динамическим языком, доступным для DLR. Возможно, он самый зрелый, но уже существуют альтернативы, такие как IronRuby и IronScheme. Одна из заявленных целей среды DLR заключается в том, чтобы упростить перспективным проектировщикам языков создание работающего языка, который имеет доступ к огромному числу библиотек .NET Framework, а также обладает хорошими возможностями взаимодействия с другими языками DLR и традиционными языками .NET вроде C#.

Зачем может понадобиться использовать IronPython из C#?

Есть много причин, по которым может потребоваться взаимодействие с динамическим языком, подобно тому, как было выгодно взаимодействовать с другими управляемыми языками на заре развития .NET. Вполне очевидно, что возможность применения библиотеки классов, написанной на C#, разработчиком VB (и наоборот) очень удобна, так почему бы это не оказалось справедливым в отношении динамических языков? Я попросил Майкла Фурда, одного из авторов книги *Iron Python in Action* (Manning, 2009 г.), подать несколько идей по использованию IronPython внутри приложения C#. Вот его список:

- написание пользовательских сценариев;
- реализация определенного уровня в приложении на IronPython;
- применение Python в качестве языка конфигурации;
- использование Python как механизма поддержки правил с хранением самих правил в текстовом виде (даже в базе данных);
- применение библиотеки, которая доступна в Python, но не имеет эквивалента в .NET;
- помещение внутрь приложения интерактивного интерпретатора, предназначенного для целей отладки.

Если вы все еще не избавились от скепсиса, то примите во внимание, что внедрение сценарного языка в широко распространенные приложения в наши дни не такая уж редкость — к примеру, компьютерная игра *Цивилизация IV* Сида Мейера[3] оснащена возможностью поддержки сценариев с помощью Python. И это не просто запоздалое решение в пользу модификаций — много основных игровых сценариев написано на Python. После построения механизма разработчики обнаружили, что он оказался более мощной средой разработки, чем предполагалось первоначально.

В этой главе я буду работать с одним примером использования Python в качестве языка конфигурации. Как и пример с COM, он будет простым, но вполне достаточным, чтобы послужить стартовой точкой для дальнейшего экспериментирования, если возникнет интерес.

Начало работы: внедрение "hello, world"

Для размещения или внедрения другого языка внутрь приложения C# в зависимости от желаемого уровня гибкости и контроля доступны разнообразные типы. Здесь будут применяться только `ScriptEngine` и `ScriptScope`, поскольку требования элементарны. В данном примере известно, что всегда будет использоваться язык Python, поэтому можно запросить у инфраструктуры IronPython создание экземпляра `ScriptEngine` напрямую; в более общих ситуациях можно применять `ScriptRuntime` для динамического выбора реализации языка по имени. В более требовательных сценариях может понадобиться работать с `ScriptHost` и `ScriptSource`, а также использовать дополнительные возможности других типов.

Не довольствуясь однократным выводом на консоль строки `hello, world`, этот начальный пример сделает это *дважды*, один раз с применением текста, переданного механизму в виде строки, и еще раз путем загрузки файла по имени `HelloWorld.py`.

В листинге 14.8 представлено все необходимое.

---

[3] Или The Way of Life в зависимости от ваших взглядов на мир и предрасположенности к играм.

Листинг 14.8. Вывод на консоль строки `hello, world` два раза с использованием Python, внедренного в C#

```
ScriptEngine engine = Python.CreateEngine();
engine.Execute("print 'hello, world'");
engine.ExecuteFile("HelloWorld.py");
```

Вы можете найти код в этом листинге либо довольно скучным, либо очень интересным, причем по одной и той же причине. Он прост для понимания и требует лишь небольшого объяснения. В терминах фактического вывода он делает немногое... и все же тот факт, что внедрить код Python в C# *настолько* просто, является причиной для торжества. Правда, уровень взаимодействия пока что близок к минимальному, но на самом деле вряд ли бы нашлось что-то более простое.

Множество форм строковых литералов в Python

Файл Python содержит единственную строку: `print "hello, world"`. Обратите внимание на двойные кавычки в файле и сравните это с одинарными кавычками в строковом литерале, который передается методу `engine.Execute()`. В обоих случаях было бы допустимым и то, и другое. В Python поддерживаются разнообразные представления строковых литералов, включая утроенные одинарные кавычки или утроенные двойные кавычки для многострочных литералов. Я упоминаю об этом только потому, что удобно, когда отсутствует необходимость в обязательной отмене двойных кавычек каждый раз, когда нужно поместить код Python в строковый литерал C#.

Следующим будет рассматриваться тип `ScriptScope`, который является критически важным для сценария конфигурации.

Хранение и извлечение информации из `ScriptScope`

Оба используемых ранее метода запуска имеют перегруженные версии со вторым параметром — областью действия. В контексте простейших терминов это может рассматриваться как словарь имен и значений. Сценарные языки часто позволяют осуществлять присваивание переменных без явного объявления, и когда это делается на верхнем уровне программы (а не в функции или классе), оно обычно затрагивает *глобальную область действия*.

Переданный в метод запуска экземпляр `ScriptScope` применяется в качестве глобальной области действия для сценария, который вы предлагаете выполнить механизму. Этот сценарий может извлекать существующие значения из области действия и создавать новые значения, как демонстрируется в листинге 14.9.

Листинг 14.9. Передача информации между хостом и сценарием с использованием `ScriptScope`

```
string python = @"
text = 'hello'                          <-- ❶ Код Python, внедренный в виде строкового литерала C#
output = input + 1
";
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
scope.SetVariable("input", 10); <-- ❷ Установка переменной для использования в коде Python
```

```
engine.Execute(python, scope);
Console.WriteLine(scope.GetVariable("text"));   ←❸ Извлечение переменных из области действия
Console.WriteLine(scope.GetVariable("input"));
Console.WriteLine(scope.GetVariable("output"));
```

В листинге 14.9 исходный код Python внедрен в код C# в виде дословного строкового литерала ❶ вместо помещения его в файл, что позволяет видеть весь код в одном месте. Я не рекомендую поступать так в производственном коде, отчасти потому, что Python чувствителен к пробельным символам — на первый взгляд безобидное переформатирование кода может привести к нарушению его работоспособности во время выполнения.

Методы `SetVariable()` и `GetVariable()` просто помещают значения в область действия ❷ и извлекают их оттуда ❸ в очевидной манере. Они объявлены с типом `object`, а не `dynamic`, как возможно вы ожидали. Но `GetVariable` также позволяет указывать аргумент типа, который действует в качестве запроса преобразования.

Это не совсем то же самое, что и приведение результата необобщенного метода, т.к. последний просто распаковывает значение, а это означает необходимость его приведения к правильному типу. Например, можно поместить в область действия целое число, но извлечь его как значение `double`:

```
scope.SetVariable("num", 20)
double x = scope.GetVariable<double>("num") ←❶ Успешное преобразование в тип double
double y = (double) scope.GetVariable("num"); ←❷ Распаковка генерирует исключение
```

Первый вызов проходит успешно: вы явно указываете методу `GetVariable()`, какой тип нужен ❶, поэтому он автоматически преобразует значение должным образом. Второй вызов ❷ сгенерирует исключение `InvalidCastException`, как это было бы в любой другой ситуации с попыткой распаковки значения с применением некорректного типа.

Область действия способна также сохранять функции, которые можно извлекать и затем вызывать динамическим образом, передавая аргументы и получая возвращаемые значения. Проще всего делать это с использованием типа `dynamic` (листинг 14.10).

Листинг 14.10. Вызов функции, объявленной в `ScriptScope`

```
string python = @"
def sayHello(user):
   print 'Hello %(name)s' % {'name' : user}
";
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
engine.Execute(python, scope);
dynamic function = scope.GetVariable("sayHello");
function("Jon");
```

Файлам конфигурации такая возможность требуется нечасто, но она может быть удобной в других ситуациях. Например, язык Python можно было бы легко применять для написания сценариев в программе графического рисования, предоставляя функцию, которая должна вызываться в каждой входной точке. Простой пример находится на вебсайте книги по адресу `http://mng.bz/6yGi`.

Существует несколько ситуаций, когда удобно иметь средство вычисления выражений определенного вида, которое запускает код, вводимый пользователем во время

выполнения, такой как проверка бизнес-правил для скидок, стоимости доставки и т.д. Также полезно иметь возможность изменять эти правила в текстовой форме без необходимости в повторной компиляции или развертывании двоичных файлов. Код в листинге 14.10 довольно скучен, но еще один пример в загружаемом исходном коде переплетает эти два языка более извилисто, демонстрируя возможность обращения в обоих направлениях: из C# в IronPython, как вы уже видели, и из IronPython в C#.

Собираем все вместе

Теперь, когда вы можете заносить значения в область действия, по существу все сделано. Потенциально область действия можно было бы поместить в оболочку другого объекта, обеспечивая доступ через индексатор или даже обращаясь к значениям динамически с использованием приемов из раздела 14.5. Код приложения может выглядеть приблизительно так:

```
static Configuration LoadConfiguration()
{
   ScriptEngine engine = Python.CreateEngine();
   ScriptScope scope = engine.CreateScope();
   engine.ExecuteFile("configuration.py", scope);
   return Configuration.FromScriptScope(scope);
}
```

Точная форма типа `Configuration` будет зависеть от вашего приложения, но вряд ли код окажется особо захватывающим. В полном исходном коде я предоставил пример динамической реализации, который позволяет извлекать значения как свойства и также напрямую вызывать функции. Конечно, вы не ограничены в своей конфигурации применением элементарных типов: код Python может быть произвольно сложным, строя коллекции, связывая компоненты и службы и т.д. Он может играть множество ролей нормального внедрения зависимостей или контейнера инверсии управления.

Важный аспект заключается в том, что теперь вы имеете файл конфигурации, который является активным, как противоположность традиционным пассивным файлам XML и `.ini`. Разумеется, вы могли бы внедрить в пассивные файлы конфигурации собственный язык программирования, но результат, скорее всего, оказался бы менее мощным и потребовал бы намного больших затрат на реализацию. В качестве примера простой ситуации, где это могло бы быть удобнее, чем полной внедрение зависимостей, можно привести необходимость конфигурирования нескольких потоков для использования в определенном компоненте фоновой обработки внутри приложения. Обычно можно применять столько потоков, сколько есть процессоров в системе, но иногда это количество нужно уменьшить, чтобы способствовать более гладкому выполнению другого приложения в той же самой системе. Файл конфигурации тогда бы изменился с такого:

```
agentThreads = System.Environment.ProcessorCount
agentThreadName = 'Processing agent'
```

на следующий:

```
agentThreads = 1
agentThreadName = 'Processing agent (single thread only)'
```

Такое изменение не требует повторной компиляции или развертывания приложения — необходимо лишь отредактировать файл и перезапустить приложение. Особо высокоинтеллектуальные приложения могли бы даже обладать способностью реконфигурирования на лету. (Я обнаружил, что эта возможность приносит больше мучений при

реализации, чем дополнительной пользы, но в определенных местах она может иметь большое значение. Способность к изменению уровней ведения журналов либо для отдельной порции кода, либо даже для конкретного пользователя, который столкнулся с проблемами, может намного упростить процесс отладки.)

Кроме выполнения функций мы еще не рассматривали применение Python действительно динамическим образом. Доступна полная мощь языка Python, и за счет использования типа `dynamic` в коде C# можно получить преимущества метапрограммирования и всех других динамических средств. Компилятор C# отвечает за представление кода в подходящей манере, а сценарный механизм — за получение этого кода и выяснение, что он означает в контексте Python. Только не думайте, что вам *придется* делать что-то совершенно нестандартное для внедрения сценарного механизма в свое приложение. Это простой шаг в направлении более мощного приложения.

Насколько большие возможности вы хотите предоставить авторам сценариев?

Если вы запускаете произвольный код, особенно код, вводимый внешними пользователями системы, то должны серьезно обдумать вопросы безопасности и возможно выполнять сценарий в специальной среде песочницы. Обсуждение этой темы выходит за рамки настоящей книги, но ей следует уделить особое внимание.

Приведенные до сих пор примеры взаимодействовали с другими системами. Тем не менее, динамическая типизация может иметь смысл даже внутри полностью управляемой системы. Давайте рассмотрим несколько примеров.

14.3.3. Динамическая типизация в полностью управляемом коде

В прошлом вы почти наверняка применяли кое-что, *подобное* динамической типизации, даже если код, выполняющий работу, не был написан лично вами. Простейшим примером является привязка данных — каждый раз, когда вы указываете что-то вроде `ListControl.DisplayMember`, вы предлагаете инфраструктуре найти во время выполнения свойство на основе его имени. Если вам когда-либо приходилось использовать напрямую рефлексию в собственном коде, вы снова применяли информацию, которая доступна только во время выполнения.

По моему опыту рефлексия подвержена ошибкам, и даже если она работает, могут понадобиться дополнительные усилия для ее оптимизации. В некоторых случаях динамическая типизация может полностью заменить код, основанный на рефлексии; в зависимости от того, что конкретно делается, динамическая типизация может также работать быстрее.

Особенно трудно использовать обобщенные типы и методы через рефлексию. Например, при наличии объекта, который, как известно, реализует `IList<T>` для заданного аргумента типа `T`, может оказаться затруднительным точное выяснение того, что собой представляет `T`. Если единственной причиной для выяснения `T` является последующий вызов другого обобщенного метода, то вы на самом деле хотите предложить компилятору вызвать то, что он все равно *вызвал бы*, будь ему известен действительный тип. Разумеется, это в точности то, что делает динамическая типизация. Такой сценарий будет применен в качестве первого примера.

Выведение типов во время выполнения

Если необходимо сделать нечто большее, чем просто вызов одиночного метода, часто лучше помещать все дополнительные действия внутрь обобщенного метода. Затем можно вызвать этот обобщенный метод динамически, но записать остаток кода с использованием статической типизации. Простой пример продемонстрирован в листинге 14.11.

Предположим, что имеется список некоторого типа и новый элемент из какой-то другой части системы. Было обещано, что они совместимы, но их типы не известны статически. Это могло бы произойти по разным причинам — например, элемент мог быть результатом десериализации. В любом случае ваш код предназначен для добавления нового элемента в конец списка, но только если список содержит менее 10 элементов. Метод возвращает признак того, был ли добавлен элемент. Очевидно, в реальности бизнес-логика была бы более сложной, но дело в том, что действительно хотелось бы применять для этих операций строгие типы. В листинге 14.11 показан статически типизированный метод и его динамический вызов.

Листинг 14.11. Использование динамического выведения типов

```
private static bool AddConditionallyImpl<T>(IList<T> list, T item)
{
  if (list.Count < 10)                          <-- ❶ Обычный статически типизированный код
  {
    list.Add(item);
    return true;
  }
  return false;
}
public static bool AddConditionally(dynamic list, dynamic item)
{
  return AddConditionallyImpl(list, item);      <-- ❷ Вызов вспомогательного метода
}
...
object list = new List<string> { "x", "y" };
object item = "z";
AddConditionally(list, item);                   <-- В конечном счете вызов
                                                    AddConditionallyImpl<string>()
```

Открытый метод имеет динамические параметры; в предыдущих версиях C# он, возможно, принимал бы `IEnumerable` и `Object`, полагаясь на сложные проверки с помощью рефлексии с целью выяснения типа списка и затем вызова обобщенного метода через рефлексию. Благодаря динамической типизации, можно просто вызвать строго типизированную реализацию ❶ с применением динамических аргументов ❷, изолируя динамический доступ единственным вызовом метода оболочки. Конечно, вызов может по-прежнему потерпеть неудачу, но зато не пришлось прикладывать усилий по определению подходящего аргумента типа.

Можно было бы также предоставить открытый строго типизированный метод, чтобы позволить вызывающему коду, в котором известны типы списков статически, избежать динамической типизации. В этом случае было бы полезно использовать для методов разные имена, чтобы не допустить случайного вызова динамической версии из-за небольшого заблуждения относительно статических типов аргументов. (К тому же, когда имена отличаются, намного проще делать правильный вызов внутри динамической версии.)

В качестве еще одного примера динамической типизации в полностью управляемом коде вспомните, что я уже жаловался на отсутствие в языке C# поддержки обобщенных операций. Не существует концепции указания ограничения в виде "тип T должен иметь операцию, которая позволяет суммировать два значения типа T". Это было показано в первоначальной демонстрации динамической типизации (листинг 14.4), так что упоминание об этом не должно стать сюрпризом. Давайте возьмем операцию запроса Sum() из LINQ и сделаем ее динамической.

Компенсация отсутствия обобщенных операций

Приходилось ли вам видеть список перегруженных версий для метода Enumerable.Sum()? Он довольно длинный. Надо признать, что половина перегруженных версий обслуживает проецирование, но даже при этом условии имеются 10 перегруженных версий, каждая из которых просто берет последовательность элементов и суммирует их друг с другом, и это даже без учета суммирования беззнаковых значений, байтов или коротких целых. Почему бы воспользоваться динамической типизацией, чтобы попытаться делать все это в одном методе?

Несмотря на то что мы будем применять динамическую типизацию внутри, сам метод, показанный в листинге 14.12, является статически типизированным. Его можно было бы объявить как необобщенный метод, суммирующий IEnumerable<dynamic>, но это не очень хорошо работало бы из-за ограничений ковариантности. Метод получил имя DynamicSum(), а не Sum(), чтобы не конфликтовать с методами в типе Enumerable. Компилятор отдаст предпочтение необобщенной перегруженной версии перед обобщенной, когда обе сигнатуры имеют одни и те же типы параметров, но проще вообще не создавать условий для конфликта.

Листинг 14.12. Суммирование произвольной последовательности элементов динамическим образом

```
public static T DynamicSum<T>(this IEnumerable<T> source)
{
   dynamic total = default(T);                  ←❶ Динамически типизированная переменная для последующего использования

   foreach (T element in source)
   {
      total = (T) (total + element);            ← Динамический выбор операции сложения
   }
   return total;
}
...
byte[] bytes = new byte[] { 1, 2, 3 };
Console.WriteLine(bytes.DynamicSum());          ← Выводит на консоль значение 6
```

Код преимущественно прямолинеен; он имеет почти такой же вид, как выглядели бы любые реализации обычных перегруженных версий Sum(). Для краткости проверка source на предмет null опущена, но остальной код достаточно прост. Есть два интересных момента, которые следует отметить.

Во-первых, для инициализации переменной total, объявленной как dynamic, используется операция default(T), что обеспечивает желаемое динамическое поведение ❶. Так или иначе, необходимо установить какое-то начальное значение; можно было бы попробовать воспользоваться первым значением в последовательности, но тогда воз-

никла бы проблема в случае, когда последовательность пуста. Для типов значений, не допускающих null, операция default(T) почти всегда дает подходящее значение: это естественный ноль. Для ссылочных типов первый элемент последовательности будет сложен с null, что может как быть, так и не быть приемлемым. Для типов значений, допускающих null, будет предприниматься попытка сложить первый элемент со значением null для этого типа, что, безусловно, *не может* считаться приемлемым.

Во-вторых, результат сложения приводится обратно к типу T, несмотря на его последующее присваивание динамической переменной. Это может казаться странным, но вы должны вспомнить о результате суммирования двух значений типа byte. Перед выполнением сложения компилятор C# обычно поднимает тип каждого операнда до int. Без приведения переменная total в итоге хранила бы значение int, которое затем вызвало бы исключение, когда оператор return попытался бы преобразовать его в byte.

Оба момента порождают более глубокие вопросы, но это не является темой настоящего раздела. Детальные исследования динамического суммирования изложены в моей статье (на английском языке), доступной на веб-сайте книги (http://mng.bz/0N37).

Для подтверждения способности кода работать не только с обычными числами в листинге 14.13 представлен пример суммирования значений типа TimeSpan.

Листинг 14.13. Суммирование списка элементов TimeSpan динамическим образом

```
var times = new List<TimeSpan>
{
  2.Hours(), 25.Minutes(), 30.Seconds(),
  45.Seconds(), 40.Minutes()
};
Console.WriteLine(times.DynamicSum());
```

Для удобства значения TimeSpan создаются с применением расширяющих методов, но суммирование является полностью динамическим, давая в результате общий промежуток времени в 3 часа, 6 минут и 15 секунд.

Утиная типизация

Иногда вы знаете, что член с определенным именем будет доступен во время выполнения, но не можете сообщить компилятору, о каком точно члене идет речь, поскольку это будет зависеть от типа. В некотором смысле это более общий пример только что решенной задачи за исключением использования нормальных методов и свойств вместо операций.

Есть и отличие: обычно вы попытались бы заключить общность в интерфейс или абстрактный базовый класс. Поступать так с операциями нельзя, но это нормальный подход для методов и свойств. К сожалению, он работает не всегда — особенно, если задействовано множество библиотек. Наиболее согласованной в данном отношении является инфраструктура .NET Framework, но вы уже видели один пример, где это не совсем работает. В главе 12 рассматривались оптимизации, доступные для подсчета элементов в коллекции, и было показано, что ICollection и ICollection<T> имеют свойство Count — однако они не располагают общим предком в виде интерфейса с таким свойством, поэтому их приходится обрабатывать отдельно.

Утиная типизация позволяет просто обращаться к свойству Count, не выполняя проверку типа самостоятельно (листинг 14.14).

Листинг 14.14. Обращение к свойству Count с помощью утиной типизации

```
static void PrintCount(IEnumerable collection)
{
  dynamic d = collection;
  int count = d.Count;
  Console.WriteLine(count);
}
...
PrintCount(new BitArray(10));
PrintCount(new HashSet<int> { 3, 5 });
PrintCount(new List<int> { 1, 2, 3 });
```

Метод PrintCount() ограничен реализациями IEnumerable по той же причине, по которой это делается для инициализаторов коллекций: это довольно хорошая подсказка о том, что свойство Count, которое в итоге применяется, является подходящим. Тестировались коллекции BitArray (которая реализует только ICollection), HashSet<int> (реализующая только ICollection<int>) и List<int> (которая реализует оба интерфейса). Во всех случаях во время выполнения находилось правильное свойство.

Явная реализация интерфейсов и dynamic сочетаются плохо

Впервые приступив к тестированию этого кода, я использовал массив int[], который неявно может быть преобразован в оба задействованных интерфейса. Я поначалу был удивлен, когда метод PrintCount() отказался работать во время выполнения, и решил обдумать ситуацию более тщательно. Связывание во время выполнения производится с применением действительного типа объекта, которым в данном случае является int[]. Типы массивов не предоставляют открытый доступ к свойству Count — для этого они используют явную реализацию интерфейсов. Работать со свойством Count можно, только если рассматривать массив объекта определенным образом.

Это лишь один пример случая, когда динамическая типизация может вести себя логичным, но неожиданным образом, если не проявить достаточную внимательность. Я непрерывно веду список таких странностей на веб-сайте книги (http://mng.bz/5y7M); не сочтите за труд уведомить меня, если обнаружите что-нибудь новое.

Мы будем придерживаться примера извлечения количества элементов, но на этот раз посмотрим на то, как распознавание перегруженных версий может предложить альтернативу явной проверке типов.

Множественная диспетчеризация

При статической типизации C# использует *одиночную диспетчеризацию*: во время выполнения точный вызываемый метод зависит только от действительного типа целевого объекта, на котором метод вызывается, благодаря переопределению. Распознавание перегруженных версий происходит на этапе компиляции. *Множественная диспетчеризация* временами может оказаться полезной для нахождения наиболее специализированной реализации метода на основе типов аргументов во время выполнения — и снова это то, что предоставляет динамическая типизация.

В листинге 14.15 показано, что множественная диспетчеризация помогает получить более разнообразную и надежную реализацию оптимизированного подсчета элементов.

Листинг 14.15. Эффективный подсчет элементов разных типов с применением множественной диспетчеризации

```
private static int CountImpl<T>(ICollection<T> collection)
{
  return collection.Count;
}
private static int CountImpl(ICollection collection)
{
  return collection.Count;
}
private static int CountImpl(string text)
{
  return text.Length;
}
private static int CountImpl(IEnumerable collection)
{
  int count = 0;
  foreach (object item in collection)
  {
    count++;
  }
  return count;
}
public static void PrintCount(IEnumerable collection)
{
  dynamic d = collection;
  int count = CountImpl(d);
  Console.WriteLine(count);
}
...
PrintCount(new BitArray(5));
PrintCount(new HashSet<int> { 1, 2 });
PrintCount("ABC");
PrintCount("ABCDEF".Where(c => c > 'B'));
```

Вы знаете, что во время выполнения, по меньшей мере, одна перегруженная версия метода `CountImpl()` будет подходящей, т.к. параметр для метода `PrintCount()` имеет тип `IEnumerable`. С помощью динамической типизации выполняется та же самая работа, как и явные шаги "если это `ICollection<T>`, то использовать одну реализацию; если это `ICollection`, то применять другую реализацию", которые использовались при выборе случайного элемента в листинге 12.17. В качестве примера того, что это — нечто большее, чем просто применение свойства `Count`, когда оно доступно, код в листинге 14.15 содержит оптимизацию для строк, позволяющую использовать свойство `Length` для быстрого получения правильного результата.

Даже при условии, что здесь применяется множественная диспетчеризация, во время выполнения по-прежнему могут возникать проблемы: что, если действительный тип

реализует и ICollection<string>, и ICollection<int> через явную реализацию интерфейсов? В зависимости от того, какая реализация Count выбрана, возможны два результата. В этом случае связывание было бы неоднозначным, приводя к генерации исключения. К счастью, патологические случаи подобного рода встречаются редко.

Это лишь некоторые примеры областей, где *могло бы* возникнуть желание использовать динамическую типизацию, даже если не предпринимается никаких попыток взаимодействовать с чем-то еще. Далее мы углубимся в детали того, как все это достигается, и завершим главу реализацией собственного динамического поведения.

Должен предупредить, что рассматриваемые вопросы становятся сложнее. В действительности все они исключительно элегантны, однако дело осложняется тем, что языки программирования предлагают широкий набор операций, а представление всей необходимой информации об этих операциях в виде данных и затем выполнение действий на них соответствующим образом являются сложными работами. Хорошая новость в том, что понимать это все вовсе не обязательно. Как обычно, чем лучше вы знаете лежащие в основе механизмы, тем больше выгоды можете получить от динамической типизации, но даже если вы будете пользоваться только приемами, показанными до сих пор, могут возникать ситуации, в которых удастся достичь намного большей продуктивности.

14.4. Заглядывая за кулисы

Несмотря на предупреждение в предыдущем абзаце, я не буду приводить *слишком* много деталей о внутренней работе динамической типизации. Тогда пришлось бы раскрывать очень много основ, со ссылкой на изменения как в инфраструктуре, так и в языке. Я не часто избегаю практически важных аспектов спецификации, но в данном случае я искренне полагаю, что изучение всего этого не принесет особенную пользу. Будут рассмотрены наиболее важные (и интересные) моменты, и я настоятельно рекомендую обращаться за дополнительными сведениями в блог Сэма Нг (http://blogs.msdn.com/b/samng/), к спецификации языка C# и на страницу проекта среды DLR (http://mng.bz/0M6A).

Моя конечная цель — помочь вам понять, что компилятор C# делает, и какой код он выдает для обеспечения динамического связывания во время выполнения. К сожалению, сгенерированный код не будет иметь никакого смысла до тех пор, пока вы не увидите механизм, поддерживающий все это — среду DLR. Возможно, вам понравится думать о статически типизированной программе как об обыкновенной пьесе с фиксированным сценарием, а о типизированной программе — как об импровизированном шоу. Среда DLR выступает в качестве мозгового центра актеров, интенсивно обдумывая, что сказать в ответ на предложения аудитории. Давайте встретимся с нашей сообразительной звездой.

14.4.1. Введение в DLR

На протяжении некоторого времени я упоминал аббревиатуру DLR, иногда расшифровывая ее как Dynamic Language Runtime (исполняющая среда динамического языка), но никогда не объяснял, что это такое. Это было сделано преднамеренно: я пытался четко изложить природу динамической типизации и ее влияние на разработчиков, а не конкретные детали реализации. Однако данная отговорка не могла продолжаться вплоть до конца главы. В чистых терминах Dynamic Language Runtime — это библиотека, которую все динамические языки и компилятор C# применяют для динамического выполнения кода.

Довольно-таки удивительно, но это на самом деле просто библиотека. Вопреки своему названию, она не находится на том же уровне, что и CLR (Common Language Runtime — общеязыковая исполняющая среда) — она не имеет дела с JIT-компиляцией, маршализацией в низкоуровневом API-интерфейсе, сборкой мусора и т.д. Однако среда DLR построена на большом объеме функциональности .NET 2.0 и .NET 3.5, особенно на типах `DynamicMethod` и `Expression`. В .NET 4 также был расширен API-интерфейс деревьев выражений, чтобы позволить среде DLR представлять большее число концепций. Нам рис. 14.1 показано, как все это сочетается друг с другом.

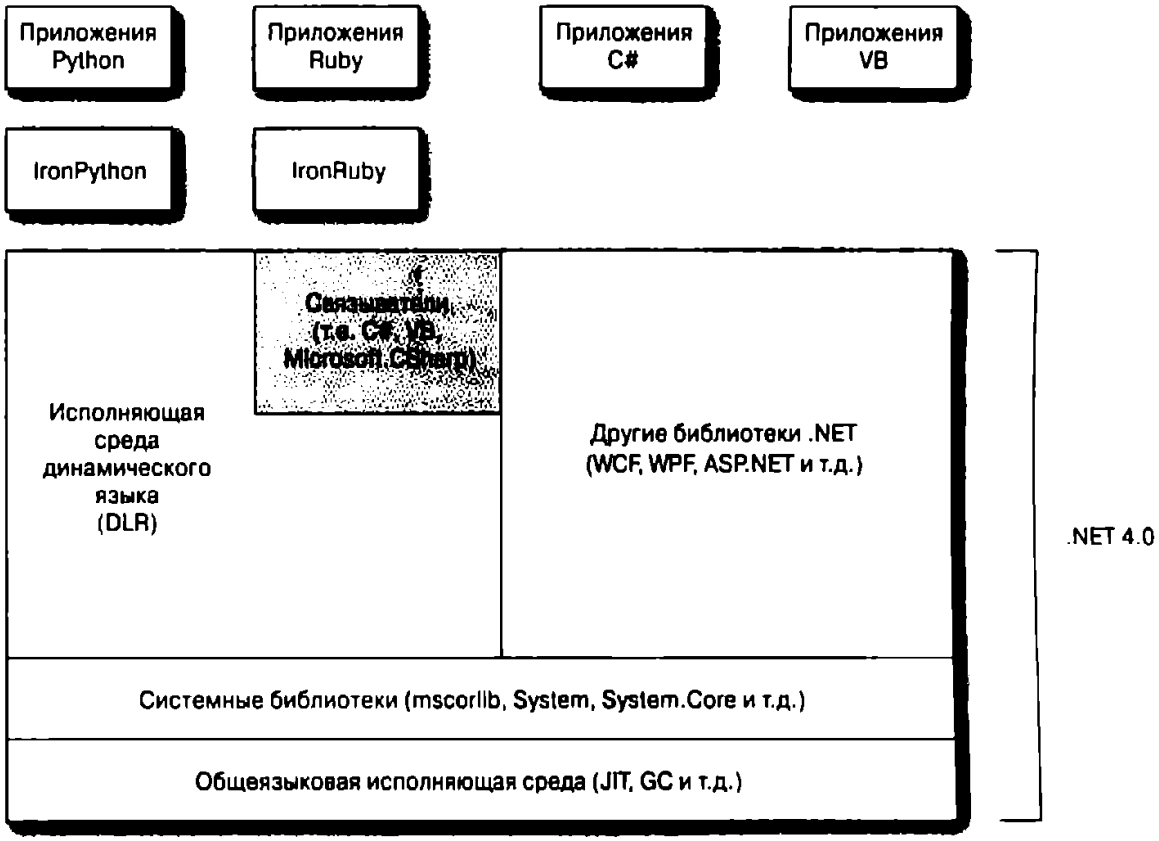

Рис. 14.1. Сочетание компонентов .NET 4 друг с другом, позволяющее статическим и динамическим языкам выполняться на одной и той же платформе

В дополнение к среде DLR на рис. 14.1 присутствует еще одна библиотека, которая может оказаться для вас новой. Одной из сборок в части связывателей на диаграмме является `Microsoft.CSharp`. Она содержит несколько типов, на которые ссылается компилятор C#, когда вы используете в своем коде `dynamic`. Немного сбивает с толку, что она не включает существующие классы `Microsoft.CSharp.Compiler` и `Microsoft.CSharp.CodeDomProvider`. (Они даже не находятся в одной сборке друг с другом!) Вы увидите, для чего предназначены новые типы, в разделе 14.4.2, где будет декомпилирован код, в котором применяется тип `dynamic`.

Еще один важный аспект отличает среду DLR от остальной инфраструктуры .NET Framework: она предлагается в виде исходного кода. Полный исходный код имеет форму проекта CodePlex (`http://dlr.codeplex.com`), так что вы можете загрузить его и проанализировать внутреннюю работу. Одно из преимуществ данного подхода заключается в том, что среду DLR не пришлось реализовать заново для Mono (`http://mono-project.com`): один и тот же код выполняется под управлением как .NET, так и межплатформенной версии.

Хотя среда DLR не обрабатывает машинный код напрямую, в некотором смысле можно считать, что она выполняет работу, *подобную* среде CLR: точно так же, как CLR преобразует код IL (Intermediate Language — промежуточный язык) в машинный код, среда DLR преобразует код, представленный с использованием связывателей, вызывающих компонентов, метаобъектов и прочих разнообразных концепций, в деревья выражений, которые затем компилируются в код IL и, в конечном счете — в низкоуровневый код средой CLR. На рис. 14.2 показано упрощенное представление жизненного цикла обработки динамического выражения.

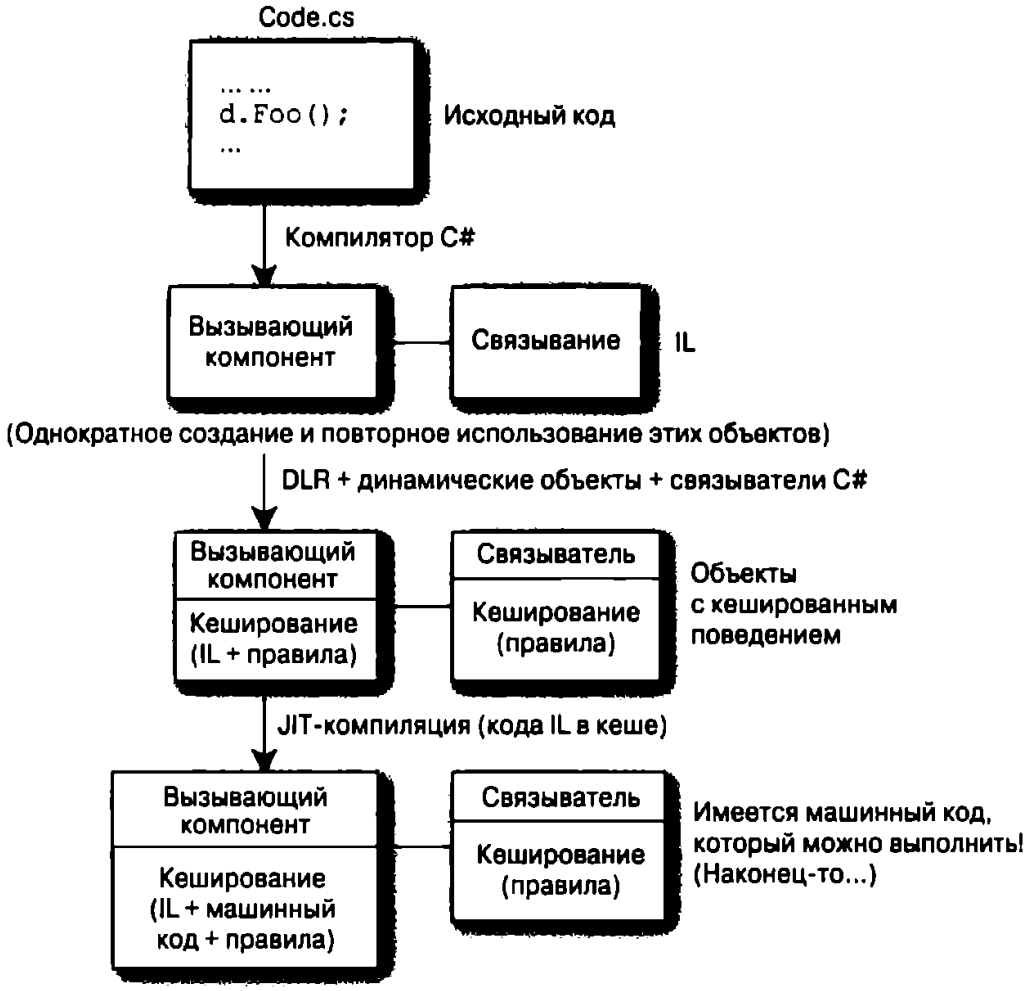

Рис. 14.2. Жизненный цикл обработки динамического выражения

Как видите, одним из важных аспектов среды DLR является многоуровневый кеш. Он критически важен для производительности, но чтобы понять эту и другие концепции, которые уже упоминались, необходимо углубиться на еще один уровень.

14.4.2. Основные концепции DLR

В *самых* общих чертах назначение среды DLR можно определить как получение высокоуровневого представления кода и запуск этого кода на основе различных порций информации, которые могут быть известны только во время выполнения. В настоящем разделе будет введено много терминов для описания работы DLR, но все они содействуют общей цели.

Вызывающие компоненты

Первой концепцией является *вызывающий компонент*. Это своего рода атом среды DLR — мельчайшая частица кода, которая может считаться одиночной выполняемой единицей. Одно выражение может содержать множество вызывающих компонентов, но поведение выстроено естественным образом, с выполнением одного вызывающего компонента за раз.

В ходе дальнейшего обсуждения мы будем считать, что имеется только один вызывающий компонент. Будет удобно ссылаться на небольшой пример вызывающего компонента, который приведен ниже, при этом d — переменная типа dynamic:

```
d.Foo(10);
```

Вызывающий компонент представлен в коде как экземпляр типа System.Runtime.CompilerServices.CallSite<T>. Полный пример создания и использования вызывающих компонентов будет показан в следующем разделе, где мы взглянем на то, что компилятор C# делает на этапе компиляции, но вот пример кода, с помощью которого может быть создан вызывающий компонент для предыдущего фрагмента:

```
CallSite<Action<CallSite, object, int>>.Create(Binder.InvokeMember(
  CSharpBinderFlags.ResultDiscarded, "Foo", null, typeof(Test),
  new CSharpArgumentInfo[] {
   CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null),
   CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.Constant |
                             CSharpArgumentInfoFlags.UseCompileTimeType,
                             null) }));
```

Теперь, когда есть вызывающий компонент, можно ли выполнить код? Пока еще нет.

Получатели и связыватели

Кроме вызывающего компонента необходимо кое-что, позволяющее решить, что он означает и как его выполнить. В среде DLR это могут решить две сущности: *получатель* вызова и *связыватель*. Получатель вызова — это просто объект, на котором вызывается член. В рассматриваемом примере вызывающего компонента получателем является объект, на который ссылается d во время выполнения. Связыватель будет зависеть от языка вызова и частично от вызывающего компонента — в этом случае видно, что компилятор C# выдает код для создания связывателя с применением Binder.InvokeMember(). Класс Binder в данном случае — это Microsoft.CSharp.RuntimeBinder.Binder, так что он действительно специфичен для C#. Связыватель C# также осведомлен о COM и будет выполнять соответствующее связывание COM, если получателем является объект IDispatch.

Среда DLR всегда отдает предпочтение получателю: если он представляет собой динамический объект, которому известно, как обработать вызов, то DLR будет использовать любой путь выполнения, предоставленный этим объектом. Объект может самостоятельно представить себя динамическим за счет реализации нового интерфейса IDynamicMetaObjectProvider. Этот интерфейс с труднопроизносимым именем содержит единственный член: GetMetaObject(). Чтобы корректно реализовать метод GetMetaObject(), нужно быть ниндзя в области деревьев выражений и прилично разбираться в DLR. Однако в умелых руках указанный интерфейс может стать мощным инструментом, предоставляющим низкоуровневое взаимодействие со средой DLR и ее кешем выполнения. Если необходимо реализовать динамическое поведение в высокопроизводительной манере, полезно выделить время на изучение деталей.

В инфраструктуру включены две открытых реализации интерфейса `IDynamicMetaObjectProvider`, которые упрощают реализацию динамического поведения в ситуациях, когда производительность не особенно критична. Мы рассмотрим все это в разделе 14.5, но пока нужно лишь знать о самом интерфейсе и о том, что он представляет способность объекта реагировать динамическим образом.

Если получатель не является динамическим, решение о том, как должен выполняться код, принимает связыватель. В коде C# это предполагает применение специфичных для C# правил к коду и выяснение, что будет делаться. Если бы вы создавали собственный динамический язык, то могли бы реализовать специальный связыватель для принятия решения относительно его общего поведения (когда объект не переопределяет поведение). Это выходит далеко за рамки материалов данной книги, но само по себе является интересной темой; одной из целей DLR было как раз упрощение реализации собственных языков.

Правила и кеши

Решение о том, как выполнять тот и иной вызов, представляется в виде *правила*. По существу оно состоит из двух элементов логики: обстоятельств, при которых вызывающий компонент должен вести себя подобным образом, и самого поведения.

Первая часть в действительности предназначена для оптимизации. Предположим, что имеется вызывающий компонент, который представляет сложение двух динамических значений, и при первой его оценке оба значения имеют тип `byte`. Связывателю придется предпринять немало усилий для выяснения того, что тип обоих операндов должен быть поднят до `int`, а результатом должна быть сумма этих двух целых чисел. Он может использовать данную операцию повторно каждый раз, когда оказывается, что оба операнда относятся к типу `byte`. Проверка набора предыдущих результатов на предмет допустимости может сохранить массу времени. Правило, применяемое в качестве примера (типы операндов должны точно совпадать, как было только что показано), является распространенным, но в DLR поддерживаются также и другие правила.

Вторая часть правила — это код, предназначенный для использования, если правило удовлетворено, и он представлен в виде дерева выражения. Код *мог бы* быть сохранен как скомпилированный делегат для вызова, но представление в форме дерева выражения означает возможность серьезной оптимизации за счет применения кеша. В среде DLR предусмотрены кеши трех уровней: L0, L1 и L2. Эти кеши хранят информацию разными способами и с отличающейся областью действия. Каждый вызывающий компонент имеет собственные кеши L0 и L1, а кеш L2 может разделяться между множеством подобных вызывающих компонентов, как показано на рис. 14.3.

Набор вызывающих компонентов, которые разделяют кеш L2, определяется их связывателями — каждый связыватель имеет ассоциированный с ним кеш L2. Компилятор (или то, что создает вызывающие компоненты) решает, сколько связывателей использовать. Компилятор может применять связыватель для множества вызывающих компонентов, только когда они представляют очень похожий код, где в случае одного и того же контекста во время выполнения вызывающие компоненты должны выполняться одинаковым образом. В действительности компилятор C# не использует эту возможность — для каждого вызывающего компонента он создает новый связыватель[4], так что для разработчиков на C# особой разницы между кешами L1 и L2 не существует. Однако

[4] Большая часть информации специфична для конкретного вызывающего компонента, поскольку правила связывания будут отличаться в зависимости от таких аспектов, как класс, из которого производится вызов.

по-настоящему динамические языки, такие как IronRuby и IronPython, применяют эту возможность чаще.

Сами кеши являются исполняемым кодом, что требует некоторого времени для понимания. Компилятор C# генерирует код, чтобы просто выполнить кеш L0 вызывающего компонента (который представляет собой делегат, доступный через свойство `Target`). Вот и все! Кеш L0 имеет единственное правило, которое проверяется при его вызове. Если правило удовлетворено, выполняется связанное поведение. Если же правило не удовлетворено (или это вызов в первый раз, так что правила вообще еще нет), в действие вводится кеш L1, который, в свою очередь, вводит в игру кеш L2. Если кеш L2 не может найти совпадающие правила, он предлагает распознать вызов получателю или связывателю. Результаты затем помещаются в кеш для следующего раза.

В случае приведенного ранее фрагмента часть, касающаяся выполнения, будет выглядеть примерно так:

```
callSite.Target(callSite, d, 10);
```

Кеши L1 и L2 просматривают свои правила довольно стандартным путем — каждый кеш имеет коллекцию правил, и каждое правило проверяется на предмет совпадения. Кеш L0 кое в чем отличается. Две части его поведения (проверка правила и делегирование управления кешу L1) скомбинированы в единственный метод, к которому затем применяется JIT-компиляция. Обновление кеша L0 предполагает повторное построение метода из нового правила.

В результате всего этого типовые вызывающие компоненты, которые часто видят похожий контекст, являются очень быстрыми; механизм диспетчеризации настолько экономичен, что близок к тому, что можно было бы получить при жестком кодировании проверок. Разумеется, это должно быть соотнесено с ценой всей задействованной генерации динамического кода, но многоуровневый кеш определенно сложен, т.к. он пытается достичь баланса между разнообразными сценариями.

Теперь вы знаете немного больше о механизме внутри DLR и в состоянии понять, что именно компилятор делает для приведения всего в движение.

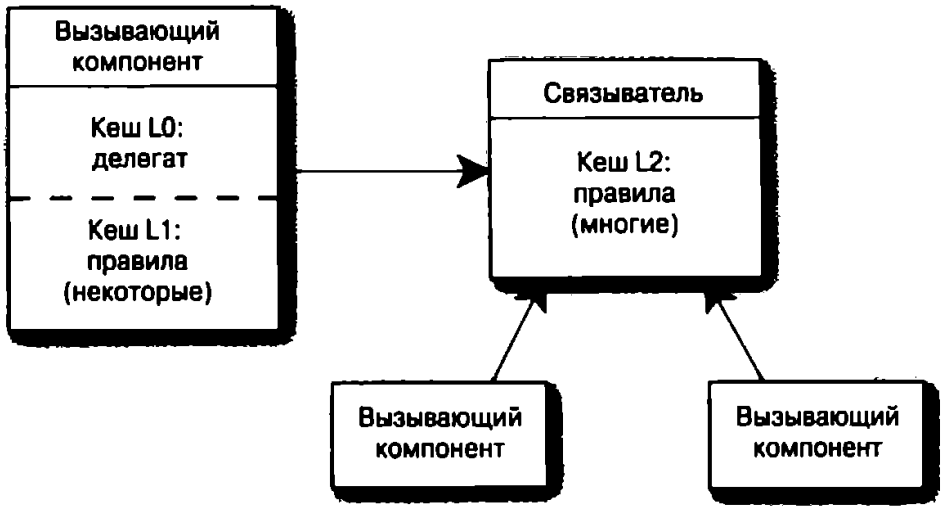

Рис. 14.3. Отношения между динамическими кешами и вызывающими компонентами

14.4.3. Как компилятор C# обрабатывает динамическое поведение

Главными работами компилятора C#, когда он добирается до динамического кода, являются выяснение, что требуется динамическое поведение, и захват всего необходимого контекста, поэтому связыватель и получатель обладают достаточной информацией для распознавания вызова во время выполнения.

Если используется `dynamic`, то вызов динамический!

Есть ситуация, которая очевидно динамическая: когда цель, на которой вызывается член, является динамической. У компилятора нет возможности узнать, каким образом будет распознан этот вызов. Это может быть по-настоящему динамический объект, который сам выполнит распознавание, или связывателю C# возможно придется распознавать его позже с помощью рефлексии. В любом случае шансы на статическое распознавание вызова отсутствуют.

Но когда динамическое значение используется в качестве *аргумента* в вызове, существуют ситуации, в которых *можно* ожидать статического распознавания вызова – особенно при наличии подходящей перегруженной версии, которая имеет тип параметра `dynamic`. Правило заключается в том, что если любая часть вызова является динамической, то сам вызов становится динамическим, и перегруженная версия будет распознаваться с применением типа динамического значения во время выполнения. В листинге 14.16 это демонстрируется на примере метода с двумя перегруженными версиями и его вызова несколькими отличающимися путями.

Листинг 14.16. Экспериментирование с перегрузкой методов и динамическими значениями

```
static void Execute(string x)
{
  Console.WriteLine("String overload");
}
static void Execute(dynamic x)
{
  Console.WriteLine("Dynamic overload");
}
...
dynamic text = "text";
Execute(text);          ← Выводит на консоль строку "String overload"
dynamic number = 10;
Execute(number);        ← Выводит на консоль строку "Dynamic overload"
```

Оба вызова метода `Execute()` связаны динамически. Во время выполнения они распознаются с использованием типов действительных значений, а именно – `string` и `int`. Параметр типа `dynamic` трактуется, как если бы он был объявлен с типом `object` везде, кроме внутренностей самого метода — взглянув на скомпилированный код, вы увидите, что он *является* параметром типа `object`, просто с примененным дополнительным атрибутом. Это также означает невозможность наличия двух методов, сигнатуры которых отличаются только типами параметров `dynamic`/`object`.

Это был пример распознавания вызовов методов, но есть много других выражений, которые предстоит рассмотреть. Поверьте, иногда ситуация не настолько проста, как приведенная выше...

Вызов динамический кроме случаев, когда это не так

Во время представления типа dynamic в разделе 14.2 я должен был проявлять осторожность и не обобщать все слишком сильно, поскольку для практически каждого правила существуют исключения. Хотя вы должны знать о них, беспокоиться об этом вовсе не нужно — вряд ли они приведут к каким-либо проблемам.

Давайте кратко пройдемся по ним.

Преобразования между типами CLR и dynamic

Преобразования между типами CLR и dynamic ограничены тем же самым способом, который не позволяет выполнять преобразование из *любого* типа CLR в object; исключениями являются типы вроде указателей и System.TypedReference. С учетом того, что dynamic — это просто object на уровне CLR, исключение указанных типов не должно вызывать удивление.

Вы могли также заметить, что речь шла о преобразовании "из выражения типа dynamic" в тип CLR, а не о преобразовании из самого типа dynamic. Это тонкое отличие оказывает помощь во время выведения типов и в других ситуациях, в которых необходимо учитывать неявные преобразования между типами; вообще говоря, когда есть два типа с неявными преобразованиями в обоих направлениях, все становится довольно затруднительным. В основном это ограничивает набор ситуаций, в которых учитывается преобразование.

Например, рассмотрим следующий неявно типизированный массив:

```
dynamic d = 0;
string x = "text";
var array = new[] { d, x };
```

Каким должен быть выведенный тип для array? Если бы существовало неявное преобразование из dynamic в string, то типом мог бы стать string[] или dynamic[], поэтому возникла бы неоднозначность и ошибка на этапе компиляции. Но поскольку преобразование предусмотрено только из *выражения* типа dynamic, компилятор видит преобразование из string в dynamic, но не наоборот, и array получает тип dynamic[]. Наверное, лучше не беспокоиться об этой тонкости, если только вы не пытаетесь отработать конкретный сценарий из спецификации.

Выражения, использующие dynamic, не всегда вычисляются динамически

В ряде случаев среда CLR вполне способна вычислить выражение с применением обычных статических путей выполнения, даже если одно из подвыражений является динамическим. Например, взгляните на показанную ниже операцию as:

```
dynamic d = GetValueDynamically();
string x = d as string;
```

Здесь нет ничего такого, что может произойти динамически — значением d либо является ссылка на строку, либо нет. Преобразования, определенные пользователем, не применяются в случае использования операции as, поэтому компилятор C# может выдавать точно такой же код IL, как применяемый в ситуации, когда переменная была бы объявлена с типом object.

Динамически вычисляемые выражения не всегда имеют тип dynamic

В некоторых случаях компилятору не известно, как в точности будет выполняться выражение, но известен тип результата (предполагается, что исключение не сгенерировано).

Например, взгляните на обращение к конструктору с указанием динамического значения в качестве аргумента:

```
dynamic d = GetValueDynamically();
SomeType x = new SomeType(d);
```

Сам вызов конструктора должен быть выполнен динамически. Во время выполнения может потребоваться распознавание между множеством перегруженных версий, но результат всегда должен быть ссылкой на SomeType. По этой причине присваивание переменной x может происходить без динамического преобразования.

Существует несколько других случаев, подобных этому; к примеру, использование динамического индекса в статически типизированном массиве может приводить только к значению, имеющему тип элементов в массиве. Но вы не должны предполагать, что это всегда будет происходить, когда его можно было ожидать. Для метода могло бы быть объявлено несколько перегруженных версий, которые все имеют один и тот же статический возвращаемый тип, но типом выражения, вызывающего этот метод, по-прежнему будет dynamic.

На этом рассмотрение вопросов о том, когда динамическое выполнение не происходит или не дает в результате динамическое значение, завершено. Давайте теперь возвратимся к ситуациям, где динамическое выполнение присутствует, и посмотрим, что компилятор C# предпринимает, чтобы все работало.

Создание вызывающих компонентов и связывателей

Для использования динамических выражений знать точные детали того, что делает с ними компилятор, не обязательно, но ознакомление с тем, как выглядит скомпилированный код, может быть поучительным. В частности, если по какой-либо причине вам необходимо декомпилировать свой код, вы не должны удивляться тому, какой вид имеют динамические части. Для такой работы лично я предпочитаю пользоваться инструментом Reflector (http://mng.bz/pMXJ), но вы можете прибегнуть и к ildasm, если хотите читать код IL напрямую.

Мы собираемся рассмотреть только один пример — я уверен, что мог бы посвятить целую главу описанию деталей реализации, но идея состоит в том, чтобы представить вам только сущность того, что предпринимает компилятор. Если вы сочтете этот пример интересным, можете продолжить эксперименты самостоятельно. Просто помните, что точные детали специфичны для реализации; в будущих версиях компилятора они вполне могут измениться, продолжая обеспечивать эквивалентное поведение. Ниже показан пример фрагмента кода, который обычно существует в методе Main() для Snippy:

```
string text = "text to cut";
dynamic startIndex = 2;
string substring = text.Substring(startIndex);
```

Довольно просто, не так ли? Однако фрагмент содержит две динамических операции — одну для вызова Substring() и одну (неявную) для динамического преобразования результата этого вызова (который как раз имеет тип dynamic на этапе компиляции) в строку. В листинге 14.17 приведен декомпилированный код для класса Snippet[5]. Для экономии пространства объявление самого класса и неявного конструктора без параметров не показано, а код сформатирован с меньшим числом пробельных символов.

---

[5] Просто чтобы напомнить: Snippet — это класс, автоматически генерируемый инструментом Snippy.

Листинг 14.17. Результаты компиляции динамического кода

```
[CompilerGenerated]
private static class <Main>o__SiteContainer0 {   ❶ Хранилище вызывающих компонентов
  public static CallSite<Func<CallSite, object, string>> <>p__Site1;
  public static CallSite<Func<CallSite, string, object, object>>
      <>p__Site2;
}
private static void Main() {
  string text = "text to cut";
  object startIndex = 2;
  if (<Main>o__SiteContainer0.<>p__Site1 == null) {   ❷ Создание вызывающего компонента для преобразования
    <Main>o__SiteContainer0.<>p__Site1 =
      CallSite<Func<CallSite, object, string>>.Create(
        new CSharpConvertBinder(typeof(string),
          CSharpConversionKind.ImplicitConversion, false));
  }
  if (<Main>o__SiteContainer0.<>p__Site2 == null) {   ❸ Создание вызывающего компонента для получения подстроки
    <Main>o__SiteContainer0.<>p__Site2 =
      CallSite<Func<CallSite, string, object, object>>.Create(
        new CSharpInvokeMemberBinder(CSharpCallFlags.None,
          "Substring", typeof(Snippet), null,
           new CSharpArgumentInfo[] {
             new CSharpArgumentInfo(
CSharpArgumentInfoFlags.UseCompileTimeType, null),   ❹ Сохранение текстового типа
             new CSharpArgumentInfo(
               CSharpArgumentInfoFlags.None, null) })));
  }
  string substring =   ❺ Активизация обоих вызовов
    <Main>o__SiteContainer0.<>p__Site1.Target.Invoke(
      <Main>o__SiteContainer0.<>p__Site1,
      <Main>o__SiteContainer0.<>p__Site2.Target.Invoke(
        <Main>o__SiteContainer0.<>p__Site2, text, startIndex));
}
```

Не знаю, как вы, но я рад, что мне никогда не приходится писать или даже сталкиваться с кодом подобного рода, кроме ситуации, когда нужно исследовать происходящие действия. Тем не менее, здесь нет ничего особо нового — код, сгенерированный для итераторных блоков, деревьев выражений и анонимных функций, также может выглядеть устрашающим.

Все вызывающие компоненты для метода сохраняются во вложенном статическом классе ❶, поскольку их необходимо создавать только однократно. (Если бы они создавались каждый раз, кеш был бы бесполезен!) Вполне возможно, что вызывающие компоненты *могли бы* создаваться более одного раза из-за многопоточности, но если это происходит, то эффективность снижается лишь незначительно и ленивое создание достигается вообще без блокировки. Совершенно не имеет значения, если один экземпляр вызывающего компонента заменяется другим. Каждый метод, использующий динамическое связывание, имеет отдельный контейнер компонентов; это *должно* предназначаться для обобщенных методов, т.к. вызывающий компонент необходимо варьировать на основе аргументов типов. Другая реализация компилятора могла бы применять один контейнер компонентов для всех необобщенных методов, еще один — для всех обобщенных методов с одиночным параметром типа и т.д.

После того, как вызывающие компоненты созданы (❹ и ❺), они активизируются. Первым инициируется вызов `Substring()` (читайте код из самой внутренней части оператора в направлении наружу) и затем на результате запускается преобразование ❻. В этой точке снова появляется статически типизированное значение, так что его можно присвоить переменной `substring`.

Я бы хотел подчеркнуть еще один аспект этого кода: способ, которым определенная информация о статическом типе сохраняется в вызывающем компоненте. Сама информация о типе представлена в сигнатуре делегата, используемой для аргумента типа, который относится к вызывающему компоненту (`Func<CallSite, string, object, object>`), а флаг в соответствующем `CSharpArgumentInfo` указывает на то, что эта информация о типе должна применяться в связывателе ❸. (Несмотря на то что это цель метода, она представлена как аргумент; методы экземпляра трактуются как статические методы с неявным первым параметром `this`.) Это критически важная часть, заставляющая связыватель вести себя так, как если бы он просто перекомпилировал ваш код во время выполнения. Давайте посмотрим, почему она настолько важна.

14.4.4. Компилятор C# становится еще интеллектуальнее

Версия C# 4 позволяет переключаться между статическим и динамическим поведением, не только связывая некоторый код статически, а некоторый динамически, но также сочетая эти две идеи в рамках единственного связывания. Все, что необходимо знать, запоминается внутри вызывающего компонента и затем эта информация разумно объединяется с типами динамических значений во время выполнения.

Сохранение поведения компилятора во время выполнения

Идеальная модель для выяснения, как должен вести себя связыватель, предполагает представление, что вместо наличия в исходном коде динамического значения имеется значение правильного типа: типа, к которому относится действительное значение во время выполнения[6]. Это применимо *только* к динамическим значениям внутри выражения; любые типы, которые известны на этапе компиляции, по-прежнему используются при поисках, таких как распознавание членов. Я приведу два примера, где это имеет значение.

В листинге 14.18 показан простой перегруженный метод в одиночном типе.

Листинг 14.18. Динамическое распознавание перегруженных версий внутри одиночного типа

```
static void Execute(dynamic x, string y)
{
  Console.WriteLine("dynamic, string");
}
static void Execute(dynamic x, object y)
{
  Console.WriteLine("dynamic, object");
}
...
```

[6] На самом деле все несколько сложнее — что, если действительный тип является внутренним в другой сборке? Например, было бы нежелательно использовать этот тип в качестве аргумента типа обобщенного метода через выведение типов. Связыватель поддерживает понятие "наилучшего доступного типа" на основе вызывающего контекста и действительного типа.

```
object text = "text";
dynamic d = 10;
Execute(d, text);  <-- Выводит на консоль строку "dynamic, object"
```

Важной переменной здесь является `text`. Ее типом *на этапе компиляции* будет `object`, но *во время выполнения* она получает значение в виде ссылки на строку. Вызов метода `Execute()` оказывается динамическим, поскольку в качестве одного из его аргументов используется динамическая переменная `d`, но при распознавании перегруженных версий применяется статический тип `text`, поэтому результатом будет вывод на консоль строки `"dynamic, object"`. Если бы переменная `text` также была объявлена как `dynamic`, использовалась бы другая перегруженная версия.

В листинге 14.19 приведен похожий код, но на этот раз имеет значение получатель вызова.

Листинг 14.19. Динамическое распознавание перегруженных версий внутри иерархии классов

```
class Base
{
  public void Execute(object x)
  {
    Console.WriteLine("object");
  }
}
class Derived : Base
{
  public void Execute(string x)
  {
    Console.WriteLine("string");
  }
}
...
Base receiver = new Derived();
dynamic d = "text";
receiver.Execute(d);  <-- Выводит на консоль строку "object"
```

В листинге 14.19 типом переменной `receiver` во время выполнения является `Derived`, поэтому можно было бы ожидать вызова перегруженной версии метода `Execute()`, определенной в классе `Derived`. Но на этапе компиляции переменная `receiver` имеет тип `Base`, так что связыватель ограничивает набор рассматриваемых методов только теми, которые *были бы* доступны, если бы связывание метода осуществлялось статически. Несмотря на все эти решения, которые должны быть приняты позже, определенные проверки на этапе компиляции доступны даже для кода, полностью связываемого во время выполнения.

Ошибки на этапе компиляции для динамического кода

Как упоминалось ближе к началу этой главы, один из недостатков динамической типизации заключается в том, что определенные ошибки, которые обычно обнаруживались бы компилятором, откладываются до периода выполнения и проявляются в виде исключений. Во многих ситуациях компилятор должен просто надеяться на то, что вы знаете, что делаете, но там, где он *может* помочь, он поможет.

Простейшим примером может быть случай, когда вы пытаетесь вызвать метод со статически типизированным получателем (или статический метод) и возможно ни одна из перегруженных версий не будет допустимой, какой бы тип не имело динамическое значение во время выполнения. В листинге 14.20 приведены три примера недопустимых вызовов, два из которых перехватываются компилятором.

Листинг 14.20. Перехват ошибок в динамических вызовах на этапе компиляции

```
string text = "cut me up";
dynamic guid = Guid.NewGuid();
text.Substring(guid);
text.Substring("x", guid);
text.Substring(guid, guid, guid);
```

В коде присутствуют три вызова метода `string.Substring()`. Компилятору известен точный набор возможных перегруженный версий, поскольку он знает тип переменной `text` статически. Он не жалуется на первый вызов, т.к. не может выяснить, какой тип будет иметь переменная `guid` — если окажется, что типом является `int`, то все будет в порядке. Но следующие два вызова приводят в выдаче сообщений об ошибках — нет каких-либо перегруженных версий, которые бы принимали `string` в первом аргументе, и нет перегруженных версий с тремя параметрами. Компилятор может гарантировать, что эти вызовы дадут сбой во время выполнения, поэтому благоразумно отказывается их компилировать.

Чуть более сложный пример связан с выведением типов. Если динамическое значение применяется для вывода аргумента типа в вызове обобщенного метода, то фактический аргумент типа не будет известен вплоть до периода выполнения и никакая заблаговременная проверка допустимости не может быть предпринята. Но любой аргумент типа, который может быть выведен без использования *любых* динамических значений, может привести к отказу выведения типов на этапе компиляции. В листинге 14.21 показан соответствующий пример.

Листинг 14.21. Выведение обобщенных типов со смешанными статическими и динамическими значениями

```
void Execute<T>(T first, T second, string other) where T : struct
{
}
...
dynamic guid = Guid.NewGuid();
Execute(10, 0, guid);
Execute(10, false, guid);
Execute("hello", "hello", guid);
```

И снова первый вызов скомпилируется, но даст сбой во время выполнения. Второй вызов не скомпилируется, потому что `T` не может быть одновременно `int` и `bool`, а между ними не существует преобразований. Третий вызов не скомпилируется, т.к. `T` выводится в `string`, а это нарушает ограничение о том, что он должен быть типом значения.

Компилятор консервативен: он будет сообщать об ошибке, только если точно знает, что некоторый код не может быть выполнен успешно, и проводит только относительно простые проверки по этому поводу. Существуют ситуации, при которых человеку может

быть очевидным (и доказуемым) тот факт, что код работать не будет, но компилятор все равно разрешает такой код. Конечно, если отдельная строка кода никогда не будет работать, то одиночный модульный тест, который ее выполняет, не пройдет, поэтому упрощенная природа проверки со стороны компилятора не играет особой роли при наличии хорошего покрытия кода тестами. Думайте об этом как о дополнительной премии в случаях, когда проблему *удается* обнаружить.

Это раскрывает наиболее важные моменты в плане того, что компилятор *может* сделать для вас. Однако нельзя применять тип `dynamic` абсолютно везде. Существуют ограничения, часть из которых болезненны, но большинство из них скрыты.

14.4.5. Ограничения, накладываемые на динамический код

Как правило, `dynamic` можно использовать везде, где обычно указывается имя типа, и затем писать нормальный код C#. Однако есть и несколько исключений. Список далеко не полон, но он охватывает случаи, с которыми можно столкнуться с довольно высокой вероятностью.

Расширяющие методы не распознаются динамически

Как вы уже видели, компилятор помещает *некоторый* контекст вызова внутрь вызывающего компонента. В частности, вызывающему компоненту известны статические типы, о которых осведомлен компилятор. Но в текущих версиях C# он *не* знает, какие директивы `using` встретятся в файле исходного кода, содержащем вызов. Это означает, что компилятор не располагает сведениями о том, какие расширяющие методы будут доступны во время выполнения.

В результате не только нельзя вызывать расширяющие методы *на* динамических значениях, но динамические значения также невозможно передавать расширяющим методам в качестве аргументов. Существуют два обходных пути, которые оба любезно предлагаются компилятором. Если вы знаете, какая перегруженная версия нужна, то можете привести динамическое значение к правильному типу внутри вызова метода. В противном случае, исходя из предположения, что вам известен статический класс, который содержит необходимый расширяющий метод, можете вызвать его как нормальный статический метод. В листинге 14.22 показан пример сбойного вызова и применение обоих обходных путей.

Листинг 14.22. Вызов расширяющих методов с динамическими аргументами

```
dynamic size = 5;
var numbers = Enumerable.Range(10, 10);
var error = numbers.Take(size);                    ← Ошибка на этапе компиляции
var workaround1 = numbers.Take((int) size);
var workaround2 = Enumerable.Take(numbers, size);
```

Оба подхода будут также работать в ситуации, когда расширяющий метод нужно вызвать с динамическим значением в качестве неявного значения `this`, хотя в таком случае приведение становится довольно неуклюжим.

Ограничения преобразований делегатов, содержащих `dynamic`

При преобразовании лямбда-выражения, анонимного метода или группы методов компилятор должен знать точный тип задействованного делегата (или выражения). Нельзя присвоить любое из них простой переменной типа `Delegate` или `object` без

приведения, и то же самое справедливо для dynamic. Однако приведения достаточно, чтобы удовлетворить компилятор. Это могло бы оказаться удобным в некоторых ситуациях, если необходимо динамически выполнить делегат позже. Если это полезно, можно также применять делегат с динамическим типом в качестве одного из параметров. В листинге 14.23 представлены примеры, часть из которых компилируется, а часть – нет.

Листинг 14.23. Динамические типы и лямбда-выражения

```
dynamic badMethodGroup = Console.WriteLine;
dynamic goodMethodGroup = (Action<string>) Console.WriteLine;

dynamic badLambda = y => y + 1;
dynamic goodLambda = (Func<int, int>) (y => y + 1);

dynamic veryDynamic = (Func<dynamic, dynamic>) (d => d.SomeMethod());
```

Обратите внимание, что из-за особенностей работы распознавания перегруженных версий это означает полную невозможность использования лямбда-выражений в динамически связываемых вызовах без приведения — даже если единственный метод, который мог бы вызываться, имеет тип делегата, известный на этапе компиляции. Например, следующий код не скомпилируется:

```
void Method(Action<string> action, string value)
{
  action(value);
}
...
dynamic text = "error";
Method(x => Console.WriteLine(x), text);     <-- Ошибка на этапе компиляции
```

Полезно отметить, что все это не теряется при взаимодействии LINQ и dynamic. Вы можете иметь дело со строго типизированной коллекцией с типом элементов dynamic и по-прежнему иметь возможность применять расширяющие методы, лямбда-выражения и даже выражения запросов. Коллекция может содержать объекты разных типов, и они будут вести себя соответствующим образом во время выполнения, как показано в листинге 14.24.

Листинг 14.24. Запрашивание коллекции динамических элементов

```
var list = new List<dynamic> { 50, 5m, 5d };
var query = from number in list
            where number > 4
            select (number / 20) * 10;

foreach (var item in query)
{
  Console.WriteLine(item);
}
```

Этот код выводит на консоль 20, 2.50 и 2.5. Я умышленно делил на 20 и затем умножал на 10, чтобы продемонстрировать разницу между decimal и double: тип decimal обеспечивает точность без нормализации, потому вместо 2.5 отображается 2.50. Первое значение является целым числом, так что применяется целочисленное деление; следовательно, получается значение 20, а не 25.

Конструкторы и статические методы

Вызывать конструкторы и статические методы динамическим образом можно в смысле указания динамических аргументов, но распознать конструктор или статический метод на динамическом типе нельзя. Дело в том, что нет никакого способа указания, какой тип имеется в виду.

Если вы попадете в ситуацию, когда *необходима* возможность делать это динамически каким-нибудь способом, подумайте об использовании методов экземпляра, например, создав фабричный тип. Может оказаться, что нужное динамическое поведение удастся получить с применением простого полиморфизма или интерфейсов, оставаясь в рамках статической типизации.

Объявления типов и параметры обобщенных типов

Не допускается объявлять, что тип имеет базовый класс `dynamic`. Кроме того, нельзя использовать `dynamic` в ограничении параметра типа или как часть набора интерфейсов, которые тип реализует. Однако `dynamic` *можно* применять в качестве аргумента типа для базового класса или при указании интерфейса для объявления переменной. Например, перечисленные ниже объявления недействительны:

- `class BaseTypeOfDynamic : dynamic`
- `class DynamicTypeConstraint<T> where T : dynamic`
- `class DynamicTypeConstraint<T> where T : List<dynamic>`
- `class DynamicInterface : IEnumerable<dynamic>`

Но следующие объявления будут допустимыми:

- `class GenericDynamicBaseClass : List<dynamic>`
- `IEnumerable<dynamic> variable;`

Большинство этих ограничений, касающихся обобщений, являются результатом того, что тип `dynamic` реально не существует как тип .NET. Среде CLR о нем ничего не известно — любые случаи использования `dynamic` в коде транслируются в тип `object`, к которому соответствующим образом применен атрибут `DynamicAttribute`. (Для типов, подобных `List<dynamic>` или `Dictionary<string, dynamic>`, этот атрибут четко указывает, какие части типа являются динамическими.) Атрибут `DynamicAttribute` применяется только тогда, когда динамическая природа должна быть представлена в метаданных; локальным переменным этот атрибут не требуется, поскольку никакие средства не нуждаются в инспектировании локальных переменных после компиляции для выявления их динамической природы.

Все динамическое поведение достигается за счет мастерства компилятора, проявляемого при трансляции кода, и мастерства *библиотеки* во время выполнения. Эквивалентность `dynamic` и `object` наглядна во многих местах, но наиболее очевидна она, пожалуй, при просмотре результатов операций `typeof(dynamic)` и `typeof(object)`, которые возвращают одну и ту же ссылку. В общем случае, если вам не удается добиться желаемого с помощью типа `dynamic`, вспомните, как он выглядит для среды CLR, и подумайте, может ли это прояснить проблему. Возможно, это и не натолкнет на решение, но, во всяком случае, вы заблаговременно получите лучшее представление о том, как все будет работать.

Вот мы и рассмотрели все детали того, как C# 4 трактует `dynamic`, но есть еще один аспект динамической типизации, на который крайне важно взглянуть, чтобы получить

хорошее представление об этой теме: динамическое реагирование. Одно дело иметь возможность *вызова* кода динамическим образом, но совсем другое дело обеспечить возможность динамического *реагирования* на такие вызовы.

Конечно, если вы просто динамически обращаетесь к стороннему коду или даже пользуетесь приемами вроде множественной диспетчеризации, показанными ранее, то беспокоиться о динамическом реагировании не понадобится. Я понимаю, что вы можете чувствовать себя сытыми по горло динамической типизацией, во всяком случае, в данный момент; мы уже разобрали огромное количество основ. Можете спокойно пропустить следующий раздел и вернуться к нему в будущем — в оставшихся главах книги нет ничего такого, что было бы основано на нем. С другой стороны, материал в этом разделе довольно интересен.

14.5. Реализация динамического поведения

Язык C# не предлагает какую-либо конкретную помощь в реализации динамического поведения, а вот инфраструктура предлагает. Чтобы реагировать динамическим образом, тип должен реализовывать интерфейс `IDynamicMetaObjectProvider`, но доступны две встроенных реализации, которые во многих случаях могут взять на себя множество работы. Мы рассмотрим их обе, а также *очень* простую реализацию `IDynamicMetaObjectProvider`, просто чтобы показать, что при этом задействуется. Все три подхода действительно отличаются, и начнем мы с простейшего из них: `ExpandoObject`.

14.5.1. Использование `ExpandoObject`

На первый взгляд тип `System.Dynamic.ExpandoObject` кажется забавным зверьком. Его единственный открытый конструктор не принимает параметров. Он не имеет открытых методов, если не считать явную реализацию разнообразных интерфейсов – критически важных `IDynamicMetaObjectProvider` и `IDictionary<string, object>`. (Все остальные реализуемые им интерфейсы связаны с тем, что `IDictionary<,>` расширяет другие интерфейсы.) И да — он запечатан, так что наследование с целью реализации полезного поведения невозможно. Тип `ExpandoObject` пригоден *только* для ссылки на него через `dynamic` или один из интерфейсов, которые он реализует.

Установка и извлечение отдельных свойств

Словарный интерфейс предоставляет подсказку по своему назначению — по существу он является способом хранения объектов по именам. Но через динамическую типизацию эти имена могут также использоваться в качестве свойств. В листинге 14.25 демонстрируются оба приема.

Листинг 14.25. Сохранение и извлечение значений с помощью `ExpandoObject`

```
dynamic expando = new ExpandoObject();
IDictionary<string, object> dictionary = expando;
expando.First = "value set dynamically";
Console.WriteLine(dictionary["First"]);

dictionary["Second"] = "value set with dictionary";
Console.WriteLine(expando.Second);
```

Строки в качестве значений в листинге 14.25 применяются просто ради удобства — можно использовать любой объект, как и ожидается от IDictionary<string, object>. Если для значения указать делегат, то затем его можно вызвать, как если бы он был методом объекта expando (листинг 14.26).

Листинг 14.26. Подделка методов в ExpandoObject с помощью делегатов

```
dynamic expando = new ExpandoObject();
expando.AddOne = (Func<int, int>) (x => x + 1);
Console.Write(expando.AddOne(10));
```

Хотя это выглядит похожим на доступ к методу, о нем можно также думать, как о доступе к свойству, которое возвращает делегат, с последующим вызовом этого делегата. Если бы вы создали статически типизированный класс со свойством AddOne типа Func<int, int>, то могли бы использовать точно такой же синтаксис. Код C#, генерируемый для вызова AddOne, на самом деле применяет операцию "активизация члена", а не пытается получить доступ к AddOne как к свойству и затем активизировать его, но типу ExpandoObject известно, что делать. При желании можно также обратиться к свойству, чтобы извлечь делегат. Давайте перейдем к чуть более длинному примеру, хотя в нем по-прежнему не будет делаться что-то особо сложное.

Создание дерева DOM

Мы создадим дерево из объектов ExpandoObject, которое зеркально отражает дерево XML DOM. Реализация будет довольно сырой, т.к. она предназначена для упрощения демонстрации, а не для реального использования. В частности, в ней предполагается, что пространства имен XML, которые пришлось бы учитывать, отсутствуют.

Каждый узел в дереве имеет две пары "имя/значение", присутствующие всегда: XElement, хранящий исходный элемент LINQ to XML, который применялся для создания узла, и ToXml, хранящий делегат, который возвращает узел в виде строки XML. Можно было бы просто вызвать node.XElement.ToString(), но такой способ дает еще один пример работы делегатов с ExpandoObject. Следует упомянуть о том, что мы будем использовать ToXml вместо ToString, поскольку установка свойства ToString объекта ExpandoObject *не приводит* к переопределению нормального метода ToString(). Поскольку из-за этого могут возникать сбивающие с толку ошибки, мы будем применять другое имя.

Интересной частью здесь являются не фиксированные имена, а те, которые зависят от реальной разметки XML. Мы будем полностью игнорировать атрибуты, но любые *элементы* в первоначальной разметке XML, являющиеся дочерними по отношению к исходному элементу, доступны через свойства, имеющие такие же имена. Например, рассмотрим следующую разметку XML:

```
<root>
  <branch>
    <leaf />
  </branch>
</root>
```

Предполагая, что динамическая переменная по имени root представляет элемент root, к узлу leaf можно обратиться с помощью двух простых доступов к свойствам, которые могут быть оформлены в виде одного оператора:

```
dynamic leaf = root.branch.leaf;
```

Если элемент встречается более одного раза внутри своего родительского элемента, это свойство ссылается на первый элемент с таким именем. Чтобы обеспечить доступ к другим элементам, каждый элемент будет также открыт через свойство, имеющее имя элемента с суффиксом *List*, которое возвращает список `List<dynamic>`, содержащий все элементы с указанным именем в порядке их следования внутри документа. Другими словами, доступ также может быть представлен как `root.branchList[0].leaf` или, возможно, `root.branchList[0].leafList[0]`. Обратите внимание, что индексатор здесь применяется к списку — определять собственное поведение индексатора для объектов `ExpandoObject` нельзя.

Реализация всего сказанного удивительно проста, с единственным рекурсивным методом, выполняющим всю работу, как показано в листинге 14.27.

Листинг 14.27. Реализация упрощенного преобразования дерева XML DOM с помощью `ExpandoObject`

```
public static dynamic CreateDynamicXml(XElement element)
{
   dynamic expando = new ExpandoObject();
   expando.XElement = element;                                    ❶ Присваивание простого свойства

   expando.ToXml = (Func<string>)element.ToString;                ❷ Преобразование группы методов в делегат для использования как свойства

   IDictionary<string, object> dictionary = expando;
   foreach (XElement subElement in element.Elements())
   {
      dynamic subNode = CreateDynamicXml(subElement);             ❸ Рекурсивная обработка подэлемента
      string name = subElement.Name.LocalName;
      string listName = name + "List";
      if (dictionary.ContainsKey(name))
      {
         ((List<dynamic>) dictionary[listName]).Add(subNode);     ❹ Добавление повторяющегося элемента в список
      }
      else
      {
         dictionary[name] = subNode;                              ❺ Создание нового списка и установка свойств
         dictionary[listName] = new List<dynamic> { subNode };
      }
   }
   return expando;
}
```

Без обработки списка код в листинге 14.27 был бы еще проще. Свойства `XElement` и `ToXml` устанавливаются динамически (❶ и ❷), но это невозможно сделать для элементов либо их списков, т.к. нужные имена на этапе компиляции не известны[7]. Взамен используется словарное представление (❹ и ❺), которое также позволяет легко проверять повторяющиеся элементы. Нельзя выяснить, содержит ли объект `ExpandoObject` значение для отдельного ключа, просто обратившись к нему как к свойству; любая попытка получения доступа к свойству, которое еще было определено, приводит к генерации исключения.

[7] В этом есть определенная ирония — имена, известные статически, могут быть установлены динамически, но для имен, известных динамически, должна использоваться статическая типизация.

Рекурсивная обработка подэлементов в динамическом коде столь же прямолинейна, как в статически типизированном коде; вы просто вызываете метод рекурсивно ❶ с каждым подэлементом, применяя результат для заполнения соответствующих свойств.

Для примера необходима какая-то разметка XML, которую удобно изобразить как графически, так и в низкоуровневом формате. Мы будем использовать простую структуру, представляющую книги. Каждая книга имеет одно название, представленное в виде атрибута, и может иметь несколько авторов, каждый со своим элементом.

На рис. 14.4 показан полный файл в виде дерева; а ниже приведена разметка XML в низкоуровневом формате:

```
<books>
  <book name="Mortal Engines">
    <author name="Philip Reeve" />
  </book>
  <book name="The Talisman">
    <author name="Stephen King" />
    <author name="Peter Straub" />
  </book>
  <book name="Rose">
    <author name="Holly Webb" />
    <excerpt>
      Rose was remembering the illustrations from
      Morally Instructive Tales for the Nursery.
    </excerpt>
  </book>
</books>
```

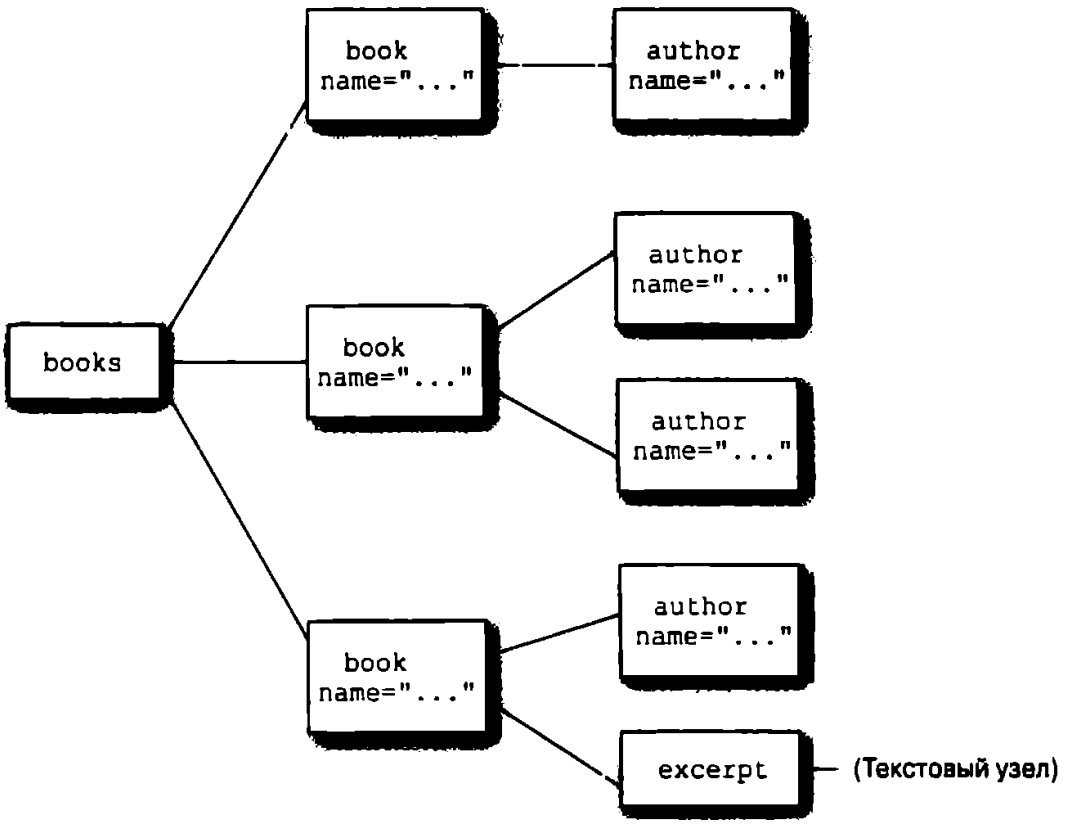

Рис. 14.4. Древовидная структура для примера файла XML

В листинге 14.28 представлен краткий пример того, как код, работающий с ExpandoObject, может применяться с этим XML-документом, включая свойства ToXml и XElement. Файл books.xml содержит XML-документ, показанный на рис. 14.4.

Листинг 14.28. Использование динамического дерева DOM, созданного из объектов ExpandoObject

```
XDocument doc = XDocument.Load("books.xml");
dynamic root = CreateDynamicXml(doc.Root);
Console.WriteLine(root.book.author.ToXml());
Console.WriteLine(root.bookList[2].excerpt.XElement.Value);
```

Код в листинге 14.28 не должен вызывать удивление, разве что если вы не знакомы со свойством XElement.Value, которое просто возвращает текст, находящийся внутри элемента. Вывод, полученный в результате выполнения кода из этого листинга, выглядит вполне ожидаемо:

```
<author name="Philip Reeve" />
Rose was remembering the illustrations from
Morally Instructive Tales for the Nursery.
```

Все это хорошо, но с применяемым деревом DOM связано несколько проблем.

- Оно вообще не обрабатывает атрибуты.
- Для каждого имени элемента требуются два свойства из-за необходимости в представлении списков.
- Было бы неплохо переопределить метод ToString(), а не добавлять дополнительное свойство.
- Результат является изменяемым — ничто не препятствует последующему добавлению в коде собственных свойств.
- Несмотря на изменяемость объекта ExpandoObject, он не будет отражать изменения лежащему в основе XElement (который также изменяемый).
- Существует много возможностей для возникновения конфликтов имен, например, наличие узла, содержащего элементы Foo и FooList либо же элементы с именами XElement или ToXml.
- Полное дерево заполняется заранее, что влечет за собой выполнение большого объема работ, даже если интересует всего несколько узлов.

Устранение этих проблем требует большего контроля, чем только возможность установки свойств. Давайте рассмотрим тип DynamicObject.

14.5.2. Использование DynamicObject

Тип DynamicObject предлагает более мощный подход к взаимодействию со средой DLR, чем тип ExpandoObject, но все равно он намного проще реализации интерфейса IDynamicMetaObjectProvider вручную. Хотя *на самом деле* он не является абстрактным классом, чтобы извлечь из него хоть какую-то пользу, необходимо унаследовать от него свой класс — его единственный конструктор объявлен защищенным, поэтому на практике данный тип можно также считать абстрактным.

Может понадобиться переопределить четыре вида методов.

- Методы вызова `TryXXX()`, представляющие динамические обращения к объекту.
- Метод `GetDynamicMemberNames()`, который может возвращать список доступных членов.
- Нормальные методы `Equals()`, `GetHashCode()` и `ToString()`, которые могут быть переопределены обычным образом.
- Метод `GetMetaObject()`, возвращающий метаобъект, который используется средой DLR.

Для улучшения представления XML DOM мы рассмотрим все методы кроме последнего, а в следующем разделе, посвященном реализации интерфейса `IDynamicMetaObjectProvider` с нуля, обсудим метаобъекты. Вдобавок может быть полезно создать в производном типе новые члены, даже при условии, что в вызывающем коде, скорее всего, его экземпляры будут применяться как динамические значения. Прежде чем приступить к выполнению любого из этих действий, необходимо создать класс для удержания всех нужных членов.

Начало работы

Поскольку мы наследуем свой тип от `DynamicObject`, а не просто вызываем его методы, то должны начать с объявления этого класса. В листинге 14.29 приведен базовый каркас, который мы будем расширять.

Листинг 14.29. Каркас класса `DynamicXElement`

```
public class DynamicXElement : DynamicObject
{
  private readonly XElement element;                          ❶ Поле типа XElement, содержащееся в этом экземпляре

  private DynamicXElement(XElement element)                   ❷ Закрытый конструктор, предотвращающий создание экземпляров напрямую
  {
    this.element = element;
  }

  public static dynamic CreateInstance(XElement element)      ❸ Открытый метод для создания экземпляров
  {
    return new DynamicXElement(element);
  }
}
```

Класс `DynamicXElement` является просто оболочкой для типа `XElement` ❶. Он будет хранить все необходимое состояние, представляя собой воплощение важного проектного решения. Ранее в `ExpandoObject` вы рекурсивно проходили по его структуре и заполняли целое отображаемое дерево. Это действительно было необходимо, т.к. иначе позже не удалось бы перехватить доступ к свойствам из специального кода. Очевидно, что такой подход более дорогостоящий, чем подход с `DynamicXElement`, при котором элементы дерева помещаются в оболочку, только когда они на самом деле нужны. Кроме того, это означает, что любые изменения, внесенные в `XElement` после создания объекта `ExpandoObject`, теряются; например, если добавить дополнительные подэлементы, они не будут видны как свойства, поскольку они не присутствовали при получении снимка дерева. Облегченный подход с помещением в оболочку всегда обеспечивает "актуальность" — любые изменения, произведенные в дереве, будут видимыми через оболочку.

Недостаток этого подхода связан с тем, что вы больше не поддерживаете ту же самую идею идентичности, которая была ранее. В случае `ExpandoObject` выражение `root.book.author` было бы оценено в ту же ссылку, если бы оно использовалось дважды. В ситуации с `DynamicXElement` выражение оценивается каждый раз, когда создаются новые экземпляры для помещения в них подэлементов. Чтобы обойти это, *можно* было бы реализовать какую-нибудь разновидность интеллектуального кеширования, но все очень быстро стало бы чрезмерно сложным.

В листинге 14.29 конструктор `DynamicXElement` является закрытым ❷, а для создания экземпляров предусмотрен открытый статический метод ❶. Этот метод имеет возвращаемый тип `dynamic`, потому что вы ожидаете, что разработчики именно таким способом будут пользоваться вашим классом. В качестве небольшой альтернативы можно было бы создать отдельный открытый статический класс с методом, расширяющим тип `XElement`, и хранить внутри него `DynamicXElement`. Сам класс является деталью реализации; в его применении мало смысла, если работа не ведется динамическим образом.

Имея каркас, можно приступать к наращиванию его функциональных возможностей. Мы начнем с действительно простого действия: добавим методы и индексаторы, как если бы это был нормальный класс.

Поддержка `DynamicObject` для простых членов

При обсуждении типа `ExpandoObject` упоминались два члена, которые добавляются *всегда*: метод `ToXml()` и свойство `XElement`. На этот раз не придется добавлять новый метод для преобразования объекта в строковое представление; можно переопределить нормальный метод `ToString()`. Можно также предоставить свойство `XElement`, как при написании любого другого класса.

Одной из замечательных особенностей типа `DynamicObject` является то, что если какое-то поведение не должно быть по-настоящему динамическим, то и не придется реализовывать его динамическим образом. Прежде чем ассоциированный метаобъект воспользуется любым из методов `TryXXX()`, он проверяет, существует ли член в форме прямолинейного члена CLR. Если это так, данный член будет вызван. Это значительно упрощает жизнь.

Также в `DynamicXElement` будут предусмотрены два индексатора, предназначенные для предоставления доступа к атрибутам и замены списков элементов. В листинге 14.30 показан новый код, добавленный в класс.

Листинг 14.30. Добавление в класс `DynamicXElement` нединамических членов

```
public override string ToString()          ←❶ Переопределение ToString() обычным образом
{
   return element.ToString();
}
public XElement XElement                   ←❷ Возвращение элемента из оболочки
{
   get { return element; }
}
public XAttribute this[XName name]         ←❸ Индексатор, извлекающий атрибут
{
   get { return element.Attribute(name); }
}
```

```
public dynamic this[int index]      ←❶ Индексатор, извлекающий родственный элемент
{
  get
  {
    XElement parent = element.Parent;
    if (parent == null)             ←❷ Является ли данный элемент корневым?
    {
      if (index != 0)
      {
        throw new ArgumentOutOfRangeException();
      }
      return this;
    }
    XElement sibling = parent.Elements(element.Name)     ←❸ Поиск соответствующего родственного элемента
                              .ElementAt(index);
    return element == sibling ? this
                              : new DynamicXElement(sibling);
  }
}
```

В листинге 14.30 присутствует немалый объем кода, но большая его часть проста. Метод `ToString()` переопределяется ❶ путем передачи вызова типу `XElement`, и если вы желаете реализовать эквивалентность значений, можете предпринять похожее действие для `Equals()` и `GetHashCode()`. Свойство, возвращающее лежащий в основе элемент ❷, и индексатор для атрибутов ❸ также просты, хотя полезно отметить, что использовать `XName` для параметра необходимо только в индексаторе атрибутов; если во время выполнения вы предоставите строку, тип `DynamicObject` позаботится о вызове неявного преобразования в `XName`.

Самый сложный аспект кода — понять, для какой работы был задуман индексатор с параметром типа `int` ❶. Вероятно, проще всего это объяснить в терминах ожидаемого применения. Идея в том, чтобы избежать необходимости в наличии дополнительного спискового свойства, заставляя элемент действовать и в качестве одиночного элемента, и в качестве списка дочерних элементов с тем же самым именем. На рис. 14.5 представлен предыдущий пример разметки XML с несколькими выражениями, позволяющими достигать различных узлов внутри нее.

После выяснения, для чего предназначен этот индексатор, реализация становится довольно простой, усложняемой только за счет возможности того, что вы уже можете находиться в корне дерева ❷. В противном случае нужно всего лишь запросить у элемента все его родственные элементы и затем выбрать тот, который интересует ❸.

До сих пор мы пока не видели ничего динамического кроме возвращаемого типа `CreateInstance()` — ни один из приведенных примеров не будет работать, т.е. еще не написан код для извлечения подэлементов. Давайте исправим положение.

Переопределение методов `TryXXX()`

В типе `DynamicObject` вы реагируете на вызовы динамическим образом путем переопределения одного из методов `TryXXX()`. Таких методов 12, и они представляют разные типы операций, как показано в табл. 14.1.

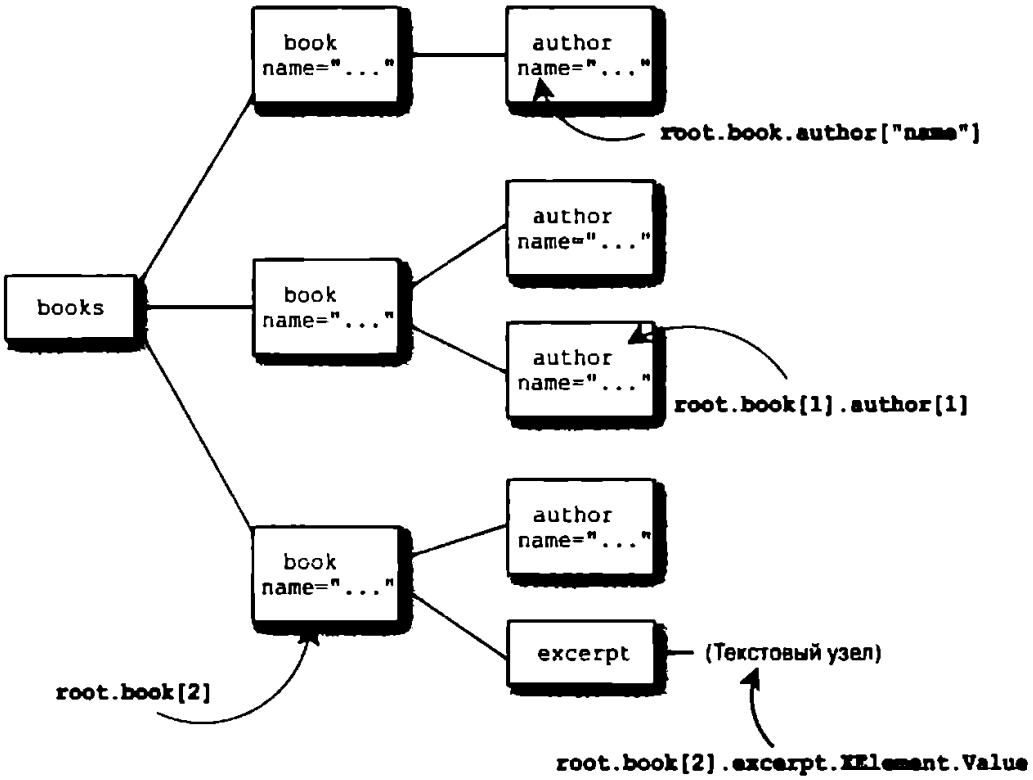

Рис. 14.5. Выбор данных с использованием `DynamicXElement`

Таблица 14.1. Виртуальные методы `TryXXX()` в типе `DynamicObject`

Имя	Тип представляемого вызова (где `x` — динамический объект)
`TryBinaryOperation()`	Бинарная операция, такая как `x + y`
`TryConvert()`	Преобразования, такие как `(Target) x`
`TryCreateInstance()`	Выражения создания объектов; эквивалентная возможность в C# отсутствует
`TryDeleteIndex()`	Операция удаления индексатора; эквивалентная возможность в C# отсутствует
`TryDeleteMember()`	Операция удаления свойства; эквивалентная возможность в C# отсутствует
`TryGetIndex()`	Средство получения индексатора, такое как `x[10]`
`TryGetMember()`	Средство получения свойства, такое как `x.Property`
`TryInvoke()`	Прямой вызов, трактующий `x` подобно делегату, такой как `x(10)`
`TryInvokeMember()`	Вызов члена, такой как `x.Method()`
`TrySetIndex()`	Средство установки индексатора, такое как `x[10] = 20`
`TrySetMember()`	Средство установки свойства, такое как `x.Property = 10`
`TryUnaryOperation()`	Унарная операция, такая как `!x` или `-x`

Каждый из этих методов имеет булевский возвращаемый тип для указания, успешно ли прошло связывание. Каждый принимает в первом параметре подходящий связыватель, и если операция логически имеет аргументы (например, аргументы для метода или индексы для индексатора), то они представлены как `object[]`. Наконец, если операция может возвращать значение (это делают все операции кроме операций установки и удаления), для его хранения определен параметр `out` типа `object`.

Точный тип связывателя зависит от операции; для каждой операции предусмотрен собственный тип связывателя. Например, полная сигнатура метода `TryInvokeMember()` выглядит следующим образом:

```
public virtual bool TryInvokeMember(InvokeMemberBinder binder,
  object[] args, out object result)
```

Переопределять необходимо лишь те методы, которые представляют операции, поддерживаемые динамически. В данном случае имеются динамические свойства, допускающие только чтение (для элементов), поэтому понадобится переопределить метод `TryGetMember()`, как показано в листинге 14.31.

Листинг 14.31. Реализация динамического свойства с помощью метода `TryGetMember()`

```
public override bool TryGetMember(GetMemberBinder binder, out object result)
{
  XElement subElement = element.Element(binder.Name);   ❶ Поиск первого подходящего подэлемента
  if (subElement != null)
  {
    result = new DynamicXElement(subElement);   ❷ Построение нового динамического элемента, если подэлемент найден
    return true;
  }
  return base.TryGetMember(binder, out result);   ❸ В противном случае использование базовой реализации
}
```

Реализация, представленная в листинге 14.31, проста. Связыватель содержит имя запрашиваемого свойства, поэтому вы ищете соответствующий подэлемент в дереве ❶. Если он найден, для него создается новый объект `DynamicXElement`, который присваивается выходному параметру `result`, и возвращается значение `true`, указывающее на то, что вызов был успешно привязан ❷. Если подэлемент с корректным именем обнаружить не удалось, вы просто вызываете базовую реализацию метода `TryGetMember()` ❸. Базовая реализация каждого метода `TryXXX()` только возвращает `false` и устанавливает выходной параметр в `null`, если он предусмотрен. Это можно было бы легко сделать явным образом, но тогда получилось бы два отдельных оператора: один для установки выходного параметра и один для возвращения `false`. Если вы предпочитаете иметь дело с несколько более длинным кодом, то нет никаких причин избегать его — базовые реализации просто чуть удобнее при выполнении действий, требуемых для указания о том, что связывание потерпело неудачу.

Я обошел один сложный момент: связыватель имеет еще одно свойство (`IgnoreCase`), которое определяет, должно ли свойство быть привязано нечувствительным к регистру способом. Например, язык Visual Basic нечувствителен к регистру, так что его реализация связывателя для этого свойства возвратила бы `true`, в то время как реализация связывателя C# — значение `false`. В такой ситуации возникает небольшое неудобство. Мало того, что в методе `TryGetMember()` пришлось бы выполнять больше работы для поиска элемента в нечувствительной к регистру манере (больше работы — всегда неприятно, однако это не является веской причиной отказа от реализации метода), но

возникает также проблема философского характера: что произойдет, когда впоследствии такой индексатор (по числу) будет применен для выбора родственных элементов? Должен ли объект запоминать, что он является чувствительным к регистру, и таким же способом позже выбирать родственные элементы? Можно было бы легко столкнуться с ситуациями, когда поведение трудно предсказать и нелегко объяснить в документации. Такая разновидность рассогласования, скорее всего, возникнет и в других похожих случаях. Если вы стремитесь к совершенству, то вероятно поставите себя в затруднительное положение. Взамен найдите прагматичное решение, которое вы уверенно сможете реализовать и сопровождать, а затем документируйте ограничения.

Теперь, когда все нужное на месте, можно приступить к тестированию класса `DynamicXElement`, как показано в листинге 14.32.

Листинг 14.32. Тестирование класса `DynamicXElement`

```
XDocument doc = XDocument.Load("books.xml");
dynamic root = DynamicXElement.CreateInstance(doc.Root);
Console.WriteLine(root.book[2]["name"]);
Console.WriteLine(root.book[1].author[1]);
Console.WriteLine(root.book);
```

Разумеется, в класс можно было бы добавить дополнительную сложность. Скажем, можно предусмотреть свойство `Parent` для прохода вверх по дереву либо изменить код с целью использования методов для доступа к подэлементам и обеспечить доступ к свойствам, представляющим атрибуты. Принцип был бы тем же: там, где имя известно заранее, оно реализуется в виде нормального члена класса. Если требуется динамическое поведение, следует переопределить соответствующий метод типа `DynamicObject`.

Осталось отшлифовать еще одну деталь применительно к типу `DynamicXElement`, прежде чем мы оставим его в покое. Пора проинформировать о том, что мы собираемся предложить.

Переопределение метода `GetDynamicMemberNames()`

Некоторые языки, подобные Python, позволяют запрашивать у объекта известные ему имена. Например, в Python можно применять функцию `dir()` для вывода их списка. Эта информация полезна в среде REPL и она также может быть удобной при отладке в IDE-среде. Среда DLR делает эту информацию доступной через метод `GetDynamicMemberNames()` в типах `DynamicObject` и `DynamicMetaObject` (последний вскоре будет рассмотрен). Все, что потребуется сделать — это переопределить данный метод для предоставления последовательности имен динамических членов, в результате чего свойства вашего объекта станут более обнаруживаемыми. В листинге 14.33 представлена реализация указанного метода для `DynamicXElement`.

Листинг 14.33. Реализация метода `GetDynamicMemberNames()` в `DynamicXElement`

```
public override IEnumerable<string> GetDynamicMemberNames()
{
    return element.Elements()
                  .Select(x => x.Name.LocalName)
                .Distinct()
                .OrderBy(x => x);
}
```

Как видите, понадобился лишь простой запрос LINQ. Так случается *не всегда*, но я подозреваю, что многие динамические реализации будут иметь возможность использовать LINQ подобным образом.

Необходимо удостовериться в том, что при наличии нескольких элементов с одинаковым именем одно и то же значение не возвращается более одного раза, и отсортировать результат в целях согласованности. В отладчике Visual Studio 2010 можно развернуть узел Dynamic View (Динамическое представление) динамического объекта и просмотреть имена и значения свойств (рис. 14.6).

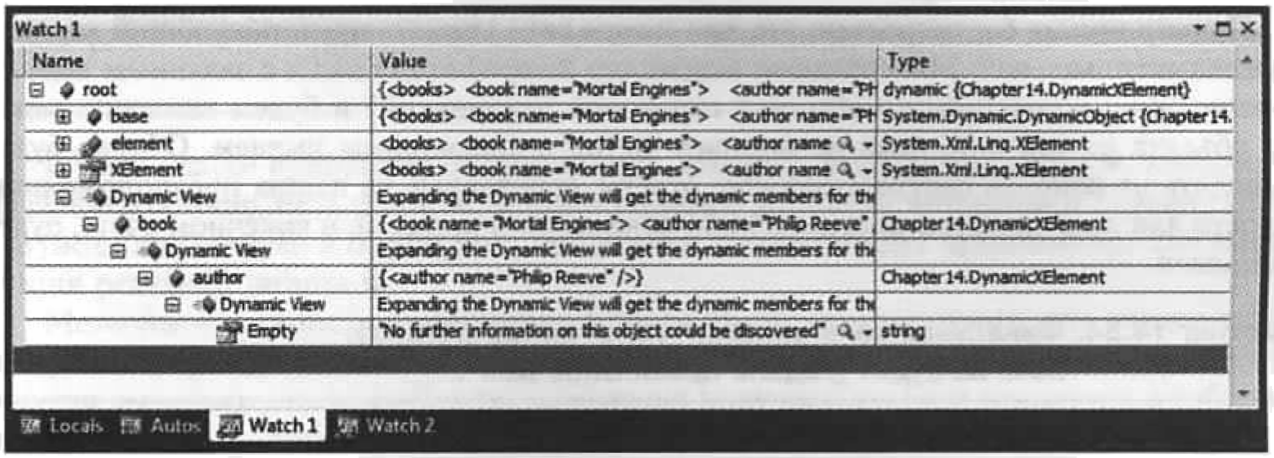

Рис. 14.6. Окно среды Visual Studio 2010, отображающее динамические свойства объекта `DynamicXElement`

Развертывая узлы Dynamic View на каждом уровне, можно углубиться в детали динамического объекта. На рис. 14.6 были последовательно развернуты узлы документа, первой книги и ее автора. Представление Dynamic View для автора показывает, что дальнейшей информации в иерархии больше нет.

Итак, класс `DynamicXElement` завершен в той степени, в какой он отвечает требованиям этой главы. Я уверен, что тип `DynamicObject` соблюдает баланс между управляемостью и простотой: его довольно легко понять, и он обладает гораздо меньшими ограничениями, чем `ExpandoObject`. Но если вам нужен действительно полный контроль над связыванием, то придется реализовать интерфейс `IDynamicMetaObjectProvider` напрямую.

14.5.3. Реализация интерфейса `IDynamicMetaObjectProvider`

В этом разделе я не буду вдаваться в слишком мелкие детали, а хочу в действительности продемонстрировать, по крайней мере, один пример низкоуровневого динамического поведения. Трудность реализации `IDynamicMetaObjectProvider` заключается не в самом интерфейсе — ее порождает экземпляр `DynamicMetaObject`, который должен быть возвращен из единственного метода этого интерфейса. Тип `DynamicMetaObject` немного похож на `DynamicObject` в том, что он содержит множество методов, часть из которых переопределяется для оказания влияния на поведение; там, где ранее вы переопределяли `DynamicObject.TryGetMember()`, здесь пришлось бы переопределять `DynamicMetaObject.BindGetMember()`. Но идея состоит в том, что внутри переопределенных методов вместо выполнения требуемого действия напрямую необходимо строить дерево выражения, *описывающее* требуемое действие и обстоятельства, при которых действие должно быть предпринято. Причина такого дополнительного уровня косвенности кроется в том, что это *метаобъект*.

Сначала будет рассмотрен пример, а затем даны краткие объяснения. На самом деле я хочу здесь четко изложить отличие на уровне взаимодействия — это немного похоже на разбор внутренностей JIT-компилятора. Большинству разработчиков на C# такие детали не нужны, а если вам они все же понадобились, то, скорее всего, вы занимаетесь написанием библиотеки, которая реагирует динамически, но должна также обладать приемлемой производительностью. В качестве альтернативы ваш интерес к данной теме может объясняться попыткой построения собственного динамического языка. Если дела обстоят именно так, то желаю удачи — и найдите более исчерпывающий ресурс, чем этот скудный пример.

Пример весьма бесхитростен; это тип `Rumpelstiltskin`, представляющий карлика Румпельштильцхена[8]. Мы создадим экземпляр `Rumpelstiltskin` с заданным именем (хранящимся в совершенно обычной строковой переменной) и будем вызывать методы объекта до тех пор, пока не вызовем метод с правильным именем. Объект будет выводить на консоль соответствующие ответы, основанные на наших предположениях. Просто для прояснения в листинге 14.34 показан код, который, в конечном счете, будет выполнен.

Листинг 14.34. Финальная цель: динамический вызов методов, пока не будет угадано правильное имя

```
dynamic x = new Rumpelstiltskin("Hermione");
x.Harry();
x.Ron();
x.Hermione();
```

Объект не мог получить имя `Rumpelstiltskin`, иначе все стало бы слишком очевидным. Вместо этого мы будем упоминать ряд других волшебников, хотя никто из них не достиг значительных высот в алхимии. Цель первых двух обращений к методам — дать в результате отрицание, а третьего вызова метода — признать поражение. Методы будут также возвращать булевское значение, указывающее на успешность предположения, но для краткости результаты возврата здесь не используются.

Давайте рассмотрим сначала тип `Rumpelstiltskin`. Не забывайте, что это *не* метаобъект — он появится позже. В листинге 14.35 приведен полный код.

Листинг 14.35. Тип `Rumpelstiltskin` без своего кода метаобъекта

```
public sealed class Rumpelstiltskin : IDynamicMetaObjectProvider
{
  private readonly string name;
  public Rumpelstiltskin(string name)                    <-- (1) Конструирование нового экземпляра
  {
    this.name = name;
  }
  public DynamicMetaObject
    GetMetaObject(Expression expression)                 <-- (2) Предоставление динамического поведения
  {
    return new MetaRumpelstiltskin(expression, this);
  }
```

[8] Если вы не читали сказку братьев Гримм о Румпельштильцхене, ознакомьтесь со статьей в Википедии (`http://ru.wikipedia.org/wiki/Румпельштильцхен`). После этого пример сразу станет более понятным!

```
    private object RespondToWrongGuess(string guess)  <-- ❸ Ответы на предположения
    {
      Console.WriteLine("No, I'm not {0}! (I'm {1}.)",
        guess, name);
      return false;
    }
    private object RespondToRightGuess()
    {
      Console.WriteLine("Curses! Foiled again!");
      return true;
    }
}
```

В этом классе присутствуют три аспекта. Имеется конструирование ❶, которое совершенно обычно. Есть реализация единственного метода из интерфейса `IDynamicMetaObjectProvider` ❷ и определены два метода, которые будут применяться для выполнения реальной работы ❸.

Метаобъекту, сконструированному в точке ❷, необходимо знать, для какого экземпляра он формирует ответ, и какое дерево выражения относится к этому экземпляру внутри вызывающего кода. Дерево выражения получается в виде параметра, а собственный экземпляр — через ссылку `this`, поэтому они и передаются конструктору.

Почему методы возвращают тип object?

Вас может интересовать, по какой причине методы объявлены как возвращающие тип `object`, а не `bool`. В моей исходной реализации на самом деле присутствовали методы `void`, но, к сожалению, вызовы динамических методов рассчитаны на возвращение чего-либо, и, согласно моему опыту, связыватель всегда ожидает тип `object`. (Предусмотрено свойство `ReturnType`, которое можно проверить.) Это приводит к тому, что вызов метода `void` сгенерирует исключение во время выполнения, и то же самое справедливо в отношении метода `bool`; для корректного сопоставления типов понадобится самостоятельно обеспечить упаковку. Можно было бы встроить упаковку в дерево выражения, но это намного сложнее, чем изменение возвращаемого типа метода. С тонкостями подобного рода придется иметь дело при реализации интерфейса `IDynamicMetaObjectProvider` в реальности.

Строго говоря, два метода ответов *не нужны*. Когда вы строите поведение для реагирования на входящие вызовы методов, то *могли бы* представить такую логику непосредственно в дереве выражения. Но поступать так относительно сложно по сравнению с обычным возвращением дерева выражения, которое вызывает правильный метод. Хотя если говорить более конкретно, то в данном случае это не было бы слишком сложным; в других ситуациях все может оказаться намного хуже. В сущности, вы создадите шлюз между статическим и динамическим мирами, реагируя на вызовы динамических методов за счет перенаправления их статическим методам с подходящими аргументами. Это приводит к более простому коду в метаобъекте.

С учетом сказанного, давайте, наконец, взглянем на код класса `MetaRumpelstiltskin` — он показан в листинге 14.36 вместе с закрытым вложенным классом.

Листинг 14.36. Реальное динамическое содержимое класса `Rumpelstiltskin` – его метаобъект

```
private class MetaRumpelstiltskin : DynamicMetaObject
{
   private static readonly
     MethodInfo RightGuessMethod =                    ←❶ Получение методов через рефлексию
     typeof(Rumpelstiltskin).GetMethod("RespondToRightGuess",
     BindingFlags.Instance | BindingFlags.NonPublic);

   private static readonly MethodInfo WrongGuessMethod =
     typeof(Rumpelstiltskin).GetMethod("RespondToWrongGuess",
     BindingFlags.Instance | BindingFlags.NonPublic);

   internal MetaRumpelstiltskin                       ←❷ Делегирование конструирования базовому классу
     (Expression expression, Rumpelstiltskin creator)
     : base(expression, BindingRestrictions.Empty, creator)
   {}

   public override DynamicMetaObject BindInvokeMember  ←❸ Реагирование на обращение к члену
     (InvokeMemberBinder binder, DynamicMetaObject[] args)
   {
     Rumpelstiltskin targetObject
       = (Rumpelstiltskin)base.Value;                 ←❹ Повторное обращение к "реальному" объекту
     Expression self = Expression.Convert(base.Expression,
        typeof(Rumpelstiltskin));

     Expression targetBehavior;                       ←❺ Определение подходящего поведения
     if (binder.Name == targetObject.name)
     {
       targetBehavior = Expression.Call(self, RightGuessMethod);
     }
     else
     {
       targetBehavior = Expression.Call(self, WrongGuessMethod,
         Expression.Constant(binder.Name));
     }

     var restrictions = BindingRestrictions.GetInstanceRestriction
         (self, targetObject);
     return new DynamicMetaObject(targetBehavior,     ←❻ Реагирование с применением поведения и ограничений
                                  restrictions);
   }
}
```

Печатая это, я почти представлял ваши остекленевшие глаза. Листинг 14.36 содержит сложный для понимания код, который выглядит как приложение огромных трудозатрат для выполнения столь простой работы. Не забывайте о том, что это вряд ли вам когда-либо понадобится, поэтому расслабьтесь и постарайтесь уловить изюминку кода, прежде чем детали поглотят вас.

Первая половина кода по-настоящему проста. Мы сохраняем данные `MethodInfo` для двух методов ответов в статических переменных ❶ (они не меняются для разных

экземпляров) и объявляем конструктор, который ничего не делает, но передает свои параметры конструктору базового класса ❷. Вся реальная работа выполняется в методе `BindInvokeMember()` ❸, который должен выяснить два момента — каким образом объект должен реагировать на вызов метода и обстоятельства, при которых это решение является действительным.

Реагировать необходимо вызовом либо `RespondToRightGuess()`, либо `RespondToWrongGuess()`, основываясь на том, что имя метода в вызове совпадает с именем объекта. Метаобъекту известен реальный экземпляр, поскольку он передается конструктору. Мы обращаемся к нему снова с использованием свойства `Value` и запоминаем его в переменной `targetObject` ❹. Также потребуется дерево выражения, которое первоначально применялось для создания метаобъекта, чтобы можно было привязывать вызов подходящего метода целиком внутри деревьев выражений. Метод `Expression.Convert()` является эквивалентом, реализованным в дереве выражения, приведения в предыдущей строке.

Зная реальный объект, можно сравнить его имя с именем привязываемого вызова метода, который доступен через свойство `InvokeMemberBinder.Name`. Вызов привязывается к соответствующему методу с использованием метода `Expression.Call()`, которому в качестве аргумента передается имя метода, если предположение оказалось ошибочным ❺. Опять-таки, я хотел бы подчеркнуть, что в этой точке вы на самом деле не вызываете метод, а только описываете его вызов.

Ограничения в данном случае просты: этот вызов будет всегда привязываться одинаково, если он производится с одним и тем же аргументом, но по-разному, если он осуществляется на разных объектах, т.к. он мог бы иметь отличающееся имя. Метод `GetInstanceRestriction()` возвращает соответствующее ограничение; если требуется обеспечить одинаковое поведение независимо от того, на каком экземпляре метод был вызван, взамен можно применить метод `GetTypeRestriction()`, тем самым указав, что вызов должен обрабатываться одинаково для любого экземпляра `Rumpelstiltskin`. Полный исходный код включает альтернативную реализацию, которая делает в точности описанное, всегда передавая действительное имя метода и помещая проверку условия внутрь нормального метода.

Наконец, мы создаем новый экземпляр `DynamicMetaObject`, представляющий результат связывания ❿. Может несколько сбивать с толку, что результат имеет тот же тип, что и объект, осуществляющий связывание, но именно так работает среда DLR.

К этому моменту все готово — постучите по дереву, запустите код и посмотрите, работает ли он... Затем проведите несколько сеансов отладки, чтобы выяснить, что идет не так, если вы в чем-то похожи на меня. Как я уже упоминал, это не то, что большинство разработчиков должны брать на вооружение — это немного похоже на LINQ, когда основная масса программистов будут пользоваться самим языком LINQ, а не реализовывать собственный поставщик LINQ, основанный на `IQueryable`. Полезно получить общее представление о том, как все работает, нежели трактовать его как "магию", но большую часть времени вы сможете получать удовольствие от результатов великолепной работы, выполненной командой проектировщиков среды DLR.

14.6. Резюме

Такое чувство, что мы довольно далеко ушли от преимущественно статически типизированного языка C#. Мы рассмотрели ситуации, где динамическая типизация могла быть удобной, а также ознакомились с тем, каким образом в C# 4 она становится воз-

можной (в терминах кода и ее внутренней работы) и как динамически реагировать на вызовы. По ходу изложения был продемонстрирован код для COM и Python, использование рефлексии и некоторые сведения о DLR.

Данная глава не задумывалась как полное руководство по функционированию среды DLR или даже по взаимодействию с ней языка C#. Дело в том, что это весьма глубокая тема с множеством темных закоулков. Многие проблемы настолько малоизвестны, что вы вряд ли столкнетесь с ними, а большинство разработчиков редко пользуются даже самыми простыми сценариями. Я уверен, что DLR заслуживает написания отдельной книги, но надеюсь, что предоставил здесь достаточно деталей, чтобы 99% разработчиков на C# смогли продолжать свою работу, не нуждаясь в какой-то дополнительной информации. Если вас интересует больше сведений, то документация на веб-сайте DLR будет хорошей отправной точкой (`http://mng.bz/0M6A`).

Если вы никогда не применяете тип `dynamic`, можете в принципе проигнорировать динамическую типизацию. Именно так я рекомендую поступать при написании подавляющего большинства кода — в частности, я бы не использовал ее в качестве средства, позволяющего избежать создания подходящих интерфейсов, базовых классов и т.д. С другой стороны, когда динамическая типизация *действительно* нужна, я применяю ее настолько умеренно, насколько это возможно. Не занимайте позицию, которую можно сформулировать так: "поскольку я использую `dynamic` в данном методе, то могу также сделать динамическим *все остальное*".

Я вовсе не хочу, чтобы все это звучало слишком негативно. Если вы окажетесь в ситуации, когда динамическая типизация полезна, уверен, что вы будете признательны проектировщикам за ее наличие в C# 4. Даже если вам она никогда не понадобится в производственном коде, я рекомендую опробовать ее на практике чисто ради интереса — лично мне нравится копаться в ней. Вы можете также счесть среду DLR удобной даже без применения динамической типизации; в большинстве примеров работы с кодом Python в этой главе возможности динамической типизации не использовались, но применялась среда DLR для запуска сценария Python, содержащего конфигурационные данные.

В этой и предыдущей главах были раскрыты все новые средства C# 4. Следующая глава посвящена версии C# 5, которая сосредоточена на единственном средстве даже *уже*, чем версия C# 4 была сфокусирована на динамической типизации. На самом деле *все* вертится вокруг асинхронности...

Часть V

C# 5: упрощение асинхронности

Описать версию C# 5 довольно просто: она имеет в точности одно крупное (асинхронные функции) и два мелких средства.

Глава 15 полностью посвящена асинхронности. Цель средства асинхронных функций (часто для краткости называемого `async/await`) — сделать асинхронное программирование легким... или, по крайней мере, более легким, чем было ранее. Оно не пытается устранить сложность, присущую асинхронности; вы по-прежнему должны продумывать последствия от завершения операций в непредвиденном порядке или щелчка пользователем на кнопке до полного завершения предшествующей операции, но исчезает много побочной сложности. Это позволяет увидеть лес за деревьями и строить надежные, читабельные решения, имея дело только с характерной для них сложностью.

В прошлом асинхронный код часто становился запутанным подобно спагетти, с логическим путем выполнения, который перепрыгивал с метода на метод по мере того, как один асинхронный вызов завершался, а другой начинался. Благодаря асинхронным функциям, можно писать код, который *выглядит* синхронным, используя знакомые управляющие структуры, такие как циклы и блоки `try/catch/finally`, но иметь асинхронный поток выполнения, который инициируется с помощью нового ключевого слова `await`. По моему опыту разница в читабельности является просто ошеломительной. Мы будем рассматривать эту тему довольно глубоко, не только в плане поведения языка, но также в том, как это реализовано компилятором Microsoft C#.

В главе 16 будут раскрыты два оставшихся средства: небольшое изменение раздражающего поведения цикла `foreach`, о котором упоминалось в главе 5, и несколько новых атрибутов, которые работают с необязательными параметрами из C# 4, чтобы сделать возможным автоматическое предоставление компилятором номера строки, имени члена, а также имени исходного файла для заданной порции кода. Затем я завершу это издание книги своими привычными заключительными размышлениями.

Может показаться, что это не выглядит чем-то серьезным, учитывая тот факт, что я умышленно преуменьшил значимость средств, раскрываемых в главе 16. Однако заблуждаться не следует; асинхронные функции — это действительно важное событие, особенно если вы занимаетесь написанием приложений Windows Store, использующих WinRT. Предлагаемый WinRT интерфейс API построен вокруг асинхронности, чтобы справиться с проблемой неотзывчивых пользовательских интерфейсов. Без асинхронных функций с ним было бы довольно трудно работать. Имея в своем распоряжении средства C# 5, по-прежнему приходится думать, но зато код может получиться настолько ясным, что трудно было даже вообразить подобное применительно к асинхронному коду. Итак, вместо того, чтобы продолжать описывать, до чего же все прекрасно, давайте перейдем непосредственно к ознакомлению со средством.

Глава 15

Асинхронность с помощью async/await

В этой главе...

- Фундаментальные цели асинхронности
- Написание асинхронных методов и делегатов
- Трансформации, выполняемые компилятором для асинхронности
- Асинхронный шаблон, основанный на задачах
- Асинхронность в WinRT

На протяжении многих лет асинхронность попортила немало крови разработчикам. Она была известна как полезный способ избежать связывания потока на время ожидания завершения произвольной задачи, но еще и как истинное наказание с учетом того, что требовалось для ее корректной реализации.

Даже внутри платформы .NET Framework (которая по-прежнему относительно нова в рамках глобальной картины) мы имеем три разных модели, с помощью которых можно попытаться упростить ситуацию.

- Подход `BeginFoo()/EndFoo()` из .NET 1.x, в котором для передачи результатов применяются `IAsyncResult` и `AsyncCallback`.
- Асинхронный шаблон, основанный на событиях, из .NET 2.0, реализованный в `BackgroundWorker` и `WebClient`.
- Библиотека параллельных задач (Task Parallel Library — TPL), которая появилась в .NET 4 и была расширена в .NET 4.5.

Несмотря на великолепное проектное решение в целом, написание надежного и читабельного кода с помощью TPL было трудным. Хотя поддержка параллелизма была хороша, существовали аспекты общей асинхронности, которые намного лучше зафиксированы в языке, а не полностью в библиотеках.

Подход async/await перевернет ваш мир

Вводный список тем может навести на мысль, что материал этой главы довольно скучен. Список точен, однако он не в состоянии передать волнение, которое меня переполняет, когда я думаю об этом средстве. К этому времени я работал с подходом `async/await` около двух лет, но он все еще превращает меня в мечтательного школьника. Я твердо уверен, что он сделает для асинхронности то, что технология LINQ сделала для обработки данных в версии C# 3, исключая лишь тот факт, что асинхронность была намного более сложной проблемой. Чтобы достичь нужного эффекта, читайте эту главу с соответствующим умственным настроем. Надеюсь, что по ходу дела я смогу заразить вас своим энтузиазмом относительно данного средства.

Главное средство C# 5 построено на основе TPL, поэтому можно писать код, выглядящий подобно синхронному, который использует асинхронность, когда это уместно. Ушло в прошлое переплетение обратных вызовов, подписок на события и фрагментированной обработки событий; взамен асинхронный код ясно выражает свои намерения, причем в форме, которая основана на структурах, уже знакомых разработчикам. Новая языковая конструкция `await` позволяет "дожидаться" асинхронной операции. Такое "ожидание" выглядит очень похожим на обычное обращение к блокированию в том, что остальной код не будет продолжать работу до тех пор, пока операция не завершится, но это управляется так, что текущий выполняющийся поток не блокируется. Не переживайте, если данное утверждение звучит совершенно противоречиво — по мере чтения этой главы все станет ясно.

В версии 4.5 платформа .NET Framework охватывает асинхронность повсеместно. Она предлагает асинхронные версии для огромного количества операций и следует новому документированному асинхронному шаблону, основанному на задачах, чтобы обеспечить согласованность среди множества API-интерфейсов. Кроме того, новая платформа Windows Runtime[1], используемая для создания приложений Windows Store в Windows 8, принудительно применяет асинхронность для всех длительно выполняющихся (или потенциально длительно выполняющихся) операций. Словом, будущее за асинхронностью, и было бы глупо не воспользоваться преимуществами новых средств языка при попытке справиться с дополнительной сложностью. На случай, если версия .NET 4.5 не применяется, в Microsoft создали пакет NuGet (`Microsoft.Bcl.Async`), который позволяет использовать эти новые средства для целевых платформ .NET 4, Silverlight 4, Silverlight 5, Windows Phone 7.5 или Windows Phone 8.

Просто ради ясности, язык C# не являются всезнающим, чтобы догадываться, где вы можете пожелать выполнить операции параллельно или асинхронно. Компилятор достаточно интеллектуален, но он вовсе не пытается устранить *неотъемлемую* сложность асинхронного выполнения. Вы по-прежнему должны тщательно все обдумывать, но красота C# 5 в том, что необходимости в написании всего утомительного и запутанного стереотипного кода больше нет. Не отвлекаясь на малозначительные детали, требуемые для превращения кода в асинхронный, все усилия можно сосредоточить на трудных элементах.

Одно предупреждение: эта тема довольно сложна. Она обладает неудачным свойством становиться чрезвычайно важной (на самом деле с годами даже начинающим раз-

[1] Она широко известна как WinRT; ее не следует путать с Windows RT, которая представляет собой версию операционной системы Windows 8, функционирующую на процессорах ARM.

работчикам нужно будет иметь хотя бы мимолетное представление о ней), но также достаточно запутанной, чтобы поначалу голова пошла кругом. Как и в остальных главах книги, я не буду избегать этой сложности — мы рассмотрим, что происходит, с достаточным числом деталей.

Вполне возможно, что я немного поморочу вам голову, но надеюсь, что со временем все станет на свои места. Если вещи начнут казаться близкими к бреду, не переживайте — дело вовсе не в вас; трудность восприятия является вполне нормальной реакцией. Хорошая новость в том, что при *использовании* C# 5 смысл всего этого лежит на поверхности. Положение дел усложняется, только если пытаться думать о том, что действительно происходит "за кулисами". Конечно, позже именно этим мы и займемся, а также взглянем, каким образом применять данное средство максимально эффективно.

Давайте приступим.

15.1. Введение в асинхронные функции

До сих пор я утверждал, что версия C# 5 делает реализацию асинхронности проще, но предоставлял лишь крошечное описание задействованных средств. Для начала исправим ситуацию, а затем рассмотрим пример.

В C# 5 появилась концепция *асинхронной функции*. Это всегда либо метод, либо анонимная функция[2], которая объявлена с модификатором `async` и может включать выражения `await`. Такие выражения `await` являются точками, где вещи становятся интересными с точки зрения языка: если значение, ожидаемое выражением, пока еще не доступно, асинхронная функция немедленно произведет возврат, и будет затем продолжена с места, в котором управление ее покинуло (в соответствующем потоке), когда значение станет доступным. Естественный поток вида "не выполнять следующий оператор до тех пор, пока не завершится выполнение текущего" по-прежнему поддерживается, но без блокирования.

Позже я разобью это неясное описание на более конкретные термины и поведение, но прежде чем это обретет хоть какой-нибудь смысл, необходимо ознакомиться с примером.

15.1.1. Первые встречи с асинхронностью

Давайте начнем с чего-то очень простого, но демонстрирующего асинхронность практическим путем. Мы часто ругаем задержку сети, приводящую к возникновению пауз в работе реальных приложений, однако задержка упрощает иллюстрацию того, насколько важна асинхронность. Взгляните на код в листинге 15.1.

Листинг 15.1. Отображение длины страницы асинхронным образом

```
class AsyncForm : Form
{
  Label label;
  Button button;

  public AsyncForm()
  {
    label = new Label { Location = new Point(10, 20),
```

[2] Просто в качестве напоминания: анонимная функция — это либо лямбда-выражение, либо анонимный метод.

```
                    Text = "Length" };
    button = new Button { Location = new Point(10, 50),
                          Text = "Click" };
    button.Click += DisplayWebSiteLength;      <-- ❶ Привязка обработчика событий
    AutoSize = true;
    Controls.Add(label);
    Controls.Add(button);
  }
  async void DisplayWebSiteLength(object sender, EventArgs e)
  {
    label.Text = "Fetching...";
    using (HttpClient client = new HttpClient())
    {
      string text =                            <-- ❷ Начало извлечения страницы
          await client.GetStringAsync("http://csharpindepth.com");
      label.Text = text.Length.ToString();     <-- ❸ Обновление пользовательского
    }                                              интерфейса
  }
}
...
Application.Run(new AsyncForm());
```

В первой части листинга 15.1 просто создается пользовательский интерфейс и привязывается обработчик события для кнопки в прямолинейной манере ❶. Интересен здесь метод `DisplayWebSiteLength()`. По щелчку на кнопке извлекается текст домашней страницы книги ❷, а метка обновляется для отражения длины HTML-разметки в символах ❸. Экземпляр `HttpClient` соответствующим образом освобождается независимо от того, успешно или неудачно прошла операция — обо всем этом слишком легко забыть, когда пишется похожий асинхронный код на C# 4.

Освобождение задач

Я внимательно отношусь к освобождению экземпляра `HttpClient`, когда завершаю пользоваться им, но не освобождаю задачу, возвращаемую методом `GetStringAsync()`, хотя класс `Task` реализует интерфейс `IDisposable`. К счастью, в целом освобождать задачи на самом деле не придется. Подноготная этого кое в чем запутана, но Стивен Тауб объясняет ее в отдельной статье своего блога: `http://mng.bz/E6L3`.

Я мог бы написать пример программы меньшего размера в виде консольного приложения, но надеюсь, что листинг 15.1 окажется более убедительной демонстрацией. В частности, если вы удалите контекстные ключевые слова `async` и `await`, измените `HttpClient` на `WebClient`, а также измените вызов `GetStringAsync()` на `DownloadString()`, то код по-прежнему будет компилироваться и работать, но на период извлечения содержимого страницы пользовательский интерфейс замораживается[3].

[3] В определенном смысле тип `HttpClient` является "обновленным и улучшенным" типом `WebClient` — он считается предпочтительным API-интерфейсом для работы с HTTP в .NET 4.5 и последующих версиях. Тип `HttpClient` содержит только асинхронные операции. При написании приложения Windows Store даже нет возможности воспользоваться `WebClient`.

Если вы запустите асинхронную версию (в идеальном случае через медленное сетевое подключение), то увидите, что пользовательский интерфейс реагирует — во время извлечения веб-страницы окно можно перемещать по экрану.

Большинству разработчиков знакомы два золотых правила работы с многопоточностью при написании приложений Windows Forms.

- Не выполнять действие, занимающее длительное время, в потоке пользовательского интерфейса.
- Не обращаться к элементам управления из потоков, отличных от потока пользовательского интерфейса.

Эти правила проще заявить, чем следовать им. В качестве упражнения может понадобиться опробовать несколько разных способов создания кода, подобного приведенному в листинге 15.1, без применения новых средств C# 5. Для такого предельно простого примера не слишком плохим решением будет использование основанного на событиях метода `WebClient.DownloadStringAsync()`. Однако когда в игру вступает более сложное управление потоком (обработка ошибок, ожидание завершения загрузки данных с нескольких страниц и т.д.), "унаследованный" код быстро становится трудным в сопровождении, тогда как код C# 5 может быть модифицирован вполне естественным образом. В настоящий момент метод `DisplayWebSiteLength()` выглядит чем-то магическим: вы знаете, что он делает необходимую работу, но не имеете представления, как он это делает. Давайте немного отвлечемся, отложив по-настоящему сложные детали на будущее.

15.1.2. Разбор первого примера

Мы начнем с небольшого расширения метода — выделим вызов метода `HttpClient.GetStringAsync()` из выражения `await`, чтобы подчеркнуть задействованные типы:

```
async void DisplayWebSiteLength(object sender, EventArgs e)
{
  label.Text = "Fetching...";
  using (HttpClient client = new HttpClient())
  {
    Task<string> task =
      client.GetStringAsync("http://csharpindepth.com");
    string text = await task;
    label.Text = text.Length.ToString();
  }
}
```

Обратите внимание, что типом `task` является `Task<string>`, но типом выражения `await task` по-прежнему остается просто `string`. В этом смысле выражение `await` выполняет "распаковывающую" операцию — во всяком случае, когда ожидаемое значение имеет тип `Task<TResult>`. (Как вы увидите, ожидать можно также и другие типы, но `Task<TResult>` представляет собой хорошую отправную точку.) Данный аспект `await` не связан напрямую с асинхронностью, но делает жизнь проще.

Главное назначение `await` — избежать блокирования, пока ожидается завершение длительно выполняющихся операций. Вас может интересовать, как все это работает в конкретных терминах многопоточности. Значение `label.Text` устанавливается и в начале, и в конце метода, поэтому разумно предположить, что оба эти оператора выполняются в потоке пользовательского интерфейса, и во время ожидания загрузки веб-страницы поток пользовательского интерфейса определенно *не* блокируется.

Трюк заключается в том, что метод на самом деле производит возврат, как только достигнуто выражение `await`. Вплоть до этой точки он выполняется синхронно в потоке пользовательского интерфейса, как это бы делал любой другой обработчик событий. Если вы поместите точку останова в первую строку и попадете на нее в отладчике, то увидите в трассировке стека, что кнопка занята инициированием своего события `Click`, включая вызов метод `Button.OnClick()`. По достижении `await` код проверяет, доступен ли уже результат, и если нет (что случится почти наверняка), он планирует *продолжение*, которое должно выполняться, когда веб-операция завершится. В данном примере продолжение выполняет оставшуюся часть метода, фактически перепрыгивая на конец выражения `await` и возвращаясь обратно в поток пользовательского интерфейса, что и требуется для манипулирования пользовательским интерфейсом.

Продолжения

Продолжение — это, в сущности, обратный вызов, который должен быть выполнен, когда асинхронная операция (или на самом деле любой экземпляр `Task`) завершается. В асинхронном методе продолжение поддерживает управляющее состояние метода; в точности как замыкание поддерживает свою среду в терминах переменных, продолжение запоминает, где оно должно быть активизировано, поэтому появляется возможность продолжить выполнение с того места, где оно было приостановлено. В классе `Task` предусмотрен метод, предназначенный специально для присоединения продолжений: `Task.ContinueWith()`.

Если вы затем поместите точку останова в код *после* выражения `await`, то увидите, что трассировка стека больше не включает вызов метода `Button.OnClick()` (предполагается, что выражение `await` нуждается в планировании продолжения). Этот метод давно завершил свое выполнение. Стек вызовов теперь содержит обычный цикл событий Windows Forms, с несколькими уровнями инфраструктуры асинхронности поверх него. Содержимое стека вызовов будет очень похоже на ситуацию с вызовом `Control.Invoke()` из фонового потока с целью соответствующего обновления пользовательского интерфейса, но все это было сделано автоматически. На первых порах такое кардинальное изменение стека вызовов прямо на глазах может лишать присутствия духа, но оно совершенно необходимо, чтобы сделать асинхронность эффективной.

В случае если вам интересно, все это обрабатывается за счет создания компилятором сложного конечного автомата. Это деталь реализации, которую поучительно исследовать, чтобы получить лучшее понимание происходящего, но сначала понадобится более конкретное описание того, чего мы пытаемся достигнуть, и что на самом деле предписывает язык.

15.2. Обдумывание асинхронности

Если вы попросите разработчика описать асинхронное выполнение, то с высокой вероятностью он начнет говорить о многопоточности. Хотя многопоточность является важной частью *типового* использования асинхронности, на самом деле она не является обязательной для асинхронного выполнения. Чтобы в полной мере оценить, как работает средство асинхронности в C# 5, лучше отбросить любые мысли о многопоточности и вернуться к основам.

15.2.1. Фундаментальные основы асинхронного выполнения

Асинхронность проникает в самое сердце модели выполнения, с которой хорошо знакомы разработчики на C#. Взгляните на следующий простой код:

```
Console.WriteLine("First");
Console.WriteLine("Second");
```

Вы ожидаете, что первый вызов завершится, а затем начнется второй вызов. Выполнение по порядку перетекает с одного оператора на другой. Однако модель асинхронного выполнения не работает подобным образом. Взамен все сводится к *продолжениям*. Когда вы начинаете делать что-то, вы сообщаете операции о том, что должно произойти по ее завершению. Возможно, вы уже слышали (или применяли) термин *обратный вызов* для описания той же идеи, но он имеет более широкий смысл, нежели то, что подразумевается здесь. В контексте асинхронности я использую этот термин для ссылки на обратные вызовы, которые сохраняют управляющее состояние программы, а не на произвольные обратные вызовы, предназначенные для других целей, такие как обработчики событий в графическом пользовательском интерфейсе.

Продолжения естественным образом представлены в .NET как делегаты, и они обычно являются действиями, которые получают результаты асинхронной операции. Именно по этой причине для применения асинхронных методов класса `WebClient` в версиях, предшествующих C# 5, необходимо был привязывать разнообразные события, чтобы указать, какой код должен выполняться в случае успеха, неудачи и т.д. Беда в том, что создание всех этих делегатов для сложной последовательности шагов, в конечном итоге, становится весьма затруднительным, даже с учетом преимуществ, предлагаемых лямбда-выражениями. Ситуация становится даже хуже, когда предпринимается попытка удостовериться в корректности обработки ошибок. (В удачные дни я могу быть достаточно уверенным в том, что пути во вручную написанном асинхронном коде, обрабатывающие успешный случай, являются корректными. Однако обычно я менее уверен в том, правильно ли будет реагировать код в сбойной ситуации.)

По существу все, что делает `await` в C# — это предлагает компилятору построить продолжение. Однако идея, выражаемая настолько просто, оказывает значительные последствия на читабельность и здравомыслие разработчиков.

Мое ранее описание асинхронности было идеализированным. Реальное положение дел в асинхронном шаблоне, основанном на задачах, несколько отличается. Вместо передачи продолжения асинхронной операции эта асинхронная операция начинается и возвращает маркер, который можно использовать для предоставления продолжения в более позднее время. Он представляет выполняющуюся операцию, которая может быть завершена перед возвратом в вызывающий код или по-прежнему продолжать свое выполнение. Затем данный маркер применяется везде, где нужно выразить следующую идею: "я не могу двигаться дальше до тех пор, пока эта операция не будет завершена". Обычно маркер имеет форму `Task` или `Task<TResult>`, но это не обязательно. Поток выполнения внутри асинхронного метода в C# 5 обычно следует описанным ниже шагам.

1. Выполнить определенную работу.
2. Начать асинхронную операцию и запомнить возвращенный ею маркер.
3. Возможно, выполнить какую-то дополнительную работу. (Часто делать что-либо еще, пока асинхронная операция не завершена, не удается, и в таком случае данный шаг будет пустым.)
4. Ожидать завершения асинхронной операции (через маркер).

5. Выполнить дополнительную работу.
6. Завершиться.

Если вас не интересует в точности, что означает шаг "ожидать", то вы могли бы сделать все это в версии C# 4. Если вас устраивает *блокировка* вплоть до завершения асинхронной операции, то маркер обычно предоставит какой-нибудь способ сделать это. В случае `Task` можно просто вызвать метод `Wait()`. Хотя в данный момент вы занимаете ценный ресурс (поток), не выполняя полезной работы. Это похоже на ситуацию, когда вы заказываете по телефону доставку пиццы и затем ждете у дверей, пока она не прибудет. На самом деле желательно было бы заняться чем-то другим, игнорируя пиццу вплоть до факта ее доставки. Здесь на выручку приходит `await`.

Когда вы "ожидаете" асинхронную операцию, то в действительности утверждаете следующее: "Пройдено столько, сколько было возможно на данный момент. Выполнение продолжится после завершения операции". Но если вы не собираетесь блокировать поток, тогда что можно предпринять? Все очень просто: можно прямо сейчас осуществить возврат. Вы будете продолжать асинхронным образом самого себя. И если необходимо, чтобы *вызывающий код* узнал, когда ваш асинхронный метод завершился, вы передаете ему обратно маркер, который при желании может блокироваться или (более вероятно) использоваться с другим продолжением. Зачастую у вас будет целый стек асинхронных методов, вызывающих друг друга — это почти как входить в "асинхронный режим" для какого-то раздела кода. В языке нет ничего такого, с помощью чего можно было заявить, что все *должно* делаться именно таким путем, но тот факт, что код, который *потребляет* асинхронные операции, также *ведет себя* как асинхронная операция, определенно поощряет это.

Контексты синхронизации

Ранее я упоминал, что одно из золотых правил кода пользовательского интерфейса заключается в том, чтобы не допускать обновление пользовательского интерфейса из другого потока. В примере с проверкой длины веб-страницы (листинг 15.1) необходимо было удостовериться, что код после выражения `await` выполняется в потоке пользовательского интерфейса. Асинхронные функции возвращаются в правильный поток с применением `SynchronizationContext` — класса, который существует, начиная с версии .NET 2.0, и используется другими компонентами, такими как `BackgroundWorker`. Класс `SynchronizationContext` обобщает идею выполнения делегата "в подходящем потоке"; его методы отправки сообщений `Post()` (асинхронный) и `Send()` (синхронный) подобны методам `Control.BeginInvoke()` и `Control.Invoke()` в Windows Forms.

Разные среды выполнения применяют разные контексты; например, один контекст может позволять любому потоку из пула потоков выполнять данное ему действие. Помимо контекста синхронизации существует дополнительная контекстная информация, но если вы начали интересоваться, каким образом асинхронные методы удается выполнять именно там, где это необходимо, имейте в виду данную врезку.

За более подробными сведениями о `SynchronizationContext` обратитесь к статье Стивена Клири в журнале MSDN Magazine, посвященной этой теме (`http://mng.bz/5cDw`). В частности, уделите статье особое внимание, если вы являетесь разработчиком на ASP.NET: неосмотрительные разработчики могут легко попасть в ловушку, устроенную контекстом ASP.NET, и создавать взаимоблокировки внутри кода, который выглядит вполне нормальным.

Вооружившись предоставленной теорией, давайте внимательнее посмотрим на конкретные детали асинхронных методов. Асинхронные анонимные функции вписываются в ту же самую мысленную модель, но с их помощью намного проще говорить об асинхронных методах.

15.2.2. Моделирование асинхронных методов

Я нахожу очень удобным думать об асинхронных методах так, как показано на рис. 15.1.

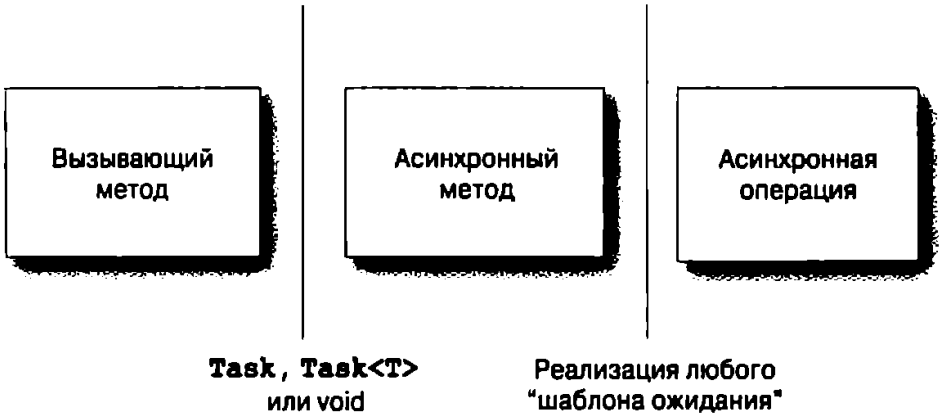

Рис. 15.1. Асинхронная модель

Здесь имеются три блока кода (методы) и две границы (возвращаемые типы методов). В качестве очень простого примера можно привести следующий код:

```
static async Task<int> GetPageLengthAsync(string url)
{
  using (HttpClient client = new HttpClient())
  {
    Task<string> fetchTextTask = client.GetStringAsync(url);
    int length = (await fetchTextTask).Length;
    return length;
  }
}
static void PrintPageLength()
{
  Task<int> lengthTask =
    GetPageLengthAsync("http://csharpindepth.com");
  Console.WriteLine(lengthTask.Result);
}
```

Пять частей на рис. 15.1 соответствуют показанному коду, как описано ниже.

- Вызывающий метод — это `PrintPageLength()`.
- Асинхронный метод — это `GetPageLengthAsync()`.
- Асинхронная операция — это `HttpClient.GetStringAsync()`.
- Граница между вызывающим методом и асинхронным методом — это `Task<int>`.
- Граница между асинхронным методом и асинхронной операцией — это `Task<string>`.

Главным образом мы заинтересованы в самом асинхронном методе, но другие методы были включены для того, чтобы продемонстрировать взаимодействие между ними. В частности, вы определенно должны знать допустимые типы на границах между методами.

В остальном материале главы будут неоднократно встречаться ссылки на эти блоки и границы, поэтому во время чтения держите рис. 15.1 в уме.

15.3. Синтаксис и семантика

Мы окончательно готовы к ознакомлению с написанием асинхронных методов и анализу их поведения. Здесь много чего придется раскрыть, т.к. смесь "что может быть сделано" и "что происходит, когда это делается" довольно обширна.

Новых порций синтаксиса только две: `async` является модификатором, используемым при объявлении асинхронного метода, а выражения `await` потребляют асинхронные операции. Однако отслеживание передачи информации между различными частями программы очень быстро становится затруднительным, особенно когда приходится обдумывать, что происходит, когда что-то идет не так. Я попытался отделить разные аспекты друг от друга, но ваш код будет иметь дело со всеми ими сразу. Если вы обнаружите, что во время чтения этого раздела задаетесь вопросом "Но как насчет?..", продолжайте читать — скорее всего, вы довольно скоро получите ответ на свой вопрос.

Давайте начнем с самого объявления метода — это простейшая часть.

15.3.1. Объявление асинхронного метода

При объявлении асинхронного метода применяется точно такой же синтаксис, как для любого другого метода, но с добавлением контекстного ключевого слова `async`. Это ключевое слово может находиться где угодно перед возвращаемым типом. Все показанные ниже объявления являются допустимыми:

```
public static async Task<int> FooAsync() { ... }
public async static Task<int> FooAsync() { ... }
async public Task<int> FooAsync() { ... }
public async virtual Task<int> FooAsync() { ... }
```

Лично я предпочитаю помещать модификатор `async` непосредственно перед возвращаемым типом, но нет никаких причин не принять другое соглашение по этому поводу. Как всегда, обсудите его с остальными членами команды и постарайтесь обеспечить согласованность в рамках одной кодовой базы.

А теперь я раскрою "страшный" секрет, связанный с контекстным ключевым словом `async`: *проектировщики языка на самом деле не должны были вообще включать его в язык*. Точно так же, как компилятор входит в своего рода "режим итераторного блока", когда в методе используется оператор `yield return` или `yield break` с подходящим возвращаемым типом, компилятор *мог бы* просто обнаружить присутствие `await` внутри метода и применить данный факт для перехода в "асинхронный режим". Однако лично я рад, что ключевое слово `async` является обязательным, т.к. оно способствует более легкому *чтению* кода, написанного с использованием асинхронных методов. Оно немедленно порождает предположения, поэтому вы активно ищете выражения `await` — и можете не менее активно искать также любые блокирующие вызовы, которые *должны* быть преобразованы в асинхронные вызовы и выражения `await`.

Тем не менее, тот факт, что модификатор `async` не имеет представления[4] в сгенерированном коде, важен. С точки зрения вызывающего метода это просто нормальный метод, возможно, возвращающий объект задачи. Вы можете изменить существующий метод (с подходящей сигнатурой) для применения `async` или двинуться в другом направлении — это изменение является совместимым в терминах как исходного кода, так и двоичной сборки.

15.3.2. Возвращаемые типы асинхронных методов

Взаимодействие между вызывающим методом и асинхронным методом осуществляется посредством возвращаемого значения. Асинхронные функции ограничены следующими возвращаемыми типами:

- `void`
- `Task`
- `Task<TResult>` (для определенного типа `TResult`, который сам может быть параметром типа)

Типы `Task` и `Task<TResult>` в .NET 4 представляют операцию, которая может быть еще не завершена; тип `Task<TResult>` является производным от `Task`. Разница между ними двумя по существу состоит в том, что `Task<TResult>` представляет операцию, которая возвращает значение типа `T`, в то время как `Task` не нуждается в генерации какого-либо результата. Тем не менее, возвращать объект `Task` все равно удобно, т.к. он позволяет вызывающему методу присоединять к возвращенной задаче собственные продолжения, обнаруживать ситуации, когда задача отказала или завершилась, и т.д. В определенном смысле о `Task` можно думать как о типе, подобном `Task<void>`, если бы такое было допустимым.

Возможность возвращения `void` из асинхронного метода предназначена для достижения совместимости с обработчиками событий. Например, взгляните на следующий обработчик щелчка на кнопке пользовательского интерфейса:

```
private async void LoadStockPrice(object sender, EventArgs e)
{
  string ticker = tickerInput.Text;
  decimal price = await stockPriceService.FetchPriceAsync(ticker);
  priceDisplay.Text = price.ToString("c");
}
```

Хотя метод асинхронный, вызывающий код (метод `OnClick()` кнопки или любой другой фрагмент кода инфраструктуры, генерирующий событие щелчка) на самом деле это совершенно не заботит. Ему даже не нужно знать, когда обработка события действительно *завершена*, т.е. когда курс акций загружен и пользовательский интерфейс обновляется. Он просто вызывает заданный обработчик событий. Тот факт, что сгенерированный компилятором код будет создавать конечный автомат, который присоединяет продолжение ко всему, возвращаемому методом `FetchPriceAsync()`, в сущности, является деталью реализации. Предыдущий метод можно подписать на событие, как если он был любым другим обработчиком событий:

```
loadStockPriceButton.Click += LoadStockPrice;
```

---

[4] Хорошо, отчасти. Как вы увидите далее, на самом деле есть примененный атрибут, но он не является частью сигнатуры метода, и его вполне можно проигнорировать. В действительности он используется для того, чтобы помочь инструментам в идентификации, где пошел "реальный" код.

В конце концов (и да, я добиваюсь этого умышленно), с точки зрения вызывающего кода это *просто нормальный метод*. Он имеет возвращаемый тип `void` и параметры типа `object` и `EventArgs`, которые делают его подходящим в качестве действия для экземпляра делегата `EventHandler`.

Подписка на событие — это практически *единственный* случай, когда я рекомендую возвращать `void` из асинхронного метода. Во всех остальных ситуациях вы не обязаны возвращать какое-то конкретное значение, но лучше объявить, что метод возвращает `Task`. Это позволит вызывающему коду ожидать завершения операции, обнаруживать отказы и т.д.

К сигнатуре асинхронного метода применимо одно дополнительное ограничение: ни один из параметров не может использовать модификаторы `out` или `ref`. Это имеет смысл, т.к. указанные модификаторы предназначены для обмена информацией с вызывающим кодом; поскольку некоторые асинхронные методы могут быть еще не выполненными к моменту возврата управления в вызывающий код, значение параметра, передаваемого по ссылке, могло оказаться неустановленным. На самом деле, ситуация могла стать даже более странной: представьте себе передачу локальной переменной для аргумента `ref` — асинхронный метод мог бы пытаться установить эту переменную после того, как вызывающий метод был уже завершен. В таких действиях мало смысла, поэтому компилятор запрещает их.

После объявления метода можно приступать к написанию его тела и ожиданию других асинхронных операций.

15.3.3. Шаблон ожидания

Асинхронный метод может содержать в основном все, что допускается в обычном методе C#, плюс выражения `await`. Можно применять все виды управления потоком — циклы, исключения, операторы `using`, словом, все что угодно. Код будет вести себя нормальным образом. Единственные интересные вещи касаются того, что делают выражения `await`, и как распространяются возвращаемые выражения.

Ограничения, накладываемые на выражения `await`

Подобно `yield return`, существуют ограничения относительно того, где можно использовать выражения `await`. Их нельзя применять в блоках `catch` или `finally`, в неасинхронных анонимных функциях[5], в теле оператора `lock` или в небезопасном коде.

Эти ограничения направлены на обеспечение безопасности — особенно ограничения, касающиеся оператора `lock`. Если вы когда-либо обнаружите, что *стремитесь* удержать блокировку на время завершения асинхронной операции, то должны перепроектировать код. Не следует обходить ограничение компилятора, вручную вызывая методы `Monitor.TryEnter()` и `Monitor.Exit()` с блоком `try/finally` — измените код так, чтобы не нуждаться в блокировке при выполнении операции. Если это совершенно затруднительно в вашей ситуации, подумайте об использовании класса `SemaphoreSlim` с его методом `WaitAsync()`.

[5] Это лямбда-выражения и анонимные методы, которые не объявлены с `async` — т.е. любое объявление анонимной функции, которое было бы допустимым в C# 4. Асинхронные анонимные функции рассматриваются в разделе 15.4.

Выражение `await` является очень простым — оно просто ожидает выполнения другого выражения. Но, конечно же, имеются ограничения на то, для чего можно выполнять ожидание. Просто в качестве напоминания: речь идет о второй границе на рис. 15.1, т.е. о том, каким образом асинхронный метод взаимодействует с другой асинхронной операцией. Неформально применять `await` можно только к тому, что описывает асинхронную операцию, другими словами, к тому, что предоставляет следующие средства:

- сообщение о том, завершена ли уже операция;
- присоединение продолжения, если операция еще не завершена;
- получение результата, который может быть возвращаемым значением, но как минимум признаком успеха или отказа.

Вы можете ожидать, что все перечисленное будет выражено через интерфейсы, но это (главным образом) не так. Задействован только один интерфейс, который покрывает часть "присоединения продолжения". Но даже несмотря на такую простоту, вам почти никогда не придется иметь с ним дело напрямую. Интерфейс находится в пространстве имен `System.Runtime.CompilerServices` и выглядит следующим образом:

```
// Реальный интерфейс в пространстве имен System.Runtime.CompilerServices
public interface INotifyCompletion
{
   void OnCompleted(Action continuation);
}
```

Основная масса работы выражается через шаблоны, немного напоминая цикл `foreach` и запросы LINQ. Чтобы прояснить форму шаблона, я кратко представлю его, *как если бы* были задействованы интерфейсы, которых на самом деле нет. Сразу после этого я раскрою действительное положение дел. Давайте взглянем на воображаемые интерфейсы:

```
// Внимание: в действительности все это не существует!
// Воображаемые интерфейсы для асинхронных операций, возвращающих значения
public interface IAwaitable<T>
{
   IAwaiter<T> GetAwaiter();
}
public interface IAwaiter<T> : INotifyCompletion
{
   bool IsCompleted { get; }
   T GetResult();
   // Унаследован от INotifyCompletion
   // void OnCompleted(Action continuation);
}
// Воображаемые интерфейсы для асинхронных операций void
public interface IAwaitable
{
   IAwaiter GetAwaiter();
}
public interface IAwaiter : INotifyCompletion
{
   bool IsCompleted { get; }
   void GetResult();
   // Унаследован от INotifyCompletion
   // void OnCompleted(Action continuation);
}
```

Возможно, это напомнит вам интерфейсы IEnumerable<T> и IEnumerator<T>. Для прохождения по коллекции в цикле foreach компилятор генерирует код, который сначала вызывает метод GetEnumerator(), а затем использует метод MoveNext() и свойство Current. Аналогично, всякий раз, когда в асинхронных методах записывается выражение await, компилятор будет генерировать код, который сначала обращается к методу GetAwaiter(), после чего применяет члены объекта ожидания, чтобы соответствующим образом дождаться результата.

Компилятор C# *требует*, чтобы объект ожидания реализовывал интерфейс INotifyCompletion. В первую очередь это объясняется причинами производительности; некоторые предварительные версии платформы вообще не содержали этого интерфейса.

Все остальные члены проверяются компилятором просто по сигнатуре. Важно отметить, что сам метод GetAwaiter() не обязан быть нормальным методом экземпляра. Он может быть расширяющим методом для любого типа, с которым необходимо использовать выражение await. Члены IsCompleted и GetResult() должны быть реальными членами любого типа, возвращаемого методом GetAwaiter(), но они не обязаны быть открытыми — они просто должны быть доступными коду, содержащему выражение await.

В предшествующем тексте было описано, что именно требуется от выражения для применения в качестве цели ключевого слова await, однако само выражение в целом также имеет интересный тип: если метод GetResult() возвращает void, то общий тип выражения await отсутствует — выражение await должно быть отдельным оператором. Иначе общий тип совпадает с возвращаемым типом GetResult().

Например, Task<TResult>.GetAwaiter() возвращает объект TaskAwaiter <TResult>, который имеет метод GetResult(), возвращающий TResult. (Надеюсь, это не является сюрпризом.) Правило, касающееся выражения await, позволяет написать следующий код:

```
using (var client = new HttpClient())
{
  Task<string> task = client.GetStringAsync(...);
  string result = await task;
}
```

Сравните это со статическим методом Task.Yield(), который возвращает структуру YieldAwaitable. В свою очередь, она имеет метод GetAwaiter(), возвращающий структуру YieldAwaitable.YieldAwaiter, которая содержит метод GetResult(), возвращающий void. Это означает, что допускается только такое использование:

```
await Task.Yield();
```

Либо, если вы действительно хотите разделить оператор — хотя и странно, но вполне может быть:

```
YieldAwaitable yielder = Task.Yield();
await yielder;
```

Здесь выражение await не возвращает значение какого-нибудь вида, поэтому его нельзя присвоить переменной, передать в качестве аргумент метода или сделать что-то еще, что обычно допускается для выражений, классифицируемых как значения.

Важно отметить, что поскольку Task и Task<TResult> реализуют шаблон ожидания, один асинхронный метод можно вызывать из другого и т.д.:

```
public async Task<int> FooAsync()
{
   string bar = await BarAsync();
   // Очевидно, что код обычно будет более сложным...
   return bar.Length;
}
public async Task<string> BarAsync()
{
   // Асинхронный код, который мог бы вызывать другие асинхронные методы...
}
```

Такая возможность объединения асинхронных операций является одним из аспектов средства асинхронности, который действительно делает его блестящим. Перейдя в асинхронный режим, очень легко в нем оставаться и писать код, который выполняется естественным образом.

Но я несколько забегаю вперед. Я описал то, что компилятору нужно для обеспечения возможности ожидания, но не то, что он в действительности делает.

15.3.4. Поток выражений `await`

Один из самых необычных аспектов средства асинхронности в C# 5 заключается в том, что `await` может быть одновременно интуитивно понятным и совершенно запутанным. Если не слишком много думать о нем, все выглядит просто. Если вы всего лишь примете, что средство асинхронности делает именно то, что вам необходимо, не определяя в точности, что действительно требуется, то, скорее всего, все будет хорошо – во всяком случае, до тех пор, пока что-то пойдет не так.

Как только вы начнете выяснять, что в точности должно происходить для достижения такого результата, все станет немного сложнее. Учитывая, что вы читаете эту книгу, я предполагаю, что вас интересует именно такой уровень подробностей. В долгосрочной перспективе я обещаю, что вы станете применять `await` более уверенно и эффективно.

Тем не менее, я призываю вас постараться развить в себе способность читать асинхронный код на двух разных уровнях в зависимости от контекста: когда нет *необходимости* думать об отдельных шагах, перечисленных здесь, не обращайте на них особого внимания. Читайте код почти так, как если бы он был синхронным, просто отмечая места, где код асинхронно ожидает завершения той или иной операции. Впоследствии, столкнувшись с проблемой, когда код ведет себя неожиданным образом, можете переключиться на более экспертный режим, выясняя, где какие потоки задействованы, и как выглядит стек вызовов в любой момент времени. (Я не обещаю, что это будет просто, однако понимание самого механизма, во всяком случае, сделает его более осуществимым.)

Разворачивание сложных выражений

Давайте начнем с некоторого упрощения. Иногда `await` используется с результатом вызова метода, а изредка со свойством, примерно так[6]:

```
string pageText = await new HttpClient().GetStringAsync(url);
```

Код выглядит так, как если бы конструкция `await` могла изменить смысл целого выражения. Правда в том, что `await` просто действует на одиночном значении.

---

[6] Этот пример слегка надуман, т.к. обычно для `HttpClient` предусматривается оператор `using`, но я надеюсь, что один раз вы простите меня за отсутствие освобождения ресурсов.

Предыдущая строка кода эквивалентна следующим двум:

```
Task<string> task = new HttpClient().GetStringAsync(url);
string pageText = await task;
```

Аналогично, результат выражения await может применяться в качестве аргумента метода или внутри другого выражения. Опять-таки, помогает возможность отделения части, специфичной для await, от всего остального.

Предположим, что есть два метода, GetHourlyRateAsync() и GetHoursWorkedAsync(), возвращающие Task<decimal> и Task<int>, соответственно. Может быть написан следующий сложный оператор:

```
AddPayment(await employee.GetHourlyRateAsync() *
           await timeSheet.GetHoursWorkedAsync(employee.Id));
```

Здесь применяются нормальные правила оценки выражений C#, и левый операнд операции * должен быть полностью оценен до оценки правого операнда, поэтому предыдущий оператор может быть развернут так, как показано ниже:

```
Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
decimal hourlyRate = await hourlyRateTask;
Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
int hoursWorked = await hoursWorkedTask;
AddPayment(hourlyRate * hoursWorked);
```

Такое разворачивание выявляет потенциальную неэффективность исходного оператора — данный код можно было бы распараллелить, запустив *обе* задачи (вызовом обоих методов Get...Async()) перед ожиданием одной из них.

В данный момент более полезный результат состоит в том, что нужно изучить поведение await только в контексте *значения*. Даже если это значение первоначально поступило из вызова метода, в целях обсуждения асинхронности вызов метода можно проигнорировать.

Видимое поведение

Когда выполнение достигает выражения await, существуют две возможности — либо ожидаемая асинхронная операция уже завершилась, либо нет.

Если операция уже завершилась, поток выполнения на самом деле прост — он продолжается. Если операция дала отказ, для представления которого предусмотрено исключение, оно и будет сгенерировано. В противном случае из операции получается результат — например, путем извлечения string из Task<string> — и происходит переход на следующую часть программы. Все это делается безо всякого переключения контекста потоков или присоединения продолжений к чему-либо.

В первую очередь вас может удивить, почему операция, которая завершается немедленно, представлена с помощью асинхронности. Это немного похоже на вызов метода Count() с последовательностью в LINQ: в общем случае может потребоваться проход по всем элементам в последовательности, но в некоторых ситуациях (например, когда последовательностью оказывается List<T>) доступна простая оптимизация. Удобно иметь единственную абстракцию, которая покрывает оба сценария, но платя за это накладными расходами во время выполнения. В качестве реального примера в случае асинхронного API-интерфейса подумайте об асинхронном чтении из потока данных, связанного с файлом на диске. Все данные, которые нужно прочитать, уже извлечены из диска в память, возможно как часть предыдущего вызова ReadAsync(), поэтому имеет смысл использовать их немедленно, не проходя весь механизм асинхронности.

Более интересный сценарий возникает, когда асинхронная операция по-прежнему выполняется. В этом случае метод асинхронно ожидает завершения операции и затем продолжается в соответствующем контексте. Такое "асинхронное ожидание" на самом деле означает, что метод вообще не выполняется. К асинхронной операции присоединяется продолжение и производится возврат из метода. Асинхронная операция отвечает за обеспечение возобновления работы метода в правильном потоке — обычно либо в потоке из пула (где не играет роли, какой поток используется), либо в потоке пользовательского интерфейса, где это имеет смысл.

С точки зрения разработчика это *выглядит*, как будто метод приостанавливается на время до завершения асинхронной операции. Компилятор гарантирует, что все локальные переменные, применяемые внутри метода, имеют те же самые значения, которые они хранили перед продолжением — почти так же, как это делается для итераторных блоков.

Я попытался отразить это на рис. 15.2, хотя классические блок-схемы в действительности не предназначены для представления асинхронного поведения.

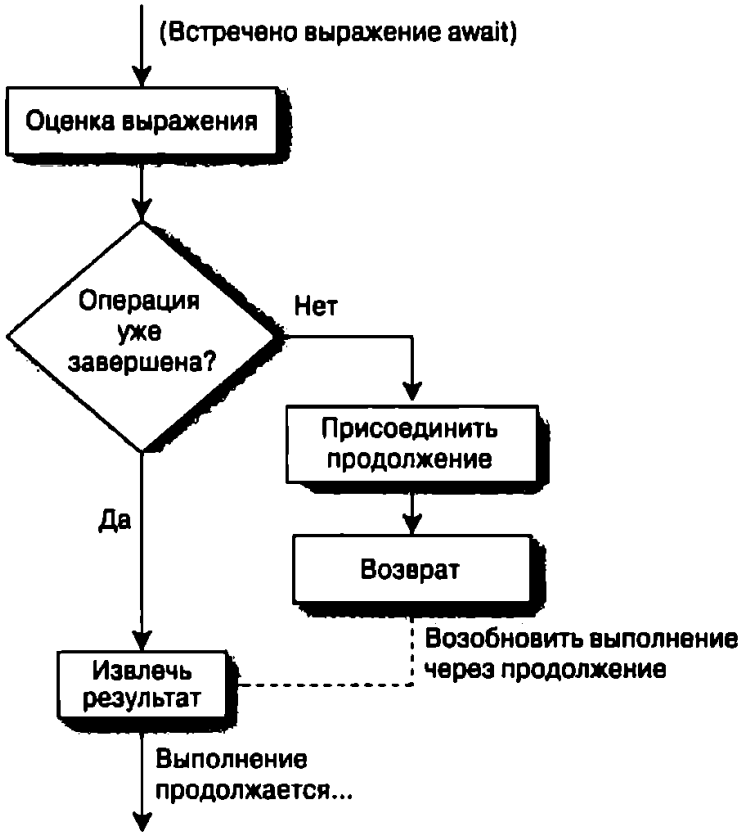

Рис. 15.2. Видимая пользователю модель обработки выражения `await`

Пунктирную линию можно считать как еще одну линию, входящую в верхнюю часть блок-схемы в качестве альтернативы. Обратите внимание, что я предполагаю наличие результата у цели выражения `await`. Если вы осуществляете ожидание обычного объекта `Task` или чего-то подобного, то "извлечь результат" на самом деле означает "проверить успешность завершения операции".

Полезно остановиться и обдумать смысл понятия "возврат" из асинхронного метода. Здесь снова есть две возможности.

- Это первое выражение `await`, которое фактически необходимо ожидать, поэтому где-то в стеке по-прежнему присутствует первоначальный вызывающий метод. (Вспомните, что до тех пор, пока не возникнет действительная потребность в ожидании, метод выполняется синхронно.)

- Вы уже ожидаете что-то еще, поэтому находитесь внутри продолжения, которое было вызвано чем-нибудь. Стек вызовов почти наверняка будет значительно изменен по сравнению с тем, который вы видели, когда впервые зашли в метод.

В первом случае все обычно сводится к возвращению объекта `Task` или `Task<T>` вызывающему методу. Очевидно, что действительный результат метода пока не доступен — даже если значение для возвращения отсутствует как таковое, вы не знаете, будет ли метод завершен без исключений. По этой причине задача, которая будет возвращена, должна быть незавершенной.

В последнем случае "что-нибудь", вызывающее асинхронный метод, зависит от контекста. Например, в пользовательском интерфейсе Windows Forms, если вы запустили свой асинхронный метод в потоке пользовательского интерфейса и не переключились преднамеренно из него, то метод целиком будет выполняться в потоке пользовательского интерфейса. В первой части метода вы будете находиться внутри какого-то обработчика события — того, который инициировал запуск асинхронного метода. Однако позже вызов был бы совершен практически прямо из конвейера обработки сообщений, как если бы использовался метод `Control.BeginInvoke(continuation)`. Здесь вызывающий код — будь он конвейером обработки сообщений Windows Forms, частью механизма пула потоков или чем-то еще — совершенно не беспокоится о вашей задаче.

Обратите внимание, что пока не достигнуто по-настоящему асинхронное выражение `await`, метод выполняется полностью синхронно. Вызов асинхронного метода *не* похож на инициирование новой задачи в отдельном потоке, и на вас возлагается ответственность за написание асинхронных методов так, чтобы они обеспечивали быстрый возврат. Надо сказать, что это зависит от контекста, в котором пишется код, но вы должны, как правило, избегать выполнения в асинхронном методе работы, требующей длительного времени. Вынесите ее в другой метод, для которого можно создать объект `Task`.

Использование членов шаблона ожидания

Теперь, когда вы понимаете, чего должны достичь, довольно легко увидеть, каким образом применяются члены шаблона ожидания. На рис. 15.3 на самом деле показана та же блок-схема, что и на рис. 15.2, но дополненная обращениями к этому шаблону.

Вас может интересовать, к чему все это беспокойство — по какой причине вообще полезно иметь языковую поддержку для таких целей? Тем не менее, присоединение продолжения сложнее, чем может казаться. В очень простых случаях, когда поток управления полностью линеен (выполняет определенную работу, ожидает чего-либо, выполняет дополнительную работу, ожидает чего-либо еще), довольно легко представить себе, что продолжение может выглядеть подобно лямбда-выражению, даже если это не особенно изящно. Однако если код содержит циклы или условия и его нужно уместить внутри одного метода, то все намного затрудняется. Именно здесь преимущества C# 5 становятся действительно явными. Хотя можно было бы утверждать, что компилятор всего лишь применяет синтаксический сахар, существует гигантская разница в читабельности между созданием продолжений вручную и поручением этой работы компилятору.

В отличие от простых преобразований, таких как автоматически реализуемые свойства, генерируемый компилятором код довольно сильно отличается от кода, который вы, возможно, написали бы вручную, даже когда сам асинхронный метод близок к тривиальному. Мы кратко рассмотрим эту трансформацию далее в главе, но вы уже частично видите "человека за ширмой" — вероятно, асинхронные методы теперь стали менее загадочными.

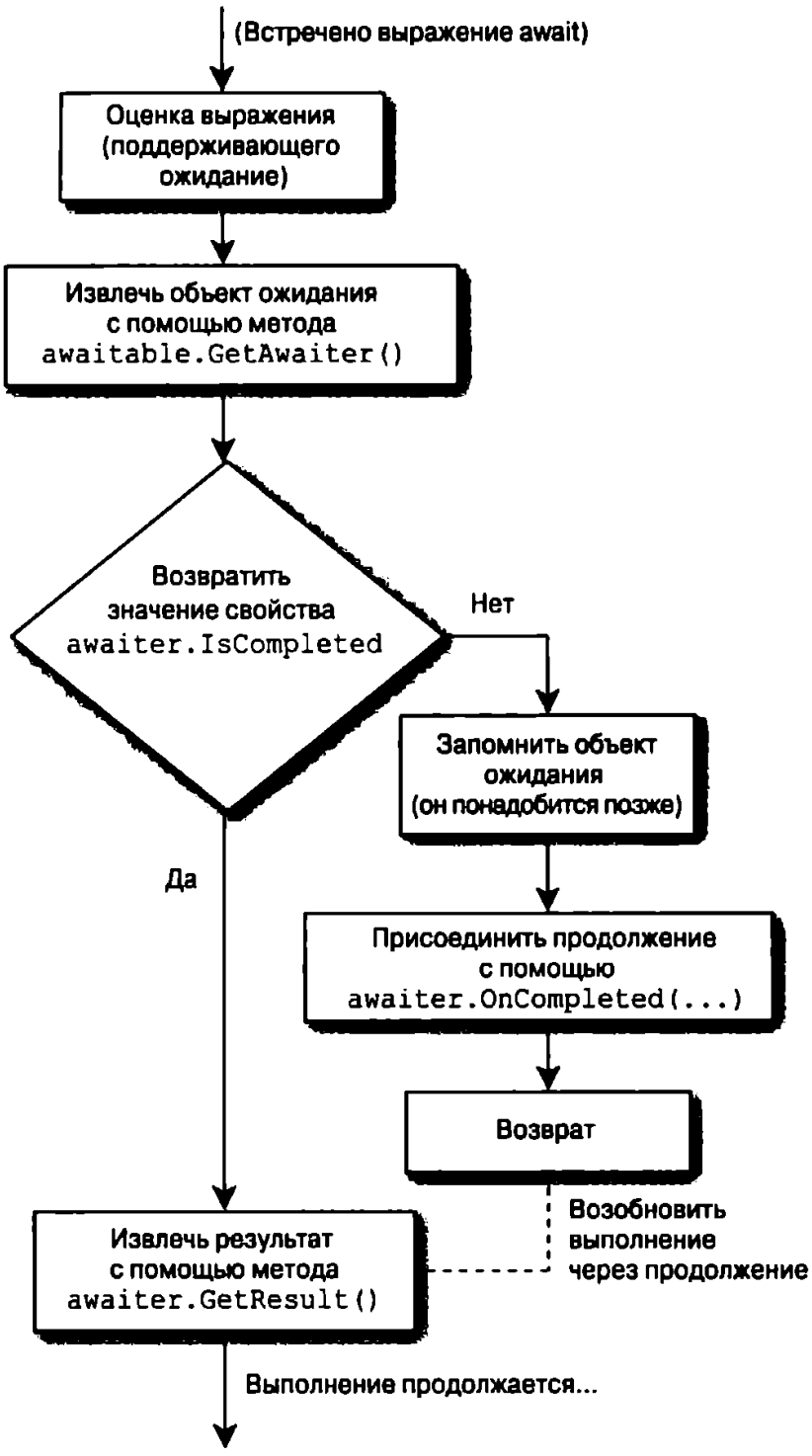

Рис. 15.3. Обработка выражения `await` посредством шаблона ожидания

Ранее были описаны ограничения, накладываемые на возвращаемые типы асинхронных методов, и вы видели, что выражение `await` распаковывает результаты асинхронной операции с помощью метода `GetResult()`. Тем не менее, пока еще не обсуждались вопросы связи между ними двумя и способ возвращения значений из асинхронных методов.

15.3.5. Возвращение значений из асинхронных методов

Вы уже видели пример, в котором возвращались данные, но давайте взглянем на него еще раз, сосредоточив внимание только на аспекте возвращения:

```
static async Task<int> GetPageLengthAsync(string url)
{
   using (HttpClient client = new HttpClient())
   {
      Task<string> fetchTextTask = client.GetStringAsync(url);
      int length = (await fetchTextTask).Length;
      return length;
   }
}
```

Типом `length` является `int`, но возвращаемый тип метода выглядит как `Task<int>`. Сгенерированный код сам позаботится об упаковке, поэтому вызывающий код получит объект `Task<int>`, который, в конечном счете, будет содержать значение, возвращенное методом, когда он завершится. Метод, возвращающий просто `Task`, подобен нормальному методу `void` — ему вообще не нужен оператор возврата, и любые *присутствующие* в нем операторы возврата должны иметь вид `return;`, без попытки указать какое-то значение. В любом случае задача будет также распространять любое исключение, сгенерированное внутри асинхронного метода.

Надеюсь, что теперь вы имеете неплохое представление о том, почему необходима упаковка: возврат в вызывающий код почти наверняка произойдет до столкновения с оператором `return`, и вызывающему коду нужно каким-то образом передавать информацию. Объект `Task<T>` (в вычислительной технике он часто называется *будущим*) является обещанием значения — либо исключения — в более позднее время.

Как и при нормальном потоке выполнения, если оператор `return` встречается внутри области действия блока `try`, который имеет связанный с ним блок `finally` (включая ситуацию, когда все это происходит из-за наличия оператора `using`), то выражение, используемое для вычисления возвращаемого значения, *оценивается* немедленно, но не становится результатом задачи до тех пор, пока не будет произведена вся необходимая очистка. Это означает, что если в блоке `finally` сгенерируется исключение, вы не получите задачу, которая одновременно успешна и неудачна — отказывает только все целиком.

Возвращаясь к моменту, упомянутому ранее, именно сочетание автоматической упаковки и распаковки обеспечивает настолько хорошую работу средства асинхронности в целом. Можете представлять это немного похожим на LINQ: вы пишете операции, выполняемые над каждым *элементом* последовательности в LINQ, а упаковка и распаковка означают возможность применения этих операций к последовательностям и получения последовательностей обратно. В асинхронном мире редко приходится явно обрабатывать задачу — вместо этого для потребления задачи к ней применяется `await`, а результирующая задача создается автоматически как часть механизма асинхронного метода.

15.3.6. Исключения

Разумеется, работа не всегда проходит гладко, и идиоматическим способом представления отказов в .NET являются исключения. Подобно возвращению значения в вызывающий код, обработка исключений требует дополнительной поддержки со стороны

языка. Когда нужно сгенерировать исключение, первоначальный вызывающий метод или асинхронный метод может не находиться в стеке; и в случае использования await для асинхронной операции, которая потерпела отказ, она может выполняться не в том же самом потоке, поэтому необходим какой-нибудь способ маршализации отказа. Если думать об отказе просто как о другой разновидности результата, то имеет смысл, чтобы исключения и возвращаемые значения обрабатывались похожим образом.

В этом разделе мы посмотрим, как исключения проходят обе границы, показанные на рис. 15.1. Давайте начнем с границы между асинхронным методом и асинхронной операцией, которую он ожидает.

Распаковка исключений при ожидании

Метод GetResult() объекта ожидания предназначен не только для извлечения возвращаемого значения, когда оно имеется, но также отвечает за передачу любых исключений из асинхронных операций обратно методу. Это делается не *настолько* просто, как может показаться, потому что в асинхронном мире единственный объект Task может представлять несколько операций, приводящих к множеству отказов. Хотя доступны и другие реализации шаблона ожидания, класс Task стоит рассмотреть специально, т.к. это тип, для которого, скорее всего, будет осуществляться ожидание в подавляющем большинстве случаев.

Класс Task обозначает исключения несколькими способами.

- Свойство Status задачи получает значение Faulted, когда асинхронная операция отказала (и свойство IsFaulted возвращает true).
- Свойство Exception возвращает объект AggregateException, содержащий все (потенциально множественные) исключения, которые привели к отказу задачи, или null, если задача не отказывала.
- Метод Wait() сгенерирует исключение AggregateException, если задача находится в состоянии отказа.
- Свойство Result класса Task<T> (который также ожидает завершения) аналогичным образом сгенерирует исключение AggregateException.

Кроме того, задачи поддерживают идею отмены посредством класса CancellationTokenSource и структуры CancellationToken.

Если задача отменена, метод Wait() и свойства Result сгенерируют исключение AggregateException, содержащее экземпляр OperationCanceledException (на деле экземпляр класса TaskCanceledException, который является производным от OperationCanceledException), но состоянием становится Canceled, а не Faulted.

Если во время ожидания задачи она откажет или будет отменена, сгенерируется исключение — но не типа AggregateException. Вместо этого для удобства (во многих ситуациях) генерируется первое исключение *внутри* AggregateException. В большинстве случаев именно это и необходимо. Характерной чертой средства асинхронности является разрешение писать асинхронный код, выглядящий очень похожим на синхронный, который пришлось бы писать в противном случае. Например, взгляните на следующий код:

```
async Task<string> FetchFirstSuccessfulAsync(IEnumerable<string> urls)
{
  // ЧТО ДЕЛАТЬ: проверить, что действительно получены адреса URL...
  foreach (string url in urls)
  {
```

```
      try
      {
         using (var client = new HttpClient())
         {
            return await client.GetStringAsync(url);
         }
      }
      catch (WebException exception)
      {
         // ЧТО ДЕЛАТЬ: занести в журнал, обновить статистику и т.д.
      }
   }
   throw new WebException("No URLs succeeded"); // URL не получены
}
```

На данный момент проигнорируйте тот факт, что все первоначальные исключения теряются, а все страницы извлекаются последовательно. Здесь я пытаюсь довести до вас то, что в коде необходимо ожидать перехвата `WebException` — предпринимается попытка выполнения асинхронной операции с помощью `HttpClient`, и если что-то пойдет не так, она сгенерирует исключение `WebException`. Вы хотите перехватить и обработать его, ведь так? Это определенно *выглядит* как то, что хотелось бы сделать, но, конечно же, метод `GetStringAsync()` не может генерировать исключение `WebException` для ошибки, такой как тайм-аут сервера, поскольку метод только *запускает* операцию. Все, что он может — это возвратить отказавшую задачу, которая *содержит* объект типа `WebException`. Если просто вызвать метод `Wait()` на этой задаче, сгенерируется исключение `AggregateException`, содержащее внутри себя `WebException`. Метод `GetResult()` объекта ожидания задачи вместо этого всего лишь генерирует исключение `WebException`, которое и перехватывается в показанном выше коде.

Разумеется, при этом *может* теряться информация. Если в отказавшей задаче имеется несколько исключений, то метод `GetResult()` может сгенерировать только одно из них и произвольно использует первое. Вы можете переписать предыдущий код так, чтобы в случае отказа вызывающий метод перехватывал исключение `AggregateException` и просматривал *все* причины отказа. Важно отметить, что определенные методы инфраструктуры, такие как `Task.WhenAll()`, делают именно это — `WhenAll()` является методом, который будет асинхронно ожидать завершения множества задач (указанных при его вызове). Если любая из задач откажет, результатом будет отказ, содержащий исключения из всех отказавших задач. Но если вы просто применяете `await` к задаче, возвращаемой методом `WhenAll()`, то будете видеть только первое исключение.

К счастью, для исправления ситуации слишком много работы не потребуется. Вы можете воспользоваться знанием шаблона ожидания и написать расширяющий метод для `Task<T>`, чтобы создать специальный поддерживающий ожидание класс, который будет генерировать внутри задачи первоначальное исключение `AggregateException`. Полный код выглядит довольно громоздким на печатной странице, но его сущность отражена в листинге 15.2.

Листинг 15.2. Повторная упаковка множества исключений, вызванных отказами задачи

```
public static AggregatedExceptionAwaitable WithAggregatedExceptions(
                                                this Task task)
{
   return new AggregatedExceptionAwaitable(task);
}
```

```
// В AggregatedExceptionAwaitable
public AggregatedExceptionAwaiter GetAwaiter()
{
   return new AggregatedExceptionAwaiter(task);
}
// В AggregatedExceptionAwaiter
public bool IsCompleted
{
    get { return task.GetAwaiter().IsCompleted; } ←❶ Делегаты для объекта ожидания задачи
}
public void OnCompleted(Action continuation)
{
   task.GetAwaiter().OnCompleted(continuation);
}
public void GetResult()
{
   task.Wait();     ←❷ Генерирует исключение AggregateException
}
```

Возможно, вам нужен похожий подход для `Task<T>`, с применением `return task.Result;` в `GetResult()` вместо вызова `Wait()`. Важный момент связан с тем, что вы делегируете нормальный объект ожидания задачи для функций, которые не хотите обрабатывать самостоятельно ❶, но обходите обычное поведение метода `GetResult()`, при котором происходит распаковка исключений. На время вызова `GetResult()` вам *известно*, что задача находится в заключительном состоянии, поэтому вызов метода `Wait()` ❷ завершится немедленно — это не нарушает асинхронность, которую вы пытаетесь достичь.

Для использования кода необходимо просто вызвать расширяющий метод и ожидать результата, как показано в листинге 15.3.

Листинг 15.3. Перехват множества исключений как `AggregateException`

```
private async static Task CatchMultipleExceptions()
{
   Task task1 = Task.Run(() => { throw new Exception("Message 1");
   });
   Task task2 = Task.Run(() => { throw new Exception("Message 2");
   });
   try
   {
      await Task.WhenAll(task1, task2).WithAggregatedExceptions();
   }
   catch (AggregateException e)
   {
      Console.WriteLine("Caught {0} exceptions: {1}",
         e.InnerExceptions.Count,
         string.Join(", ",
            e.InnerExceptions.Select(x => x.Message)));
   }
}
```

Метод `WithAggregatedExceptions()` возвращает ваш специальный объект, поддерживающий ожидание; с этого момента метод `GetAwaiter()`, в свою очередь, поставляет этот специальный объект ожидания, который поддерживает операции, требуемые компилятором C# для ожидания результата. Обратите внимание, что можно было бы объединить объект, поддерживающий ожидание, и объект ожидания — ничто не говорит о том, что они *должны* быть разных типов, — но их разделение выглядит немного яснее.

Ниже приведен вывод кода из листинга 15.3:

```
Caught 2 exceptions: Message 1, Message 2
```

Делать такое захочется относительно редко — достаточно редко для того, чтобы в Microsoft не предусмотрели какую-либо поддержку в рамках инфраструктуры, — однако полезно знать о такой возможности.

Это все, что необходимо знать об обработке исключений для второй границы, по крайней мере, сейчас. Но как насчет первой границы, между асинхронным методом и вызывающим методом?

Упаковка исключений при генерации

Вы можете быть в состоянии предсказать, что здесь будет: асинхронные методы *никогда* не генерируют исключения прямо при вызове. Взамен для асинхронных методов, возвращающих `Task` или `Task<T>`, любые исключения, которые были сгенерированы внутри метода (в том числе те, что распространились из других операций, синхронных или асинхронных), как вы уже видели, просто перемещаются в задачу. Если вызывающий метод ожидает задачу напрямую, будет получен объект `AggregateException`, содержащий исключение, но если вызывающий метод применяет вместо этого `await`, то исключение будет распаковано из задачи. Асинхронные методы, возвращающие `void`, будут сообщать об исключении исходному объекту `SynchronizationContext`, который обработает его в зависимости от контекста[7].

Если только вы действительно не заботитесь об упаковке и распаковке для отдельного контекста, то можете просто перехватывать исключение, сгенерированное вложенным асинхронным методом. В листинге 15.4 демонстрируется, насколько знакомым это выглядит.

Листинг 15.4. Обработка асинхронных исключений в знакомой манере

```
static async Task MainAsync()
{
  Task<string> task = ReadFileAsync("garbage file");
  try
  {
    string text = await task;
    Console.WriteLine("File contents: {0}", text);
  }
  catch (IOException e)
  {
    Console.WriteLine("Caught IOException: {0}", e.Message);
  }
}
```

❶ Начало асинхронного чтения

❷ Ожидание содержимого

❸ Обработка отказов ввода-вывода

7 Контексты обсуждаются более подробно в разделе 15.6.4.

```
static async Task<string> ReadFileAsync(string filename)
{
  using (var reader = File.OpenText(filename)) ←❸ Открытие файла синхронным образом
  {
    return await reader.ReadToEndAsync();
  }
}
```

Здесь вы получаете исключение IOException при вызове File.OpenText ❸ (если только не создали файл по имени garbage file), но вы увидите тот же самый путь выполнения, если откажет задача, возвращенная вызовом ReadToEndAsync(). Внутри метода MainAsync() обращение к ReadFileAsync() ❶ происходит *перед* входом в блок try, однако это случается только, когда осуществляется ожидание задачи ❷, исключение в которой видит вызывающий метод и перехватывает его блоком catch ❶, точно как в показанном ранее примере WebException. Опять-таки, код ведет себя очень знакомым образом, возможно, не считая времени возникновения исключения.

Подобно итераторным блокам, в терминах проверки допустимости аргументов это порождает некоторые сложности. Предположим, что после проверки на предмет значений null в параметрах, в асинхронном методе необходимо выполнить определенную работу. Если вы проверяете параметры подобно тому, как это делаете в нормальном синхронном коде, вызывающий метод не будет иметь никакого указания на наличие проблемы до тех пор, пока не будет организовано ожидание завершения задачи.

В листинге 15.5 приведен соответствующий пример.

Листинг 15.5. Нарушенная проверка допустимости аргументов в асинхронном методе

```
static async Task MainAsync()
{
  Task<int> task = ComputeLengthAsync(null); ← Умышленная передача недопустимого аргумента
  Console.WriteLine("Fetched the task");
                  // Извлечение задачи
  int length = await task;                   ←❶ Ожидание результата
  Console.WriteLine("Length: {0}", length);
}

static async Task<int> ComputeLengthAsync(string text)
{
  if (text == null)
  {
    throw new ArgumentNullException("text"); ←❷ Немедленная генерация исключения
  }
  await Task.Delay(500);                     ← Эмуляция реальной асинхронной работы
  return text.Length;
}
```

Код в листинге 15.5 выводит на консоль строку Fetched the task и затем отказывает. Фактически исключение было сгенерировано синхронно перед выводом на консоль, т.к. до проверки допустимости ❷ никаких выражений await не встречалось, но вызывающий код не увидит это исключение вплоть до ожидания возвращаемой задачи ❶. Как правило, для проверки допустимости аргументов, которая разумно может быть выполнена без длительных временных затрат (или вовлечения других асинхронных операций), было бы лучше сообщать об отказе незамедлительно, до того, как

система может столкнуться с дальнейшими проблемами. В качестве примера метод HttpClient.GetStringAsync() сгенерирует исключение немедленно после передачи ссылки null.

В C# 5 существуют два подхода к принудительной генерации исключения "энергичным образом". Первый из них предполагает отделение проверки допустимости аргументов от реализации тем же способом, как это делалось для итераторных блоков в листинге 6.9. В листинге 15.6 представлена исправленная версия метода ComputeLengthAsync().

Листинг 15.6. Отделение проверки допустимости аргументов от асинхронной реализации

```
static Task<int> ComputeLengthAsync(string text)
{
   if (text == null)
   {
      throw new ArgumentNullException("text");
   }
   return ComputeLengthAsyncImpl(text);
}

static async Task<int> ComputeLengthAsyncImpl(string text)
{
   await Task.Delay(500); // Эмуляция реальной асинхронной работы
   return text.Length;
}
```

В листинге 15.6 метод ComputeLengthAsync() сам по себе не является асинхронным с точки зрения языка — он не имеет модификатора async. Метод выполняется с использованием нормального потока выполнения, поэтому если проверка допустимости аргументов в начале метода генерирует исключение, то и метод на самом деле генерирует исключение. Однако если проверка проходит успешно, возвращаемой задачей будет та, что создана методом ComputeLengthAsyncImpl(), где происходит действительная работа. В более реалистичном сценарии метод ComputeLengthAsync(), возможно, был бы открытым или внутренним, а метод ComputeLengthAsyncImpl() должен быть закрытым, поскольку он *предполагает*, что проверка допустимости аргументов уже выполнена.

Другой подход к энергичной проверке допустимости предусматривает применение *асинхронных анонимных функций* — мы возвратимся к данному примеру в разделе 15.4, где будут рассматриваться такие функции.

Существует еще один вид исключения, которое внутри асинхронных методов обрабатывается по-другому — отмена.

Обработка отмены

Библиотека параллельных задач (Task Parallel Library — TPL) ввела в .NET 4 унифицированную модель отмены, использующую два типа: CancellationTokenSource и CancellationToken. Идея состоит в том, что можно создать экземпляр CancellationTokenSource, а затем запросить у него объект CancellationToken, который передается асинхронной операции. Отмену можно выполнять только на источнике (CancellationTokenSource), но это отражается на признаке (CancellationToken). (Это означает, что один и тот же признак можно передавать нескольким операциям и

не переживать, что они будут создавать помехи друг другу.) Существуют разнообразные способы применения признака отмены, но наиболее идиоматический подход предполагает вызов метода `ThrowIfCancellationRequested()`, который сгенерирует исключение `OperationCanceledException`, если сам признак был отменен, и ничего не делает в противном случае. То же самое исключение генерируется синхронными вызовами (такими как `Task.Wait()`), если они отменяются.

Взаимодействие этого с асинхронными методами описано в спецификации C# 5. Согласно спецификации, если в теле асинхронного метода генерируется *любое* исключение, то задача, возвращаемая методом, будет находиться в состоянии отказа. Точный смысл состояния "отказа" специфичен для реализации, но в действительности, когда асинхронный метод генерирует исключение `OperationCanceledException` (или производного от него типа, такого как `TaskCanceledException`), возвращаемая задача получает состояние `Canceled`. Код в листинге 15.7 доказывает, что это на самом деле исключение, которое приводит к отмене задачи.

Листинг 15.7. Создание отмененной задачи путем генерации исключения `OperationCanceledException`

```
static async Task ThrowCancellationException()
{
  throw new OperationCanceledException();
}
...
Task task = ThrowCancellationException();
Console.WriteLine(task.Status);
```

Код выводит на консоль слово `Canceled`, а не `Faulted`, как можно было предположить на основе спецификации. Когда вызывается `Wait()` на задаче или запрашивается ее результат (в случае `Task<T>`), исключение по-прежнему генерируется внутри `AggregateException`, так что вряд ли понадобится явно выполнять проверку на предмет отмены для каждой используемой задачи.

Отсутствие состязаний?

Вас может интересовать, присутствует ли условие состязаний в листинге 15.7. В конце концов, вы вызываете асинхронный метод и затем ожидаете, что состояние немедленно будет фиксированным. Если бы вы на самом деле запустили новый поток, то подобная опасность возникла бы, однако это не так. Вспомните, что до первого выражения `await` асинхронный метод выполняется синхронно — он по-прежнему реализует упаковку результатов и исключений, но тот факт, что это делается в асинхронном методе, *не* обязательно означает участие каких-то дополнительных потоков. Метод `ThrowCancellationException()` не содержит выражений `await`, поэтому метод целиком (все его строки кода) выполняется синхронно; вам известно, что к моменту возврата из него результат будет доступен. В действительности среда Visual Studio предупреждает о наличии асинхронного метода, для которого не предусмотрено выражений `await`, но в данном случае именно это и требуется.

Важно отметить, что если ожидается операция, которая отменена, то генерируется первоначальное исключение `OperationCanceledException`. Это означает, что если не предпринять никакого прямого действия, то задача, возвращаемая из асинхронного метода, также будет отменена — отмена распространяется естественным образом.

В листинге 15.8 приведен несколько более реалистичный пример отмены задачи.

Листинг 15.8. Отмена асинхронного метода через отмененную задержку

```
static async Task DelayFor30Seconds(CancellationToken token)
{
  Console.WriteLine("Waiting for 30 seconds...");
                  // Ожидание в течение 30 секунд
  await Task.Delay(TimeSpan.FromSeconds(30), token); ←❶ Запуск асинхронной задержки

}
...
var source = new CancellationTokenSource();
var task = DelayFor30Seconds(source.Token); ←❷ Вызов асинхронного метода
source.CancelAfter(TimeSpan.FromSeconds(1)); ←❸ Запрос признака отложенной отмены
Console.WriteLine("Initial status: {0}", task.Status);
                // Начальное состояние
try
{
  task.Wait();   ←❹ Ожидание завершения (синхронным образом)
}
catch (AggregateException e)
{
  Console.WriteLine("Caught {0}", e.InnerExceptions[0]);
                  // Перехвачено исключение
}
Console.WriteLine("Final status: {0}", task.Status); ←❺ Отображение состояния задачи
                // Финальное состояние
```

Первым делом запускается асинхронная операция ❷, которая просто вызывает метод `Task.Delay()` для эмуляции реальной работы ❶, но предоставляет признак отмены. На этот раз *действительно* задействовано несколько потоков: достигнув выражения `await`, управление возвращается в вызывающий метод, и в этой точке запрашивается признак отмены, которая произойдет через 1 секунду ❸. Затем организуется ожидание (синхронным образом) завершения задачи ❹, которая должна привести к исключению. Наконец, на консоль выводится состояние задачи ❺.

Вывод кода из листинга 15.8 выглядит следующим образом:

```
Waiting for 30 seconds...
Initial status: WaitingForActivation
Caught System.Threading.Tasks.TaskCanceledException: A task was canceled.
Final status: Canceled
```

Это можно представлять в терминах транзитивности отмены: если операция A ожидает операцию B, а операция B отменяется, то операцию A также следует считать отмененной.

Разумеется, вы не обязаны поступать подобным образом. Можно было бы перехватить исключение `OperationCanceledException` в методе `DelayFor30Seconds()` и продолжить делать что-то еще, немедленно выполнить возврат или даже сгенерировать другое исключение. Опять-таки, средство асинхронности — это не устранение контроля, а просто предоставление удобного стандартного поведения.

Осторожно запускайте этот код!

Код в листинге 15.8 работает нормально в консольном приложении либо в случае его вызова внутри потока из пула, но при его выполнении в потоке пользовательского интерфейса Windows Forms (или в любом другом однопоточном контексте синхронизации) он приведет к взаимоблокировке. Можете ли вы понять, почему? Подумайте о том, в какой поток попытается возвратиться метод `DelayFor30Seconds()`, когда отложенная задача завершится, и затем учтите, в каком потоке выполняется вызов `task.Wait()`. Это относительно простой пример, но заблуждение того же самого вида приводит к возникновению проблем у некоторых разработчиков, когда они впервые приступают к написанию асинхронного кода. По сути, проблема кроется в применении вызова метода `Wait()` или свойства `Result`, которые оба будут блокироваться до тех пор, пока не завершится связанная с ними задача. Речь идет не о том, чтобы отказаться от их использования, а о том, что во время их применения следует проявлять особую осторожность. Вместо этого вы должны обычно использовать `await` для асинхронного ожидания результатов задач.

Мы практически раскрыли поведение асинхронных методов. Вполне вероятно, что в большинстве случаев работы со средством асинхронности C# 5 будут задействованы асинхронные методы, но у них есть и ближайший родственник.

15.4. Асинхронные анонимные функции

Я не собираюсь тратить слишком много времени на рассмотрение асинхронных анонимных функций. Как и можно было предположить, они являются комбинацией двух средств: анонимных функций (лямбда-выражений и анонимных методов) и асинхронных функций (кода, который может содержать выражения `await`). В своей основе они позволяют создавать делегаты[8], которые представляют асинхронные операции. Все, что вы изучили до сих пор об асинхронных методах, в равной степени применимо и к асинхронным анонимным функциям. Асинхронная анонимная функция создается подобно любому анонимному методу или лямбда-выражению, но с модификатором `async` в своем начале. Вот пример:

```
Func<Task> lambda = async () => await Task.Delay(1000);
Func<Task<int>> anonMethod = async delegate()
{
  Console.WriteLine("Started");
  await Task.Delay(1000);
  Console.WriteLine("Finished");
  return 10;
};
```

Создаваемый делегат должен иметь сигнатуру с возвращаемым типом `void`, `Task` или `Task<T>`, точно так же, как асинхронный метод. Можно захватывать переменные, как в других анонимных функциях, и добавлять параметры. Кроме того, асинхронная операция не начинается до тех пор, пока делегат не будет вызван, а многочисленные обращения создают множество операций. Тем не менее, вызов делегата на самом деле *запускает* операцию; как и ранее, он не ожидает начала операции, и вы совершенно не *обязаны* использовать `await` с результатом асинхронной анонимной функции. В листинге 15.9 показан чуть более полный (хотя по-прежнему бесполезный) пример.

[8] На тот случай, если вы интересуетесь, асинхронные анонимные функции нельзя использовать для создания деревьев выражений.

Листинг 15.9. Создание и вызов асинхронной анонимной функции с помощью лямбда-выражения

```
Func<int, Task<int>> function = async x =>
{
  Console.WriteLine("Starting... x={0}", x);
  await Task.Delay(x * 1000);
  Console.WriteLine("Finished... x={0}", x);
  return x * 2;
};
Task<int> first = function(5);
Task<int> second = function(3);
Console.WriteLine("First result: {0}", first.Result);
Console.WriteLine("Second result: {0}", second.Result);
```

Значения здесь были намеренно выбраны такими, чтобы вторая операция завершилась быстрее первой. Но поскольку ожидание завершения первой операции производится до вывода на консоль результатов (с применением свойства `Result`, которое блокируется до тех пор, пока задача не будет завершена — и снова проявляйте осторожность, запуская этот код!), вывод выглядит так:

```
Starting... x=5
Starting... x=3
Finished... x=3
Finished... x=5
First result: 10
Second result: 6
```

Результаты получаются в точности теми же самыми, как если бы асинхронный код был помещен в асинхронный метод.

Лично я не нахожу асинхронные анонимные функции особо привлекательными, но они занимают свою нишу. Несмотря на то что они не могут включаться в выражения запросов LINQ, все равно есть случаи, когда может понадобиться выполнить трансформации данных асинхронным образом. Необходимо просто думать обо всем процессе немного по-другому.

Мы возвратимся к этой идее, когда будем обсуждать объединение, но прежде я хочу продемонстрировать одну область, где асинхронные анонимные функции действительно очень удобны. Ранее я обещал, что покажу другой способ выполнения энергичной проверки аргументов на предмет допустимости в начале асинхронного метода. Вы помните, что перед передачей главной операции значение параметра необходимо проверить на равенство `null`. В листинге 15.10 приведен единственный метод, который достигает тех же результатов, что и раздельная реализация в листинге 15.6.

Листинг 15.10. Проверка допустимости аргументов с использованием асинхронной анонимной функции

```
static Task<int> ComputeLengthAsync(string text)
{
  if (text == null)
  {
    throw new ArgumentNullException("text");   <-- 1 Проверка достоверности полностью синхронным образом
  }
```

```
    Func<Task<int>> func = async () =>      <-- ❷ Создание асинхронной функции
    {
      await Task.Delay(500);                <-- Эмуляция реальной асинхронной работы
      return text.Length;
    };
    return func();                          <-- ❸ Вызов асинхронной функции
}
```

Вы заметите, что это *не* асинхронный метод. Если бы он был таковым, то исключение оказалось бы упакованным в задачу вместо того, чтобы генерироваться непосредственно. Тем не менее, по-прежнему необходимо возвратить задачу, поэтому после проверки допустимости ❶ нужная работа просто помещается в асинхронную анонимную функцию ❷, вызывается делегат ❸ и возвращается результат.

Хотя код выглядит *несколько* неуклюжим, он яснее, чем версия с разделением метода на две части. Однако при этом следует учитывать влияние на производительность: такая дополнительная упаковка достается вовсе не бесплатно. В большинстве случаев все проходит нормально, но если вы пишете библиотеку, которая может быть задействована в критических к производительности работах, то перед принятием данного подхода должны оценить затраты в действительном сценарии.

Превосходство VB?

В версии 11 язык Visual Basic, в конце концов, получил поддержку итераторных блоков, которая существовала в C#, начиная с версии 2. Эта задержка позволила команде проектировщиков поразмышлять о недостатках C# — реализация Visual Basic допускает анонимные итераторные функции, разрешая такой же вид внутриметодного разделения между энергичным и отложенным выполнением. Аналогичная возможность в языке C# (пока еще) не появилась...

Итак, вы ознакомились практически со всеми аспектами средства асинхронности в C# 5. В оставшихся разделах главы мы погрузимся в некоторые детали реализации и затем посмотрим, как извлечь максимум из этого средства. Все это подразумевает, что вы вполне освоились с материалом, предоставленным к данному моменту — если вы еще не апробировали какой-нибудь код примера (или в идеальном случае собственный экспериментальный код), самое время сделать это сейчас. Даже если вы считаете, что хорошо понимаете теорию, полезно поработать с `async` и `await`, чтобы действительно *прочувствовать* программирование асинхронности в кое-где синхронном стиле.

15.5. Детали реализации: трансформация компилятора

Я очень ярко помню вечер 28 октября 2010 года. Андерс Хейлсберг представил шаблон `async/await` на конференции профессиональных разработчиков (Professional Developers Conference — PDC), и незадолго до начала его разговора была сделана доступной лавина загружаемого материала — включая черновик изменений в спецификации C#, версия CzTP (Community Technology Preview) компилятора C# 5 и набор слайдов презентации. Какое-то время я просматривал живую трансляцию и листал слайды, пока версия CTP устанавливалась. К моменту завершения доклада Андерса я написал асинхронный код и опробовал его в работе.

На протяжении следующих нескольких недель я начал разбирать все на части — просматривая точный код, генерируемый компилятором, пытаясь написать собственную упрощенную реализацию библиотеки, поступившей в составе CTP, и в целом рассматривая ее под самыми разными углами. По мере появления новых версий я выяснял, что изменилось, и постепенно обретал все лучшее и лучшее понимание происходящего внутри. Чем больше я узнавал, тем больше преисполнялся благодарностью за то, насколько большой объем рутинного кода компилятор благополучно генерировал за меня. Это было сродни изучению красивого цветка под микроскопом: красота по-прежнему существует, чтобы ею восхищались, но ее намного больше, чем можно заметить с первого взгляда.

Конечно, не все похожи на меня. Если вы всего лишь хотите опираться на уже описанное поведение и просто верите в то, что компилятор делает все правильно, то это *совершенно нормально*. Кроме того, вы ничего не потеряете, если пока что пропустите данный раздел и вернетесь к его изучению позже — в книге нет ничего, что было бы основано на его материале. Маловероятно, что вам придется отлаживать код до настолько низкого уровня, какой будет рассматриваться в этом разделе, но я уверен, что приведенные здесь сведения позволят глубже проникнуть в суть функционирования средства в целом. Шаблон ожидания определенно обретет больший смысл после того, как вы взглянете на сгенерированный код, и вы увидите ряд типов, которые инфраструктура предоставляет, чтобы оказать помощью компилятору. Несколько отпугивающих деталей присутствуют только по причине оптимизации; проектное решение и реализация очень тщательно настроены для устранения ненужных выделений в памяти кучи и переключений контекста, например.

В грубом приближении мы притворимся, что компилятор C# выполняет трансформацию из "кода C#, в котором применяется `async/await`" в "код C# без использования `async/await`". На деле внутренности компилятора нам не доступны, и более чем вероятно, что такая трансформация происходит на уровне, находящемся ниже C#. Безусловно, сгенерированный код IL не всегда может быть выражен с помощью неасинхронного кода C#, т.к. в C# накладываются более жесткие ограничения на управление потоком, чем в IL. Однако проще думать об этом, как о языке C#, в терминах того, как фрагменты кода сочетаются друг с другом.

Сгенерированный код напоминает луковицу, имея уровни сложности. Мы начнем снаружи и проложим путь в направлении сложных вещей — выражений `await` и танца объектов ожидания совместно с продолжениями.

15.5.1. Обзор сгенерированного кода

Вы все еще со мной? Тогда приступим. Я не буду здесь вдаваться во *все* подробности, в какие мог бы (это потребовало бы сотен страниц), но предоставлю вам достаточно данных, чтобы удалось понять общую структуру. Затем вы сможете либо почитать различные статьи, которые я писал в своем блоге на протяжении последней пары лет, чтобы ознакомиться с более запутанными деталями, или просто подготовить какой-то асинхронный код и декомпилировать его. Кроме того, я раскрою только асинхронные методы, что позволит продемонстрировать все интересные механизмы, и вам не придется иметь дело с дополнительным уровнем косвенности, который привносится асинхронными анонимными функциями.

Предупреждение отважному путешественнику: здесь будут детали реализации!

В этом разделе документируются некоторые аспекты реализации, которые найдены в компиляторе Microsoft C# 5, входящем в состав .NET 4.5. Если сравнивать версии CTP и бета-версию, то несколько деталей изменилось довольно существенно, и в будущем они вполне могут снова измениться. Но я считаю маловероятным слишком сильное изменение фундаментальных *идей*. Если вы освоите достаточный объем материала из этого раздела, чтобы понять, что никакой "магии" здесь нет, а есть лишь действительно искусный код, сгенерированный компилятором, то должны быть в состоянии принять любые будущие изменения.

Как уже несколько раз упоминалось, реализация (как в настоящем приближении, так и в коде, который генерируется реальным компилятором) по существу имеет форму *конечного автомата*. Компилятор генерирует закрытую вложенную структуру для представления асинхронного метода и должен также включить метод, имеющий ту же самую сигнатуру, как у объявленного вами. Я называю его *каркасным методом* — в нем не особенно много существенного, но от него зависит все остальное.

Каркасный метод должен создать конечный автомат, заставить его выполнить один шаг (где *шаг* — это любой код, выполняемый перед первым настоящим ожиданием выражения `await`) и затем возвратить задачу для представления хода работ конечного автомата. (Не забывайте, что до тех пор, пока не достигнуто первое выражение `await`, которое действительно необходимо ожидать, выполнение является синхронным.) На этом работа данного метода завершена — конечный автомат затем просматривает что-то другое, а продолжения, присоединенные к другим асинхронным операциям, просто сообщают конечному автомату о том, что нужно выполнить еще один шаг. Конечный автомат сигнализирует, когда достигает конца, предоставлением подходящего результата задаче, которая была возвращена ранее. На рис. 15.4 показана блок-схема описанных действий в лучшем виде, в каком я только смог ее представить.

Разумеется, шаг "Выполнить тело метода" начинается с начала метода только при первом вызове из каркасного метода. После этого каждый раз, когда вы попадаете в данный блок, это связано с продолжением, когда выполнение фактически продолжается с места, где оно было оставлено.

Теперь у нас есть две вещи, которые необходимо рассмотреть: каркасный метод и конечный автомат. На протяжении большей части этого раздела будет применяться один пример асинхронного метода, который представлен в листинге 15.11.

Листинг 15.11. Простой асинхронный метод для демонстрации трансформаций компилятора

```
static async Task<int> SumCharactersAsync(IEnumerable<char> text)
{
   int total = 0;
   foreach (char ch in text)
   {
      int unicode = ch;
      await Task.Delay(unicode);
      total += unicode;
   }
   await Task.Yield();
   return total;
}
```

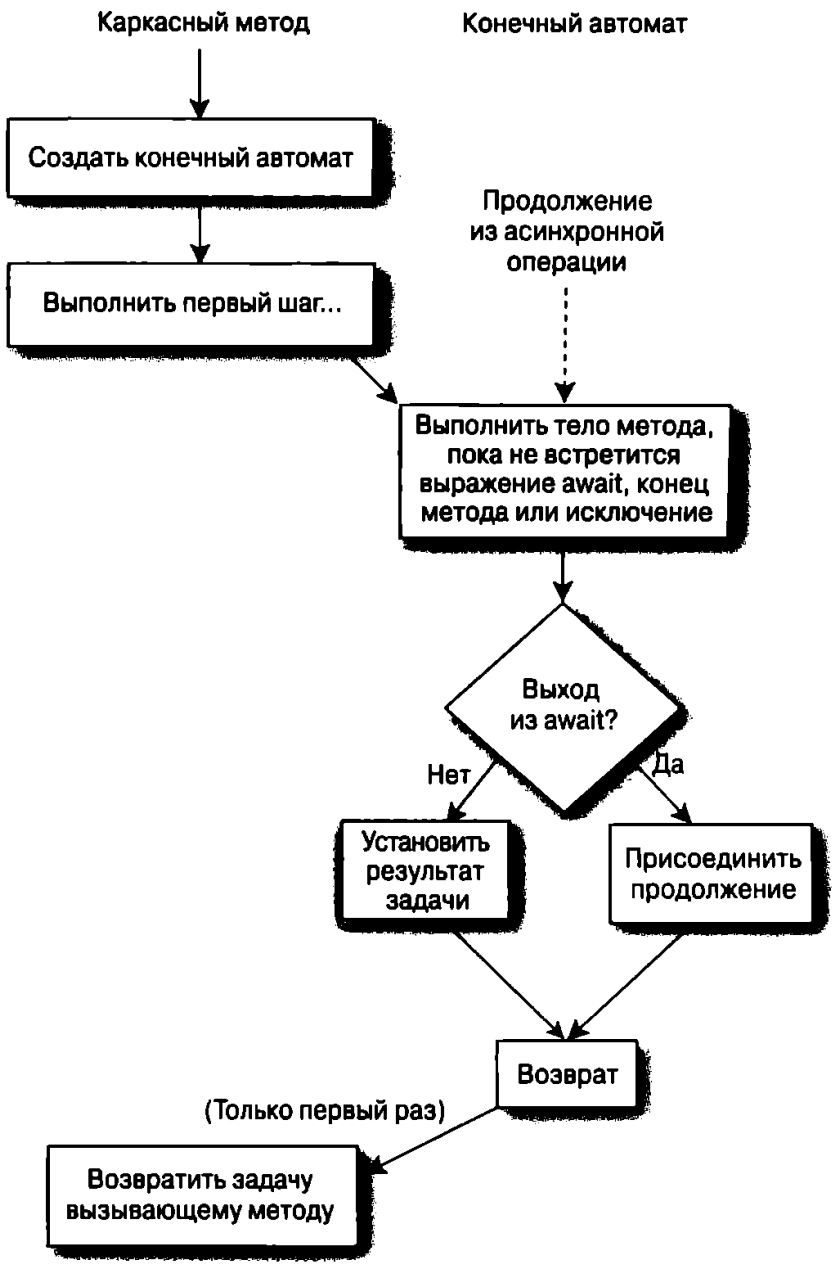

Рис. 15.4. Блок-схема работы сгенерированного кода

В листинге 15.11 не делается ничего *полезного*, но нас интересует только поток управления. Прежде чем начать, полезно отметить несколько моментов относительно этого метода.

- Метод имеет параметр (`text`).
- Он содержит цикл, внутрь которого фактически необходимо переходить, когда выполняется продолжение.
- Он имеет два выражения `await` разных типов: `Task.Delay()` возвращает объект `Task`, а `Task.Yield()` — объект `YieldAwaitable`.
- Он имеет очевидные локальные переменные (`total`, `ch` и `unicode`), которые понадобится отслеживать между вызовами.
- Он имеет неявную локальную переменную, созданную вызовом `text.GetEnumerator()`.
- В конце метод возвращает значение.

Первоначальная версия этого кода имеет text в качестве параметра типа string, но компилятору C# известно о проходе по строкам эффективным способом с использованием свойства Length и индексатора, которые делают декомпилированный код более сложным.

Я не буду представлять *полный* декомпилированный код, хотя он доступен в загружаемом коде. В следующих нескольких разделах мы рассмотрим ряд наиболее важных частей в нем. Если вы декомпилируете код самостоятельно, то не увидите в точности такой же код; я переименовал переменные и типы, чтобы их смысл стал более отчетливым, но на самом деле код аналогичен.

Давайте начнем с простейшей части — каркасного метода.

15.5.2. Структура каркасного метода

Несмотря на простоту кода в каркасном методе, он дает ряд подсказок об ответственностях конечного автомата. Каркасный метод, сгенерированный для листинга 15.11, выглядит следующим образом:

```
[DebuggerStepThrough]
[AsyncStateMachine(typeof(DemoStateMachine))]
static Task<int> SumCharactersAsync(IEnumerable<char> text)
{
   var machine = new DemoStateMachine();
   machine.text = text;
   machine.builder = AsyncTaskMethodBuilder<int>.Create();
   machine.state = -1;
   machine.builder.Start(ref machine);
   return machine.builder.Task;
}
```

Тип AsyncStateMachineAttribute — это всего лишь один из новых атрибутов, введенных для async. В действительности он полезен только для инструментов — вряд ли возникнет потребность в его самостоятельном применении, и вы не должны начинать декорировать им собственные методы.

Уже сейчас можно видеть три поля конечного автомата.

- Поле для параметра (text). Очевидно, что таких полей будет столько же, сколько параметров.
- Поле для AsyncTaskMethodBuilder<int>. Эта структура, в сущности, отвечает за связывание вместе конечного автомата и каркасного метода. Доступен ее необобщенный эквивалент для методов, возвращающих просто Task, и структура AsyncVoidMethodBuilder для методов, возвращающих void.
- Поле для state, которое хранит значения, начиная с -1. Начальным значением всегда является -1, а позже мы посмотрим, какой смысл имеют другие возможные значения.

Учитывая, что конечный автомат является структурой и AsyncTaskMethodBuilder<int> — также структура, вы пока сознательно не выполняете *какого-либо* выделения памяти в куче. Конечно, это вполне возможно для разнообразных вызовов, которые должны быть сделаны, но полезно отметить, что по возможности код пытается избежать их. Природа асинхронности означает, что если нужно действительно ожидать любые выражения await, понадобится много таких значений в куче, однако код обеспечивает упаковку только там, где она необходима. Все это представляет собой деталь

реализации, в точности так, как деталями реализации являются куча и стек, но чтобы сделать async практичным в максимально возможном числе ситуаций, команды разработчиков, задействованные в Microsoft, свели количество выделений памяти к абсолютному минимуму.

Интересен вызов machine.builder.Start(ref machine). Использование здесь передачи по ссылке позволяет избежать создания копии конечного автомата (и таким образом копии построителя) — это сделано в целях производительности и корректности. Компилятору действительно *хотелось* бы трактовать конечный автомат и построитель как классы, так что в коде обильно применяется модификатор ref. Чтобы использовать интерфейсы, различные методы принимают объект построителя (или ожидания) в качестве параметра обобщенного типа, который ограничен как реализующий интерфейс (такой как IAsyncStateMachine для конечного автомата). Это позволяет членам интерфейса быть вызванными без какой-либо упаковки. *Действие* метода описать просто — он заставляет конечный автомат выполнить первый шаг синхронным образом, осуществляя возврат, только когда метод либо завершен, либо достигнута точка, где необходимо ожидать асинхронную операцию.

После завершения первого шага каркасный метод запрашивает у построителя задачу для возвращения. Конечный автомат применяет построитель для установки результатов или исключений, когда вся работа сделана.

15.5.3. Структура конечного автомата

Общая структура конечного автомата довольно прямолинейна. Конечный автомат всегда реализует интерфейс IAsyncStateMachine (появившийся в .NET 4.5), используя явную реализацию интерфейсов. Он содержит только два метода, объявленные этим интерфейсом (MoveNext() и SetStateMachine()). Вдобавок имеется множество полей, часть из которых закрытые, а часть — открытые.

Например, ниже показано свернутое объявление конечного автомата для листинга 15.11:

```
[CompilerGenerated]
private struct DemoStateMachine : IAsyncStateMachine
{
  public IEnumerable<char> text;
  public IEnumerator<char> iterator;
  public char ch;
  public int total;
  public int unicode;
  private TaskAwaiter taskAwaiter;
  private YieldAwaitable.YieldAwaiter yieldAwaiter;
  public int state;
  public AsyncTaskMethodBuilder<int> builder;
  private object stack;
  void IAsyncStateMachine.MoveNext() { ... }
  [DebuggerHidden]
  void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine machine)
  { ... }
}
```

❶ Поля для параметров

❷ Поля для локальных переменных

❸ Поля для объектов ожидания

❹ Общая инфраструктура

В этом примере я разделил поля на различные разделы. Вы уже видели, что поле `text` ❶, представляющее первоначальный параметр, устанавливается каркасным методом, а также знакомы с полями `builder` и `state`, которые являются общей инфраструктурой, разделяемой всеми конечными автоматами.

Каждая локальная переменная также имеет собственное поле ❷, поскольку необходимо сохранять значения между вызовами метода `MoveNext()`. Иногда встречаются локальные переменные, которые применяются только между двумя отдельными выражениями `await` и *не нуждаются* в сохранении внутри полей, но согласно моему опыту текущая реализация в любом случае предусматривает для них поля. Помимо всего прочего, это улучшает процесс отладки, т.к. обычно ожидается, что локальные переменные не должны терять свои значения, даже если в дальнейшем коде они больше не используются.

Для каждого типа объекта ожидания, применяемого в асинхронном методе, предусмотрено одно поле, если это тип значения, и еще одно поле для всех объектов ожидания, которые являются ссылочными типами (в терминах их типов на этапе компиляции). В данном случае имеются два выражения `await`, которые используют два разных типа структур ожидания, поэтому получаются два поля ❸. Если бы второе выражение `await` также работало с `TaskAwaiter`, или если бы `TaskAwaiter` и `YieldAwaiter` оба были классами, тогда было бы создано единственное поле. В каждый момент времени активным может быть только один объект ожидания, поэтому совершенно не важно, что сохранять можно лишь одно значение за раз. Вы должны передавать объекты ожидания между выражениями `await`, чтобы после завершения операции можно было получить результат.

Из числа полей общей инфраструктуры ❹ вы уже видели `state` и `builder`. В качестве напоминания, `state` применяется для отслеживания состояния, поэтому продолжение может получить правильную точку внутри кода. Поле `builder` используется для разнообразных действий, включая создание экземпляра `Task` или `Task<T>` для возвращения каркасным методом — задачи, которая впоследствии будет заполнена корректным результатом, когда асинхронный метод завершится. Поле `stack` более загадочно — оно применяется в случае, если выражение `await` встречается как часть оператора, которому необходимо отслеживать дополнительное состояние, не представленное нормальными локальными переменными. Пример этого будет приведен в разделе 15.5.6 — в конечном автомате, сгенерированном для листинга 15.11, данное поле не используется.

Метод `MoveNext()` — это то место, где в игру вступают все интеллектуальные возможности компилятора, но прежде чем его рассматривать, нужно *очень* кратко взглянуть на метод `SetStateMachine()`. В каждом конечном автомате он имеет одну и ту же реализацию, которая выглядит следующим образом:

```
void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine machine)
{
  builder.SetStateMachine(machine);
}
```

В двух словах, этот метод применяется для того, чтобы позволить упакованной копии конечного автомата иметь ссылку на саму себя внутри построителя. Я не буду вдаваться в детали управления всей упаковкой — достаточно лишь понимать, что конечный автомат *упаковывается* там, где это необходимо, а разнообразные аспекты механизма асинхронности гарантируют, что впоследствии одиночная упакованная копия будет использоваться согласованным образом. Это действительно важно, т.к. мы говорим об *изменяемом типе значения* (трепещите!).

Если бы к одной копии конечного автомата применялись одни изменения, а к другой копии — другие, то все очень скоро бы развалилось.

При желании можно представить все по-другому, и это будет важно, если вы *действительно* начнете думать о том, каким образом передаются переменные экземпляра конечного автомата. Во избежание излишних выделений памяти в куче конечный автомат реализован в виде структуры, однако большая часть кода пытается *действовать* так, как будто бы он на самом деле является классом. Работоспособность всего этого обеспечивается ловкими фокусами со ссылками внутри метода `SetStateMachine()`.

Итак, теперь на месте все кроме действительного кода, который был в асинхронном методе. Давайте займемся методом `MoveNext()`.

15.5.4. Одна точка входа для управления всем

Если вы когда-либо декомпилируете асинхронный метод (на самом деле я надеюсь, что вы сделаете это), то увидите, что метод `MoveNext()` в конечном автомате становится очень длинным и очень быстрым, по большей части как функция от количества имеющихся выражений `await`. Он содержит всю логику первоначального метода, *а также* изысканную балетную партию[9], требуемую для обработки всех переходов между состояниями, *и* код оболочки для обработки окончательного результата или исключения.

При написании асинхронного кода вручную вы обычно будете помещать продолжения в отдельные методы: начало в одном методе, затем продолжение в другом и, возможно, завершение в третьем. Однако это усложняет применение средств управления потоком, таких как циклы, к тому же оно не нужно компилятору C#. Задача достижения сгенерированным кодом высокой читабельности не ставилась. Конечный автомат имеет единственную точку входа — метод `MoveNext()`, который используется с самого начала и для продолжений всех выражений `await`. Каждый раз, когда вызывается `MoveNext()`, конечный автомат выясняет с помощью поля `state`, в какой точке метода он находится. Это либо логическая начальная точка метода, либо конец выражения `await`, когда все готово для оценки результата. Каждый конечный автомат выполняется только раз. Фактически имеется оператор `switch`, основанный на `state`, который содержит различные конструкции `case` с соответствующими операторами `goto` для переходов на разные метки.

Метод `MoveNext()` обычно выглядит примерно так:

```
void IStateMachine.MoveNext()
{
  // Для асинхронного метода, объявленного с возвращаемым типом Task<int>
  int result;
  try
  {
    bool doFinallyBodies = true;
    switch (state)
    {
      // Код для перехода в правильное место...
    }
    // Тело метода
  }
```

[9] Код действительно похож на танец, с замысловатыми шагами, которые должны быть выполнены в нужное время и в нужном месте.

```
        catch (Exception e)
        {
            state = -2;
            builder.SetException(e);
            return;
        }
        state = -2;
        builder.SetResult(result);
    }
```

Начальным состоянием всегда является -1, и это *также* будет состоянием, когда метод выполняет ваш код (как противоположность приостановке на время ожидания). Любые неотрицательные состояния указывают цель продолжения. Конечный автомат попадает в состояние -2, когда он завершен. В конечных автоматах, созданных в отладочных конфигурациях, вы увидите ссылку на состояние -3 — но никогда не следует ожидать действительного *попадания* в это состояние. Оно предназначено для того, чтобы избежать получения вырожденного оператора switch, который приводил бы к снижению удобства отладки.

Переменная result устанавливается в ходе метода, в точке, где исходный асинхронный метод имеет оператор return. Затем она применяется в вызове builder.SetResult(), когда достигается логический конец метода. Даже необобщенные типы AsyncTaskMethodBuilder и AsyncVoidMethodBuilder содержат методы SetResult(); первый из них сообщает о факте завершения метода задаче, которая была возвращена из каркасного метода, а второй сигнализирует о завершении исходному объекту SynchronizationContext. (Исключения передаются исходному объекту SynchronizationContext тем же самым способом. Это довольно грубый подход к отслеживанию происходящего, однако он предлагает решение для ситуаций, где действительно *должны* быть методы void.)

Переменная doFinallyBodies используется для выяснения, должны ли любые блоки finally в первоначальном коде (в том числе неявные из операторов using или foreach) быть выполнены, когда поток выполнения покидает область действия блока try. Концептуально блок finally желательно выполнять только в случае покидания блока try нормальным путем. Если же произошел возврат из метода, который ранее имел продолжение, присоединенное к объекту ожидания, то метод логически "приостанавливается", поэтому выполнять блок finally не нужно. Любые блоки finally наряду со связанным с ними блоком try находились бы внутри раздела кода, помеченного как "Тело метода".

Большая часть тела метода узнаваема в понятиях первоначального асинхронного метода. Правда, придется привыкнуть к тому, что все локальные переменные теперь выглядят как переменные экземпляра в конечном автомате, но это не слишком трудно. Как и можно было предположить, все сложности касаются выражений await.

15.5.5. Поток управления для выражений await

Просто чтобы напомнить: любое выражение await представляет разветвление в смысле возможных путей выполнения. Сначала для ожидаемой асинхронной операции извлекается объект ожидания, а затем проверяется его свойство IsCompleted. Если оно возвращает true, можно получить результат немедленно и продолжить.

В противном случае понадобится выполнить следующие действия:

- запомнить объект ожидания для более позднего применения;
- обновить состояние, чтобы указать, откуда продолжать;
- присоединить продолжение к объекту ожидания;
- осуществить возврат из `MoveNext()`, удостоверившись, что ни один блок `finally` не был выполнен.

Затем при вызове продолжения необходимо перейти в правильную точку, извлечь объект ожидания и сбросить состояние перед продолжением.

В качестве примера возьмем первое выражение `await` в листинге 15.11:

```
await Task.Delay(unicode);
```

Сгенерированный для него код выглядит следующим образом:

```
TaskAwaiter localTaskAwaiter = Task.Delay(unicode).GetAwaiter();
  if (localTaskAwaiter.IsCompleted)
  {
    goto DemoAwaitCompletion;
  }
  state = 0;
  taskAwaiter = localTaskAwaiter;
  builder.AwaitUnsafeOnCompleted(ref localTaskAwaiter, ref this);
  doFinallyBodies = false;
  return;
DemoAwaitContinuation:
  localTaskAwaiter = taskAwaiter;
  taskAwaiter = default(TaskAwaiter);
  state = -1;
DemoAwaitCompletion:
  localTaskAwaiter.GetResult();
  localTaskAwaiter = default(TaskAwaiter);
```

Если бы ожидание было организовано для операции, возвращающей значение, например, присваивание результата `await client.GetStringAsync(...)` с использованием `HttpClient`, то вызов `GetResult()` ближе к концу будет местом получения значения.

Метод `AwaitUnsafeOnCompleted()` присоединяет продолжение к объекту ожидания, и оператор `switch` в начале метода `MoveNext()` обеспечит при повторном выполнении `MoveNext()` передачу управления методу `DemoAwaitContinuation()`.

Сравнение методов `AwaitOnCompleted()` и `AwaitUnsafeOnCompleted()`

Ранее был показан значимый набор интерфейсов, где `IAwaiter<T>` расширял `INotifyCompletion` с его методом `OnCompleted()`. Имеется также интерфейс `ICriticalNotifyCompletion` с методом `UnsafeOnCompleted()`. Конечный автомат вызывает `builder.AwaitUnsafeOnCompleted()` для объектов ожидания, реализующих `ICriticalNotifyCompletion`, или `builder.AwaitOnCompleted()` для объектов ожидания, которые реализуют только `INotifyCompletion`. Мы рассмотрим отличия между этими двумя вызовами в разделе 15.6.4, когда будем обсуждать, каким образом шаблон ожидания взаимодействует с контекстами.

Обратите внимание, что компилятор очищает локальные переменные и переменные экземпляра для объекта ожидания, так что они могут быть обработаны сборщиком мусора в подходящее время.

После того, как вы можете определить блок вроде этого как соответствующий одиночному выражению `await`, сгенерированный код на самом деле не *настолько* труден для чтения в декомпилированной форме. Из-за ограничений, накладываемых средой CLR, операторов `goto` (и связанных с ними меток) может быть больше, чем вы ожидали, но, согласно моему опыту, осмысление шаблона `await` является критически важным для понимания асинхронности.

Есть еще один момент, который мне следует объяснить — загадочная переменная `stack` а конечном автомате.

15.5.6. Отслеживание стека

Когда вы размышляете о стековом фрейме, вероятно, вы думаете о локальных переменных, объявленных в методе. Бесспорно, вы можете быть осведомлены о ряде скрытых локальных переменных, подобных итератору для цикла `foreach`, но это не все, что помещается в стек, во всяком случае, логически[10]. В разнообразных ситуациях существуют промежуточные выражения, которые не могут применяться до тех пор, пока не будут оценены какие-то другие выражения. Простейшими примерами являются бинарные операции вроде сложения и вызовы методов.

Ниже приведен тривиальный пример:

```
var x = y * z;
```

С помощью псевдокода, основанного на стеке, его можно представить следующим образом:

```
push y
push z
multiply
store x
```

Теперь предположим, что есть такое выражение `await`:

```
var x = y * await z;
```

Перед ожиданием `z` необходимо оценить переменную `y` и сохранить где-то ее значение, но, в конечном счете, также может произойти немедленный возврат из `MoveNext()`, поэтому для хранения `y` нужен логический стек. Когда выполняется продолжение, значение восстанавливается и участвует в умножении. В данном случае компилятор может присвоить значение `y` переменной экземпляра `stack`. При этом задействуется упаковка, но это означает использование одиночной переменной.

Рассмотренный пример был простым. А теперь представим ситуацию, при которой должно быть сохранено несколько переменных:

```
Console.WriteLine("{0}: {1}", x, await task);
```

Здесь необходимо, чтобы в логическом стеке находилась и строка формата, и значение переменной `x`. На этот раз компилятор создает экземпляр `Tuple<string, int>`, содер-

[10] Как любит говорить Эрик Липперт, стек является деталью реализации — определенные переменные, которые вы можете ожидать увидеть в стеке, в действительности находятся в куче, а некоторые переменные могут существовать только в регистрах. В этом разделе речь пойдет лишь о том, что логически происходит в стеке.

жащий два значения, и сохраняет ссылку на него в поле stack. Подобно объекту ожидания, в любой момент времени необходим только один логический стек, поэтому вполне нормально применять ту же самую переменную[11]. Внутри продолжения отдельные аргументы могут быть извлечены из кортежа и использованы в вызове метода. В загружаемом коде содержится полная декомпилированная версия этого примера, с обоими предшествующими операторами (LogicalStack.cs и LogicalStackDecompiled.cs).

В итоге второй оператор использует примерно такой код:

```
    string localArg0 = "{0} {1}";
    int localArg1 = x;
    localAwaiter = task.GetAwaiter();
    if (localAwaiter.IsCompleted)
    {
      goto SecondAwaitCompletion;
    }
    var localTuple = new Tuple<string, int>(localArg0, localArg1);
    stack = localTuple;
    state = 1;
    awaiter = localAwaiter;
    builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
    doFinallyBodies = false;
    return;
SecondAwaitContinuation:
    localTuple = (Tuple<string, int>) stack;
    localArg0 = localTuple.Item1;
    localArg1 = localTuple.Item2;
    stack = null;
    localAwaiter = awaiter;
    awaiter = default(TaskAwaiter<int>);
    state = -1;
SecondAwaitCompletion:
    int localArg2 = localAwaiter.GetResult();
    Console.WriteLine(localArg0, localArg1, localArg2);
```

Полужирным здесь отмечены строки, в которых задействованы элементы логического стека. К этому моменту, возможно, вы зашли настолько далеко, насколько было необходимо — если вы поняли весь предоставленный материал, то знаете о происходящем "за кулисами" больше, чем об этом подозревают 99% разработчиков. Тем не менее, вполне нормально, если на первый раз не удалось усвоить абсолютно все — когда чтение кода этих конечных автоматов ни к чему не приводит, имеет смысл сделать паузу и вернуться к нему позже.

15.5.7. Дополнительные сведения

Интересуетесь даже еще большим количеством деталей? Займитесь декомпилятором. Я бы посоветовал воспользоваться очень маленькой программой для исследования того, что делает компилятор — если вы пишете что-то нетривиальное, то довольно легко заблудиться в лабиринте неуловимых мелких продолжений, которые все похожи друг на друга. Возможно, вам потребуется уменьшить уровень оптимизации, выполняемой декомпи-

[11] Правда, бывают случаи, когда компилятор может быть лучше осведомлен о типе переменной либо вообще избегать ее включения, если она никогда не понадобится, но все это может быть добавлено в более поздней версии в качестве дополнительной оптимизации.

лятором, чтобы приблизиться к низкоуровневому представлению, а не к его интерпретации. В конце концов, безупречный декомпилятор просто бы воспроизвел ваши асинхронные функции, что свело бы на нет цель данного упражнения!

Код, генерируемый компилятором, не всегда может быть декомпилирован в допустимый код C#. Постоянно возникает проблема умышленного применения непроизносимых имен для переменных и типов, но более важно то, что в ряде случаев допустимый код IL не имеет прямого эквивалента в C#. Например, в IL вполне законно разветвлять инструкцию, которая находится внутри цикла — в конце концов, язык IL *не имеет* даже концепции цикла как таковой. В языке C# не допускается помещать внутрь цикла оператор `goto` с меткой, которая находится за пределами цикла, поэтому инструкция подобного рода не может быть представлена полностью корректно. Даже компилятор C# не может трактовать все это по-своему: язык IL по-прежнему обладает рядом ограничений относительно целей переходов, поэтому вы часто будете обнаруживать, что компилятору приходится проходить через последовательность переходов для достижения нужного места.

Подобным же образом я видел, что некоторые декомпиляторы слегка путают точный порядок операторов присваивания, связанных с логическим стеком, временами перемещая присваивание временных переменеенных (`localArg0` и `localArg1`, например) на неправильную сторону проверки свойства `IsCompleted`. Я уверен, это объясняется тем, что код не очень похож на нормальный вывод компилятора C#. Это не так уже плохо, когда вы знаете, что искать, однако означает, что иногда придется опускаться на уровень кода IL.

15.6. Практическое использование `async/await`

К настоящему моменту вы ознакомились с тем, как ведут себя асинхронные функции, и узнали, каким образом они выглядят "за кулисами". Так что же, вы теперь можете считать себя экспертом в асинхронном программировании? Определенно нет[12]. Подобно многим аспектам программирования, для обретения опыта нужно многое, и к этому времени лишь небольшое число разработчиков достаточно плотно имели дело с асинхронными функциями. Я пока не могу поделиться с вами собственным опытом, но предоставлю советы и подсказки, которые хотя бы немного упростят вам жизнь.

На время написания эти строк команды разработчиков, обладающих наибольшим пониманием асинхронного программирования на C# 5, входили в состав Microsoft; они жили и дышали им на протяжении всей разработки и получали отзывы от бета-тестировщиков и других людей. С этой целью я *настоятельно* рекомендую почитать блог команды параллельного программирования (Parallel Programming Team) по адресу `http://blogs.msdn.com/b/pfxteam/`, в котором приводится намного больше рекомендаций, чем в этой главе.

Разумеется, это вовсе не означает, что у меня нет для вас нескольких советов.

15.6.1. Асинхронный шаблон, основанный на задачах

Одним из преимуществ средства асинхронных функций в C# 5 является то, что оно предлагает согласованный подход к асинхронности. Однако его легко можно было бы нарушить, если бы любой мог выдвигать собственные пути его использования — как именовать асинхронные методы, каким образом должны генерироваться исключения, и тому подобное. В Microsoft решили данную проблему за счет опубликования *асинхрон-*

[12] Конечно, вы *можете* быть экспертом в асинхронном программировании, но отнюдь не только благодаря чтению одной этой главы.

ного шаблона, основанного на задачах (Task-based Asynchronous Pattern — TAP) — набора соглашений, которым должны следовать все. Он доступен в виде автономного документа (`http://mng.bz/B68W`), а также отдельных страниц в MSDN (`http://mng.bz/4N39`).

Конечно, в Microsoft также следуют этому шаблону — инфраструктура .NET 4.5 содержит огромное число асинхронных API-интерфейсов для всех видов сценариев. Почти как в случае обычных соглашений .NET по именованию, проектированию типов и прочим аспектам, если вы следуете тем же соглашениям, что и инфраструктура, то существенно облегчаете работу с вашим кодом другим разработчикам.

Шаблон TAP весьма читабелен и занимает всего лишь 38 страниц — я настоятельно рекомендую прочитать документ целиком. В оставшихся материалах этого раздела будут раскрыты наиболее важные его части.

Асинхронные методы должны иметь имена, заканчивающиеся на суффикс `Async` — `GetAuthenticationTokenAsync()`, `FetchUserPortfolioAsync()` и т.д. В .NET Framework это привело к ряду конфликтов — класс `WebClient` уже имел методы вроде `DownloadStringAsync()`, следующие асинхронному шаблону, основанному на событиях, из-за чего новые методы, соответствующие TAP, имеют несколько неуклюжие имена `DownloadStringTaskAsync()`, `UploadDataTaskAsync()` и тому подобные. Добавлять суффикс `TaskAsync` рекомендуется, если у вас также возникают конфликты имен. Там, где асинхронность очевидна, суффикс вообще можно не указывать — примерами могут служить `Task.Delay()` и `Task.WhenAll()`. В качестве общего правила, если вся бизнес-логика метода направлена на асинхронность, а не на достижение какой-то бизнес-цели, то *возможно* безопаснее отбросить суффикс.

Методы TAP, как правило, возвращают `Task` или `Task<T>` — опять-таки, существуют исключения, такие как `Task.Yield()`, где в игру вступает шаблон ожидания, но они встречаются нечасто. Важно то, что задача, возвращаемая из метода TAP, должна быть *горячей*. Другими словами, представляемая задачей операция должна уже находиться в состоянии выполнения — вызывающий метод не должен запускать ее вручную. Для большинства разработчиков это может выглядеть очевидным, но есть другие платформы, где соглашение предусматривает создание холодной задачи, которая не запускается до тех пор, пока это не будет запрошено явно — немного похоже на итераторный блок в C#. В частности, такое соглашение принято в языке F#, и его также следует принимать во внимание в Reactive Extensions (Rx).

Существуют четыре перегруженных версии, о предоставлении которых необходимо *подумать* при создании асинхронного метода. Все они принимают одни и те же базовые параметры, но предоставляют разные возможности по сообщению о ходе работ и отмене. Предположим, что решено разработать асинхронный метод, который является логическим эквивалентом следующего синхронного метода:

```
Employee LoadEmployeeById(string id)
```

По соглашениям TAP можно было бы предоставить любой или все перечисленные ниже методы:

```
Task<Employee> LoadEmployeeById(string id)
Task<Employee> LoadEmployeeById(string id,
CancellationToken cancellationToken)
Task<Employee> LoadEmployeeById(string id, IProgress<int> progress)
Task<Employee> LoadEmployeeById(string id,
  CancellationToken cancellationToken, IProgress<int> progress)
```

Здесь `IProgress<int>` мог бы быть `IProgress<T>` для любого типа `T`, который подходит для использования при сообщении о ходе работ. Например, если асинхронный

метод находит коллекцию записей и затем обрабатывает их по одной, можно было бы принимать `IProgress<Tuple<int, int>>`, который позволил бы сообщать, сколько всего записей и сколько из них обработано.

Я бы избегал попыток втиснуть сообщение о ходе работ внутрь операций, в которых это не имеет никакого смысла. Отмена обычно проще в плане поддержки, поскольку ее уже поддерживают очень многие методы инфраструктуры. Если асинхронный метод в основном состоит из выполнения множества других асинхронных операций (возможно с зависимостями), то может оказаться проще принимать признак отмены и передавать его дальше.

Асинхронные операции должны выполнять проверки на предмет ошибочного применения (обычно ситуации с недопустимыми аргументами) синхронным образом. Это слегка неуклюже, но может быть реализовано либо за счет разделения метода, как было показано в разделе 15.3.6, либо с помощью единственного метода, использующего анонимную асинхронную функцию, как объяснялось в разделе 15.4. Хотя заманчиво проверять допустимость аргументов ленивым образом, вы поплатитесь тем, что выяснить причины возможного отказа окажется труднее, чем должно быть.

Операции, основанные на вводе-выводе, когда работа передается либо диску, либо другому компьютеру, являются великолепными кандидатами для асинхронности безо всяких видимых недостатков. Задачи, интенсивно загружающие центральный процессор, для этого подходят меньше. Определенную работу легко выгрузить в пул потоков, и благодаря методу `Task.Run()` в .NET 4.5 это делать даже проще, чем было ранее, но выполнение такого действия внутри библиотечного кода означало бы принятие предположений от имени вызывающего кода. Разный вызывающий код может также иметь отличающиеся требования; если вы просто откроете доступ к синхронному методу, то тем самым предоставите вызывающему коду гибкость в плане работы наиболее подходящим для него способом. Он может либо запустить новую задачу, если это необходимо, либо вызвать метод синхронно, если вполне приемлемо загрузить текущий поток выполнением метода в течение некоторого времени. Задачи, которые являются смесью ожидания результатов из других систем и последующей их обработки с потенциально большими затратами времени, будут сложнее. Хотя я считаю, что жесткие и быстрые руководящие принципы вряд ли окажутся полезными, важно *документировать* поведение. Если вы собираетесь занимать много ресурсов центрального процессора в контексте вызывающего кода, то должны сделать это предельно ясным.

Другой способ избежать применения контекста вызывающего кода предполагает использование метода `Task.ConfigureAwait()`. Этот метод в настоящее время имеет только один параметр, `continueOnCapturedContext`, хотя для ясности при его указании полезно применять именованный аргумент. Метод возвращает реализацию шаблона ожидания. Когда аргумент равен `true`, объект, поддерживающий ожидание, ведет себя в точности как нормальный объект, так что если асинхронный метод вызывается в потоке пользовательского интерфейса, например, то продолжение после выражения `await` будет выполняться по-прежнему в потоке пользовательского интерфейса. Это удобно, *если* вам необходим доступ к элементам пользовательского интерфейса. Тем не менее, если специальные требования отсутствуют, для аргумента можно указать значение `false`, и тогда продолжение будет выполняться обычно в том же контексте, что и завершенная первоначальная операция[13].

---

[13] Как правило, но не всегда. Детали не документированы явно, но существуют случаи, когда выполнение в том же самом контексте на самом деле нежелательно. Вы должны считать, что вызов `ConfigureAwait(false)` говорит "меня не заботит, где выполняется продолжение", а не явно присоединять его к специфичному контексту.

Для смешанной рабочей нагрузки, при которой определенные данные извлекаются, обрабатываются и затем сохраняются в базе данных, может быть предусмотрен код следующего вида:

```
public static async Task<int> ProcessRecords()
{
   List<Record> records = await FetchRecordsAsync()
      .ConfigureAwait(continueOnCapturedContext: false);

   // ... обработка записей ...
   await SaveResultsAsync(results)
      .ConfigureAwait(continueOnCapturedContext: false);

   // Позволить вызывающему коду узнать, сколько записей было обработано
   return records.Count;
}
```

Большая часть кода метода пригодна для выполнения в потоке из пула; это именно то, что нужно, т.к. не делается ничего такого, что требует выполнения в исходном потоке. (Выражаясь профессиональной лексикой, операция не имеет никакой *привязки к потоку*.) Однако это не влияет на вызывающий код; если асинхронный метод пользовательского интерфейса ожидает результата вызова метода `ProcessRecords()`, то этот асинхронный метод будет выполняться по-прежнему в потоке пользовательского интерфейса. Только код внутри метода `ProcessRecords()` заявляет, что его не заботит контекст выполнения.

Вероятно, здесь нет действительной необходимости вызывать метод `ConfigureAwait()` во втором выражении `await`, т.к. осталось очень мало работы, но, в общем, он должен использоваться в *каждом* выражении `await`, и неплохо, чтобы делать это согласованно вошло в привычку. Если вы хотите предоставить вызывающему коду гибкость в отношении контекста, в котором метод выполняется, потенциально можно было бы предусмотреть для этого параметр в асинхронном методе.

Обратите внимание, что метод `ConfigureAwait()` влияет только на часть *синхронизации* контекста выполнения. Другие аспекты, такие как заимствование прав, распространяются безотносительно к этому, как будет показано в разделе 15.6.4.

Поток данных TPL

Хотя TAP — это просто набор соглашений и ряд примеров, в Microsoft также создали отдельную библиотеку под названием TPL Dataflow (Поток данных TPL), которая предназначена для предоставления высокоуровневых строительных блоков, ориентированных на специфичные сценарии, особенно на те, которые могут быть смоделированы с применением шаблонов "производитель/потребитель". Чтобы приступить к ее использованию, проще всего загрузить пакет NuGet (`Microsoft.Tpl.Dataflow`). Он бесплатен и по нему доступно много инструкций. Даже если вы не планируете работать с этой библиотекой напрямую, полезно взглянуть на нее, чтобы получить представление о том, как параллельные программы в принципе *могут* быть спроектированы.

Но и без дополнительных библиотек можно по-прежнему строить элегантный асинхронный код, следующий нормальным проектным принципам, и одним из наиболее важных аспектов этого является объединение.

15.6.2. Объединение асинхронных операций

Больше всего в асинхронности C# 5 мне нравится исключительно естественный способ объединения. Это проявляется двумя разными путями. Наиболее очевидная форма объединения связана с тем, что асинхронные методы возвращают задачи и обычно предусматривают вызов других методов, которые возвращают задачи. Они могут быть прямыми асинхронными операциями (так сказать, находиться внизу цепочки) или просто дополнительными асинхронными методами. Вся упаковка и распаковка, требуемая для перевода результатов в задачи и обратно, обрабатывается компилятором.

Другая форма объединения — это способ, которым можно создавать нейтральные к операциям строительные блоки для управления тем, как обрабатываются задачи. Такие строительные блоки не нуждаются в знании чего-либо о том, что делают задачи, а находятся исключительно на уровне абстракции `Task<T>`. Они немного напоминают операции LINQ, но работают с задачами, а не последовательностями. Некоторые строительные блоки встроены в инфраструктуру, однако допускается также писать и собственные.

Сбор результатов в единственном вызове

В качестве примера давайте рассмотрим задачу извлечения множества URL. В разделе 15.3.6 это делалось по одному URL за раз, с остановом в случае успеха. Предположим, что на этот раз необходимо запускать запросы параллельно и затем фиксировать в журнале результат для каждого URL. Вспомнив, что асинхронные методы возвращают уже выполняющиеся задачи, запустить задачу для каждого URL довольно легко:

```
var tasks = urls.Select(async url =>
{
  using (var client = new HttpClient())
  {
    return await client.GetStringAsync(url);
  }
}).ToList();
```

Обратите внимание, что обращение к `ToList()` требуется для активизации запроса LINQ. Это гарантирует, что каждая задача запускается один и только один раз — иначе при каждом проходе по `tasks` будет начинаться другой набор извлечений. (Код мог быть еще проще, если не заботиться об освобождении `HttpClient`, но даже с учетом этого недостатка он выглядит весьма неплохо.)

Библиотека TPL предлагает метод `Task.WhenAll()`, объединяющий результаты множества задач, каждая из которых предоставляет одиночный результат, в единую задачу с множеством результатов. Сигнатура перегруженной версии, которая будет применяться, выглядит следующим образом:

```
static Task<TResult[]> WhenAll<TResult>(IEnumerable<Task<TResult>> tasks)
```

Объявление выглядит устрашающим, но назначение метода довольно простое, как вы поймете, приступив к его использованию. Вы получили `List<Task<string>>`, поэтому можно написать такой код:

```
string[] results = await Task.WhenAll(tasks);
```

Здесь будет организовано ожидание завершения всех задач и сбор результатов в массив. В случае если несколько задач сгенерируют исключения, только первое из них будет сгенерировано непосредственно, однако всегда можно пройти по задачам

и выяснить, какие из них отказали и почему, или применить расширяющий метод WithAggregatedExceptions(), показанный в листинге 15.2.

Если вас интересует только первый запрос, возвращенный обратно, то есть еще один метод по имени Task.WhenAny(), который не ожидает первого *успешного* завершения задачи, а ожидает первой задачи, достигшей терминального состояния.

В данном случае может понадобиться что-то слегка отличающееся. Более удобно сообщать все результаты, когда они поступают.

Сбор результатов по мере их поступления

Несмотря на то что метод Task.WhenAll() был примером трансформационного строительного блока, входящего в состав инфраструктуры .NET, в следующем примере показано, как можно построить собственные методы аналогичным образом. В документации TAP приведен очень похожий пример кода, создающий метод по имени Interleaved(), и мы рассмотрим слегка отличающуюся версию.

Идея кода в листинге 15.12 заключается в том, чтобы позволить передачу последовательности входных задач, и метод будет возвращать последовательность выходных задач. Результаты задач в двух последовательностях будут теми же самыми, но с одной ключевой разницей: выходные задачи будут завершаться в порядке их предоставления, так что к ним можно применять await по одной за раз и знать, что результаты будут получены, как только они станут доступными. Теперь это может звучать сродни "магии" — для меня так оно и есть — так что взгляните на код и посмотрите, как он работает.

Листинг 15.12. Трансформация последовательности задачи в новую коллекцию согласно порядку завершения

```
public static IEnumerable<Task<T>> InCompletionOrder<T>
    (this IEnumerable<Task<T>> source)
{
  var inputs = source.ToList();
  var boxes = inputs.Select(x => new TaskCompletionSource<T>())
                    .ToList();
  int currentIndex = -1;
  foreach (var task in inputs)
  {
    task.ContinueWith(completed =>
    {
      var nextBox = boxes[Interlocked.Increment(ref currentIndex)];
      PropagateResult(completed, nextBox);
    }, TaskContinuationOptions.ExecuteSynchronously);
  }
  return boxes.Select(box => box.Task);
}
```

Код в листинге 15.12 полагается на очень важный тип в библиотеке TPL — TaskCompletionSource<T>. Данный тип позволяет создавать экземпляр Task с пока отсутствующим результатом и затем позже предоставлять результат (либо исключение). Это построено на основе той же самой инфраструктуры, которую структура AsyncTaskMethodBuilder<T> использует для предоставления экземпляра Task, воз-

вращаемого асинхронным методом, делая возможным заполнение задачи результатом, когда тело метода завершается.

Чтобы объяснить несколько необычные имена переменных, я часто думаю о задачах как о картонных коробках, внутри которых в какой-то момент появится значение (или отказ). Тип `TaskCompletionSource<T>` подобен коробке с отверстием на задней стенке — ее можно дать кому-то и затем позже исподтишка подсмотреть значение через отверстие[14]. Именно это делает метод `PropagateResult()` — он не особенно интересен, поэтому здесь не показан, но, в сущности, данный метод передает результат завершенной задачи `Task<T>` в `TaskCompletionSource<T>`. Если исходная задача завершилась нормально, то в источник завершения задачи копируется возвращаемое значение. Если исходная задача потерпела отказ, то в источник завершения задачи копируется исключение. Если же исходная задача была отменена, то отменяется и источник завершения задачи.

Действительно искусной частью (и моей заслуги здесь нет — данное предложение пришло ко мне по электронной почте) является то, что когда этот метод выполняется, ему не известно соответствие между `TaskCompletionSource<T>` и входными задачами. Взамен он просто присоединяет к каждой задаче одно и то же продолжение, которое выражает такое действие: "Найти следующий источник `TaskCompletionSource<T>` (атомарно инкрементируя счетчик) и передать ему результат". Другими словами, коробки заполняются в выходном порядке по мере завершения исходных задач.

На рис. 15.5 показаны три входных задачи и соответствующие выходные задачи, возвращаемые методом. Выходные задачи завершаются в порядке возвращения, несмотря на то, что входные задачи завершаются в другом порядке.

Имея этот замечательный расширяющий метод, можно написать код, приведенный в листинге 15.13, который принимает коллекцию URL, запускает запросы для каждого из URL параллельно, выводит на консоль длину каждой страницы по мере завершения запросов и возвращает суммарную длину.

Листинг 15.13. Отображение длин страниц по мере возвращения данных

```
static async Task<int> ShowPageLengthsAsync(params string[] urls)
{
   var tasks = urls.Select(async url =>
   {
      using (var client = new HttpClient())
      {
         return await client.GetStringAsync(url);
      }
   }).ToList();
   int total = 0;
   foreach (var task in tasks.InCompletionOrder())
   {
      string page = await task;
      Console.WriteLine("Got page length {0}", page.Length);
      total += page.Length;
   }
   return total;
}
```

[14] Любые совпадения с реальностью чисто случайны, и я не несу никакой ответственности за эксперименты с кошками (представленными с помощью `Task<Cat>`).

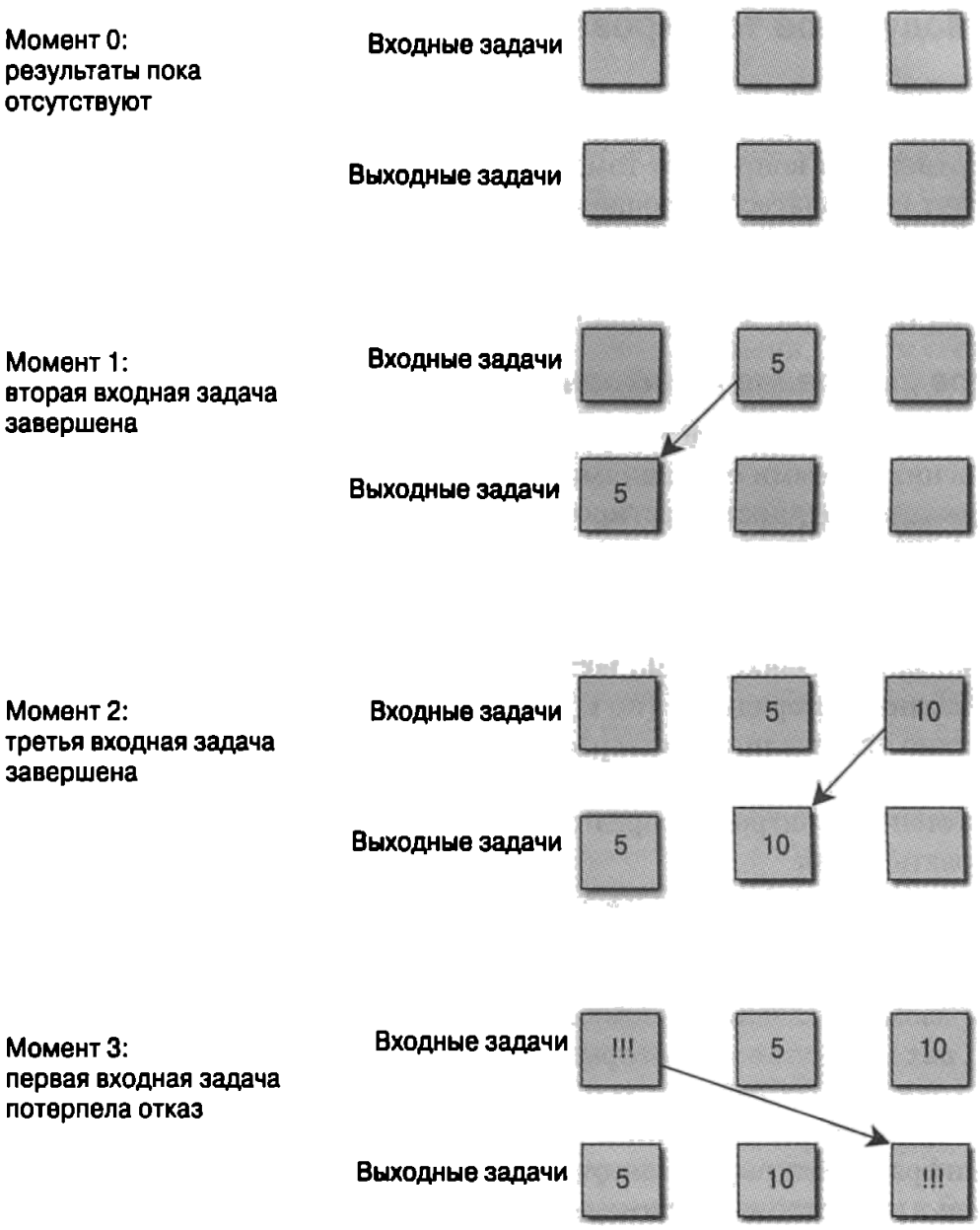

Рис. 15.5. Визуализация упорядочения

С кодом в листинге 15.13 связаны две небольших проблемы.

- Если одна задача терпит отказ, то отказывает вся асинхронная операция, никак не отражая остальные результаты. Это может быть приемлемым или может понадобиться регистрировать в журнале каждый отказ. (В отличие от .NET 4, разрешение исключениям из задач оставаться необнаруженными по умолчанию не нарушает работу процесса, но вы должны хотя бы подумать о том, что произойдет с другими задачами.)
- Теряется информация о том, с какого URL поступила та или иная страница.

Обе проблемы довольно легко устраняются за счет написания незначительного объема дополнительного кода, и они могут даже предложить новые многократно используемые строительные блоки. Целью показанных примеров было не исследование отдельных требований, а демонстрация возможностей, предлагаемых объединением.

Метод `Interleaved()` — не единственный пример в официальной документации TAP; она содержит множество идей и примеров кода, помогающих понять их.

15.6.3. Модульное тестирование асинхронного кода

Я слегка нервничал, приступая к написанию этого раздела. На данный момент я не верю, что сообщество разработчиков обладает достаточным опытом, чтобы дать окончательные ответы на вопросы о том, как тестировать асинхронный код. Я уверен, что по пути будут совершаться оплошности, и без сомнений появится несколько конкурирующих подходов. Важный аспект в том, что подобно синхронному коду, если вы проектируете с учетом тестируемости с самого начала, то сможете эффективно проводить модульное тестирование асинхронного кода.

Безопасное внедрение асинхронности

В этом разделе будет представлен подход для ситуаций, когда имеется возможность управлять асинхронными операциями, от которых зависит асинхронный код. Он не пытается разрешить трудности тестирования кода, в котором применяется `HttpClient` и аналогичные трудные для имитации типы, но в нем нет ничего нового — если есть зависимости, которые трудно использовать в тестах, всегда будут возникать проблемы.

Предположим, что нужно протестировать код "магического упорядочения" из предыдущего раздела. Вам необходима возможность создавать задачи, которые будут завершаться в указанном порядке, и (по крайней мере, в некоторых тестах) удостоверяться в том, что можно устанавливать утверждения между завершениями задач. Кроме того, все это желательно делать без привлечения любых других потоков — требуется максимально высокая степень контроля и предсказуемости. По сути, вы хотите иметь возможность контролировать время.

Мое решение данной проблемы, в сущности, сводится к имитации времени с применением класса `TimeMachine`, который предлагает способ программного продвижения во времени с запланированными задачами, которые завершаются определенным образом в указанные моменты времени. За счет объединения его с классом `SynchronizationContext`, который фактически представляет собой ручную версию знакомого конвейера обработки сообщений Windows Forms, получается довольно рациональное средство тестирования. Я не буду здесь приводить весь используемый для этого код инфраструктуры, т.к. он очень длинный и относительно скучный, но полный код доступен в загружаемых примерах. Тем не менее, я покажу пару тестов.

Давайте начнем с полностью успешного случая: если вы запрограммировали три задачи для завершения в моменты времени 1, 2 и 3, но вызвали метод `InCompletionOrder()` с этими задачами в другом порядке, то должны получить *результаты*, которые по-прежнему упорядочены:

```
[TestMethod]
public void TasksCompleteInOrder()
{
  var tardis = new TimeMachine();
  var task1 = tardis.ScheduleSuccess(1, "t1");
  var task2 = tardis.ScheduleSuccess(2, "t2");
  var task3 = tardis.ScheduleSuccess(3, "t3");

  var tasksOutOfOrder = new[] { task2, task3, task1 };

  tardis.ExecuteInContext(advancer =>
  {
    var inOrder = tasksOutOfOrder.InCompletionOrder().ToList();
    advancer.AdvanceTo(3);
    Assert.AreEqual("t1", inOrder[0].Result);
```

```
    Assert.AreEqual("t2", inOrder[1].Result);
    Assert.AreEqual("t3", inOrder[2].Result);
  });
}
```

Метод ExecuteInContext() временно заменяет объект SynchronizationContext текущего потока объектом ManuallyPumpedSynchronizationContext (ищите этот класс в загружаемом коде) и затем предоставляет объект продвижения (advancer) делегату, указанному аргументом метода. Этот объект продвижения может применяться для продвижения времени на заданные промежутки с задачами, завершающимися (и выполняющими продолжения) в подходящие моменты времени. В этом тесте производится просто быстрый переход вперед до тех пор, пока все задачи не будут завершены.

Ниже приведен второй тест, который демонстрирует возможность управления временем более детализированным путем:

```
// Шаги по настройке не показаны; они такие же, как в предыдущем тесте.
tardis.ExecuteInContext(advancer =>
{
  var inOrder = tasksOutOfOrder.InCompletionOrder().ToList();

  Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[0].Status);
  Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[1].Status);
  Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

  advancer.Advance();
  Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[0].Status);
  Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[1].Status);
  Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

  advancer.Advance();
  Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[1].Status);
  Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

  advancer.Advance();
  Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[2].Status);
});
```

Здесь можно увидеть, что выходные задачи завершаются в правильном порядке.

Вас может заинтересовать, почему моменты времени представлены обычными целыми числами — возможно, вы ожидали, что будут задействованы типы DateTime и TimeSpan. Это сделано умышленно — единственная временная шкала, которая на самом деле имеется, является искусственной и устанавливается классом TimeMachine, а интересующими точками во времени будут только те, где задачи завершаются.

Разумеется, тестируемый метод несколько необычен в двух отношениях:

- он в действительности не реализован с использованием async;
- он получает задачи напрямую как аргументы.

Если бы вы тестировали более сосредоточенный на бизнес-логике асинхронный метод, то вполне вероятно запланировали бы все результаты для своих зависимостей, продвигали бы время для завершения их всех и затем проверяли бы результат возвращенной задачи. Вы должны были бы иметь возможность обеспечить своему производственному коду имитации нормальным образом — единственное отличие, привносимое асинхронностью, заключается в том, что вместо использования заглушек или имитированных объектов для возвращения прямых результатов из вызовов вы запрашивали бы возврат задач, генерируемых TimeMachine. Все традиционные преимущества инверсии управления по-прежнему применимы — вам просто нужен способ создания подходящих задач.

Одна эта идея, очевидно, не может служить панацеей, но я надеюсь, что мне удалось хотя бы убедить вас в *возможности* модульного тестирования асинхронного кода без произвольных вызовов `Thread.Sleep()` и постоянного риска получения хрупких тестов.

Выполнение асинхронных тестов

Тесты в предыдущем разделе были полностью синхронными. Внутри самих этих тестов ключевые слова `async` или `await` вообще не использовались. В случае применения класса `TimeMachine` для всех тестов это вполне обоснованно, но в других ситуациях может понадобиться написать тестовые методы, декорированные с помощью `async`.

Это делается легко:

```
[Test] // Тип TestAttribute из NUnit
public async void BadTestMethod()
{
   // Код, в котором используется await
}
```

Такой код *скомпилируется* с любой нормальной инфраструктурой тестирования, но он может не делать то, что ожидается. В частности, может оказаться, что все тесты запущены параллельно и вполне вероятно "завершаются" до того, как выдвигают хоть какие-то утверждения.

Так случилось, что инфраструктура NUnit поддерживает асинхронные тесты, начиная с версии 2.6.2, и показанный выше метод работал бы благодаря определенному мастерству, заложенному в реализацию. Однако если вы попытаетесь запустить его с более старыми версиями, тест начнется и затем завершится, поскольку запускающий тест код посчитал, что достигнуто первое "медленное" выражение `await`. Любые сбои, происходящие в методе позже, будут сообщаться объекту `SynchronizationContext` теста, который может этого не ожидать.

Для инфраструктур тестирования, которые поддерживают асинхронные тесты, гораздо более удачный подход предусматривает обеспечение возврата такими тестами экземпляра `Task`, примерно так:

```
[Test]
public async Task GoodTestMethod()
{
   // Код, в котором используется await
}
```

Теперь инфраструктуре тестирования намного легче узнать, когда тесты завершены и необходимо выполнять проверки на предмет сбоя. Подход имеет дополнительное преимущество того, что инфраструктуры тестирования, которые *не* поддерживают асинхронные тесты, могут даже не пытаться запускать их, взамен выдавая предупреждение, что намного лучше, чем некорректное выполнение тестов. На момент написания этих строк все последние версии инфраструктур NUnit, xUnit и Visual Studio Unit Testing Framework (также неформально называемой MS Test) поддерживали асинхронные тесты — другие инфраструктуры тоже могут делать это. Прежде чем приступать к написанию тестов подобного рода, обязательно проверяйте конкретную инфраструктуру и ее версию.

Вы должны также проявлять осторожность, памятуя о возможных взаимоблокировках. В отличие от тестов с помощью класса `TimeMachine` из предыдущего раздела, *вряд ли* вы захотите, чтобы все продолжения выполнялись в единственном потоке, если только этот поток также не будет подкачиваться сообщениями как поток пользовательского ин-

терфейса. Иногда вы управляете всеми задействованными задачами и можете обосновать свой способ использования однопоточного контекста, а в других ситуациях вы должны быть гораздо более осторожны и также обеспечить для множества потоков возможность запускать продолжения, при условии, что сам тестирующий код не выполняется параллельно. Я переживаю в этом случае за *модульные* тесты, но если вы применяете тот же самый вид инфраструктуры для функциональных тестов, интеграционных тестов или даже зондирования производственной версии, то обычно хотите, чтобы тесты запускались в отношении реальных задач, а не имитаций, предоставляемых классом `TimeMachine`.

Я уверен, что со временем сообщество разработает ряд замечательных инструментов, помогающих тестировать все больше и больше кода. Убежден, что значительная доля будущего кода будет по своей природе асинхронной, и совершенно уверен в том, что никогда не захочу писать какой-либо код без тестов. К настоящему моменту мы почти закончили с асинхронностью, но ранее я обещал возвратиться к показанному ранее интересному вызову метода `AwaitUnsafeOnCompleted()` в сгенерированном коде.

15.6.4. Возвращение к шаблону ожидания

В разделе 15.3.3 было представлено несколько воображаемых интерфейсов, которые иллюстрируют правильную базовую идею, заложенную в шаблон ожидания. Даже когда я объяснял, что это была не совсем реальность, то немого сжульничал. Если только вы не реализуете шаблон ожидания самостоятельно или внимательно изучаете его в декомпилированном коде, то на самом деле не нуждаетесь в знании об этом жульничестве, но если вы зашли настолько далеко, то вероятно хотите знать все.

Подлинным интерфейсом, который упоминался ранее, был `INotifyCompletion`, который выглядит следующим образом:

```
public interface INotifyCompletion
{
  void OnCompleted(Action continuation);
}
```

Тем не менее, есть еще один интерфейс, который его расширяет и также находится в пространстве имен `System.Runtime.CompilerServices`:

```
public interface ICriticalNotifyCompletion : INotifyCompletion
{
  void UnsafeOnCompleted(Action continuation);
}
```

Все причины существования этих интерфейсов имеют в своей основе *контекст*. В этой главе класс `SynchronizationContext` уже упоминался несколько раз, и вы можете хорошо знать о нем; это контекст синхронизации, который позволяет вызовам маршализироваться в подходящий поток, будь он специфичным потоком из пула, одиночным потоком пользовательского интерфейса или любым другим необходимым потоком. Однако это не единственный задействованный контекст. Их много — `SecurityContext`, `LogicalCallContext` и `HostExecutionContext`, например. Тем не менее, общим их предком является `ExecutionContext`. Он действует в качестве контейнера для всех других контекстов, и именно на нем будет сосредоточено внимание в настоящем разделе.

Очень важно, что `ExecutionContext` заполняет точки с `await`; вам не нужно возвращаться в асинхронный метод, когда задача завершена, разве только для выяснения права какого пользователя заимствуются, если вы забыли это, например. Для заполнения контекста он должен быть *захвачен* во время присоединения продолжения и за

тем *восстановлен* при выполнении продолжения. Это достигается посредством методов `ExecutionContext.Capture()` и `ExecutionContext.Run()`, соответственно.

Существуют две порции кода, которые могут выполнять такую пару действий "захват/восстановление": объект ожидания и класс `AsyncTaskMethodBuilder<T>` (вместе с родственными ему классами). Вы могли ожидать, что нужно просто сделать выбор в пользу одного или другого и оставить их. Однако в игру вступают разнообразные компромиссы. Довольно легко забыть заполнить контекст выполнения в объекте ожидания, так что имеет смысл реализовать его *однажды* в коде построителя метода. С другой стороны, объект ожидания будет напрямую доступен любому использующему его коду, поэтому вряд ли вы захотите открывать потенциальную брешь в безопасности, надеясь, что все вызывающие методы, которые пользуются сгенерированным кодом, предполагают наличие заполнения контекста в коде объекта ожидания. Но еще менее желательно, чтобы захват и восстановление контекста выполнялись дважды, порождая избыточность. Как разрешить эту дилемму?

Вы уже видели ответ: применение двух разных интерфейсов с тонким отличием в их смысле. Если вы реализуете шаблон ожидания, то ваш метод `OnCompleted()` (который является обязательным) *должен* заполнять контекст выполнения. Если вы решите реализовать интерфейс `ICriticalNotifyCompletion`, то ваш метод `UnsafeOnCompleted()` *не* должен заполнять контекст выполнения, и должен быть декорирован атрибутом `[SecurityCritical]`, чтобы предотвратить его вызов в коде, не являющимся доверенным. Конечно, построители методов относятся к доверенному коду, и они заполняют контекст, поэтому тут все в порядке — вызывающий код с частичным доверием может по-прежнему эффективно использовать ваш объект ожидания, но потенциальные нарушители не получат возможность обойти заполнение контекста.

Я намеренно оставил этот раздел довольно кратким; я нахожу всю тему, связанную с контекстами, несколько запутанной, и есть много дополнительных сложностей, которых мы даже не касались. Если вы реализуете собственный объект ожидания, не делегируя его работу существующему объекту подобного рода (или, *возможно*, не нуждаясь в этом), то определенно должны почитать статью в блоге Стивена Тауба под названием "`ExecutionContext` vs `SynchronizationContext`" ("Сравнение `ExecutionContext` и `SynchronizationContext`"), доступную по адресу `http://mng.bz/Ye65`.

15.6.5. Асинхронные операции в WinRT

Операционная система Windows 8 ввела в экосистему приложений магазин Windows Store — и наряду с ним WinRT. Дополнительные сведения о WinRT будут даны в приложении B, а сейчас можно сказать, что эта система была задумана как современная, объектно-ориентированная, неуправляемая среда. Во многих отношениях это новая библиотека Win32. Некоторые известные типы .NET в WinRT не доступны, но даже те, что доступны, главным образом были лишены блокирующих вызовов, имеющих отношение к вводу-выводу. Как вы уже видели, типы, которые по-прежнему находятся в CLR, обычно предоставляют асинхронные операции через `Task<T>`, но в самой среде WinRT такой тип не существует. Вместо этого имеется набор интерфейсов, которые все расширяют один ключевой интерфейс `IAsyncInfo`:

- `IAsyncAction`
- `IAsyncActionWithProgress<TProgress>`
- `IAsyncOperation<TResult>`
- `IAsyncOperationWithProgress<TResult, TProgress>`

Можете считать, что отличие между типами Action и типами Operation подобно отличию между Task и Task<T> или между Action и Func: тип Action не имеет возвращаемого значения, тогда как Operation имеет. Версии WithProgress встраивают сообщение о ходе работ внутрь одиночного типа, а не требуют перегрузки методов в IProgress<T>, как в случае шаблона TAP.

Детали этих интерфейсов выходят за рамки тематики данной книги, но доступно немало ресурсов, объясняющих их. Я советую начать со статьи в блоге Стивена Тауба под названием "Diving deep with WinRT and await" ("Более глубокие исследования WinRT и await"), которая находится по адресу http://mng.bz/F1TF.

В терминах поддержки этих интерфейсов в C# 5 необходимо отметить несколько важных моментов.

- Расширяющие методы GetAwaiter() позволяют ожидать действия и операции напрямую.
- Расширяющие методы AsTask() позволяют рассматривать действие или операцию как задачу, с поддержкой признаков отмены и сообщением о ходе работ через IProgress<T>.
- Расширяющие методы AsAsyncOperation() и AsAsyncAction() двигаются в обратном направлении, получая задачу и возвращая дружественную к WinRT оболочку.

Все это предоставляется классом System.WindowsRuntimeSystemExtensions, находящимся в сборке System.Runtime.WindowsRuntime.dll.

Вы еще раз увидели ценность шаблона ожидания. Компилятор C# в действительности не заботит тот факт, что для ожидания асинхронной операции вызывается расширяющий метод. Это просто другой тип, поддерживающий ожидание. Большую часть времени вы, скорее всего, сможете оставлять асинхронную операцию с ее собственным типом и организовывать ожидание обычным образом. Тем не менее, хорошо располагать гибкостью трактовки асинхронной операции WinRT как знакомого экземпляра Task<T> в более сложных сценариях.

Еще одной возможностью для выполнения кода в рамках модели асинхронности WinRT является применение метода Run() из класса System.Runtime.InteropServices.WindowsRuntime.AsyncInfo. Такое использование яснее, чем вызов Task.Run(...).AsAsyncOperation, если требуется передать IAsyncOperation (или IAsyncAction) какому-то другому коду.

На самом деле асинхронность не является необязательной при написании приложений WinRT. В большинстве случаев платформа не даст возможности писать синхронный код для ввода-вывода. Разумеется, вы *можете* сделать всю работу самостоятельно, но применение средств C# 5 значительно упрощает использование WinRT. Я уверен, что не случайно язык усилился асинхронностью почти одновременно с выходом WinRT. Здесь в Microsoft не просто подвели итоги; это то, каким образом вы *будете* писать приложения Windows Store на C#.

15.7. Резюме

Надеюсь, что более сложные и глубокие разделы этой главы не помешали оценить элегантность асинхронных средств C# 5. Возможность написания эффективного асинхронного кода в рамках более знакомой модели выполнения является крупным шагом вперед, и я уверен в том, что он будет способен к преобразованиям, после того, как

хорошо усвоится. У меня был опыт проведения презентаций по асинхронности, когда многие разработчики очень легко запутывались, когда впервые сталкивались с этим средством и пробовали его использовать. Оно полностью поддается пониманию, и не допускайте, чтобы оно отталкивало вас. Надеюсь, что эта глава помогла вам получить ответы хотя бы на некоторые вопросы. Кроме того, доступен большой объем документации, и, конечно же, многие люди на Stack Overflow готовы оказать вам помощь.

Говоря о других ресурсах, я должен подчеркнуть, что пытался здесь раскрыть в основном *языковые* аспекты асинхронности в соответствии с остальными главами книги. Тем не менее, кроме знания языковых средств асинхронная разработка предполагает и многое другое, и я настоятельно рекомендую почитать все, что удастся найти по библиотеке TPL. Даже если вы пока не можете применять C# 5, когда вы имеете дело с .NET 4, можете начать использовать `Task<T>` в качестве чистой модели для асинхронных операций. Всякий раз, когда вы стремитесь прибегнуть к низкоуровневому методу класса `Thread`, подумайте о том, не может ли библиотека TPL предложить высокоуровневую абстракцию, которая позволит достичь той же самой цели более легким путем.

Подведем итог: асинхронные функции — это краеугольный камень в C# 5. Хотя мы рассмотрели еще не все. Существует еще пара крошечных средств, которые я просто обязан раскрыть, прежде чем завершить настоящее издание.

Глава 16

Дополнительные средства C# 5 и заключительные размышления

В этой главе...

- Изменения в захваченных переменных
- Атрибуты информации о вызывающем компоненте
- Заключительные размышления

В C# 2 было множество мелких несопоставимых друг с другом средств наряду с крупными средствами. В C# 3 присутствовало несколько небольших средств, направленных на построение LINQ. Даже в C# 4 имелись относительно мелкие средства, которые стоили детального рассмотрения.

По большому счету в C# 5 других средств кроме асинхронности нет. Есть только два крошечных дополнения. Команда проектировщиков C# всегда оценивает затраты на внедрение средства (в терминах проектирования, реализации, тестирования, документирования и обучения разработчиков) и сравнивает их с получаемой выгодой. Я уверен, что было много невыполненных запросов на средства, которые команда *не прочь* была бы удовлетворить, но, по-видимому, сравнительная выгода от этих средств оказалась не настолько большой, чтобы они прошли отборочный тур.

Первое изменение является в большей степени не *средством*, а исправлением ошибки, ранее существовавшей в проектном решении языка.

16.1. Изменения в захваченных переменных внутри циклов `foreach`

В разделе 5.5.5 я давал предупреждение о коде, в котором внутри цикла `foreach` используется анонимная функция (обычно лямбда-выражение), захватывающая переменную цикла. В листинге 16.1 приведен простой пример такого кода, который выглядит так, как будто бы выводит на консоль x, затем y и, наконец, z.

Листинг 16.1. Использование захваченных переменных итерации

```
string[] values = { "x", "y", "z" };
var actions = new List<Action>();

foreach (string value in values)
{
  actions.Add(() => Console.WriteLine(value));
}

foreach (Action action in actions)
{
  action();
}
```

В версиях C# 3 и C# 4 этот код *в действительности* выводит на консоль z три раза — переменная цикла (`value`) захватывается лямбда-выражением, и теоретически существует лишь один "экземпляр" переменной, который изменяет значение на каждой итерации цикла. Все три делегата ссылаются на одну и ту же переменную, и ко времени, когда они, наконец, выполнятся, значением этой переменной окажется z. Это не ошибка в реализации компилятора; таким было *задано* поведение языка в спецификации.

В версии C# 5 язык работает так, как он него можно было ожидать: каждая итерация цикла фактически вводит отдельную переменную. Каждый делегат будет ссылаться на отличающуюся переменную, которая получает значение из данной итерации цикла.

Больше сказать об этом средстве нечего — оно действительно представляет собой просто исправление той области языка, которая создавала проблемы у многих разработчиков. (Возможно, вы были бы удивлены, узнав, сколько вопросов это вызвало на сайте Stack Overflow.)

Тем не менее, я хочу дать одно предупреждение: если вы находитесь в довольно необычном положении, когда приходится писать код, который необходимо компилировать с применением разных версий компилятора C#, то должны осознавать, что поведение будет меняться. При компиляции кода в листинге 16.1 не выдается никаких предупреждений ни в одной из версий C# — *в C# 5 поведение просто молча изменяется*. Будьте осторожны и удостоверьтесь, что у вас на этот случай предусмотрены модульные тесты!

Наконец, рассмотрим финальное средство.

16.2. Атрибуты информации о вызывающем компоненте

Некоторые средства носят очень общий характер — лямбда-выражения, неявно типизированные локальные переменные, обобщения и тому подобное. Другие средства более специфичны — LINQ на самом деле предназначен для запрашивания данных в

той или иной форме, хотя он рассчитан на обобщение множества разных источников данных. Финальное средство C# 5 отличается исключительной направленностью: есть два существенных сценария использования (один очевидный, а другой чуть менее очевидный), и я действительно не ожидаю, что данное средство будет особо применяться за рамками этих ситуаций.

16.2.1. Базовое поведение

В .NET 4.5 появились три новых атрибута: `CallerFilePathAttribute`, `CallerLineNumberAttribute` и `CallerMemberNameAttribute`, которые все находятся в пространстве имен `System.Runtime.CompilerServices`. Как и с другими атрибутами, при их применении можно не указывать суффикс *`Attribute`*, и поскольку это наиболее распространенный способ использования атрибутов, в оставшейся части главы их имена будут сокращаться.

Все три атрибута могут применяться только к параметрам, и они полезны, лишь когда применяются к *необязательным* параметрам. Идея проста: если вызывающий компонент не предоставляет аргумент, то для заполнения аргумента компилятор использует текущий файл, номер строки либо имя члена вместо того, чтобы брать нормальное стандартное значение. Если вызывающий компонент *все-таки* предоставляет аргумент, то компилятор оставляет именно его.

В листинге 16.2 показаны примеры обоих случаев.

Листинг 16.2. Использование атрибутов информации о вызывающем компоненте обычным и необычным образом

```
static void ShowInfo([CallerFilePath] string file = null,
                     [CallerLineNumber] int line = 0,
                     [CallerMemberName] string member = null)
{
  Console.WriteLine("{0}:{1} - {2}", file, line, member);
}
...
ShowInfo();                                    <-- Компилятор заполняет всю информацию
ShowInfo("LiesAndDamnedLies.java", -10);       <-- Компилятор заполняет только имя
```

Вывод кода из листинга 16.2 будет выглядеть приблизительно так:

```
c:\Users\Jon\Code\Chapter16\CallerInfoDemo.cs:21 - Main
LiesAndDamnedLies.java:-10 - Main
```

Разумеется, обычно вы не будете предоставлять поддельное значение каждому из этих аргументов, но удобно иметь возможность передавать значение явно, особенно если нужно зарегистрировать в журнале вызывающий компонент *текущего* метода, используя те же самые атрибуты.

Имя члена работает для всех членов обычно в очевидной манере, со следующими довольно-таки предсказуемыми специальными именами:

- Статический конструктор: `.cctor`
- Конструктор: `.ctor`
- Финализатор: `Finalize`

Имя, применяемое как часть вызова метода во время выполнения инициализатора поля, является именем этого поля.

Существуют две ситуации, в которых информация о вызывающем компоненте *не* заполняется. Первая из них — инициализация атрибутов; в листинге 16.3 представлен пример атрибута, который, как можно было *ожидать*, получит имя члена, к которому он был применен, но, к сожалению, в этом случае компилятор ничего не заполняет автоматически.

Листинг 16.3. Попытка использования атрибутов информации о вызывающем компоненте в объявлении атрибута

```
[AttributeUsage(AttributeTargets.All)]
public class MemberDescriptionAttribute : Attribute
{
  public MemberDescriptionAttribute([CallerMemberName]
    string member = null)
  {
    Member = member;
  }
  public string Member { get; set; }
}
```

Это определенно может быть полезно. Я наблюдал ситуации, когда разработчики находили атрибуты через рефлексию, но должны были заполнять собственную структуру данных для поддержки отображения между именем члена и атрибутом, что могло бы делаться компилятором автоматически.

Отказ от поддержки атрибутами динамической типизации объяснить гораздо легче. В листинге 16.4 демонстрируется разновидность применения, которая, к сожалению, не работает.

Листинг 16.4. Попытка использования атрибутов информации о вызывающем компоненте с динамическим вызовом

```
class TypeUsedDynamically
{
  internal void ShowCaller([CallerMemberName] string caller = "Unknown")
  {
    Console.WriteLine("Called by: {0}", caller);
  }
}
...
dynamic x = new TypeUsedDynamically();
x.ShowCaller();
```

Код из листинга 16.4 выведет на консоль только `Called by: Unknown`, как если бы атрибут отсутствовал. Хотя это может показаться разочаровывающим, подумайте об альтернативе: чтобы обеспечить работоспособность, компилятору пришлось бы внедрять имя члена, имя файла и номер строки в каждый динамический вызов, который, *возможно*, требовал бы этой информации. В целом, я думаю, что затраты перевесили бы преимущества для большинства разработчиков.

16.2.2. Регистрация в журнале

Самый очевидный случай, когда информация о вызывающем компоненте полезна, связан с записью в журнальный файл. Ранее при ведении журнала обычно строилась трассировка стека (с применением `System.Diagnostics.StackTrace`, например) для выяснения, откуда поступила информация, заносимая в журнал. Как правило, это скрывалось из виду инфраструктур регистрации в журнале, однако присутствовало — и было неуклюжим. Потенциально оно служило источником проблем с производительностью, и являлось хрупким перед лицом встраивания, выполняемого JIT-компилятором.

Довольно легко увидеть, каким образом инфраструктура регистрации могла бы воспользоваться новым средством, чтобы позволить информации об одном лишь вызывающем компоненте заноситься в журнал с минимальными затратами, даже сохраняя номера строк и имена членов для сборок с отброшенной отладочной информацией и даже после обфускации. Конечно, это не поможет в случаях, когда необходимо фиксировать в журнале полную трассировку стека, но в любом случае это не отбирает у вас возможности сделать нужное.

На момент написания этих строк мне не было известно ни одной инфраструктуры регистрации в журнале, которая бы воспользовалась преимуществами нового средства; для начала, это требует компиляции для целевой платформы .NET 4.5 или же инфраструктуры с явно объявленными атрибутами, как будет показано в разделе 16.2.4. Но должно быть проще написать собственные классы оболочек, которые работают с любой предпочитаемой инфраструктурой регистрации и предоставляют информацию о вызывающем компоненте. Я уверен, что со временем инфраструктуры наверстают упущенное и станут предлагать такую функциональность в готовом виде.

16.2.3. Реализация интерфейса `INotifyPropertyChanged`

Менее очевидный случай применения одного из атрибутов, `[CallerMemberName]`, может быть *совершенно* очевидным, если приходится часто реализовывать интерфейс `INotifyPropertyChanged`.

Этот интерфейс очень прост — он содержит единственное событие типа `PropertyChangedEventHandler`. Событие имеет тип делегата со следующей сигнатурой:

```
public delegate void PropertyChangedEventHandler(Object sender,
                                                  PropertyChangedEventArgs e)
```

В свою очередь, тип `PropertyChangedEventArgs` имеет единственный конструктор:

```
public PropertyChangedEventArgs(string propertyName)
```

Типичная реализация интерфейса `INotifyPropertyChanged` до выхода C# 5 может выглядеть примерно так, как показано в листинге 16.5.

Листинг 16.5. Реализация интерфейса `INotifyPropertyChanged` в старом стиле

```
class OldPropertyNotifier : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;

  private int firstValue;
  public int FirstValue
  {
    get { return firstValue; }
```

```
    set
    {
      if (value != firstValue)
      {
        firstValue = value;
        NotifyPropertyChanged("FirstValue");
      }
    }
  }
  // Другие свойства с тем же самым шаблоном
  private void NotifyPropertyChanged(string propertyName)
  {
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
    {
      handler(this, new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

Цель вспомогательного метода — избежать необходимости в помещении проверки на предмет `null` внутрь каждого свойства. Разумеется, можно было бы сделать его расширяющим методом, устранив повторение в каждой реализации.

Код не столько многословный (в версии C# 5 это не изменилось), сколько хрупкий. Проблема в том, что имя свойства (`FirstValue`) указано как строковый литерал, и при проведении рефакторинга с целью переименования свойства легко забыть изменить также и этот строковый литерал. Если повезет, инструменты и тесты помогут обнаружить ошибку, но все равно реализация выглядит совершенно неуклюжей.

В C# 5 большая часть кода остается той же самой, но вы можете предложить компилятору самостоятельно заполнить имя свойства, применив во вспомогательном методе атрибут `CallerMemberName`, как демонстрируется в листинге 16.6.

Листинг 16.6. Реализация интерфейса `INotifyPropertyChanged` с использованием информации о вызывающем компоненте

```
// Внутри средства установки
if (value != firstValue)
{
  firstValue = value;
  NotifyPropertyChanged();
}
...
void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
{
  // В точности тот же самый код, что и ранее
}
```

В листинге 16.6 приведены только те разделы кода, которые изменились — все очень просто. Теперь в случае изменения имени свойства компилятор будет применять новое имя. Улучшение, конечно, не шокирующее, но, без сомнения, приятное.

16.2.4. Использование атрибутов информации о вызывающем компоненте без .NET 4.5

Подобно расширяющим методам, атрибуты информации о вызывающем компоненте лишь позволяют вам предложить компилятору слегка вмешаться в ваш код во время процесса компиляции. Они не используют какую-либо информацию, которую бы вы не смогли предоставить сами — просто вы должны соблюдать аккуратность, делая это. Как и в случае расширяющих методов, атрибуты можно применять и при ориентации на более раннюю версию .NET, нежели та, которая *действительно* содержит атрибуты — нужно просто объявить их самостоятельно. Это сводится всего лишь к копированию объявления из MSDN. Сами атрибуты не имеют каких-либо параметров, поэтому придется только предоставить пустое тело для объявлений классов, которые по-прежнему должны быть в пространстве имен `System.Runtime.CompilerServices`.

Компилятор C# будет трактовать ваши пользовательские атрибуты точно таким же образом, как он трактовал бы подлинные атрибуты из .NET 4.5. Недостаток такого подхода заключается в том, что вы столкнетесь с проблемами, если когда-либо начнете компилировать этот же код с .NET 4.5. Чтобы не запутать компилятор, вручную созданные атрибуты в таком случае придется удалить.

Если вы используете .NET 4, Silverlight 4 или Silverlight 5 либо Windows Phone 7.5, то другой вариант предполагает применение пакета NuGet под названием `Microsoft.Bcl`. Он предоставляет все эти атрибуты, а также ряд других удобных типов, к которым вы могли привыкнуть.

Вот и все — мы завершили рассмотрение C# 5.

16.3. Заключительные размышления

Первые два издания этой книги завершались главой, посвященной будущему, как я его представлял на время их написания. Если у вас есть одно из изданий (а то и оба!), можете заглянуть в них и тихо посмеяться. Я не думаю, что писал тогда что-то возмутительно неправильное, но вполне понятно, что я имел лишь смутное представление о том, насколько многое могло измениться всего за пару лет.

Я также хотел бы отметить, что совершенно не представлял себе, что могло появиться в C# 4 или C# 5 до того, как эти версии были анонсированы Microsoft. И динамическая типизация, и асинхронные функции стали для меня крупными неожиданностями. Я имел счастье представить свои идеи относительно C# 5 на конференции, в которой принимали участие члены команды проектировщиков C#, и я чрезвычайно признателен им, что они пошли своим путем. На тот случай, если мне не удалось ясно выразиться, подход `async/await` — это *краеугольное* по важности средство, и оно далеко выходит за рамки того, что я был в состоянии придумать.

Что еще осталось в резерве отрасли? Больше мобильности, больше сенсорного ввода, больше распределенных облачных служб, возможность дополненной реальности — сейчас на все это можно смело делать ставки. Но если они окажутся наиболее движущими силами в индустрии к концу 2014 года, я буду весьма разочарован. Лучшие вещи в вычислениях, кажется, приходят из ниоткуда — конечно, после многих лет упорного труда вовлеченных в это людей — и удивляют всех.

Того же рода размышления применимы и к C#. Я по-прежнему имею свой список пожеланий относительно небольших средств, и может быть, версия C# 6 будет выпуском, приводящим все в порядок, с множеством мелких средств, а не крупных, как это

было в прошлом. Возможно, язык будет дополнен возможностью расширения, позволив другим разработчикам самостоятельно создавать нужные им небольшие средства. А может быть новое убийственное средство будет чем-то таким, что я даже не могу себе вообразить, для чего оно может понадобиться.

Безусловно, команды проектировщиков C# и .NET не сидят, сложа руки. Даже если оставить в стороне C# 5 и всю работу, требуемую для интеграции .NET в пользовательский интерфейс Windows 8, мы знаем *один* проект, над которым они интенсивно трудятся: Roslyn. Получивший свое имя из-за расположения офиса Эрика Липперта, когда он работал над проектом, Roslyn — это другое название идеи "компилятора как службы", о которой так долго говорили. Проект Roslyn предоставит API-интерфейс, который разработчики смогут использовать для анализа кода C# (или VB), модификации его программным образом, компиляции его в код IL и т.д. Я подозреваю, что потребность в нем возникнет лишь у *относительно* немногих разработчиков, но те, кому оно понадобится, будут безмерно рады, и они смогут создавать замечательные вещи для всех остальных. Только подумайте о возможности написания собственных инструментов для рефакторинга, более сложного анализа соглашений кода, генерации кода и тому подобного — и все это с помощью API-интерфейса, спроектированного быть настолько мощным и высокопроизводительным, чтобы служить механизмом для будущих выпусков Visual Studio. Пожалуй, более важно то, что Roslyn предоставляет команде проектировщиков C# площадку, на которой относительно легко реализовывать новые средства. Возможно, в будущем они станут даже еще более смелыми и амбиционными!

Хотя одну вещь я могу утверждать с достаточной степенью уверенности: я продолжу наслаждаться написанием, разговорами и применением C# на протяжении определенного времени независимо от того, будет язык развиваться или нет. Мне трудно поверить, что программирование станет *менее* интересным в следующем десятилетии.

Как и в предыдущих изданиях, я советую делать фантастические вещи. Напишите невероятно ясный код, с которым понравится работать вашим коллегам. Разработайте Будущую Крупную Вещь в мире открытого кода. Помогайте другим разработчикам на сайте Stack Overflow. Поговорите в группах пользователей, на конференциях, в кругу друзей и с теми, кто готов послушать о предмете вашего страстного увлечения. Я желаю вам большой удачи во всем, что бы вы ни предпринимали, и надеюсь, что эта книга хотя бы немного поспособствовала достижению ваших амбиционных целей.

Приложение А

Стандартные операции запросов LINQ

В LINQ существует множество стандартных операций запросов, часть из которых поддерживается напрямую в выражениях запросов C#, а другие должны вызываться вручную как нормальные методы. Некоторые стандартные операции запросов демонстрировались в главах книги, и все они перечислены в настоящем приложении.

В большинстве примеров используются следующие две последовательности:

```
string[] words = {"zero", "one", "two", "three", "four"};
int[] numbers = {0, 1, 2, 3, 4};
```

Для полноты в это приложение включены операции, которые вы уже видели, хотя в большинстве случаев глава 11 содержит больше подробностей, чем представлено здесь.

Описанное поведение относится к LINQ to Objects; другие поставщики могут работать по-другому. Для каждой операции указано, какое выполнение она применяет — отложенное или немедленное. Если операция использует отложенное выполнение, также указывается, организует она поток или буферизирует свои данные.

Недавно я заново реализовал LINQ to Objects в проекте под названием Edulinq и документировал каждую отдельную операцию, описав возможности оптимизации, ленивой оценки и т.д. Дополнительные детали по LINQ to Objects можно узнать на домашней странице проекта Edulinq по адресу `http://edulinq.googlecode.com`.

A.1. Агрегирование

Все операции агрегирования (табл. A.1) в результате дают одиночное значение, а не последовательность. Операции `Average()` и `Sum()` работают либо с последовательностью чисел (любого встроенного числового типа), либо с последовательностью элементов и делегатом для преобразования каждого элемента в один из встроенных числовых типов. Операции `Min()` и `Max()` имеют перегруженные версии для числовых типов, но могут также работать с любыми последовательностями, либо применяя стандартный компаратор для типа элементов, либо используя делегат для преобразования. Операции `Count()` и `LongCount()` являются эквивалентными друг другу и просто имеют раз-

ные возвращаемые типы. Для обеих определены перегруженные версии — одна просто подсчитывает длину последовательности, а другая получает предикат и подсчитывает только элементы, удовлетворяющие этому предикату.

Таблица А.1. Примеры операций агрегирования

Выражение	Результат
`numbers.Sum()`	`10`
`numbers.Count()`	`5`
`numbers.Average()`	`2`
`numbers.LongCount(x => x % 2 == 0)`	`3` (как `long`; имеется три четных числа)
`words.Min(word => word.Length)`	`3` (`"one"` и `"two"`)
`words.Max(word => word.Length)`	`5` (`"three"`)
`numbers.Aggregate("seed", (current, item) => current + item, result=> result.ToUpper())`	`"SEED01234"`

Самая общая операция агрегирования (показанная в нижней строке табл. А.1) называется `Aggregate()`. Все другие операции агрегирования могут быть выражены в форме вызовов `Aggregate()`, хотя это требует относительно больших усилий. Базовая идея заключается в том, что всегда есть "результат до настоящего времени", стартующий с исходного начального значения. Делегат агрегирования применяется к каждому элементу входной последовательности; делегат получает результат до настоящего времени и входной элемент и производит следующий результат. В качестве финального дополнительного шага применяется преобразование из результата агрегирования в возвращаемое значение метода. При необходимости данное преобразование может дать в результате другой тип. Это не настолько сложно, как может показаться, но вряд ли вы будете использовать его часто.

Все операции агрегирования применяют немедленное выполнение. Перегруженная версия `Count()`, которая не использует предикат, оптимизирована для реализации интерфейсов `ICollection` и `ICollection<T>`; в этой ситуации она будет применять свойство `Count` коллекции, не читая какие-либо данные[1].

А.2. Конкатенация

Существует единственная операция конкатенации: `Concat()` (табл. А.2). Как и можно было ожидать, она работает с двумя последовательностями и возвращает одну последовательность, состоящую из всех элементов первой последовательности, за которыми следуют все элементы второй последовательности. Две входных последовательности должны быть одного и того же типа, выполнение является отложенным, а все данные организуются в поток.

[1] Такого сокращения для `LongCount()` не предусмотрено. Лично я никогда не видел, чтобы этот метод использовался в LINQ to Objects.

Таблица А.2. Пример `Concat()`

Выражение	Результат
`numbers.Concat(new[] {2, 3, 4, 5, 6})`	`0, 1, 2, 3, 4, 2, 3, 4, 5, 6`

А.3. Преобразование

Операции преобразования используются довольно широко, но встречаются парами.

В примерах в табл. А.3 применяются две дополнительных последовательности для демонстрации операций `Cast()` и `OfType()`:

```
object[] allStrings = {"These", "are", "all", "strings"};
object[] notAllStrings = {"Number", "at", "the", "end", 5};
```

Таблица А.3. Примеры преобразований

Выражение	Результат
`allStrings.Cast<string>()`	`"These", "are", "all", "strings"` (как `IEnumerable<string>`)
`allStrings.OfType<string>()`	`"These", "are", "all", "strings"` (как `IEnumerable<string>`)
`notAllStrings.Cast<string>()`	Во время прохода по последовательности в точке отказа преобразования генерируется исключение
`notAllStrings.OfType<string>()`	`"Number", "at", "the", "end"` (как `IEnumerable<string>`)
`numbers.ToArray()`	`0, 1, 2, 3, 4` (как `int[]`)
`numbers.ToList()`	`0, 1, 2, 3, 4` (как `List<int>`)

`words.ToDictionary(w => w.Substring(0, 2))`	Содержимое словаря:
	`"ze": "zero"`
	`"on": "one"`
	`"tw": "two"`
	`"th": "three"`
	`"fo": "four"`

`// Ключом является первый символ слова`
`words.ToLookup(word => word[0])`
	Содержимое списка поиска:
	`'z': "zero"`
	`'o': "one"`
	`'t': "two", "three"`
	`'f': "four"`

`words.ToDictionary(word => word[0])`	Исключение: можно иметь только одну запись для ключа, поэтому происходит отказ на `'t'`

Операции `ToArray()` и `ToList` не требуют особых пояснений: они читают целую последовательность в память и возвращают ее либо в виде массива, либо как `List<T>`. Обе операции используют немедленное выполнение.

Операции `Cast()` и `OfType()` преобразуют нетипизированную последовательность в типизированную, либо генерируя исключение (`Cast()`), либо игнорируя (`OfType()`)

элементы входной последовательности, которые не могут быть неявно преобразованы в тип элементов выходной последовательности с применением распаковывающего или ссылочного преобразования. Они также могут использоваться для преобразования типизированных последовательностей в последовательности более специализированных типов, например, IEnumerable<object> в IEnumerable<string>. Обе операции применяют отложенное выполнение и организуют поток для своих входных данных.

Операции ToDictionary() и ToLookup() принимают делегат, который получает ключ для отдельного элемента. Операция ToDictionary() возвращает словарь, отображающий ключ на тип элемента, а операция ToLookup() — соответствующим образом типизированный ILookup<,>. Список поиска подобен словарю, в котором значение, ассоциированное с ключом, является не одиночным элементом, а последовательностью элементов. Списки поиска обычно используются, когда в результате выполнения нормальной операции ожидаются дублированные ключи, в то время как дублированный ключ приведет к генерации исключения в ToDictionary(). Более сложные перегруженные версии обоих методов позволяют указывать для сравнения ключей специальную реализацию IEqualityComparer<T>, и к каждому элементу перед его помещением в словарь или список поиска применяется делегат преобразования. Оба метода используют немедленное выполнение.

Есть еще две операции, для которых примеры не приводились: AsEnumerable() и AsQueryable(). Они не воздействуют на результаты очевидным образом, поэтому продемонстрировать их здесь невозможно. Взамен они оказывают влияние на то, каким способом выполняется запрос. Queryable.AsQueryable() — это расширяющий метод в IEnumerable, который возвращает IQueryable (оба типа являются обобщенными или необобщенными в зависимости от того, какая перегруженная версия была выбрана). Если последовательность IEnumerable, на которой производится вызов, уже представляет собой IQueryable, возвращается та же самая ссылка; в противном случае создается оболочка вокруг первоначальной последовательности. Эта оболочка позволяет применять все нормальные расширяющие методы Queryable с передачей деревьев выражений, но когда запрос выполняется, дерево выражения компилируется в обычный код IL и выполняется напрямую, используя метод LambdaExpression.Compile(), который был показан в разделе 9.3.2.

Enumerable.AsEnumerable() — это расширяющий метод в IEnumerable<T>, который имеет тривиальную реализацию, просто возвращающую ссылку, на которой осуществлялся вызов. Никакие оболочки не задействуются — лишь возвращается та же самая ссылка. Это вынуждает применять в последующих операциях LINQ расширяющие методы Enumerable. Взгляните на следующие выражения запросов:

```
// Фильтровать пользователей в базе данных с использованием LIKE
from user in context.Users
where user.Name.StartsWith("Tim")
select user;

// Фильтровать пользователей в памяти
from user in context.Users.AsEnumerable()
where user.Name.StartsWith("Tim")
select user;
```

Второе выражение запроса приводит к тому, что типом на этапе компиляции источника становится IEnumerable<User> вместо IQueryable<User>, поэтому вся обработка происходит в памяти, а не в базе данных. Компилятор будет использовать расширяющие методы Enumerable (принимающие параметры в виде делегатов) вместо таких

методов Queryable (принимающих параметры в виде деревьев выражений). Обычно желательно выполнять как можно больше обработки в SQL, но когда есть трансформации, требующие локального кода, иногда приходится заставлять LINQ применять подходящие расширяющие методы Enumerable. Разумеется, это не является специфичным для баз данных; прием с вынуждением заключительной части запроса использовать Enumerable применим также и к другим поставщикам, если они основаны на IQueryable или чем-то аналогичном.

А.4. Операции элементов

Это еще один набор операций запросов, которые сгруппированы парами (табл. А.4). На этот раз все пары работают одинаково. Имеется простая версия, которая выбирает одиночный элемент, когда это возможно, или генерирует исключение, если указанный элемент не существует, и версия с суффиксом OrDefault в конце имени. Все операции используют немедленное выполнение.

Таблица А.4. Примеры выбора одиночного элемента

Выражение	Результат
`words.ElementAt(2)`	`"two"`
`words.ElementAtOrDefault(10)`	`null`
`words.First()`	`"zero"`
`words.First(w => w.Length == 3)`	`"one"`
`words.First(w => w.Length == 10)`	Исключение: подходящие элементы отсутствуют
`words.FirstOrDefault (w =>w.Length == 10)`	`null`
`words.Last()`	`"four"`
`words.Single()`	Исключение: обнаружено более одного элемента
`words.SingleOrDefault()`	Исключение: обнаружено более одного элемента
`words.Single(word => word.Length == 5)`	`"three"`
`words.Single(word => word.Length == 10)`	Исключение: подходящие элементы отсутствуют
`words.SingleOrDefault (w =>w.Length == 10)`	`null`

Имена операций понять легко: First() и Last() возвращают, соответственно, первый и последний элементы последовательности, генерируя исключение InvalidOperationException, если последовательность пуста. Операция Single() возвращает единственный элемент в последовательности, генерируя исключение, если последовательность пуста или содержит более одного элемента. Операция ElementAt() возвращает специфичный элемент по индексу — например, пятый элемент. Исключение

`ArgumentOutOfRangeException` генерируется, если индекс отрицательный или превышает действительное количество элементов в коллекции. Кроме того, для всех операций кроме `ElementAt()` существуют перегруженные версии для предварительной фильтрации последовательности — скажем, операция `First()` может возвращать первый элемент, который соответствует заданному условию.

Версии `OrDefault` этих методов подавляют только что описанные исключения (взамен возвращая стандартное значение для типа элементов) кроме одного случая: `SingleOrDefault()` будет возвращать стандартное значение, если последовательность пуста, но если в ней находится более одного элемента, метод по-прежнему генерирует исключение, как и `Single()`. Это рассчитано на ситуации, при которых, когда все идет нормально, последовательность будет иметь ноль или один элемент. Чтобы справиться с последовательностями, которые могут содержать больше элементов, применяйте вместо этого `FirstOrDefault()`.

Все перегруженные версии, которые не имеют параметра с предикатом, оптимизированы для экземпляров `IList<T>`, т.к. они могут получить доступ к нужному элементу без выполнения итерации. Когда задействован предикат, оптимизация отсутствует — это не имело бы смысла в большинстве вызовов, хотя была бы ощутимая разница, если бы поиск *последнего* подходящего элемента происходил путем перемещения с конца списка по направлению к его началу. На момент написания этих строк данный случай не был оптимизирован, но в будущей версии это может измениться.

A.5. Эквивалентность

Есть только одна стандартная операция эквивалентности: `SequenceEqual()` (табл. А.5). Она просто поэлементно сравнивает две последовательности на предмет эквивалентности, включая порядок. Например, последовательность 0, 1, 2, 3, 4 не эквивалентна последовательности 4, 3, 2, 1, 0. Перегруженная версия позволяет использовать при сравнении элементов специальную реализацию `IEqualityComparer<T>`. Возвращаемое значение имеет булевский тип, а операция применяет немедленное выполнение.

Таблица А.5. Примеры определения эквивалентности последовательностей

Выражение	Результат
`words.SequenceEqual (new[]{"zero","one","two","three","four"})`	`True`
`words.SequenceEqual (new[]{"ZERO","ONE","TWO","THREE","FOUR"})`	`False`
`words.SequenceEqual (new[]{"ZERO","ONE","TWO","THREE","FOUR"}, StringComparer.OrdinalIgnoreCase)`	`True`

И снова в LINQ to Objects не хватает трюка в плане оптимизации: если обе последовательности располагают эффективным способом извлечения их счетчиков, имело бы смысл проверять, равны ли они, прежде чем приступать к сравнению самих элементов. Текущая реализация просто проходит по обеим последовательностям до тех пор, пока не достигнет конца или не обнаружит неэквивалентную пару элементов.

А.6. Генерация

Из всех генерирующих операций (табл. А.6) только одна действует на существующей последовательности: `DefaultIfEmpty()`. Она возвращает либо исходную последовательность, если она не пуста, либо последовательность с единственным элементом в противном случае. Этот элемент обычно является стандартным значением для типа последовательности, но есть перегруженная версия, которая позволяет указать используемое значение.

Три других генерирующих операции — это просто статические методы в `Enumerable`.

- `Range()` генерирует последовательность целых чисел на основе параметров, указывающих первое значение и общее количество значений.
- `Repeat()` генерирует последовательность любого типа, повторяя указанное одиночное значение заданное количество раз.
- `Empty()` генерирует пустую последовательность любого типа.

Все генерирующие операции применяют отложенное выполнение и организуют поток для своего вывода — другими словами, они не просто заранее заполняют коллекцию и возвращают ее. Исключением является операция `Empty()`, которая возвращает пустой массив нужного типа. Пустой массив полностью неизменяем, поэтому тот же самый массив может возвращаться при каждом вызове для того же самого типа элементов.

Таблица А.6. Примеры генерации

Выражение	Результат
`numbers.DefaultIfEmpty()`	`0, 1, 2, 3, 4`
`new int[0].DefaultIfEmpty()`	`0` (внутри `IEnumerable<int>`)
`new int[0].DefaultIfEmpty(10)`	`10` (внутри `IEnumerable<int>`)
`Enumerable.Range(15, 2)`	`15, 16`
`Enumerable.Repeat(25, 2)`	`25, 25`
`Enumerable.Empty<int>()`	Пустая последовательность `IEnumerable<int>`

А.7. Группирование

Существуют две операции группирования, но одной из них является `ToLookup()`, которую вы уже видели в разделе А.3 как операцию преобразования. Остается только операция `GroupBy()`, которую мы исследовали в разделе 11.6.1 в форме конструкции `group...by` внутри выражений запросов. Она использует отложенное выполнение, но буферизирует свои результаты: когда вы начинаете проходить по результирующей последовательности групп, потребляется полная входная последовательность.

Результатом операции `GroupBy()` является последовательность соответствующим образом типизированных элементов `IGrouping<,>`. Каждый элемент имеет ключ и последовательность элементов, относящихся к этому ключу. Во многих отношениях это лишь другой способ взгляда на список поиска — вместо произвольного доступа к группам по ключу группы перебираются по очереди. Порядок, в котором группы возвраща-

ются, является порядком обнаружения соответствующих им ключей. Внутри группы порядок элементов совпадает с их порядком в исходной последовательности.

Операция `GroupBy()` имеет обескураживающее количество перегруженных версий, позволяя указывать не только, как выводить ключи из элемента (что требуется всегда), но также дополнительно следующие аспекты.

- Способ сравнения ключей.
- Проекция из исходного элемента в элемент внутри группы.
- Проекция, которая получает ключ и последовательность соответствующих элементов. Общим результатом в этом случае будет последовательность элементов результирующего типа проекции.

В табл. А.7 приведены примеры второго и третьего вариантов, а также простейшая форма операции. Специальные сравнения ключей несколько многословнее для демонстрации, но работают они вполне очевидным образом.

Таблица А.7. Примеры `GroupBy()`

Выражение	Результат
`words.GroupBy(word => word.Length)`	Ключ: 4; последовательность: `"zero", "four"` Ключ: 3; последовательность: `"one", "two"` Ключ: 5; последовательность: `"three"`

`words.GroupBy`
`  (word => word.Length,  // Ключ`
`   word => word.ToUpper()  // Элемент группы`
`  )`
	Ключ: 4; последовательность: `"zero", "four"`
	Ключ: 3; последовательность: `"one", "two"`
	Ключ: 5; последовательность: `"three"`

`// Проецировать каждую пару (ключ, группа)`
`// в строку`
`words.GroupBy`
`   (word => word.Length,`
`    (key, g) => key + ": " + g.Count())`
	`"4: 2", "3: 2", "5: 1"`

Согласно моему опыту, последний вариант применяется редко.

А.8. Соединения

Есть две операции соединения, `Join()` и `GroupJoin()`, которые вы видели в разделе 11.5 при использовании, соответственно, конструкций `join` и `join...into` выражений запросов. Каждый метод принимает несколько параметров: две последовательности, селектор ключей для каждой последовательности, проекцию для применения к каждой согласованной паре элементов и необязательное сравнение ключей.

Для операции `Join()` проекция берет один элемент из каждой последовательности и строит результат; для операции `GroupJoin()` проекция берет элемент из левой последовательности и последовательность соответствующих элементов из правой последовательности. Обе операции используют отложенное выполнение и организуют поток для левой последовательности, но читают правую последовательность полностью, когда запрашивается первый результат.

Для примеров соединений в табл. А.8 будет сопоставляться последовательность имен (`Robin`, `Ruth`, `Bob`, `Emma`) с последовательностью цветов (`Red`, `Blue`, `Beige`, `Green`). При этом просматриваются первые символы в имени и цвете, так что, скажем, `Robin` будет соединяться с `Red`, а `Bob` — с `Blue` и `Beige`.

Обратите внимание, что имени `Emma` не соответствует ни один из цветов — это имя отсутствует во всех результатах первого примера, но присутствует в результате второго примера, с пустой последовательностью цветов.

Таблица А.8. Примеры `Join()`

Выражение	Результат

`names.Join // Левая последовательность`
`  (colors, // Правая последовательность`
`  name => name[0], // Левый селектор ключей`
`  color=>  color[0], // Правый селектор ключей`
`  // Проекция для пар результатов`
`  (name, color) => name + " - " + color`
`  )`
	`"Robin - Red",`
	`"Ruth - Red",`
	`"Bob - Blue",`
	`"Bob - Beige"`

`names.GroupJoin`
`  (colors,`
`  name => name[0],`
`  color => color[0],`
`  // Проекция для пар ключ/последовательность`
`  (name, matches) => name + ": " +`
`  string.Join("/", matches.ToArray())`
`  )`
	`"Robin: Red",`
	`"Ruth: Red",`
	`"Bob: Blue/Beige",`
	`"Emma: "`

А.9. Разделение

Операции разделения либо *пропускают* начальную часть последовательности, возвращая только ее остаток, либо *берут* только начальную часть последовательности, игнорируя остаток. В каждом случае можно или указывать количество элементов в первой части последовательности, или задавать условие — первая часть последовательности продолжается до тех пор, пока условие не станет ложным. После того, как условие оказывается ложным в первый раз, оно заново не проверяется — не играет никакой роли, что последующие элементы в последовательности могут быть подходящими. Все операции разделения применяют отложенное выполнение и организуют поток для своих данных.

Разделение фактически делит последовательность на две индивидуальных части, либо по позиции, либо по предикату. В каждом случае, если выполнить конкатенацию результатов `Take()` или `TakeWhile()` с результатами соответствующей операции `Skip()` или `SkipWhile()`, предоставляя один и тот же аргумент обоим вызовам, будет получена первоначальная последовательность: каждый элемент будет встречаться строго один раз в исходном порядке. Это демонстрируется в табл. А.9.

Таблица А.9. Примеры разделения

Выражение	Результат
`words.Take(2)`	`"zero", "one"`
`words.Skip(2)`	`"two", "three", "four"`
`words.TakeWhile(word => word.Length <= 4)`	`"zero", "one", "two"`
`words.SkipWhile(word => word.Length <= 4)`	`"three", "four"`

А.10. Проецирование

Вы видели две операции проецирования (`Select()` и `SelectMany()`) в главе 11. Операция `Select()` — это простая проекция "один к одному" из исходного элемента в результирующий элемент. Операция `SelectMany()` используется, когда в выражении запроса присутствует несколько конструкций `from`; каждый элемент в исходной последовательности применяется для генерации новой последовательности. Обе операции проецирования (табл. А.10) используют отложенное выполнение.

Таблица А.10. Примеры проецирования

Выражение	Результат
`words.Select(word => word.Length)`	`4, 3, 3, 5, 4`
`words.Select ((word, index) => index.ToString() + ": " +word)`	`"0: zero", "1: one", "2: two", "3: three", "4: four"`
`words.SelectMany (word => word.ToCharArray())`	`'z', 'e', 'r', 'o', 'o', 'n', 'e', 't', 'w', 'o', 't', 'h', 'r', 'e', 'e', 'f', 'o', 'u', 'r'`
`words.SelectMany ((word, index) => Enumerable.Repeat(word, index))`	`"one", "two", "two", "three", "three", "three", "four", "four", "four", "four"`

Существуют дополнительные перегруженные версии, которые в главе 11 не рассматривались. Оба метода имеют перегруженные версии, которые позволяют применять в проекции индекс внутри исходной последовательности, а метод `SelectMany()` либо выравнивает все сгенерированные последовательности в единую последовательность, вообще не включая исходный элемент, либо он использует проекцию, чтобы сгенерировать результирующий элемент для каждой пары элементов. Множество конструкций `from` всегда применяют перегруженную версию, которая принимает проекцию. (Примеры этого довольно многословны и здесь не показаны. За дополнительной информацией обращайтесь в главу 11.)

В .NET 4 появилась новая операция `Zip()`. Она не является официальной стандартной операцией запроса, описанной в MSDN, но в любом случае о ней полезно знать. Операция `Zip()` принимает две последовательности и применяет указанную проекцию к каждой паре: к первым элементам из обеих последовательностей, затем ко вторым

элементам из обеих последовательностей и т.д. Результирующая последовательность завершается, когда заканчивается *любая* из исходных последовательностей. В табл. А.11 приведены два примера использования `Zip()`, в которых задействованы последовательности имен и цветов из раздела А.8. Операция `Zip()` применяет отложенное выполнение и организует поток для своих данных.

Таблица А.11. Примеры `Zip()`

Выражение	Результат

`names.Zip(colors, (x, y) => x + "-" + y)`	`"Robin-Red",`
	`"Ruth-Blue",`
	`"Bob-Beige",`
	`"Emma-Green"`

`// Вторая последовательность останавливается раньше`
`names.Zip(colors.Take(3),`
`(x, y) => x + "-" + y)`
	`"Robin-Red",`
	`"Ruth-Blue",`
	`"Bob-Beige"`

А.11. Квантификаторы

Операции квантификаторов, показанные в табл. А.12, возвращают булевское значение и используют немедленное выполнение.

- `All()` проверяет, удовлетворяют ли все элементы в последовательности заданному предикату.
- `Any()` проверяет, удовлетворяет ли хотя бы один элемент в последовательности заданному предикату, либо есть ли вообще элементы в последовательности (перегруженная версия без параметров).
- `Contains()` проверяет, содержит ли последовательность конкретный элемент с дополнительным указанием применяемого сравнения.

Таблица А.12. Примеры квантификаторов

Выражение	Результат
`words.All(word => word.Length > 3)`	`false` (`"one"` и `"two"` имеют точно три буквы)
`words.All(word => word.Length > 2)`	`true`
`words.Any()`	`true` (последовательность не пуста)
`words.Any(word => word.Length == 6)`	`false` (слова с шестью буквами отсутствуют)
`words.Any(word => word.Length == 5)`	`true` (`"three"` удовлетворяет условию)
`words.Contains("FOUR")`	`false`
`words.Contains("FOUR",` `StringComparer.OrdinalIgnoreCase)`	`true`

Особенно удобной операцией, о которой часто забывают, является `Any()`. Если вы пытаетесь выяснить, содержит ли последовательность какие-то элементы (или элементы, соответствующие предикату), то намного лучше использовать `source.Any(...)`, чем `source.Count(...) > 0`. Операции должны давать одинаковые результаты, но `Any()` может остановиться, как только найдет первый элемент, а `Count()` подсчитает *все* элементы, даже если необходимо лишь знать, что их количество отлично от нуля.

Перегруженная версия `Contains()`, которая не принимает специального сравнения, оптимизирована для случая, когда источник реализует `ICollection<T>`, делегируя работу реализации данного интерфейса. Это означает, что `Enumerable.Contains()` по-прежнему будет быстрее при вызове на `HashSet<T>`, например.

A.12. Фильтрация

Доступны две операции фильтрации — `OfType()` и `Where()`. За подробными сведениями и примерами применения операции `OfType()` обращайтесь в раздел А.3. Операция `Where()` возвращает последовательность, содержащую все элементы, которые соответствуют заданному предикату. Она имеет перегруженную версию, позволяющую предикату учитывать индекс элемента. Требование индекса довольно необычно, и конструкция `where` в выражениях запросов эту перегруженную версию не использует. Операция `Where()` всегда применяет отложенное выполнение и организует поток для своих данных. В табл. А.13 демонстрируется использование обеих перегруженных версий.

Таблица А.13. Примеры фильтрации

Выражение	Результат
`words.Where(word => word.Length > 3)`	`"zero", "three", "four"`

`words.Where`
`((word, index) => index < word.Length)`
	`"zero", // index=0, length=4`
	`"one", // index=1, length=3`
	`"two", // index=2, length=2`
	`"three", // index=3, length=5`
	`// Not "four", index=4, length=4`

A.13. Операции, основанные на множествах

Возможность трактовки двух последовательностей как множеств элементов вполне естественна. Все четыре операции, основанные на множествах, имеют по две перегруженных версии, из которых одна применяет стандартное сравнение эквивалентности для типа элементов, а другая позволяет указывать сравнение в дополнительном параметре. Все они используют отложенное выполнение.

Операция `Distinct()` наиболее проста — она действует на одной последовательности и возвращает новую последовательность с несовпадающими элементами, отбрасывая дубликаты. Другие операции также обеспечивают возвращение только несовпадающих элементов, но имеют дело с двумя последовательностями.

- `Intersect()` возвращает элементы, которые присутствуют в обеих последовательностях.
- `Union()` возвращает элементы, которые находятся в одной или другой последовательности.

- `Except()` возвращает элементы, которые находятся в первой последовательности, но не во второй. (Элементы, которые присутствуют во второй последовательности, но не в первой, *не* возвращаются.)

Примеры применения этих операций в табл. А.14 работают с двумя новыми последовательностями: `abbc` (`"a"`, `"b"`, `"b"`, `"c"`) и `cd` (`"c"`, `"d"`).

Таблица А.14. Примеры операций, основанных на множествах

Выражение	Результат
`abbc.Distinct()`	`"a", "b", "c"`
`abbc.Intersect(cd)`	`"c"`
`abbc.Union(cd)`	`"a", "b", "c", "d"`
`abbc.Except(cd)`	`"a", "b"`
`cd.Except(abbc)`	`"d"`

Все операции, основанные на множествах, используют отложенное выполнение, но особенности буферизации и организации потоков более сложны. Операции `Distinct()` и `Union()` организуют потоки для своих входных последовательностей, тогда как `Intersect()` и `Except()` читают целиком правую входную последовательность, но затем организуют поток для левой входной последовательности способом, похожим на то, как это делают операции соединения. *Все* эти операции хранят множество элементов, которые они уже возвратили, чтобы не включать дубликаты. Это означает, что даже операции `Distinct()` и `Union()` не подходят для последовательностей, которые являются слишком длинными, чтобы уместиться в памяти, если только заранее не известно, что множество несовпадающих элементов будет ограниченным.

А.14. Сортировка

Все операции сортировки вы уже видели ранее: `OrderBy()` и `OrderByDescending()` предоставляют первичную сортировку, а `ThenBy()` и `ThenByDescending()` обеспечивают последующее упорядочение для элементов, которые не были различены в результате первичной сортировки. В каждом случае указывается проекция из элемента в его ключ сортировки и может быть также задано сравнение (между ключами). В отличие от ряда других алгоритмов сортировки в инфраструктуре (таких как `List<T>.Sort()`), упорядочение LINQ является *устойчивым* — другими словами, если два элемента рассматриваются как эквивалентные в плане своих ключей сортировки, они будут возвращаться в порядке, в котором находились в исходной последовательности.

Последняя операция сортировки — это `Reverse()`, которая изменяет порядок последовательности на противоположный. Все операции сортировки (табл. А.15) применяют отложенное выполнение и буферизуют свои данные.

Таблица А.15. Примеры сортировки

Выражение	Результат
`words.OrderBy(word => word)`	`"four", "one", "three", "two", "zero"`
`// Упорядочить слова по второму символу` `words.OrderBy(word => word[1])`	`"zero", "three", "one", "four", "two"`

`// Упорядочить слова по длине;`
`// слова с одинаковой длиной возвращаются`
`// в исходном порядке`
`words.OrderBy(word => word.Length)`
	`"one", "two", "zero", "four", "three"`

`words.OrderByDescending` `(word => word.Length)`	`"three", "zero",s` `"four", "one", "two"`

`// Упорядочить слова по длине и затем`
`// в алфавитном порядке`
`words.OrderBy(word => word.Length)`
`.ThenBy(word => word)`
	`"one", "two", "four", "zero", "three"`

`// Упорядочить слова по длине и затем`
`// в порядке, обратном алфавитному`
`words.OrderBy(word => word.Length)`
`.ThenByDescending(word => word)`
	`"two", "one", "zero", "four", "three"`

`words.Reverse()`	`"four", "three", "two", "one", "zero"`

Приложение Б

Обобщенные коллекции в .NET

В .NET доступно много обобщенных коллекций и со временем их количество растет. В этом приложении раскрыты наиболее важные интерфейсы и классы обобщенных коллекций, о существовании которых необходимо знать. В пространствах имен `System.Collections`, `System.Collections.Specialized` и `System.ComponentModel` определены дополнительные необобщенные коллекции, но здесь они не рассматриваются. Аналогично, я не будут упоминать интерфейсы LINQ, такие как `ILookup<TKey, TValue>`. Это приложение является больше справочником, чем руководством — считайте его альтернативой просмотру MSDN во время написания кода. Очевидно, что в большинстве случаев MSDN предоставит больший объем деталей, но цель данного приложения в том, чтобы бегло ознакомиться с разнообразными интерфейсами и реализациями при выборе конкретной коллекции для использования в своем коде.

Безопасность в отношении потоков для каждой коллекции не указывается, а подробные сведения можно почерпнуть из MSDN. Ни одна из нормальных коллекций не поддерживает множество параллельных средств записи; некоторые поддерживают одиночное средство записи и параллельные средства чтения. В разделе Б.6 приведен список параллельных коллекций, добавленных в .NET 4. Кроме того, в разделе Б.7 обсуждаются интерфейсы коллекций, допускающие только чтение, которые появились в .NET 4.5.

Б.1. Интерфейсы

Почти все интерфейсы, которые нужно знать, находятся в пространстве имен `System.Collections.Generic`. На рис. Б.1 показано, как были связаны основные интерфейсы до выхода версии .NET 4.5; здесь также присутствует необобщенный `IEnumerable` в качестве корневого интерфейса. Чтобы излишне не усложнять диаграмму, в нее не были включены интерфейсы, предназначенные только для чтения, которые появились в версии .NET 4.5.

Как вы уже видели несколько раз, наиболее фундаментальным интерфейсом обобщенной коллекции является `IEnumerable<T>`, представляющий последовательность элементов, по которой можно осуществлять проход. Интерфейс `IEnumerable<T>` позволяет запрашивать итератор типа `IEnumerator<T>`.

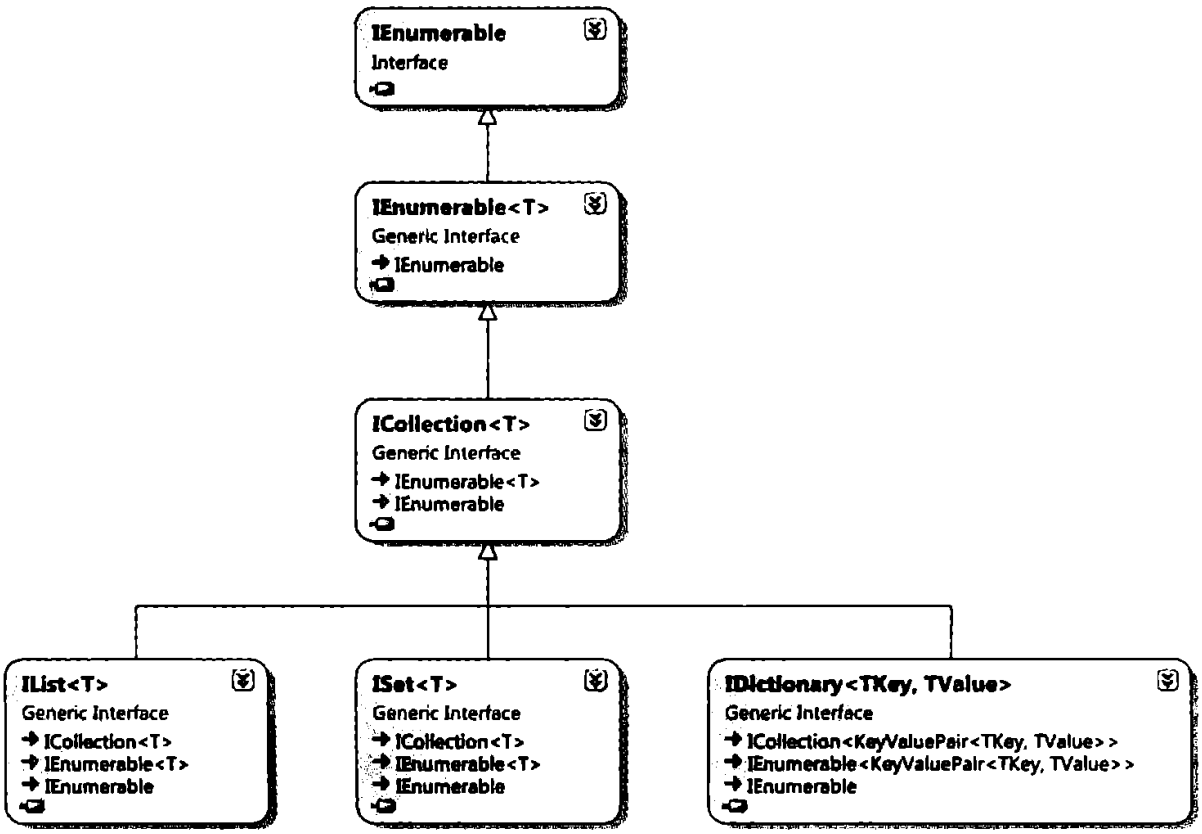

Рис. Б.1. Интерфейсы в пространстве имен `System.Collections.Generic` до выхода .NET 4

Отделение поддерживающей проход последовательности от итератора делает возможным выполнение множества итераторов независимым образом на одной и той же последовательности одновременно. Если вы хотите думать терминами баз данных, то `IEnumerable<T>` является таблицей, а `IEnumerator<T>` — курсором. Это единственные вариантные интерфейсы коллекций, раскрываемые в настоящем приложении, которые в .NET 4 становятся `IEnumerable<out T>` и `IEnumerator<out T>`; все остальные интерфейсы задействуют значения типа элементов, передаваемые в и из членов, поэтому они должны быть инвариантными.

Интерфейс `ICollection<T>` расширяет `IEnumerable<T>`, добавляя два свойства (`Count` и `IsReadOnly`), методы изменения (`Add()`, `Remove()` и `Clear()`), метод `CopyTo()` (копирует содержимое в массив) и метод `Contains()` (определяет, содержит ли коллекция заданный элемент). Этот интерфейс реализуют все стандартные реализации обобщенных коллекций.

Интерфейс `IList<T>` поддерживает позиционирование: он предоставляет индексатор, методы `InsertAt()` и `RemoveAt()` (соответствующие `Add()`/`Remove()`, но с позициями), а также метод `IndexOf()` (для определения позиции элемента внутри коллекции). Проход по `IList<T>` в большинстве случаев приводит к возвращению элемента по индексу 0, затем элемента по индексу 1 и т.д. Это не документировано строго, а является разумным предположением. Подобным же образом обычно ожидается, что произвольный доступ в `IList<T>` по индексу будет эффективным.

Интерфейс `IDictionary<TKey, TValue>` представляет отображение между уникальным ключом и значением для этого ключа. Значения не обязаны быть уникальными, и могут быть `null`; ключи не могут быть `null`. Словари могут рассматриваться как коллекции пар "ключ/значение"; именно поэтому `IDictionary<TKey, TValue>`

расширяет ICollection<KeyValuePair<TKey, TValue>>. Значения можно извлекать с помощью индексатора или метода TryGetValue(); в отличие от необобщенного типа IDictionary, при попытке извлечь значение для отсутствующего ключа индексатор интерфейса IDictionary<TKey, TValue> генерирует исключение KeyNotFoundException. Назначение метода TryGetValue() — позволить обнаруживать отсутствующие ключи в ситуациях, когда это ожидается при обычной работе.

ISet<T> является новым интерфейсом в .NET 4 и представляет отдельное множество значений. Он был задним числом применен к HashSet<T> из .NET 3.5, а в .NET 4 введена новая реализация — SortedSet<T>.

Обычно при *реализации* функциональности вполне понятно, какой интерфейс (и даже реализацию) необходимо использовать. Значительно труднее может оказаться решение о том, каким образом открыть доступ к этой коллекции как к части API-интерфейса; чем более специфично то, что возвращается, тем в большей степени вызывающий код сможет полагаться на дополнительную функциональность, указанную данными типами. Это может упростить написание вызывающего кода ценой будущей гибкости в рамках вашей реализации. Я обычно предпочитаю применять в качестве возвращаемых типов методов и свойств интерфейсы, а не гарантировать наличие конкретного класса реализации. Вы также должны тщательно подумать, прежде чем открывать изменяемую коллекцию в API-интерфейсе, особенно если такая коллекция представляет часть состояния объекта или типа. Как правило, предпочтительнее возвращать либо копию, либо оболочку для коллекции, допускающую только чтение, если только основное намерение метода не заключается в обеспечении изменения через возвращаемую коллекцию.

Б.2. Списки

Во многих отношениях списки являются простейшим и наиболее естественным типом коллекций. Внутри инфраструктуры доступно множество их реализаций, обладающих разными возможностями и характеристиками производительности. Несколько популярных реализаций используются повсеместно, другие, более экзотические применяются в специализированных ситуациях.

Б.2.1. List<T>

Класс List<T> — это стандартный выбор списка в большинстве случаев. Он реализует интерфейс IList<T> и, следовательно, ICollection<T>, IEnumerable<T> и IEnumerable. Кроме того, он реализует необобщенные интерфейсы ICollection и IList, выполняя при необходимости упаковку и распаковку, а также проверку типов во время выполнения, чтобы удостовериться в том, что новые элементы всегда имеют тип, совместимый с T.

Внутренне List<T> хранит массив и отслеживает логический размер списка и размер поддерживающего массива. Добавление элемента является либо простым случаем установки очередного значения в массиве, либо (если массив уже заполнен) копированием существующего содержимого в новый массив большего размера и затем установки в нем значения. Это означает, что операция имеет сложность O(1) или O(n) в зависимости от того, требуется ли копирование значений. Стратегия расширения не документирована (и, следовательно, не гарантирована), но на практике всегда используется подход с удвоением размера. Это дает в результате *амортизированную сложность* O(1) при добавлении элемента в конец списка; иногда она будет выше, но по мере роста списка такое случается все реже.

Размером поддерживающего массива можно управлять явно, читая и устанавливая свойство `Capacity`; метод `TrimExcess()` делает емкость в точности равной текущему размеру. На деле необходимость в этом возникает редко, но если вам известен окончательный размер списка при его создании, можете передать конструктору начальную емкость, избежав нежелательного копирования.

Удаление элемента из `List<T>` требует копирования расположенных за ним элементов на позицию назад, поэтому его сложность составляет $O(n - k)$, где k — индекс удаляемого элемента; устранение хвоста списка является менее дорогостоящей операцией, чем удаление его головы. С другой стороны, попытка удалить элемент по значению, а не по индексу (`Remove()` вместо `RemoveAt()`) приводит к выполнению операции со сложностью $O(n)$, где бы элемент ни находился: каждый элемент должен либо проверен на предмет эквивалентности, либо перемещен.

Разнообразные методы в `List<T>` действуют как своего рода предшественник LINQ. Метод `ConvertAll()` проецирует один список в другой; метод `FindAll()` фильтрует исходный список в новый список, содержащий только значения, которые удовлетворяют указанному предикату. Метод `Sort()` выполняет сортировку с применением или стандартного компаратора эквивалентности для типа, или компаратора, переданного в аргументе. Однако между методами `Sort()` и `OrderBy()` из LINQ существует крупное отличие: `Sort()` модифицирует содержимое исходного списка, а не выдает упорядоченную копию. К тому же метод `Sort()` неустойчив, тогда как `OrderBy()` устойчив; при использовании `Sort()` эквивалентные элементы в исходном списке могут быть переупорядочены. Есть один аспект `List<T>`, который не поддерживается LINQ — двоичный поиск: если список уже отсортирован подходящим для искомого значения образом, то метод `BinarySearch()` будет более эффективным, чем `IndexOf()`, применяющий линейный поиск[1].

Одним отчасти спорным аспектом `List<T>` является метод `ForEach()`. Он делает в точности то, о чем говорит его имя — проходит по списку и для каждого значения выполняет делегат (указанный в аргументе метода). Многие разработчики требовали, чтобы это было добавлено в виде расширяющего метода для `IEnumerable<T>`, но данному предложению до сих пор пока сопротивлялись; Эрик Липперт в своем блоге описывает затруднения с философской точки зрения (`http://mng.bz/Rur2`). Вызов `ForEach()`, используя лямбда-выражение, выглядит для меня излишеством; с другой стороны, если у вас уже есть делегат, который необходимо выполнить для каждого элемента, можно также поручить сделать это методу `ForEach()`, раз уж он доступен.

Б.2.2. Массивы

Массивы в некотором смысле представляют собой самый низкий уровень коллекций в .NET. Все массивы унаследованы напрямую от `System.Array`, и они являются единственными коллекциями с прямой поддержкой в среде CLR. Одномерные массивы реализуют интерфейс `IList<T>` (а также интерфейсы, которые он расширяет) и необобщенные интерфейсы `IList` и `ICollection`; прямоугольные массивы поддерживают только необобщенные интерфейсы. Массивы всегда изменяемы в терминах своих элементов, но всегда фиксированы в терминах их размера. Все изменяемые методы интерфейсов коллекций (вроде `Add()` и `Remove()`) реализованы явно и генерируют исключение `NotSupportedException`.

---

[1] Двоичный поиск обладает сложностью O(log n), а линейный поиск — O(n).

Массивы ссылочных типов всегда ковариантны; например, существует неявное преобразование из ссылки `Stream[]` в `Object[]` и явное преобразование в обратном направлении[2]. Это означает, что изменения, вносимые в массив, должны проверяться во время выполнения — самому массиву известен его тип, так что если вы попытаетесь сохранить ссылку, отличную от `Stream`, в `Stream[]`, предварительно преобразовав ссылку на массив в `Object[]`, сгенерируется исключение `ArrayTypeMismatchException`.

С точки зрения CLR есть две разновидности массивов. *Вектор* — это одномерный массив с нижней границей 0; все остальное считается *массивом*. Векторы работают лучше и почти всегда применяются в C#. Массив в форме `T[][]` по-прежнему является вектором, но с типом элементов `T[]`; только *прямоугольные массивы* в C#, такие как `string[10, 20]`, в конечном итоге являются массивами в терминологии CLR. Создать массив с ненулевой нижней границей прямо в C# невозможно — для этого необходимо использовать метод `Array.CreateInstance()`, который позволяет указывать нижние границы, длины и тип элементов по отдельности. Созданный одномерный массив с ненулевой нижней границей не получится успешно привести к `T[]` — компилятор разрешит приведение, но во время выполнения произойдет отказ.

Компилятор C# имеет встроенную поддержку массивов несколькими путями. Ему известно не только то, как их создавать и индексировать, он также поддерживает их напрямую в циклах `foreach`. В случае итерации с применением выражения, о котором на этапе компиляции известно, что оно является массивом, такая итерация будет использовать свойство `Length` и индексатор массива, а не создавать объект итератора. Это более эффективно, но разница в производительности обычно незначительна.

Подобно `List<T>`, массивы поддерживают такие методы, как `ConvertAll()`, `FindAll()` и `BinarySearch()`, хотя в случае массивов они представляют собой статические методы класса `Array`, принимающие массив в первом параметре.

Возвращаясь к моей первой точке зрения, массивы — это довольно-таки низкоуровневые структуры данных. Они являются важными строительными блоками для многих других коллекций и эффективны в подходящих ситуациях, но вы должны дважды подумать, прежде чем применять их слишком интенсивно. И снова в блоге Эрика Липперта имеется статья по этой теме, озаглавленная "Arrays considered somewhat harmful" ("Массивы считаются отчасти вредными") и доступная по адресу `http://mng.bz/3jd5`. Я вовсе не хочу преувеличивать его мнение, однако при выборе типа коллекции полезно знать о недостатках массивов.

Б.2.3. `LinkedList<T>`

Когда список не является списком? Когда это связный список. Класс `LinkedList<T>` считается списком во многих отношениях — в частности, он представляет собой коллекцию, которая поддерживает порядок при добавлении элементов, — однако он не реализует интерфейс `IList<T>`. Причина в том, что он не подчиняется общепринятому контракту об эффективном доступе по индексу. Это классический в вычислительной технике двухсвязный список: он поддерживает головной и хвостовой узлы, а каждый узел имеет ссылки на следующий и предыдущий узлы в списке. Каждый узел доступен как экземпляр `LinkedListNode<T>`, что удобно в случае, когда требуется реализовать вставку или удаление где-то в середине списка. Список явно поддерживает размер, поэтому доступ к свойству `Count` эффективен.

---

[2] Хотя это слегка запутывает, но данный факт также означает наличие неявного преобразования из `Stream[]` в `IList<Object>`, несмотря на то, что сам по себе интерфейс `IList<T>` инвариантен.

Связные списки неэффективны в плане занимаемого пространства по сравнению со списками, основанными на массивах, и они не поддерживают операции с индексами, но являются быстродействующими при вставке или удалении элементов в произвольных точках списка, если имеется ссылка на узел в нужной точке. Такие операции обладают сложностью O(1), поскольку все, что требуется — это корректировка ссылок на следующий и предыдущий узлы в окружающих узлах. Вставка либо удаление из головы или хвоста списка представляет собой просто специальный случай, при котором всегда производится непосредственный доступ к изменяемому узлу. Проход по списку (вперед или назад) также эффективен, т.к. сводится всего лишь к следованию по цепочке ссылок.

Несмотря на то что `LinkedList<T>` реализует стандартные методы наподобие `Add()` (который добавляет узел в хвост списка), я советую использовать явные методы `AddFirst()` и `AddLast()`, делая свои намерения совершенно ясными. Существуют соответствующие методы `RemoveFirst()` и `RemoveLast()`, а также свойства `First` и `Last`. Все они возвращают узлы списка, а не значения этих узлов; свойства возвращают ссылку `null`, если список пуст.

Б.2.4. Collection<T>, BindingList<T>, ObservableCollection<T> и KeyedCollection<TKey, TItem>

Класс `Collection<T>` входит в состав пространства имен `System.Collections.ObjectModel`, как и все остальные списки, которые мы еще рассмотрим. Подобно `List<T>`, он реализует обобщенные и необобщенные интерфейсы коллекций.

Хотя класс `Collection<T>` можно применять и сам по себе, чаще всего он используется в качестве базового класса. Он всегда служит оболочкой для другого списка: список либо указывается в конструкторе, либо "за кулисами" создается новый список `List<T>`. Все изменяющие действия над коллекцией выполняются посредством защищенных виртуальных методов (`InsertItem()`, `SetItem()`, `RemoveItem()` и `ClearItems()`); производные классы могут перехватывать эти методы, инициировать события или реализовывать другое специальное поведение. Внутренний список доступен производным классам через свойство `Items`. Если этот список допускает только чтение, то открытые изменяющие методы генерируют исключение, а не вызывают виртуальные методы; при их переопределении нет необходимости проверять это повторно.

Классы `BindingList<T>` и `ObservableCollection<T>` унаследованы от `Collection<T>` для предоставления возможностей привязки. Класс `BindingList<T>` доступен, начиная с версии .NET 2.0, но `ObservableCollection<T>` появился вместе с Windows Presentation Foundation (WPF). Разумеется, вы не обязаны применять их для привязки данных в пользовательских интерфейсах — вы можете иметь собственные причины заинтересованности в изменении списка. В этом случае вы должны выяснить, какая коллекция предоставляет уведомления в более удобной форме, и выбрать именно ее. Обратите внимание, что уведомления будут поступать только об изменениях, вносимых через оболочку; если лежащий в основе список модифицирует другой код, то никаких событий в оболочке не возникнет.

Класс `KeyedCollection<TKey, TItem>` — это своего рода гибрид списка и словаря, позволяющий извлекать элемент по ключу, а также по индексу. В отличие от нормальных словарей, ключ должен быть встроен внутрь элемента, а не существовать независимо. Во многих случаях это естественно; например, для представления заказчика может

быть определен тип Customer со свойством CustomerID. Класс KeyedCollection<,> является абстрактным; производные классы реализуют метод GetKeyForItem() для предоставления способа извлечения ключа из любого элемента, добавляемого в коллекцию. В сценарии с заказчиком метод GetKeyForItem() мог бы просто возвращать идентификатор данного заказчика. Подобно словарю, ключ должен быть уникальным в рамках коллекции — попытка добавления еще одного элемента с тем же самым ключом приведет к отказу с генерацией исключения. Хотя ключи null не разрешены, метод GetKeyForItem() может возвращать null (если ключ имеет ссылочный тип), в случае чего ключ будет проигнорирован (а элемент по этому ключу не будет доступен).

Б.2.5. ReadOnlyCollection<T> и ReadOnlyObservableCollection<T>

Финальные два списка представляют собой в большей степени оболочки, обеспечивая доступ только по чтению, даже если лежащий в основе список является изменяемым. Они реализуют обобщенные и необобщенные интерфейсы коллекций. Используется смесь явных и неявных реализаций интерфейсов, поэтому вызывающий код, в котором присутствует выражение конкретного типа на этапе компиляции, будет препятствовать применению изменяющих операций, которые дадут отказ.

Класс ReadOnlyObservableCollection<T> унаследован от ReadOnlyCollection<T> и реализует те же интерфейсы INotifyCollectionChanged и INotifyPropertyChanged, что и ObservableCollection<T>. Экземпляр ReadOnlyObservableCollection<T> может быть сконструирован только с поддерживающим списком ObservableCollection<T>. Хотя эта коллекция с точки зрения вызывающего кода по-прежнему допускает только чтение, в коде можно наблюдать за изменениями, вносимыми в поддерживающий список.

Хотя обычно я бы советовал использовать интерфейс, когда принимается решение относительно возвращаемого типа для методов в API-интерфейсе, может быть удобно преднамеренно указать ReadOnlyCollection<T>, чтобы предоставить вызывающему коду ясное указание на невозможность модификации возвращенной коллекции. Но все равно понадобится документировать, может ли лежащая в основе коллекция изменяться где-то в другом месте или же она является действительно постоянной.

Б.3. Словари

По сравнению со списками, варианты для словарей в инфраструктуре намного более ограничены. Есть только три главных непараллельных реализации интерфейса IDictionary<TKey, TValue>, хотя он также реализован классами ExpandoObject (как было показано в главе 14), ConcurrentDictionary (который мы рассмотрим вместе с другими параллельными коллекциями) и RouteValueDictionary (применяемый для маршрутизируемых веб-запросов, особенно в ASP.NET MVC).

В качестве напоминания: основное назначение словаря — предоставление эффективного поиска значения по ключу.

Б.3.1. Dictionary<TKey, TValue>

Если не существуют специализированные требования, то класс Dictionary<TKey, TValue> является стандартным выбором словаря в той же манере, как List<T> считается стандартной реализацией списка. Для реализации эффективного поиска он исполь-

зует хеш-таблицу, хотя это означает, что эффективность словаря зависит от того, насколько хороша функция хеширования. Можно либо применять стандартные функции хеширования и эквивалентности (вызовы `Equals()` и `GetHashCode()` внутри самих объектов ключей), либо передать реализацию `IEqualityComparer<TKey>` в аргументе конструктора.

Простейший случай использования предусматривает реализацию словаря со строковыми ключами, которые применяются нечувствительным к регистру способом, как показано в следующем коде:

```
var comparer = StringComparer.OrdinalIgnoreCase;
var dict = new Dictionary<String, int>(comparer);
dict["TEST"] = 10;
Console.WriteLine(dict["test"]);   ← Выводит на консоль значение 10
```

Несмотря на то что ключи внутри словаря должны быть уникальными, хеш-кодов это требование не касается. Вполне приемлема ситуация, что два разных ключа имеют один и тот же хеш-код; это называется *конфликтом хеш-кодов*, и хотя эффективность словаря слегка снижается, он по-прежнему функционирует корректно. Словарь *даст* отказ, если ключи являются изменяемыми и меняют свои хеш-коды после того, как были вставлены в словарь. Изменяемые ключи словаря почти всегда будут неудачным решением, но если обойтись без них *не удается*, то необходимо обеспечить, чтобы они не изменялись после вставки в словарь.

Точные детали реализации хеш-таблицы не указаны и со временем могут измениться, но один важный аспект иногда вызывает путаницу: *внутри `Dictionary<TKey, TValue>` нет никакого гарантированного порядка следования элементов, даже если они выглядят упорядоченными*. Если вы добавляете элементы в словарь и затем проходите по ним, они могут поступать в порядке, в котором были вставлены, *но рассчитывать на это нельзя*. Эта индивидуальная особенность реализации, просто добавляющей элементы даже без попытки сберечь упорядочение, несколько огорчает; реализация, которая бы естественным образом нарушала порядок, возможно, вызывала бы меньше вопросов.

Подобно `List<T>`, класс `Dictionary<TKey, TValue>` хранит свои элементы в массиве и при необходимости расширяет его, приводя к амортизированной сложности O(1). Доступ по ключу также имеет сложность O(1), предполагая рациональный хеш; если все ключи имеют один и тот же хеш-код, сложность доступа станет O(n), потому что словарю придется проверять на равенство все ключи по очереди. В большинстве практических сценариев эта проблема не возникает.

Б.3.2. SortedList<TKey, TValue> и SortedDictionary<TKey, TValue>

Случайный наблюдатель может допустить, что класс по имени `SortedList<,>` должен был быть списком, но это не так. Оба эти типа в действительности являются словарями, и ни один из них не реализует интерфейс `IList<T>`. Возможно, имена `ListBackedSortedDictionary` и `TreeBackedSortedDictionary` были бы более информативными, но теперь уже слишком поздно что-то менять.

У этих двух классов есть много общего: для сравнения ключей они оба используют интерфейс `IComparer<TKey>`, а не `IEqualityComparer<TKey>`, и оба поддерживают ключи в отсортированном виде на основе данного сравнения. Оба класса обеспечивают производительность O(log n) при поиске значений, фактически выполняя двоичный поиск. Но их внутренние структуры данных серьезно отличаются: `SortedList<,>`

поддерживает массив элементов, которые хранятся отсортированными, тогда как SortedDictionary<,> применяет структуру красно-черного дерева (кратко описано в Википедии по адресу http://ru.wikipedia.org/wiki/Красно-черное_дерево). Это приводит к значительным отличиям при вставке и удалении, а также в эффективности использования памяти. Если вы создаете словарь с применением в основном отсортированных данных, экземпляр SortedList<,> будет заполняться эффективно. Если вы представите шаги, предпринимаемые для обеспечения отсортированного вида List<T>, то увидите, что добавление одиночного элемента в конец списка не требует больших затрат (O(1), если проигнорировать расширение), в то время как добавление элементов произвольным образом является дорогостоящим, поскольку вовлекает копирование существующих элементов (O(n) в худшем случае). Добавление элементов к сбалансированному дереву в SortedDictionary<,> всегда характеризуется довольно небольшими затратами (сложность O(log n)), но предусматривает наличие в куче отдельного узла дерева для каждого элемента, приводя к большим накладным расходам и фрагментации памяти, чем структуры в виде массивов элементов "ключ/значение" в SortedList<,>.

Обе коллекции предоставляют доступ к своим ключам и значениям как к отдельным коллекциям, и в обоих случаях возвращаемые коллекции являются актуальными, отражая изменения в словарях. Однако коллекция в SortedList<,> реализует интерфейс IList<T>, так что возможен доступ к элементам по индексу в форме отсортированного ключа, если это действительно необходимо.

Я не хотел бы вас отпугнуть всеми этими разговорами о сложности. Если только вы не имеете дело с очень большим объемом данных, то, скорее всего, особо переживать по поводу того, какую реализацию использовать, не придется. Если наличие огромного количества элементов в словаре вполне вероятно, то вы должны тщательно проанализировать характеристики производительности обеих коллекций, чтобы выяснить, какая из них более приемлема.

Б.3.3. ReadOnlyDictionary<TKey, TValue>

После ознакомления с классом ReadOnlyCollection<T>, который обсуждался в разделе Б.2.5, класс ReadOnlyDictionary<TKey, TValue> не должен стать сюрпризом. Опять-таки, это просто оболочка вокруг существующей коллекции (на этот раз IDictionary<TKey, TValue>), которая скрывает все изменяемые операции за явной реализацией интерфейса и генерирует исключение NotSupportedException, если они все же вызываются.

Как и в случае списков, предназначенных только для чтения, это на самом деле только оболочка; если лежащая в основе коллекция (та, что передается конструктору) модифицируется, модификации будут видны через оболочку.

Б.4. Множества

До появления версии .NET 3.5 инфраструктура вообще не содержала открытые коллекции для множеств. Когда разработчикам необходимо было что-нибудь для представления множества в .NET 2.0, они обычно применяют Dictionary<,>, используя элементы множества в качестве ключей и предоставляя фиктивные значения. Ситуация в определенной мере улучшилась благодаря появлению HashSet<T> в .NET 3.5, а теперь в .NET 4 добавлен класс SortedSet<T> и общий интерфейс ISet<T>.

Хотя логически интерфейс множества мог бы содержать только операции Add()/Remove()/Contains(), в ISet<T> указано несколько других операций, предназначенных для манипулирования множеством (ExceptWith(), IntersectWith(), SymmetricExceptWith() и UnionWith()) и для проверки разнообразных более сложных условий (SetEquals(), Overlaps(), IsSubsetOf(), IsSupersetOf(), IsProperSubsetOf() и IsProperSupersetOf()). Параметры для всех этих методов выражены в терминах интерфейса IEnumerable<T>, а не ISet<T>, что поначалу вызывает удивление, но означает возможность взаимодействия множеств с LINQ естественным образом.

Б.4.1. HashSet<T>

Класс HashSet<T> фактически является Dictionary<,> без значений. Он обладает теми же самыми характеристиками производительности, и можно указывать IEqualityComparer<T> для настройки способа сравнения элементов. Опять-таки, не следует полагаться на то, что HashSet<T> поддерживает порядок, в котором к нему добавляются значения.

Одним дополнительным средством, предлагаемым HashSet<T>, является метод RemoveWhere(), который удаляет любой элемент, соответствующий заданному предикату. Он позволяет сократить множество, не беспокоясь об обычном запрете изменения коллекции во время прохода по ней.

Б.4.2. SortedSet<T> (.NET 4)

Подобно тому, как класс HashSet<T> сравним с Dictionary<,>, класс SortedSet<T> похож на SortedDictionary<,> без значений. Он поддерживает красно-черное дерево значений, обеспечивая сложность O(log n) для добавления, удаления и проверки наличия. При проходе по множеству значения будут выдаваться в отсортированном порядке. Класс SortedSet<T> предоставляет тот же самый метод RemoveWhere(), что и HashSet<T> (несмотря на то, что он не находится в интерфейсе), и дополнительные свойства (Min и Max) для возвращения минимального и максимального значений. Более интригующим методом является GetViewBetween(), который возвращает еще один экземпляр SortedSet<T>, предлагающий представление исходного множества в пределах нижней и верхней границ включительно. Это изменяемое и активное представление – вносимые в него изменения отражаются в исходном множестве и наоборот. Такое представление демонстрируется в следующем коде:

```
var baseSet = new SortedSet<int> { 1, 5, 12, 20, 25 };
var view = baseSet.GetViewBetween(10, 20);
view.Add(14);
Console.WriteLine(baseSet.Count);    <-- Выводит на консоль 6
foreach (int value in view)
{
  Console.WriteLine(value);          <-- Выводит на консоль 12, 14
}
```

Хотя метод GetViewBetween() удобен, он достается не совсем бесплатно: операции на представлении могут быть более дорогостоящими, чем ожидалось, из-за необходимости в поддержке внутренней согласованности. В частности, доступ к свойству Count через представление является операцией со сложностью O(n), если лежащее в основе множество изменилось с момента последнего обхода дерева. Подобно всем мощным инструментам, это должно применяться осторожно.

Класс
обладает еще одной возможностью: он предоставляет метод `Reverse()`, который позволяет проходить по множеству в обратном порядке. Он не использует метод `Enumerable.Reverse()`, буферизирующий содержимое последовательности, на которой он вызывается. Если заранее известно, что понадобится доступ к отсортированному множеству в обратном порядке, может быть удобно оставить для выражения тип `SortedSet<T>`, а не более общий интерфейсный тип, чтобы иметь дело с более эффективной реализацией.

Б.5. Queue<T> и Stack<T>

Очереди и стеки входят в число основных элементов в дисциплинах, связанных с вычислительной техникой. Иногда на них ссылаются как на структуры FIFO (first in, first out — первым пришел, первым обслужен) и LIFO (last in, first out — последним пришел, первым обслужен), соответственно. Базовая идея обеих структур данных одна: в коллекцию сначала добавляются элементы, а затем в какой-то другой точке они удаляются. Разница в порядке их удаления: очередь действует подобно обычной очереди в магазине, где первый человек, ставший в очередь, первым и будет обслужен; стек работает подобно стопке тарелок, когда первой будет взята тарелка, которая была положена последней. Распространенным случаем применения очередей и стеков является поддержка списка действий, подлежащих выполнению.

Подобно `LinkedList<T>`, несмотря на *возможность* использования для доступа к очередям и стекам обычных методов, определенных в интерфейсах коллекций, я рекомендую применять методы, специфичные для этих классов, чтобы сделать код более ясным.

Б.5.1. Queue<T>

Класс `Queue<T>` реализован с использованием кольцевого буфера: фактически он поддерживает массив с индексом, запоминающим следующую позицию, куда будет добавляться элемент, и еще одним индексом, в котором хранится следующая позиция, откуда элемент будет извлекаться. Если индекс для добавления настигает индекс для удаления, содержимое копируется в массив большего размера.

Класс `Queue<T>` предоставляет методы `Enqueue()` и `Dequeue()`, предназначенные для добавления и удаления элементов; метод `Peek()` позволяет просмотреть элемент, который будет извлечен из очереди следующим, не удаляя его. Методы `Dequeue()` и `Peek()` генерируют исключение `InvalidOperationException`, если вызываются на пустой очереди. Проход по очереди выдает значения в порядке их помещения в очередь.

Б.5.2. Stack<T>

Реализация класса `Stack<T>` даже проще, чем `Queue<T>` — ее можно считать подобной `List<T>`, но с методом `Push()`, предназначенным для добавления нового элемента в конец списка, `Pop()` для удаления последнего элемента и `Peek()` для просмотра последнего элемента без его удаления. Опять-таки, методы `Pop()` и `Peek()` генерируют исключение `InvalidOperationException`, когда вызываются на пустом стеке. Проход по стеку выдает значения в порядке, обратном их помещению — значение, добавленное последним, выдается первым.

Б.6. Параллельные коллекции (.NET 4)

В качестве части Parallel Extensions в .NET 4 в новом пространстве имен `System.Collections.Concurrent` доступно несколько новых коллекций. Они спроектированы безопасными в отношении параллельных операций, выполняемых во множестве потоков, с относительно малой долей блокирования. Указанное пространство имен также содержит три класса, которые применяются для разбиения коллекций для параллельных операций, но здесь они рассматриваться не будут.

Б.6.1. IProducerConsumerCollection<T> и BlockingCollection<T>

Три коллекции из набора новых коллекций реализуют новый интерфейс `IProducerConsumerCollection<T>`, который предназначен для использования с `BlockingCollection<T>`. При описании очередей и стеков упоминалось, что они часто применяются для хранения действий, подлежащих выполнению; шаблон "производитель/потребитель" представляет собой способ выполнения этих действий параллельным образом. Иногда имеется единственный поток производителя, создающий работу, и несколько потоков потребителей, выполняющих действия в рамках этой работы. В других случаях потребители могут быть также и производителями; например, поисковый агент может обрабатывать веб-страницу и выявлять внутри нее дополнительные ссылки для дальнейшего следования по ним.

Интерфейс `IProducerConsumerCollection<T>` действует в качестве абстракции для хранилища данных в шаблоне "производитель/потребитель", а класс `BlockingCollection<T>` помещает это в простую для использования оболочку и также предоставляет возможность ограничения количества одновременно буферизируемых элементов. Класс `BlockingCollection<T>` предполагает, что напрямую во внутреннюю коллекцию ничего другого добавляться не будет; все заинтересованные стороны должны применять эту оболочку как для добавления, так и для удаления действий. Перегруженные версии конструктора, которые не принимают параметр `IProducerConsumerCollection<T>`, используют для поддерживающего хранилища тип `ConcurrentQueue<T>`.

Интерфейс `IProducerConsumerCollection<T>` предоставляет только три довольно интересных метода: `ToArray()`, `TryAdd()` и `TryTake()`. Метод `ToArray()` копирует текущее содержимое коллекции в новый массив; это является снимком коллекции в момент вызова метода. Методы `TryAdd()` и `TryTake()` следуют нормальному шаблону `TryXXX()`, возвращая булевское значение для указания на успешное выполнение или отказ, и делают они то, что и можно было ожидать: пытаются добавить элемент в коллекцию или удалить его из нее. Обеспечение эффективного режима отказа снижает потребность в блокировании. Например, в `Queue<T>` пришлось бы удерживать блокировку для объединения операций "проверить, если в очереди какие-либо элементы" и "извлечь из очереди элемент, если он присутствует в ней" — иначе метод `Dequeue()` мог бы сгенерировать исключение.

В классе `BlockingCollection<T>` поведение блокирования укладывается поверх неблокирующих методов, с семейством перегруженных версий, позволяющих указывать тайм-ауты и признаки отмены. Обычно нет необходимости напрямую работать с `BlockingCollection<T>` или `IProducerConsumerCollection<T>`; чаще всего приходится обращаться к другим частям Parallel Extensions, которые будут применять эти типы. Тем не менее, о них полезно знать на тот случай, когда понадобится построить собственное специальное поведение.

Б.6.2. ConcurrentBag<T>, ConcurrentQueue<T> и ConcurrentStack<T>

Инфраструктура поступает с тремя реализациями интерфейса IProducerConsumer Collection<T>. По существу они отличаются в терминах порядка, в котором извлекаются элементы; очередь и стек действуют в той же манере, как их непараллельные эквиваленты, тогда как ConcurrentBag<T> не гарантирует какое-либо упорядочение.

Все три класса реализуют интерфейс IEnumerable<T> безопасным к потокам образом. Итератор, возвращаемый методом GetEnumerator(), будет проходить по снимку коллекции; во время прохода коллекцию можно модифицировать, но изменения не будут видны в итераторе. Все три класса также предлагают метод TryPeek(), который подобен TryTake(), но не удаляет значение из коллекции. В отличие от TryTake(), этот метод не указан в интерфейсе IProducerConsumerCollection<T>.

Б.6.3. ConcurrentDictionary<TKey, TValue>

Класс ConcurrentDictionary<TKey, TValue> реализует стандартный интерфейс IDictionary<TKey, TValue> (тогда как ни одна параллельная коллекция не реализует интерфейс IList<T>) и, по сути, является безопасным в отношении потоков словарем, основанным на хеш-таблице. Он поддерживает множество потоков, параллельно выполняющих чтение и запись, и также позволяет проходить по словарю безопасным к потокам образом, хотя в отличие от трех коллекций из предыдущего раздела, изменения, вносимые в словарь, могут быть, а могут и не быть отражены в итераторе.

Этот класс обеспечивает не только безопасный в отношении потоков доступ. В то время как обычные реализации словарей в основном предоставляют добавление или обновление через индексатор и добавление или генерацию исключения через метод Add(), класс ConcurrentDictionary<TKey, TValue> предлагает по-настоящему разнообразный набор вариантов. Можно обновлять значение, ассоциированное с ключом, на основе его предыдущего значения, получать значение на основе ключа или добавлять его, если ключ ранее в словаре не присутствовал, условно обновлять значение, только если оно было заранее ожидаемым, и выполнять много других действий, причем все они являются атомарными. Поначалу все это приводит в замешательство, но Стивен Тауб из команды разработчиков Parallel Extensions в своем блоге описывает детали того, когда должен использоваться тот или иной метод (http://mng.bz/WMdW).

Б.7. Интерфейсы, допускающие только чтение (.NET 4.5)

В .NET 4.5 появились три новых интерфейса коллекций: IReadOnlyCollection<T>, IReadOnlyList<T> и IReadOnlyDictionary<TKey, TValue>. На момент написания этих строк они применялись не особенно широко — однако о них полезно знать, причем в большей степени то, чем они *не* являются. На рис. Б.2 показано, как они соотносятся друг с другом и с интерфейсами IEnumerable.

Если вы думали, что класс ReadOnlyCollection<T> несколько кривил душой в своем имени, то эти интерфейсы еще более коварны. Они не просто разрешают изменения, выполняемые в коде, но даже позволяют делать их через тот же самый объект, если так случилось, что он оказался изменяемой коллекцией. Например, List<T> реализует IReadOnlyList<T>, несмотря на то, что определенно не является коллекцией, допускающей только чтение.

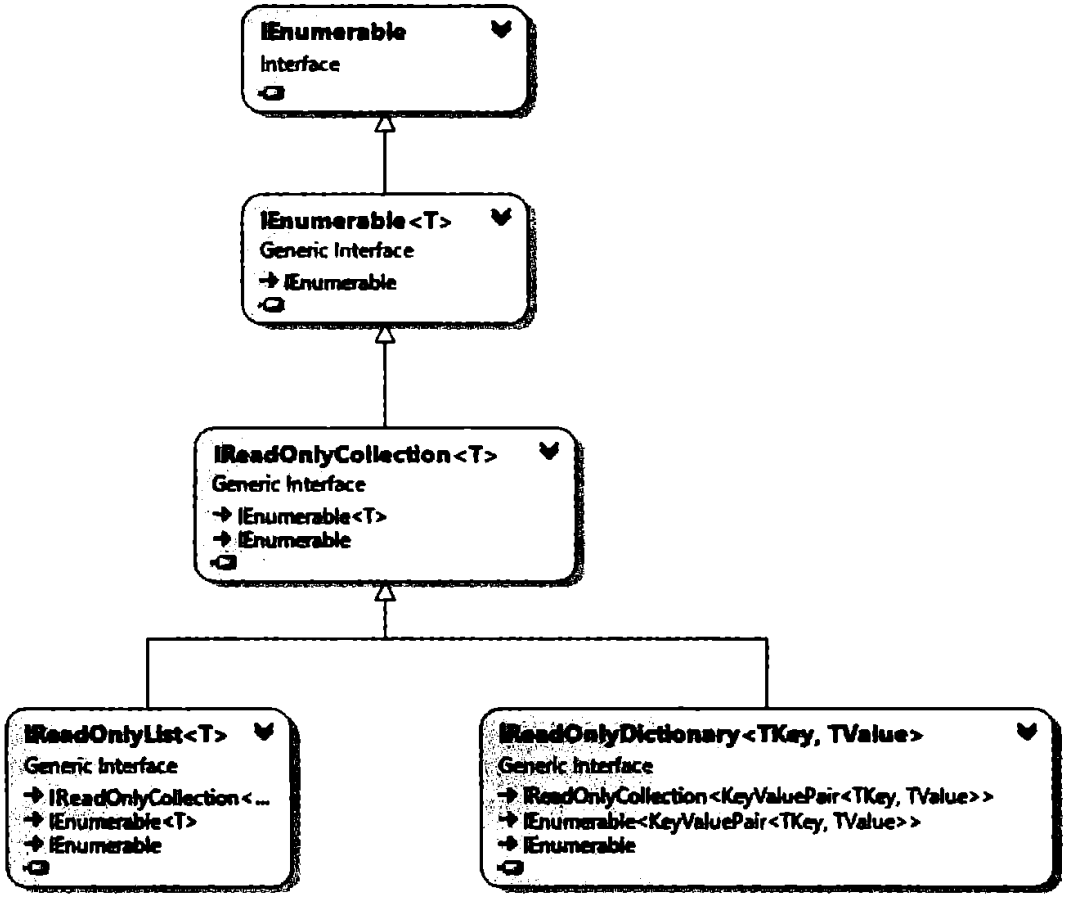

Рис. Б.2. Интерфейсы, допускающие только чтение, в .NET 4.5

Конечно, это не говорит о том, что данные интерфейсы совершенно бесполезны. В частности, `IReadOnlyCollection<T>` и `IReadOnlyList<T>` ковариантны в `T`, точно как `IEnumerable<T>`, но предоставляют больше операций. К сожалению, интерфейс `IReadOnlyDictionary<TKey, TValue>` инвариантен в обоих параметрах типов, частично из-за того, что он реализует `IEnumerable<KeyValuePair<TKey, TValue>>`, который инвариантен, поскольку `KeyValuePair<TKey, TValue>` представляет собой структуру, являющуюся инвариантной сама по себе. Кроме того, ковариантность интерфейса `IReadOnlyList<T>` означает, что он не может предложить какие-либо методы, принимающие `T`, такие как `Contains()` и `IndexOf()`. Крупное преимущество этого интерфейса в том, что он *предоставляет* индексатор для извлечения элементов по индексу.

Я пока не вижу для себя особо много сценариев использования этих интерфейсов, но в будущем, думаю, они станут очень важны. В конце 2012 года в Microsoft выпустили первую ознакомительную версию пакета NuGet для неизменяемых коллекций под названием `Microsoft.Bcl.Immutable`. В блоге команды разработчиков библиотеки базовых классов можно найти больше подробностей (`http://mng.bz/Xlqd`), но по существу они являются полностью неизменяемыми коллекциями наряду с замораживаемыми (изменяемыми до тех пор, пока не будут заморожены) коллекциями. Разумеется, если тип элементов изменяемый (такой как `StringBuilder`), то это далеко не все, но меня по-прежнему волнуют все нормальные причины, по которым такая неизменяемость полезна.

Б.8. Резюме

Инфраструктура .NET Framework содержит богатый набор коллекций (хотя не настолько богатый набор множеств). Набор постепенно разрастался наряду с остальными частями инфраструктуры, несмотря на то, что к настоящему времени наиболее распространенными коллекциями, пожалуй, можно назвать только `List<T>` и `Dictionary<TKey, TValue>`.

Определенно есть структуры данных, которые могли бы быть добавлены в будущем, но всегда должна производиться оценка преимуществ от добавления чего-либо в основную инфраструктуру по сравнению с требуемыми для этого затратами. Возможно, вскоре мы увидим API-интерфейсы, явно основанные на деревьях, вместо их применения в качестве деталей реализации внутри существующих коллекций. Может быть, мы дождемся кучи на базе последовательностей Фибоначчи, кеша со слабыми ссылками и тому подобного — но, как вы видели, задач для разработчиков уже очень много и существует риск информационной перегрузки.

Если для проекта необходима конкретная структура данных, имеет смысл поискать в Интернете ее реализацию с открытым кодом; проект Power Collections от Wintellect имеет особенно убедительную историю как альтернатива встроенным коллекциям (`http://powercollections.codeplex.com`). Но в большинстве случаев инфраструктура, вероятно, окажется вполне адекватной для существующих потребностей. Надеюсь, что это приложение слегка расширило ваш кругозор в плане того, что доступно в готовом виде в рамках инфраструктуры.

Приложение B

Итоговые сведения по версиям

Номера версий в .NET иногда вызывают путаницу. Инфраструктура, исполняющая среда, Visual Studio и C# нумеруются по отдельности. Это приложение служит кратким руководством, посвященным тому, как все они соотносятся друг с другом, и основным возможностям каждого выпуска. В каждом случае описаны средства, начиная с выпусков 2.0 в выше; перечислять все возможности .NET 1.0 и .NET 1.1 не имело бы смысла.

B.1. Главные выпуски инфраструктуры для настольных приложений

Когда разработчики ссылаются на выпуски .NET, они обычно имеют в виду главные выпуски инфраструктуры для настольных приложений. В большинстве случаев выпуск инфраструктуры сопровождался выпуском среды Visual Studio (или Visual Studio .NET, как она называлась в выпусках 2002 и 2003). Исключением была .NET 3.0, которая, в сущности, представляла собой набор библиотек (хотя эти библиотеки были весьма значимыми). Набор расширений Visual Studio 2005 стал доступным для новых средств, но в Visual Studio 2008 содержалась их большая поддержка. В табл. B.1 показаны версии и появившиеся в них аспекты инфраструктуры.

Когда была выпущена .NET 3.5, появились также пакеты обновлений .NET 2.0 SP1 и .NET 3.0 SP1; они содержали среду CLR и библиотеку базовых классов (BCL) для версии 2.0 SP1. Аналогично, выпуск .NET 3.5 SP1 совпал по времени с выходом .NET 2.0 SP2 и .NET 3.0 SP2.

Среда Visual Studio 2008 была первым выпуском, ориентированным на поддержку *множества целевых платформ*, и позволяла выбирать версию инфраструктуры, для которой нужно было проводить компиляцию. Во многих случаях новые средства языка C# можно использовать, выбрав в качестве целевой платформу более раннего выпуска — в основном это ситуации, когда средство реализовано исключительно магией компилятора, безо всякой поддержки со стороны среды CLR или библиотек. Дополнительная информация о том, как сделать это, доступна (на английском языке) на веб-сайте книги (`http://mng.bz/YpRB`) — в некоторых случаях существуют обходные пути, если средство не работает нормально с самого начала.

Таблица B.1. Выпуски инфраструктуры для настольных приложений и их компоненты

Дата	Инфраструктура	Visual Studio	C#	CLR
Февраль 2002 г.	1.0	2002	1.0	1.0
Апрель 2003 г.	1.1	2003	1.2	1.1
Ноябрь 2005 г.	2.0	2005	2.0	2.0
Ноябрь 2006 г.	3.0	(Расширения выпуска 2005)	—	2.0
Ноябрь 2007 г.	3.5	2008	3.0	2.0 SP1
Апрель 2010 г.	4	2010	4.0	4.0 (версия 3.0 отсутствовала)
Август 2012 г.	4.5	2012	5.0	4.0 или 4.5 [a]

[a] Зависит от вашей точки зрения. Дополнительные сведения будут даны позже.

Полезно отметить, что если вы укажете в качестве целевой платформы .NET 2.0 (выбрать .NET 1.0 или .NET 1.1 нельзя) в Visual Studio 2008 или Visual Studio 2010, то фактически будете направлены на соответствующий пакет обновлений (2.0 SP1 или 2.0 SP2); это означает возможность компиляции кода, в котором применяются новые средства из пакета обновлений (одним примечательным введением была структура `System.DateTimeOffset` в 2.0 SP1), с последующим отказом его работы на компьютере, где установлен только первоначальный выпуск .NET 2.0. Лично я попытался бы провести модернизацию компьютеров хотя бы последним пакетом обновлений, а в идеальном случае — актуальным выпуском полной инфраструктуры.

B.2. Средства языка C#

Если вы прочитали книгу целиком, то должны быть способны написать данный раздел самостоятельно. (Было бы заманчиво оставить несколько пустых строк, чтобы вы заполнили их самостоятельно, но я же не *настолько* ленив.) Один тривиальный факт: номер версии 1.2 в табл. B.1 не является опечаткой; согласно спецификации, в Microsoft действительно пропустили версию 1.1 и выпустили компилятор C# 1.2 вместе с .NET 1.1. Изменения в версии 1.2 были в основном незначительными, но в долгосрочном плане появилось одно существенное изменение: начиная с C# 1.2 и далее, транслированный код для цикла `foreach` проверяет факт реализации итератором интерфейса `IDisposable` и освобождает итератор соответствующим образом. Как вы уже видели, это изменение критически важно для итераторных блоков, которые имеют ресурсы, подлежащие очистке.

В любом случае ради завершенности ниже упоминаются языковые средства со ссылками на главы, в которых они рассматриваются более подробно.

B.2.1. C# 2.0

Крупными средствами C# 2 были обобщения (глава 3), допускающие `null` типы (глава 4), анонимные методы и другие улучшения, связанные с делегатами (глава 5), а также итераторные блоки (глава 6). Кроме того, появилось несколько небольших средств: частичные типы, статические классы, свойства с отличающимися модифика-

торами доступа для средств получения и установки, псевдонимы пространств имен, директивы `pragma` и буферы фиксированного размера. За дополнительной информацией обращайтесь в главу 7.

В.2.2. C# 3.0

Версия C# 3 главным образом ориентирована на LINQ, хотя многие средства полезны и в других местах. Автоматические свойства, неявная типизация массивов и локальных переменных, инициализаторы объектов и коллекций, а также анонимные типы рассматриваются в главе 8. Лямбда-выражения и деревья выражений (глава 9) еще более увеличили прогресс, достигнутый в области делегатов в версии 2.0, а расширяющие методы (глава 10) предоставили последний ингредиент для выражений запросов (глава 11). Частичные методы появились только в C# 3, но были раскрыты вместе с частичными типами в главе 7.

В.2.3. C# 4.0

Версия C# 4.0 имеет некоторые средства, способствующие взаимодействию, но не обладает такой же целеустремленностью, как C# 3.0. Опять-таки, существует довольно ясное разделение между мелкими средствами, показанными в главе 13 (именованные аргументы, необязательные параметры, улучшенное взаимодействие с COM, обобщенная вариантность), и крупным средством динамической типизации (глава 14).

В.2.4. C# 5.0

Версия C# 5.0 всецело ориентирована на асинхронность, описанную в главе 15, а также имеет два других очень маленьких средства (изменения в захвате переменной в цикле `foreach` и атрибуты информации о вызывающем компоненте), которые были раскрыты в главе 16. Несмотря на то что асинхронность вводит только один новый вид выражения (`await` внутри функции `async`), она чрезвычайно сильно изменяет модель выполнения. Я мог бы утверждать, что даже если команда проектировщиков C# была готова предоставить другие крупные новые средства языка (и, насколько мне известно, это так), было бы разумно попридержать их на какое-то время. Важно, чтобы сообщество разработчиков на C# на самом деле тщательно исследовало подход `async/await`, а это требует некоторого времени.

В.3. Средства библиотек инфраструктуры

Перечислить здесь все новые средства инфраструктуры в удобной форме попросту невозможно. В частности, каждая область инфраструктуры (Windows Forms, ASP.NET и т.д.) в каждом выпуске получала дополнительные средства — и не только в рамках основной библиотеки базовых классов. Я включил только те средства, которые считаю наиболее важными. В MSDN содержится намного более полный список: `http://mng.bz/6tiZ`.

В.3.1. .NET 2.0

Крупнейшие средства в библиотеках .NET 2.0 направлены на поддержку возможностей среды CLR и языка: обобщения и типы, допускающие `null`. Несмотря на то что типы, допускающие `null`, не требовали множества изменений, несколько обобщенных

коллекций, которыми мы пользуемся до сих пор, существуют со времен версии .NET 2.0, и API-интерфейс рефлексии должен был соответствующим образом обновиться.

Многие области подверглись относительно небольшим обновлениям, таким как поддержка для сжатия, получения множественных активных результирующих наборов (multiple active result set — MARS) через одиночное подключение к SQL Server и многих статических вспомогательных методов ввода-вывода наподобие `File.ReadAllText()`. Возможно, справедливо будет отметить, что модификации не были настолько значительными, как изменения, внесенные в инфраструктуры для построения пользовательских интерфейсов.

В ASP.NET появились мастер-страницы, возможности предварительной компиляции и разнообразные новые элементы управления. В Windows Forms был совершен большой скачок в плане возможностей компоновки с помощью `TableLayoutPanel` и похожих классов, а также обеспечена лучшая поддержка для улучшений производительности, таких как двойная буферизация, новая модель привязки данных и развертывание ClickOnce. В .NET 2.0 появился класс `BackgroundWorker`, позволяющий упростить безопасное обновление пользовательского интерфейса в многопоточных приложениях; строго говоря, он не является частью Windows Forms, хотя это был его основной сценарий применения вплоть до появления Windows Presentation Foundation в рамках версии .NET 3.0.

B.3.2. .NET 3.0

Версия .NET 3.0 кое в чем необычна в качестве главного выпуска — в ней не были внесены изменения ни в среду CLR, ни в язык, ни в существующие библиотеки. Вместо этого появились четыре новых библиотеки.

- Windows Presentation Foundation (WPF) — инфраструктура для построения пользовательских интерфейсов следующего поколения; она стала революционным скачком, а не дальнейшим развитием Windows Forms, хотя эти две инфраструктуры могут сосуществовать бок о бок. По сравнению с Windows Forms она имеет совершенно другую модель, являясь более композиционной по своей природе. Пользовательский интерфейс Silverlight основан на WPF.
- Windows Communication Foundation (WCF) — архитектура для построения приложений, ориентированных на службы; она является расширяемой, а не ограниченной единственным протоколом, и призвана унифицировать существующие каналы RPC-подобных коммуникаций, такие как удаленный доступ.
- Windows Workflow Foundation (WF) — система для построения приложений, основанных на рабочих потоках.
- Windows CardSpace — система безопасной идентификации.

Из указанных четырех областей преуспели только WPF и WCF, тогда как WF и CardSpace не получили широкого распространения. Это не говорит о том, что технологии WF и CardSpace не используются или не обретут большую важность в будущем, но на момент написания этих строк они применялись нечасто.

B.3.3. .NET 3.5

Крупным новым средством в .NET 3.5 стала технология LINQ, поддерживаемая C# 3.0 и VB 9. Она включает LINQ to Objects, LINQ to SQL, LINQ to XML и лежащую в их основе поддержку деревьев выражений.

Другие области также получили важные возможности: намного упростилось использование AJAX в ASP.NET; в инфраструктуры WCF и WPF было внесено немало усовершенствований; появилась инфраструктура дополнений (`System.AddIn`); включены разнообразные новые криптографические алгоритмы и добавлено многое другое. Как разработчик, заинтересованный в параллелизме и API-интерфейсах, связанных со временем, я просто обязан привлечь ваше внимание к факту появления класса `ReaderWriterLockSlim` и весьма востребованных типов `TimeZoneInfo` и `DateTimeOffset`. Если вы работаете с .NET 3.5 или последующими версиями, но по-прежнему везде полагаетесь на тип `DateTime`, то должны знать, что доступны более удобные варианты[1].

Наиболее заметными библиотечными средствами .NET 3.5 SP1 была инфраструктура Entity Framework и связанные с ней технологии ADO.NET, но другие технологии также подверглись небольшим усовершенствованиям. Важно и то, что в .NET 3.5 SP1 появился клиентский профиль (Client Profile) — небольшая версия настольной инфраструктуры .NET Framework, которая не включает множество библиотек, предназначенных для разработки серверной стороны. Это позволяет уменьшить площадь развертывания для полностью клиентских приложений.

B.3.4. .NET 4

На протяжении длительного времени в библиотеки .NET 4 было вложено немало труда в той или иной форме. Крупным добавлением стала среда DLR, а также средство Parallel Extensions, которое кратко рассматривалось в других главах. Как обычно, технологии для построения пользовательских интерфейсов получили множество усовершенствований, хотя в значительной степени внимание было сосредоточено на изменениях для многофункциональных клиентов в WPF, а не в Windows Forms. Существующие основные API-интерфейсы были подвергнуты многим настройкам, направленным на облегчение работы с ними, например, метод `String.Join()` стал принимать реализацию `IEnumerable<T>`, а не лишь строковый массив. Улучшения нельзя назвать поразительными, но если они хоть немного упростят жизнь каждого разработчика, то совокупное их влияние окажется весьма сильным. Вы уже видели, что некоторые существующие обобщенные интерфейсы и делегаты стали ковариантными или контравариантными (например, `IEnumerable<T>` стал `IEnumerable<out T>`, а `Action<T>` превратился в `Action<in T>`), но есть также новые типы для ознакомления.

Появилось новое пространство имен для цифровых вычислений, `System.Numeric`. На время написания этой книги оно содержало только типы `BigInteger` и `Complex`, но я не удивлюсь добавлению в него типа `BigDecimal` в ближайшем будущем. В пространство имен `System` включены другие новые типы, такие как `Lazy<T>` для лениво инициализируемых значений и семейство обобщенных классов `Tuple`, которые предоставляют ту же самую функциональность, что и класс `Pair<T1, T2>` из главы 3, но принимают до восьми параметров типов. Класс `Tuple` также поддерживает *структурные сравнения*, представляемые с помощью новых интерфейсов `IStructuralEquatable` и `IStructuralComparable` в пространстве имен `System.Collections`. Хотя все классы Reactive Extensions, рассмотренные в главе 12, не находятся в .NET 4, основные интерфейсы `IObserver<T>` и `IObservable<T>` определены в пространстве имен `System`.

---

[1] По моим личным ощущениям поддержка для такого сложного и интригующего мира, связанного с датами и временем, все еще недостаточна, поэтому я начал проект Noda Time (`https://code.google.com/p/noda-time/`). Однако теперь тип `TimeZoneInfo`, по крайней мере, предоставляет ясный способ представления часового пояса, отличного от местного.

Я привел эти специфичные элементы потому, что слишком большое внимание привлекают новые области наподобие Managed Extensibility Framework (MEF), и такие простые типы очень легко упустить из виду. Приятно видеть, что время уделяется всей инфраструктуре целиком, а не только ярким нововведениям.

B.3.5. .NET 4.5

Крупнейшей движущей силой изменений в .NET 4.5 является в сущности одна асинхронность. Асинхронные версии доступны практически для каждого API-интерфейса, которому это могло бы понадобиться: если действие занимает некоторое время, должна быть возможность выполнять его асинхронно. Чтобы также помочь в этом, библиотека параллельных задач (Task Parallel Library) из .NET 4 была расширена (и оптимизирована). В .NET 4.5 было внесено очень много других изменений, описать которые здесь попросту нереально. Даже страница MSDN с перечнем наиболее примечательных изменений (`http://mng.bz/6tiZ`) настолько длинная, что не может быть включена в данное приложение. Однако большинство этих изменений будет зависеть от разрабатываемого проекта, хотя асинхронность, повсеместно встречающаяся во всей платформе, вероятно, коснется каждого.

B.4. Средства исполняющей среды (CLR)

Изменения в CLR часто менее заметны многим разработчикам, чем новые средства в библиотеках и языке. Понятно, что есть особенно яркие средства наподобие обобщений, которые привлекут внимание кого угодно, но другие средства менее очевидны. Кроме того, по сравнению с языком или библиотеками инфраструктуры среда CLR изменялась менее часто, во всяком случае, в смысле главных выпусков.

B.4.1. CLR 2.0

В дополнение к обобщениям, среда CLR требовала еще одного изменения для поддержки новых языковых средств C# 2: поведения упаковки и распаковки типов значений, допускающих `null`, которое исследовалось в главе 4.

В CLR 2.0 были внесены и другие крупные изменения. Самым значительным из них стала поддержка для 64-разрядных процессоров (x64 и IA64) и возможность хостинга CLR внутри SQL Server 2005. Интеграция с SQL Server потребовала проектирования новых API-интерфейсов хостинга, чтобы хост мог иметь намного больший контроль над средой CLR, включая способ распределения памяти и организации многопоточности. Это позволяет прилежному хосту удостовериться в том, что код, выполняемый в CLR, не скомпрометирует другие аспекты критически важного процесса, такого как база данных.

Версия .NET 3.5 содержит CLR 2.0 SP1, а .NET 3.5 SP1 — CLR 2.0 SP2; они претерпели относительно небольшие изменения, вроде подстройки доступа кода в классе `DynamicMethod` к закрытым членам другого типа. Команда проектировщиков CLR также всегда ищет пути улучшения производительности, с усовершенствованиями сборки мусора, JIT-компиляции, времени запуска и т.д.

B.4.2. CLR 4.0

Хотя среда CLR не нуждалась в каких-либо изменениях, чтобы приспособиться к DLR, команда проектировщиков все равно немало потрудилась над этой версией.

Ниже перечислены наиболее примечательные изменения.

- Улучшение производительности маршализации и согласованности с помощью повсеместных заглушек IL (дополнительные сведения можно найти в статье блога .NET Framework Blog: `http://mng.bz/56H6`).
- Фоновый сборщик мусора, пришедший на замену параллельному сборщику мусора из CLR 2.0.
- Усовершенствованная модель безопасности, основанная на концепции прозрачности, которая является преемником безопасности доступа кода (Code Access Security — CAS).
- Эквивалентность типов, используемая для поддержки средства встраивания сборок PIA в C# 4.
- Выполнение бок о бок разных версий CLR в рамках одного процесса.

Среда CLR в .NET 4.5 включает несколько усовершенствований, в основном связанных со сборкой мусора. В сущности, ее можно считать второстепенным выпуском. Наряду с преимуществами, касающимися чисто производительности, 64-разрядная среда CLR также поддерживает опцию конфигурации `<gcAllowVeryLargeObjects>`, которая позволяет создавать массивы громадных размеров, даже когда их элементами являются крупные структуры... при условии наличия достаточного объема памяти, конечно. В терминах номера версии картина несколько затруднена. В документации вы вполне можете видеть, что на эту версию среды ссылаются как на CLR 4.5. Тем не менее, она по-прежнему представляет себя как версию 4.0, если проверить значение свойства `Environment.Version`. Например, на время написания этого приложения среда CLR, с которой я имел дело, сообщала версию 4.0.30319.18033. С течением времени номера сборки и ревизии потенциально могут измениться из-за выхода пакетов обновлений.

Больший объем сведений обо всех новых средствах доступен в блоге .NET Framework Blog (`http://blogs.msdn.com/b/dotnet`).

B.5. Связанные инфраструктуры

При вычислениях редко применяется модель "на все случаи жизни", и .NET тому не исключение. Даже настольная инфраструктура в действительности не является единственной версией: существует клиентский профиль, 32- и 64-разрядные JIT-компиляторы, а также среды CLR для сервера и рабочей станции, настроенные на выполнение разных задач. Кроме того, доступны отдельные инфраструктуры, которые имеют собственные хронологии версий, приспособленные к разным средам.

B.5.1. Compact Framework

Инфраструктура Compact Framework первоначально предназначалась для мобильных устройств, функционирующих под управлением операционной системы Windows Mobile. С тех пор она была переориентирована на Xbox 360, Windows Phone 7 и Symbian S60. График главных выпусков Compact Framework традиционно отражает выпуски настольной инфраструктуры, хотя версия, соответствующая .NET 3.0, не выходила. Если интересно, то самым последним выпуском (используемым некоторыми устройствами Windows Mobile и WP7) была версия 3.7.

В ранних версиях Compact Framework отсутствовала некоторая основная функциональность, что было восполнено усилиями сообщества разработчиков; в поздних выпусках многие существенные пробелы были ликвидированы, хотя эти выпуски, очевидно, по-прежнему являлись подмножеством настольной инфраструктуры. Уровень графического пользовательского интерфейса зависел от целевой платформы; например, в Xbox 360

применялся XNA, Windows Mobile поддерживала Windows Forms, а в WP7 поддерживались XNA и Silverlight. Код, выполняемый в среде Compact Framework, подвергался JIT-компиляции и сборке мусора, хотя сборщик мусора Compact Framework не был основан на понятии поколений объектов, как это было в настольной инфраструктуре.

B.5.2. Silverlight

Инфраструктура Silverlight (`http://silverlight.net/`) предназначена для выполнения приложений либо внутри браузеров, либо (версия Silverlight 3) в среде песочницы, обычно первоначально устанавливаемой из браузера. По существу он является естественным конкурентом Flash; его очевидное преимущество в том, что он предоставляет разработчикам на C# возможность писать приложения на знакомом языке и применять знакомую библиотеку. Silverlight устанавливает упрощенную среду CLR (называемую CoreCLR — `http://mng.bz/G32M`) и библиотеку классов — например, не поддерживаются необобщенные коллекции и отсутствует инфраструктура Windows Forms. Уровень представления Silverlight основан на WPF, однако они не идентичны. Он имеет особенно серьезную поддержку для воспроизведения медиа, с такими возможностями, как глубокое масштабирование и адаптивное потоковое видео.

Версия Silverlight 1 была выпущена в сентябре 2007 года, хотя она ограничивалась смесью XAML для конструирования пользовательского интерфейса и JavaScript для реализации логики. С выходом в октябре 2008 года версии Silverlight 2 стала реальной практика доставки приложений Silverlight, построенных с помощью C#. Некоторые средства из CoreCLR (хостинг сред CLR бок о бок внутри одного процесса и декларативная модель безопасности, основанная на концепции прозрачности) теперь появились в настольной версии CLR 4.0. Она также включает раннюю версию Dynamic Language Runtime.

Прогресс остановить было сложно, и в июле 2009 была выпущена версия Silverlight 3 с большим числом элементов управления и видеокодеков, а также автономными и внебраузерными приложениями. Команда разработчиков Silverlight повторила девятимесячный цикл выпусков, предоставив версию Silverlight 4 на той же неделе, что и .NET 4, с еще одним длинным списком новых средств. Операционная система Windows Phone 7 поддерживала Silverlight 3 и некоторые средства Silverlight 4, а затем, когда был выпущен набор Windows Phone 7.1 SDK (для поддержки телефона с потребительским клеймом версии 7.5, добавляя путаницу), спектр поддерживаемых средств Silverlight 4 расширился. Обе версии Windows Phone 7.x использовали развитие CLR из Compact Framework.

Операционная система Windows Phone 8 поддерживает API-интерфейс Silverlight для обратной совместимости, но также и новый API-интерфейс Windows Phone Runtime, который более близок к API-интерфейсу WinRT, применяемому для приложений Windows Store. Кроме того, в Windows Phone 8 используется версия среды CoreCLR, а не CLR из Compact Framework. Сама по себе инфраструктура Silverlight теперь нежизнеспособна в плане будущего развития. Хотя я уверен, что многие разработчики по-прежнему применяют ее, никакие новые версии выпускаться не будут. Тем не менее, инфраструктура WinRT должна показаться разработчикам Silverlight очень знакомой. В Microsoft старались сделать переход от приложений Silverlight на приложения Windows Store как можно более гладким.

B.5.3. Micro Framework

Инфраструктура Micro Framework (`http://mng.bz/D9qy`) — это очень маленькая реализация .NET, предназначенная для запуска на сильно ограниченных устройствах. Она не поддерживает обобщения, использует интерпретацию, а не JIT-компиляцию, и поставляется с ограниченным набором классов, но она *все-таки* включает уровень

представления, построенный на основе WPF. Для сохранения пространства необходимо развертывать только те части инфраструктуры, которые действительно нужны — минимально это может занять не более 390 Кбайт. Очевидно, что она имеет довольно специфическую область применения, но возможность написания управляемого кода для встраиваемых устройств весьма привлекательна. Инфраструктура Micro Framework не подойдет для всех ситуаций — к примеру, она не является системой реального времени, — но там, где это применимо, она, скорее всего, значительно улучшит продуктивность разработчиков. Хронология выпусков Micro Framework вообще никак не согласована с настольной инфраструктурой: впервые она была замечена в SPOT-часах в 2004 году, но версия 1.0 вышла в 2006 году. С тех пор довольно быстро было выпущено несколько новых версий. Версия 4.0 инфраструктуры Micro Framework появилась 19 ноября 2009 года — и к моей радости и удивлению, большая часть этой версии была выпущена в виде открытого кода, регламентированного лицензией Apache 2.0. По разным причинам некоторые библиотеки, такие как реализации стека TCP/IP и криптографии, по-прежнему закрыты; такие сопровождающие библиотеки могут быть загружены в двоичной форме для специфичных архитектур.

B.5.4. Windows Runtime (WinRT)

WinRT не является еще одной версией .NET — это полностью новая платформа Windows, введенная в Windows 8. Она нацелена на предоставление среды с песочницей в архитектурах процессоров x86 и ARM и поддерживает множество языков — в первую очередь, C# и VB через .NET, C++/CX (новая разновидность C++, специально ориентированная на WinRT) и JavaScript. Это неуправляемый API-интерфейс, но он спроектирован для очень тесной интеграции с .NET, так что разработчики на C# и VB.NET могут на самом деле пользоваться теми же самыми API-интерфейсами, что и разработчики на C++/CX и JavaScript. Нет нужды в наличии какого-либо API-интерфейса оболочки вокруг них, как было в случае работы Win32 из Windows Forms. С самого начала этот API-интерфейс проектировался с учетом асинхронности; применение асинхронности является *естественным* путем разработки приложений для целевой платформы WinRT.

Поскольку Windows 8 — довольно новая операционная система, нам еще предстоит увидеть, насколько хорошо это получится в долгосрочной перспективе. Разработчики, желающие создавать приложения для запуска под управлением Windows 8, по-прежнему могут ориентироваться на традиционный рабочий стол, однако вполне ясно — в Microsoft уверены в том, что WinRT является важным движением вперед в рамках разработки клиентской стороны. В частности, API-интерфейсы Windows Phone и Windows Store в будущем, скорее всего, будут все больше и больше сближаться друг с другом.

B.6. Резюме

Из-за такого большого количества версий и различных компонентов очень легко запутаться — а еще проще запутать кого-то другого. В качестве финального совета (и он действительно финальный — довольно трудно выдать что-то глубокое и значимое в предметном указателе) я рекомендую вам стараться быть как можно более ясными при общении на эту тему с другими. Если вы не используете ничего кроме настольной инфраструктуры, то так и скажите. Если же вы собираетесь упомянуть номер версии, точно укажите, что имеете в виду — например, "3.0" может означать применение C# 2.0 и .NET 3.0 либо C# 3.0 и .NET 3.5. В конце концов, после прочтения этой книги вы не получите *абсолютно никаких оправданий*, утверждая о том, что используете "C# 3.5" или "C# 4.5", если только умышленно не пытаетесь вывести меня из душевного равновесия.

Предметный указатель

B

BCL (Base Class Library), 80

C

CCR (Concurrency and Coordination Runtime), 208
CLI (Common Language Infrastructure), 50
CLR (Common Language Runtime), 50
COM, 45

D

DLR (Dynamic Language Runtime), 45; 81; 407; 448; 468
DSL (Domain-specific language), 314; 451

G

GUID (Globally Unique Identifier), 229

I

IL (Intermediate Language), 49
IronPython, 457

L

LINQ (Language Integrated Query), 42; 320
LINQ to Objects, 388; 400
LINQ to Rx, 394; 397
LINQ to SQL, 364
LINQ to XML, 380

N

NaN (Not-a-number), 136
.NET Passport, 49

O

ORM (Object-relational mapping), 216

P

Parallel LINQ, 388
PIA (Primary Interop Assembly), 429

R

REPL (read, evaluate, print loop), 449

T

TPL (Task Parallel Library), 527

X

XAML (Extensible Application Markup Language), 216

А

Автомат
  конечный, 537
Агрегирование, 312; 567
Аргумент
  именованный, 409; 416
    с модификаторами out и ref, 417
  именованный в C# 4, 32
  оценка аргументов, 419
Асинхронность, 502
Атрибут
  InternalsVisibleTo, 234

Б

База данных, 365
Библиотека
  CCR, 208
  LINQ to Rx, 399
  TPL Dataflow, 547
  базовых типов (BCL), 80
  инфраструктуры, 50
  параллельных задач (TPL), 527
Буфер фиксированного размера, 213; 230

В

Вариантность, 122
Вектор, 585
Выполнение
  немедленное, 323
  отложенное, 322
Выражения запросов, 42
  вырожденные, 336
  интеграция выражений, 376

Г

Генерация, 573
Группирование, 312; 353; 398; 573
  с помощью конструкции group...by, 353

Д

Данные
  конвейер данных, 188
Действие, 165
Делегаты, 55; 77
  объединение и удаление делегатов, 60
  объявление типа делегата, 55
  создание экземпляра делегата, 57
Дерево выражения, 275
  простое, 276
Динамическое связывание, 447
Директива
  pragma, 213; 228
  #pragma checksum, 229
  #pragma warning, 228
Диспетчеризация
  множественная, 466

З

Запрос
  LINQ to SQL, 364
  в которых задействованы соединения, 370
  внутренний, 42
  выражения запросов, 42
  для одиночных узлов, 385

операции запросов, 386
параллельный, 392
продолжение запроса, 356

И

Идентификатор
GUID, 229
прозрачный, 339; 340
Инвариантность, 122; 433
Индексатор
именованный, 428
Инициализаторы
коллекций, 252
проекций, 261
Интерфейс, 581
API-, 380
IDynamicMetaObjectProvider, 495
IQueryable, 379
допускающий только чтение (.NET 4.5), 593
обобщенный, 107
явная реализация интерфейса, 69
Исключения, 521
Итератор, 188
Итерация
обобщенная, 114

К

Квантификатор, 577
Класс
BindingList<T>, 586
Collection<T>, 586
Dictionary<TKey, TValue>, 587
Dictionary<TKey,TValue>, 89
dynamic, 483
DynamicXElement, 489
Enumerable, 305
FakeQueryProvider, 379
InternalsVisibleToAttribute, 213
KeyedCollection<TKey, TItem>, 586
List<T>, 583
Nullable, 142
ObservableCollection<T>, 586
Queryable, 376
Queue<T>, 591
ReadOnlyCollection<T>, 587
ReadOnlyObservableCollection<T>, 587
SortedSet<T>, 590; 591
Stack<T>, 591
статический, 213; 220
сущностный, 365
Ключевое слово var, 243
Ковариантность, 67; 78; 122; 166; 433
возвращаемых типов, 69
возвращаемых типов делегатов, 168
обобщенная, 79
Код инициализации (C# 4), 33
Коллекция
запрашивание коллекций, 37
обобщенная, 581
строго типизированная в C# 2, 31
Компилятор JIT, 112
Конвейер данных, 188
Конкатенация, 568
Конструирование
декларативное, 382
Конструктор, 483
Контравариантность, 78; 122; 166; 433
для параметров делегата, 167
обобщенная, 79
типов параметров, 69
Кортеж, 108; 158

Л

Лямбда-выражение, 36; 39; 78; 266; 291; 376; 482

М

Массив, 584; 585
неявно типизированный, 256
параметров, 411
прямоугольный, 585
Метод
анонимный, 39; 171
асинхронный, 510; 511
каркасный, 536
обобщенный, 90
переопределение методов, 491
расширяющий, 37; 295; 298
статический, 483
частичный, 218
Множества, 589
Множество Мандельброта, 389
Модель, 365
Модификатор
?, 144
private, 223

Н

Нотация
точечная, 359

О

Обобщения, 117
Java, 130
обобщенная итерация, 114
обобщенные типы, 90
обобщенный словарь, 88
простые, 88
реализация обобщений, 104
Оператор
try/finally, 198
yield, 192
yield break, 197
yield return, 193

Операции, основанные на множествах, 578
Операция
  асинхронная, 556
  запросов, 324; 386
  перегрузка операций, 324

П

Параметр
  массив параметров, 411
  необязательный, 409
  формальный, 409
Переменная
  диапазона, 332
  локальная, 42
Платформа .NET 4, 108
Преобразование, 569
Проецирование, 397; 576
Пространство имен, 213
  глобальное, 225
Псевдоним
  внешний, 226

Р

Разделение, 575
Распаковка, 75; 139
Рефлексия, 117; 284

С

Сборка взаимодействия
  основная (PIA), 429
Сборка мусора, 57
Связывание, 449
Связыватель, 471
Селектор ключей, 353
Словарь, 587
  обобщенный, 88
Соединение, 342; 574
  внешнее, 370
  групповое, 347
  перекрестное, 349
  эквисоединение, 349
Сортировка, 34; 272; 579
  по длине имени пользователя, 340
  с использованием делегата, 36
  с использованием метода OrderBy(), 310
Списки, 583
Сравнение
  ссылочное, 106
Среда DLR, 468
Стек, 542

Т

Технология
  LINQ to Objects, 388
  LINQ to Rx, 394
  LINQ to XML, 380
  Parallel LINQ, 388
Тип
  анонимный, 79; 258
  аргументы типов, 90
  базовый, 138
  данных, 29
  делегата, 55
  динамический, 482
  допускающий значение null, 133
  закрытый, 91
  значения, 70; 71
  обобщенный, 90
  ограничения типов, 97
  открытый, 91
  параметр типа, 90
  сконструированный, 90
  соглашения об именовании, 93
  ссылочный, 71
  стирание типов, 131
  частичный, 213
Типизация
  динамическая, 449; 455
  неявная, 65; 243
  статическая, 456
  утиная, 465
  явная, 65
Точечная нотация, 359

У

Упаковка, 75; 139

Ф

Фильтрация, 34; 272; 397; 578
  с использованием конструкции where, 335
Функция
  асинхронная, 504
  анонимная, 267; 530

Х

Хеш-код, 109
  конфликт хеш-кодов, 588

Ш

Шаблон
  асинхронный, 544
  построителя (Builder), 256

Э

Эквивалентность, 572

Я

Язык
  C#, 49
  DSL, 314
  IL, 49
  IronPython, 457
  LINQ, 45
  XAML, 216
  исполняющая среда динамического языка (DLR), 45

C#
ДЛЯ ПРОФЕССИОНАЛОВ
ТОНКОСТИ ПРОГРАММИРОВАНИЯ

Третье издание

Новый перевод

Джон Скит

Если вы занимаетесь разработкой приложений .NET, то будете использовать C# как при построении сложного приложения уровня предприятия, так и при ускоренном написании какого-нибудь чернового приложения. В C# 5 можно делать удивительные вещи с помощью обобщений, лямбда-выражений, динамической типизации, LINQ, итераторных блоков и других средств. Однако прежде их необходимо должным образом изучить.

Это издание было полностью пересмотрено с целью раскрытия новых средств версии C# 5, включая тонкости написания сопровождаемого асинхронного кода. Вы увидите всю мощь языка C# в действии и научитесь работать с ценнейшими средствами, которые эффективно впишутся в применяемый набор инструментов. Кроме того, вы узнаете, как избегать скрытых ловушек при программировании на C# с помощью простых и понятных объяснений вопросов, касающихся внутреннего устройства языка.

ЧТО ВНУТРИ

- Обновления, появившиеся в C# 5
- Новое средство async/await
- Как и почему работает C#

Джон Скит — старший инженер по программному обеспечению в Google, а также видный участник групп новостей, групп пользователей, международных конференций и сайта Stack Overflow Q&A. Большая часть ежедневной работы Джона связана с кодированием на Java, но его сердце принадлежит C#.

Категория: программирование
Предмет рассмотрения: C# / Microsoft .NET
Уровень: для пользователей средней и высокой квалификации