

Исчерпывающее руководство

Профессиональное программирование

Scala

5-е издание



Обновлено для Scala 3

 ПИТЕР®
artima

Мартин Одерски
Лекс Спун
Билл Веннерс
Фрэнк Сомерс

Programming in Scala

Fifth Edition

Martin Odersky, Lex Spoon, Bill Venners,
and Frank Sommers

artima

ARTIMA PRESS

WALNUT CREEK, CALIFORNIA

Профессиональное программирование

Scala

5-е издание

Мартин Одерски
Лекс Спун
Билл Веннерс
Фрэнк Соммерс



Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.1
УДК 004.43
О-41

Одерски Мартин, Спун Лекс, Веннерс Билл, Соммерс Фрэнк

О-41 Scala. Профессиональное программирование. 5-е изд.. — СПб.: Питер, 2022. — 608 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1914-1

«Scala. Профессиональное программирование» — главная книга по Scala, популярному языку для платформы Java, в котором сочетаются концепции объектно-ориентированного и функционального программирования, благодаря чему он превращается в уникальное и мощное средство разработки.

Этот авторитетный труд, написанный создателями Scala, поможет вам пошагово изучить язык и идеи, лежащие в его основе.

Пятое издание значительно обновлено, чтобы охватить многочисленные изменения, появившиеся в Scala 3.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с Artima Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0997148008 англ.

© 2019 Martin Odersky, Lex Spoon, Bill Venner, Frank Sommers.
All rights reserved

ISBN 978-5-4461-1914-1

© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Библиотека программиста», 2022
© Павлов А., перевод с английского языка, 2022

Краткое содержание

Отзывы на предыдущие издания книги	18
Предисловие	22
Благодарности	27
Введение	31
Глава 1. Масштабируемый язык	37
Глава 2. Первые шаги в Scala	56
Глава 3. Дальнейшие шаги в Scala	69
Глава 4. Классы и объекты	91
Глава 5. Основные типы и операции	104
Глава 6. Функциональные объекты	125
Глава 7. Встроенные управляющие конструкции	145
Глава 8. Функции и замыкания	168
Глава 9. Управляющие абстракции	189
Глава 10. Композиция и наследование	203
Глава 11. Трейты	229
Глава 12. Пакеты, импорты и экспорты	247
Глава 13. Сопоставление с образцом	266
Глава 14. Работа со списками	299
Глава 15. Работа с другими коллекциями	331

Глава 16. Изменяемые объекты	350
Глава 17. Иерархия Scala	372
Глава 18. Параметризация типов	388
Глава 19. Перечисления	410
Глава 20. Абстрактные члены	421
Глава 21. Гивены	447
Глава 22. Методы расширения	468
Глава 23. Классы типов	482
Глава 24. Углубленное изучение коллекций	511
Глава 25. Утверждения и тесты	576
Глоссарий	588
Библиография	604
Об авторах	607

Оглавление

Отзывы на предыдущие издания книги	18
Предисловие	22
Благодарности	27
Введение	31
Целевая аудитория	31
Как пользоваться книгой	31
Как изучать Scala	32
Условные обозначения	33
Структура книги	33
Ресурсы	35
Исходный код	36
От издательства	36
Глава 1. Масштабируемый язык	37
1.1. Язык, который растет вместе с вами	38
Растут новые типы.	39
Растут новые управляющие конструкции	41
1.2. Почему язык Scala масштабируемый?	41
Scala — объектно-ориентированный язык.	42
Scala — функциональный язык.	43
1.3. Почему именно Scala	45
Scala — совместимый язык.	46
Scala — лаконичный язык	47
Scala — высокоуровневый язык	48
Scala — статически типизированный язык	50
1.4. Истоки Scala	53
Резюме	55

Глава 2. Первые шаги в Scala	56
Шаг 1. Осваиваем Scala REPL	57
Шаг 2. Объявляем переменные	58
Шаг 3. Определяем функции	60
Шаг 4. Пишем Scala-скрипты	62
Шаг 5. Организуем цикл с while и принимаем решение с if	64
Шаг 6. Перебираем элементы с foreach и for-do	66
Резюме	68
Глава 3. Дальнейшие шаги в Scala	69
Шаг 7. Параметризуем массивы типами	69
Шаг 8. Используем списки	73
Шаг 9. Используем кортежи	78
Шаг 10. Используем множества и отображения	79
Шаг 11. Учимся распознавать функциональный стиль	84
Шаг 12. Преобразование с отображениями и for-yield	87
Резюме	90
Глава 4. Классы и объекты	91
4.1. Классы, поля и методы	91
4.2. Когда подразумевается использование точки с запятой	96
4.3. Объекты-одиночки	96
4.4. Case-классы	99
4.5. Приложение на языке Scala	101
Резюме	103
Глава 5. Основные типы и операции	104
5.1. Некоторые основные типы	104
5.2. Литералы	105
Целочисленные литералы	106
Литералы чисел с плавающей точкой.	107
Большие числовые литералы	107
Символьные литералы	108
Строковые литералы	109
Булевы литералы.	110
5.3. Интерполяция строк	110
5.4. Все операторы являются методами	112
5.5. Арифметические операции	115
5.6. Отношения и логические операции	116

5.7. Поразрядные операции	117
5.8. Равенство объектов	119
5.9. Приоритет и ассоциативность операторов	120
5.10. Обогащающие операции	123
Резюме	124
Глава 6. Функциональные объекты	125
6.1. Спецификация класса Rational	125
6.2. Конструирование класса Rational	126
6.3. Переопределение метода toString	128
6.4. Проверка соблюдения предварительных условий	129
6.5. Добавление полей	130
6.6. Собственные ссылки	132
6.7. Вспомогательные конструкторы	132
6.8. Приватные поля и методы	134
6.9. Определение операторов	135
6.10. Идентификаторы в Scala	137
6.11. Перегрузка методов	140
6.12. Методы расширения	142
6.13. Предостережение	143
Резюме	143
Глава 7. Встроенные управляющие конструкции	145
7.1. Выражения if	146
7.2. Циклы while	147
7.3. Выражения for	150
Обход элементов коллекций	150
Фильтрация	152
Вложенные итерации	153
Привязки промежуточных переменных	153
Создание новой коллекции	154
7.4. Обработка исключений с помощью выражений try	155
Генерация исключений	155
Перехват исключений	156
Условие finally	157
Выдача значения	157
7.5. Выражения match	158
7.6. Программирование без break и continue	160

7.7. Область видимости переменных	162
7.8. Рефакторинг кода, написанного в императивном стиле	165
7.9. Резюме	167
Глава 8. Функции и замыкания	168
8.1. Методы	168
8.2. Локальные функции	169
8.3. Функции первого класса	171
8.4. Краткие формы функциональных литералов	173
8.5. Синтаксис заместителя	173
8.6. Частично примененные функции	174
8.7. Замыкания	177
8.8. Специальные формы вызова функций	180
Повторяющиеся параметры	180
Именованные аргументы	181
Значения параметров по умолчанию	182
8.9. Тип SAM	183
8.10. Хвостовая рекурсия	184
Трассировка функций с хвостовой рекурсией	185
Ограничения хвостовой рекурсии	187
Резюме	188
Глава 9. Управляющие абстракции	189
9.1. Сокращение повторяемости кода	189
9.2. Упрощение клиентского кода	193
9.3. Карринг	195
9.4. Создание новых управляющих конструкций	196
9.5. Передача параметров по имени	199
Резюме	202
Глава 10. Композиция и наследование	203
10.1. Библиотека двумерной разметки	203
10.2. Абстрактные классы	204
10.3. Определяем методы без параметров	205
10.4. Расширяем классы	208
10.5. Переопределяем методы и поля	210
10.6. Определяем параметрические поля	211
10.7. Вызываем конструктор суперкласса	212

10.8. Используем модификатор <code>override</code>	213
10.9. Полиморфизм и динамическое связывание	215
10.10. Объявляем финальные элементы	217
10.11. Используем композицию и наследование	218
10.12. Реализуем методы <code>above</code> , <code>beside</code> и <code>toString</code>	220
10.13. Определяем фабричный объект	223
10.14. Методы <code>heighten</code> и <code>widen</code>	224
10.15. Собираем все вместе	227
Резюме	228
Глава 11. Трейты	229
11.1. Как работают трейты	229
11.2. Сравнение «тонких» и «толстых» интерфейсов	232
11.3. Трейты как наращиваемые модификации	235
11.4. Почему не используется множественное наследование	239
11.5. Параметры трейтов	243
Резюме	245
Глава 12. Пакеты, импорты и экспорты	247
12.1. Помещение кода в пакеты	247
12.2. Краткая форма доступа к родственному коду	249
12.3. Импортирование кода	252
12.4. Неявное импортирование	255
12.5. Модификаторы доступа	256
Приватные члены	256
Защищенные члены	257
Публичные члены	258
Область защиты	258
Видимость и объекты-компаньоны	260
12.6. Определения верхнего уровня	261
12.7. Экспорты	262
Резюме	265
Глава 13. Сопоставление с образцом	266
13.1. Простой пример	266
case-классы	267
Сопоставление с образцом	268
Сравнение <code>match</code> со <code>switch</code>	270

13.2. Разновидности паттернов	271
Подстановочные паттерны.	271
Паттерны-константы	272
Паттерны-переменные	272
Паттерны-конструкторы.	274
Паттерны-последовательности.	275
Паттерны-кортежи.	276
Типизированные паттерны.	276
Затирание типов	279
Привязка переменной	280
13.3. Ограждение образца	280
13.4. Наложение паттернов	281
13.5. Запечатанные классы	283
13.6. Сопоставление паттерна Options	285
13.7. Паттерны повсюду	286
Паттерны в определениях переменных	286
Последовательности вариантов в качестве частично примененных функций.	287
Паттерны в выражениях for	290
13.8. Большой пример	291
Резюме	298
Глава 14. Работа со списками	299
14.1. Литералы списков	299
14.2. Тип List	300
14.3. Создание списков	300
14.4. Основные операции над списками	301
14.5. Паттерны-списки	302
14.6. Методы первого порядка класса List	304
Конкатенация двух списков	304
Принцип «разделяй и властвуй»	305
Получение длины списка: length	307
Обращение к концу списка: init и last	307
Реверсирование списков: reverse	308
Префиксы и суффиксы: drop, take и splitAt	309
Выбор элемента: apply и indices	310
Линеаризация списка списков: flatten.	310
Объединение списков: zip и unzip.	311
Отображение списков: toString и mkString	311

Преобразование списков: <code>iterator</code> , <code>toArray</code> , <code>copyToArray</code>	312
Пример: сортировка слиянием	313
14.7. Методы высшего порядка класса <code>List</code>	315
Отображения списков: <code>map</code> , <code>flatMap</code> и <code>foreach</code>	316
Фильтрация списков: <code>filter</code> , <code>partition</code> , <code>find</code> , <code>takeWhile</code> , <code>dropWhile</code> и <code>span</code>	317
Применение предикатов к спискам: <code>forall</code> и <code>exists</code>	318
Свертка списков: <code>foldLeft</code> и <code>foldRight</code>	319
Пример: реверсирование списков с помощью свертки	321
Сортировка списков: <code>sortWith</code>	322
14.8. Методы объекта <code>List</code>	323
Создание списков из их элементов: <code>List.apply</code>	323
Создание диапазона чисел: <code>List.range</code>	323
Создание единообразных списков: <code>List.fill</code>	324
Табулирование функции: <code>List.tabulate</code>	324
Конкатенация нескольких списков: <code>List.concat</code>	324
14.9. Совместная обработка нескольких списков	325
14.10. Понимание имеющегося в <code>Scala</code> алгоритма вывода типов	326
Резюме	330
Глава 15. Работа с другими коллекциями	331
15.1. Последовательности	331
Списки.	331
Массивы	332
Буферы списков.	333
Буферы массивов.	333
Строки (реализуемые через <code>StringOps</code>).	334
15.2. Множества и отображения	335
Использование множеств.	336
Применение отображений	338
Множества и отображения, используемые по умолчанию	340
Отсортированные множества и отображения	341
15.3. Выбор между изменяемыми или неизменяемыми коллекциями	342
15.4. Инициализация коллекций	344
Преобразование в массив или список.	346
Преобразования между изменяемыми и неизменяемыми множествами и отображениями	347
15.5. Кортежи	347
Резюме	349

Глава 16. Изменяемые объекты	350
16.1. Что делает объект изменяемым	350
16.2. Переназначаемые переменные и свойства	352
16.3. Практический пример: моделирование дискретных событий	356
16.4. Язык для цифровых схем	357
16.5. API моделирования	360
16.6. Моделирование электронной логической схемы	364
Класс Wire	365
Метод inverter	366
Методы andGate и orGate	367
Вывод симуляции.	368
Запуск симулятора.	368
Резюме	370
Глава 17. Иерархия Scala	372
17.1. Иерархия классов Scala	372
17.2. Как реализованы примитивы	376
17.3. Низшие типы	378
17.4. Определение собственных классов значений	379
Уход от монокультурности типов	380
17.5. Типы пересечений	382
17.6. Типы объединения	383
17.7. Прозрачные трейты	386
Резюме	387
Глава 18. Параметризация типов	388
18.1. Функциональные очереди	388
18.2. Соккрытие информации	392
Приватные конструкторы и фабричные методы	392
Альтернативный вариант: приватные классы	393
18.3. Аннотации вариантности	394
Вариантность и массивы	397
18.4. Проверка аннотаций вариантности	399
18.5. Нижние ограничители	402
18.6. Контравариантность	404
18.7. Верхние ограничители	407
Резюме	409

Глава 19. Перечисления	410
19.1. Перечисляемые типы данных	410
19.2. Алгебраические типы данных	414
19.3. Обобщенные ADT	416
19.4. Что делает типы ADT алгебраическими	417
Резюме	420
Глава 20. Абстрактные члены	421
20.1. Краткий обзор абстрактных членов	421
20.2. Члены-типы	422
20.3. Абстрактные val-переменные	423
20.4. Абстрактные var-переменные	424
20.5. Инициализация абстрактных val-переменных	425
Параметрические поля трейтов	427
Ленивые val-переменные	428
20.6. Абстрактные типы	431
20.7. Типы, зависящие от пути	434
20.8. Уточняющие типы	436
20.9. Практический пример: работа с валютой	437
Резюме	446
Глава 21. Гивены	447
21.1. Как это работает	448
21.2. Параметризованные given-типы	451
21.3. Анонимные given-экземпляры	455
21.4. Параметризованные given-экземпляры в виде классов типов	456
21.5. Импорт гивенов	459
21.6. Правила для контекстных параметров	461
21.7. Когда подходит сразу несколько гивенов	463
21.8. Отладка гивенов	465
Резюме	467
Глава 22. Методы расширения	468
22.1. Основы	468
22.2. Обобщенные расширения	471
22.3. Групповые расширения	472
22.4. Использование класса типов	474

22.5. Методы расширения для заданных экземпляров	477
22.6. Где Scala ищет методы расширения	480
Резюме	481
Глава 23. Классы типов	482
23.1. Зачем нужны классы типов	482
23.2. Границы контекста	487
23.3. Главные методы	490
23.4. Многостороннее равенство	494
23.5. Неявные преобразования	499
23.6. Пример использования класса типов: сериализация JSON	502
Резюме	510
Глава 24. Углубленное изучение коллекций	511
24.1. Изменяемые и неизменяемые коллекции	513
24.2. Согласованность коллекций	514
24.3. Трейт Iterable	516
Подкатегории Iterable	523
24.4. Трейты последовательностей Seq, IndexedSeq и LinearSeq	523
Буферы	528
24.5. Множества	530
24.6. Отображения	534
24.7. Конкретные классы неизменяемых коллекций	538
Списки.	539
Ленивые списки.	539
Неизменяемые ArraySeq.	540
Векторы.	541
Неизменяемые очереди	542
Диапазоны	543
Сжатые коллекции HAMT	543
Красно-черные деревья	544
Неизменяемые битовые множества	544
Векторные отображения	545
Списочные отображения	545
24.8. Конкретные классы изменяемых коллекций	546
Буферы массивов.	546
Буферы списков.	546
Построители строк.	547

ArrayDeque	547
Очереди	547
Стеки	548
Изменяемые ArraySeq	548
Хеш-таблицы	548
Слабые хеш-отображения	549
Совместно используемые отображения	550
Изменяемые битовые множества	550
24.9. Массивы	551
24.10. Строки	555
24.11. Характеристики производительности	556
24.12. Равенство	558
24.13. Представления	559
24.14. Итераторы	563
Буферизованные итераторы.	570
24.15. Создание коллекций с нуля	571
24.16. Преобразования между коллекциями Java и Scala	573
Резюме	575
Глава 25. Утверждения и тесты	576
25.1. Утверждения	576
25.2. Тестирование в Scala	578
25.3. Информативные отчеты об ошибках	579
25.4. Использование тестов в качестве спецификаций	581
25.5. Тестирование на основе свойств	584
25.6. Подготовка и проведение тестов	586
Резюме	587
Глоссарий	588
Библиография	604
Об авторах	607

Отзывы на предыдущие издания книги

Это, вероятно, одна из лучших книг по программированию, которые я когда-либо читал. Мне нравятся стиль изложения, краткость и подробные объяснения. Книга, кажется, отвечает на каждый вопрос, который приходит мне в голову, — она всегда на шаг впереди меня. Авторы не просто дают вам код — они объясняют самую суть, чтобы вы действительно понимали, о чем идет речь. Мне это очень нравится.

*Кен Эджервари (Ken Egervari),
ведущий разработчик программного обеспечения*

Эта книга написана четко, основательно и понятно. В ней есть отличные примеры и полезные советы. Она позволила нашей организации быстро и эффективно освоить язык Scala. Эта книга отлично подходит для любого программиста, который хочет разобраться в гибкости и элегантности этого языка.

*Ларри Моррони (Larry Morroni),
владелец Morroni Technologies, Inc.*

Это отличный учебник по языку Scala. В нем хорошо проработана каждая глава, основанная на концепциях и примерах, описанных в предыдущих главах. Подробно объясняются конструкции языка, часто приводятся примеры того, чем язык отличается от Java. Помимо основного языка, также рассматриваются такие библиотеки, как контейнеры и факторы.

С этим пособием действительно легко работать, и, вероятно, это одна из лучших технических книг, которые я читал за последнее время. Я настоятельно рекомендую эту книгу любому программисту, желающему узнать больше о языке Scala.

Мэтью Тодд (Matthew Todd)

Я поистине впечатлен работой, проделанной авторами этой книги. Она бесценное руководство по языку Scala — средству для эффективного написания кода, постоянный источник вдохновения для разработки и реализации масштабируемого ПО. Если бы только у меня была Scala в ее нынешнем зрелом состоянии и эта книга на моем столе еще в 2003 году, когда я участвовал в разработке и внедрении части инфраструктуры портала Олимпийских игр 2004 года в Афинах!

Всем читателям: независимо от вашего опыта, я думаю, программирование на Scala покажется вам очень гибким, и эта книга станет вашим верным другом в этом путешествии.

*Христос К. К. Ловердос (Christos KK Loverdos),
консультант по программному обеспечению, исследователь*

«Scala. Профессиональное программирование» — это превосходное углубленное введение в Scala, а также отличный справочник. Я бы сказал, что эта книга занимает видное место на моей полке, если не считать, что я почти везде ношу ее с собой.

*Брайан Клэппер (Brian Clapper),
президент ArdenTex, Inc.*

Отличная книга, хорошо написанная, с вдумчивыми примерами. Рекомендую как опытным программистам, так и новичкам.

Говард Ловатт (Howard Lovatt)

Эта книга рассказывает не только о том, как разрабатывать программы на языке Scala, но и, что более важно, о том, зачем это делать. Прагматичный подход книги к представлению возможностей сочетания объектно-ориентированного и функционального программирования не оставляет читателю никаких сомнений в том, что такое Scala на самом деле.

*Доктор Эрвин Варга (Ervin Varga),
генеральный директор и основатель EXPRO I.T. Consulting*

Это отличное введение в функциональное программирование для объектно-ориентированных программистов. Моей основной целью было изучение функционального программирования, и на этом пути меня поджидали некоторые приятные сюрпризы Scala, такие как case-классы и сопоставление с образцом. Scala — интригующий язык, и эта книга хорошо его раскрывает.

В подобного рода книгах всегда есть риск сказать слишком много или слишком мало. Я считаю, что в «Scala. Профессиональное программирование» как раз достигнут идеальный баланс.

Джефф Хон (Jeff Hon), программист-аналитик

Понятность и техническая полнота — отличительные черты любой хорошо написанной книги, и я поздравляю Мартина Одерски, Лекса Спуна и Билла Веннерса с действительно очень хорошо выполненной работой! Книга «Scala. Профессиональное программирование» начинается с описания базовых концепций и поднимает пользователя до среднего уровня и выше. Эту книгу, безусловно, необходимо купить всем, кто хочет изучить Scala.

*Джаган Намби (Jagan Nambi),
архитектор корпоративных бизнес-решений,
GMAC Financial Services*

Читать эту книгу — одно удовольствие. Это одна из тех хорошо написанных технических книг, которые обеспечивают глубокое и всестороннее освещение предмета исключительно кратким и элегантным образом. Книга построена очень естественно и логично. Она одинаково хорошо подходит как для инженера-любителя, который просто хочет быть в курсе текущих тенденций, так и для профессионала, стремящегося к глубокому пониманию основных функций языка и его дизайна. Настоятельно рекомендую эту книгу всем, кто интересуется функциональным программированием в целом. Разработчикам Scala эту книгу, безусловно, необходимо прочитать.

Игорь Хлыстов, архитектор программного обеспечения/ведущий программист, Greystone Inc.

По мере изучения книги «Scala. Профессиональное программирование» начинаешь понимать, какое огромное количество труда в нее вложено. Я никогда раньше не читал настолько всеобъемлющего пособия для новичков. Большинство авторов, стараясь объяснить материал на доступном уровне и не запутать читателя, опускают некоторые сложные аспекты. Это оставляет довольно неприятный осадок, так как возникает навязчивое ощущение, что материал до конца не усвоен. Всегда есть остаточная «магия», которую не объяснили и о которой читатель вообще не может судить. В этой книге такого никогда не происходит, она никогда не принимает ничего как должное: подробно описана каждая деталь и, если этого

недостаточно, дается ссылка на более глубокое объяснение. В самом деле, в тексте много перекрестных ссылок, поэтому составить полную картину сложной темы относительно легко.

*Джеральд Леффлер (Gerald Loeffler),
Java-архитектор корпоративных бизнес-решений*

Во времена, когда хорошие книги по программированию редки, «Scala. Профессиональное программирование» Мартина Одерски, Лекса Спуна и Билла Веннерса действительно выделяется — это отличное введение для программистов среднего уровня. Здесь вы найдете все необходимое, чтобы выучить этот многообещающий язык.

Кристиан Нойкирхен (Christian Neukirchen)

Предисловие

Забавно наблюдать за рождением нового языка программирования. Большинство тех, кто пользуется языками программирования — будь то новичок или профессионал, — не задумываются об их происхождении. Подобно молотку или топору, язык программирования — это инструмент, который позволяет нам выполнять свою работу. Мы редко размышляем о том, как появился этот инструмент, каков был процесс его разработки. У нас может быть свое мнение о его синтаксисе, но обычно просто принимаем его как есть и движемся вперед.

Однако создание языка программирования открывает совершенно иную перспективу. Появляются новые возможности того, что могло показаться безграничным. Но в то же время язык программирования должен удовлетворять бесконечному списку ограничений. Странное противоречие.

Новые языки программирования создаются по многим причинам: из-за личного желания сделать что-то свое, глубокой академической проницательности, технического долга или анализа других архитектур компиляторов и даже политики. Scala 3 — это комбинация некоторых из них.

Какая бы ни была комбинация, все началось с того, что однажды Мартин Одерски (Martin Odersky) исчез, появившись несколько дней спустя, чтобы объявить на собрании исследовательской группы, что он начал эксперименты по воплощению DOT-исчисления в жизнь, написав новый компилятор с нуля¹.

Мы были группой аспирантов и кандидатов, которые до недавнего времени играли важную роль в разработке и сопровождении Scala 2. В то время Scala достигал, казалось, непостижимых высот успеха, особенно для эзотерического и академического языка программирования из школы с забавным названием в Швейцарии. Scala недавно стал популярным среди стартапов в Области

¹ DOT-исчисления, или зависимые типы объектов, — это попытка теоретически обосновать систему типов языка Scala.

залива Сан-Франциско, и для поддержки, сопровождения и управления выпусками Scala 2 недавно была создана компания Typesafe, позже названная Lightbend. Так почему же вдруг появился новый компилятор и, возможно, новый и другой язык программирования? Большинство были настроены скептически. Мартина это не испугало.

Прошли месяцы. Как по будильнику, в 12 часов дня вся лаборатория стягивалась в коридор, соединяющий все наши офисы. После того как нас собиралось изрядное количество, мы вместе с Марином отправлялись в один из многочисленных буфетов ФПШЛ (Федеральная политехническая школа Лозанны), чтобы пообедать, а затем выпить кофе. Каждый день во время этого ритуала идеи для нового компилятора были постоянной темой для обсуждения. Обсуждения были жаркими, мы прыгали с одной темы на другую: от чего-то, что «на 150 %» совместимо со Scala 2 (чтобы избежать фиаско, как при переходе с Python 2 на Python 3), до создания нового языка с полным спектром зависимых типов.

Один за другим, все скептики в исследовательской группе в итоге начинали увлекаться каким-либо из аспектов Scala 3, будь то оптимизация реализации проверки типов, новая архитектура компилятора или мощные дополнения к системе типов. Со временем большая часть сообщества также пришла к мысли, что Scala 3 значительно улучшен по сравнению со Scala 2. У разных людей были разные причины для этого. Для некоторых это было улучшение читаемости за счет решения сделать необязательными фигурные и круглые скобки вокруг условий в условных операторах. Для других это были улучшения в системе типов, например сопоставление типов для улучшенного программирования на уровне типов. Список был бесконечным.

Вместо того чтобы слепо продвигаться вперед в разработке Scala 3, основываясь лишь на догадках, я могу с уверенностью сказать, что Scala 3 — это результат большого изучения решений прошлого и многолетнего взаимодействия с исследовательской группой ФПШЛ и сообществом Scala. И не было другого выхода, кроме как начать с чистого листа и строить на чистом фундаменте. Благодаря такому подходу с нуля возник, по сути, новый язык программирования, и имя ему — Scala 3. Конечно, он может быть совместим со Scala 2 и считаться третьим крупным релизом уже существующего языка программирования, но не дайте себя обмануть: Scala 3 представляет собой существенную оптимизацию многих экспериментальных идей, впервые появившихся в Scala 2.

Возможно, самое уникальное в Scala 3 — это то, что случилось с имплицитами. Scala с момента своего создания использовался умными программистами

для достижения функциональности, которую мало кто считал возможной даже с учетом набора функций Scala, не говоря уже о том, для чего этот язык был разработан. Функция, ранее известная как имплициты, — это, пожалуй, самая известная функция Scala, которая применялась для использования Scala 2 самым неожиданным образом.

Примеры использования имплицитов включают в себя добавление метода к классу задним числом без расширения и повторной компиляции этого класса. Или, учитывая сигнатуру типа, используемую в некотором контексте, автоматический выбор правильной реализации для этого контекста. Это лишь верхушка айсберга — мы даже написали исследовательскую работу, в которой пытались каталогизировать множество способов, которыми разработчики использовали имплициты [Kri19].

Это все равно что дать пользователю кнопки и рычаги и предоставить ему возможность построить отлаженный механизм, как механический калькулятор (рис. 1). Но часто вместо этого получается что-то наподобие кинетической скульптуры Тео Янсена (Theo Jansen) (рис. 2), а не что-то имеющее очевидное применение¹. Проще говоря, вы даете сообществу программистов нечто столь же простое, как кнопка и рычаг, и бесстрашные люди будут искать творческие способы их использования. Такова природа человека. Но, возможно, ошибкой Scala 2 была идея предоставить в первую очередь что-то столь же универсальное, как кнопки и рычаги.

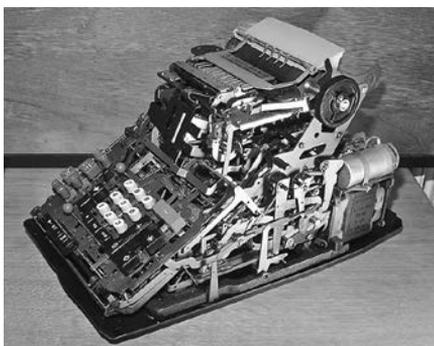


Рис. 1. Что мы задумали...



Рис. 2. ...И что получили

¹ Динамические изображения кинетических скульптур Тео Янсена, получивших название Strandbeest, см. в видео: https://www.youtube.com/watch?v=LewVEF2B_pM.

Дело здесь в том, что в Scala 2 был бесконечный набор возможностей для использования имплицитов, что потребовало исследовательской работы, и сообщество в целом не могло прийти к согласию относительно того, как разумно их использовать. Никакая языковая функция не должна иметь столь туманного предназначения. И тем не менее они были. Имплициты рассматривались одними как уникальная и мощная особенность Scala, которой, по сути, не было ни в одном другом языке, а другими — как загадочный и часто разочаровывающий механизм, который агрессивно переписывал ваш код, чтобы он стал чем-то другим.

Возможно, вы слышали часто повторяемую мантру о том, что Scala 3 во многих отношениях представляет собой упрощение Scala/Scala 2. История имплицитов — отличный тому пример. Осознавая все кульбиты, которые программисты делали с имплицитами в попытке реализовать более широкие шаблоны программирования, такие как вывод классов, Мартин, не без помощи, пришел к выводу, что нам не следует сосредотачиваться на имплицитах как на механизме, который люди могут использовать в самых общих случаях. Скорее, мы должны сосредоточиться на том, что программисты хотят делать с имплицитами, и сделать это проще и эффективнее. Отсюда и мантра: «Scala 3 фокусируется на намерении, а не на механизме».

В Scala 3 вместо того, чтобы сосредоточиться на универсальности имплицитов как механизма, было принято решение сосредоточиться на конкретных сценариях их использования, которые разработчики имели в виду при выборе имплицитов в первую очередь, и сделать эти шаблоны более доступными для использования по прямому назначению. Примеры включают в себя передачу контекстной или конфигурационной информации неявными методами, без необходимости передавать повторяющиеся аргументы программисту, добавление методов в классы задним числом и преобразование между типами, такими как `Ints` и `Doubles`, во время вычислений. Теперь Scala 3 делает эти варианты использования доступными для программистов без необходимости применять некоторую «глубокую» интуицию в отношении того, как компилятор Scala решит использовать имплициты. Вместо этого вы можете просто сосредоточиться на таких задачах, как «добавить метод `foo` в класс `Bar` без необходимости его перекомпиляции». Докторская степень не требуется. Просто замените предыдущее понятие «неявный» другими, более прямыми ключевыми словами, которые соответствуют конкретным вариантам использования, например такими, как `given` и `using`. Подробнее об этом читайте в главах 21 и 22.

Эта история о том, что «приоритет отдается намерению над механизмом», не останавливается на пересмотре имплицитов. Скорее, философия затрагивает почти все аспекты языка. Примерами могут служить дополнения и оптимизация многих аспектов системы типов Scala от типов объединения и перечислений до сопоставления типов или даже чистки синтаксиса Scala: необязательные фигурные скобки для улучшения читаемости и более читаемый «тихий» синтаксис для конструкций `if`, `else` и `while`, в результате чего условные выражения больше напоминают английский язык, чем машинный код.

Не верьте мне на слово. Независимо от того, являетесь ли вы новичком в Scala или опытным разработчиком, я надеюсь, что вы найдете новшества, вошедшие в Scala 3, такими же свежими и простыми, какими их нахожу я!

*Хизер Миллер (Heather Miller),
Лозанна, Швейцария,
1 июня 2021 г.*

Благодарности

Мы благодарны за вклад в эту книгу многим людям.

Сам язык Scala — плод усилий множества специалистов. Свой вклад в проектирование и реализацию версии 1.0 внесли Филипп Альтер (Philippe Altherr), Винсент Кремет (Vincent Cremet), Жиль Дюбоше (Gilles Dubochet), Бурак Эмир (Burak Emir), Стефан Мишель (Stéphane Micheloud), Николай Михайлов (Nikolay Mihaylov), Мишель Шинц (Michel Schinz), Эрик Стенман (Erik Stenman) и Матиас Зенгер (Matthias Zenger). К разработке второй и текущей версий языка, а также инструментальных средств подключились Фил Багвелл (Phil Bagwell), Антонио Куней (Antonio Cuneì), Юлиан Драгос (Iulian Dragos), Жиль Дюбоше (Gilles Dubochet), Мигель Гарсиа (Miguel Garcia), Филипп Халлер (Philipp Haller), Шон Макдирмид (Sean McDirmid), Инго Майер (Ingo Maier), Донна Малайери (Donna Malayeri), Адриан Мурс (Adriaan Moors), Хуберт Плоциничак (Hubert Plociniczak), Пол Филлипс (Paul Phillips), Александр Прокопец (Aleksandar Prokorec), Тиарк Ромпф (Tiark Rompf), Лукас Рыц (Lukas Rytz) и Джефффри Уошберн (Geoffrey Washburn).

Следует также упомянуть тех, кто участвовал в работе над структурой языка. Эти люди любезно делились с нами своими идеями в оживленных и вдохновляющих дискуссиях, вносили важные фрагменты кода в открытую разработку и делали весьма ценные замечания по поводу предыдущих версий. Это Гилад Браха (Gilad Bracha), Натан Бронсон (Nathan Bronson), Коаюан (Саоюан), Эймон Кэннон (Aemon Cannon), Крейг Чамберс (Craig Chambers), Крис Конрад (Chris Conrad), Эрик Эрнст (Erik Ernst), Матиас Феллизен (Matthias Felleisen), Марк Харра (Mark Harrah), Шрирам Кришнамурти (Shriram Krishnamurti), Гэри Ливенс (Gary Leavens), Дэвид Макивер (David MacIver), Себастьян Манит (Sebastian Maneth), Рикард Нильссон (Rickard Nilsson), Эрик Мейер (Erik Meijer), Лалит Пант (Lalit Pant), Дэвид Поллак (David Pollak), Джон Претти (Jon Pretty), Клаус Остерман (Klaus Ostermann), Хорхе Ортис (Jorge Ortiz), Дидье Реми (Didier Rémy), Майлз Сабин (Miles Sabin), Виджей Сарасват (Vijay Saraswat), Даниэль Спивак (Daniel Spiewak), Джеймс Страчан (James Strachan), Дон Симе (Don Syme), Эрик Торреборре (Erik Torreborre), Мэдс Торгерсен (Mads Torgersen), Филип Уодлер (Philip

Wadler), Джейми Уэбб (Jamie Webb), Джон Уильямс (John Williams), Кевин Райт (Kevin Wright) и Джейсон Зауг (Jason Zaugg). Очень полезные отзывы, которые помогли нам улучшить язык и его инструментальные средства, были получены от людей, подписанных на наши рассылки по Scala.

Джордж Бергер (George Berger) усердно работал над тем, чтобы процесс создания и размещения книги в интернете протекал гладко. Как результат, в данном проекте не было никаких технических сбоев.

Ценные отзывы о начальных вариантах текста книги были получены нами от многих людей. наших благодарностей заслуживают Эрик Армстронг (Eric Armstrong), Джордж Бергер (George Berger), Алекс Блевитт (Alex Blewitt), Гилад Браха (Gilad Bracha), Уильям Кук (William Cook), Брюс Экель (Bruce Eckel), Стефан Мишель (Stéphane Micheloud), Тод Мильштейн (Todd Millstein), Дэвид Поллак (David Pollak), Филип Уодлер (Philip Wadler) и Матиас Зенгер (Matthias Zenger). Спасибо также представителям Silicon Valley Patterns group за их весьма полезный обзор. Это Дейв Астелс (Dave Astels), Трейси Бялик (Tracy Bialik), Джон Брюер (John Brewer), Эндрю Чейз (Andrew Chase), Брэдфорд Кросс (Bradford Cross), Рауль Дюк (Raoul Duke), Джон П. Эйрих (John P. Eugich), Стивен Ганц (Steven Ganz), Фил Гудвин (Phil Goodwin), Ральф Йочем (Ralph Jocham), Ян-Фа Ли (Yan-Fa Li), Тао Ма (Tao Ma), Джеффри Миллер (Jeffery Miller), Суреш Пай (Suresh Pai), Русс Руфер (Russ Rufer), Дэйв У. Смит (Dave W. Smith), Скотт Торнквест (Scott Turnquest), Вальтер Ваннини (Walter Vannini), Дарлин Уоллах (Darlene Wallach) и Джонатан Эндрю Уолтер (Jonathan Andrew Wolter). Кроме того, хочется поблагодарить Дуэйна Джонсона (Dewayne Johnson) и Кима Лиди (Kim Leedy) за помощь в художественном оформлении обложки, а также Фрэнка Соммерса (Frank Sommers) — за работу над алфавитным указателем.

Хотелось бы выразить особую благодарность и всем нашим читателям, приславшим комментарии. Они нам очень пригодились для повышения качества книги. Мы не в состоянии опубликовать имена всех приславших комментарии, но объявим имена тех читателей, кто прислал не менее пяти комментариев на стадии eBook PrePrint™. Отсортируем их имена по убыванию количества комментариев. наших благодарностей заслуживают Дэвид Бизак (David Biesack), Дон Стефан (Donn Stephan), Матс Хенриксон (Mats Henricson), Роб Диккенс (Rob Dickens), Блэр Захак (Blair Zajac), Тони Слоан (Tony Sloane), Найджел Харрисон (Nigel Harrison), Хавьер Диас Сото (Javier Diaz Soto), Уильям Хелан (William Heelan), Джастин Фурдер (Justin Forder), Грегор Перди (Gregor Purdy), Колин Перкинс (Colin Perkins), Бьярте С. Карлсен (Bjarte S. Karlsen), Эрвин Варга (Ervin Varga), Эрик Уиллигерс (Eric Willigers), Марк Хейс (Mark Hayes), Мартин Элвин (Martin Elwin), Калум Маклин (Calum MacLean), Джо-

натан Уолтер (Jonathan Wolter), Лес Прушински (Les Pruszyński), Сет Тисуе (Seth Tisue), Андрей Формига (Andrei Formiga), Дмитрий Григорьев (Dmitry Grigoriev), Джордж Бергер (George Berger), Говард Ловетт (Howard Lovatt), Джон П. Эйрих (John P. Eurich), Мариус Скуртеску (Marius Scurtescu), Джефф Эрвин (Jeff Ervin), Джейми Уэбб (Jamie Webb), Курт Зольман (Kurt Zoglmann), Дин Уэмплер (Dean Wampler), Николай Линдберг (Nikolaj Lindberg), Питер Маклейн (Peter McLain), Аркадиуш Стрыйски (Arkadiusz Strujski), Шанки Сурана (Shanku Surana), Крейг Борделон (Craig Bordelon), Александр Пэтри (Alexandre Patry), Филип Моэнс (Filip Moens), Фред Янон (Fred Japon), Джефф Хеон (Jeff Heon), Борис Лорбир (Boris Lorbeer), Джим Менард (Jim Menard), Тим Аццопарди (Tim Azzopardi), Томас Юнг (Thomas Jung), Уолтер Чанг (Walter Chang), Йерун Дийкмейер (Jeroen Dijkmeijer), Кейси Боумен (Casey Bowman), Мартин Смит (Martin Smith), Ричард Даллауэй (Richard Dallaway), Энтони Стаббс (Antony Stubbs), Ларс Вестергрэн (Lars Westergren), Маартен Хэйзвинкель (Maarten Hazewinkel), Мэтт Расселл (Matt Russell), Ремигиуш Михаловски (Remigiusz Michalowski), Андрей Толорко (Andrew Tolorko), Куртис Стэнфорд (Curtis Stanford), Джошуа Каф (Joshua Cough), Земен Денг (Zemian Deng), Кристофер Родригес Масиас (Christopher Rodrigues Macias), Хуан Мигель Гарсия Лопес (Juan Miguel Garcia Lopez), Мишель Шинц (Michel Schinz), Питер Мур (Peter Moore), Рэндольф Кал (Randolph Kahle), Владимир Кельман (Vladimir Kelman), Даниэль Гронау (Daniel Gronau), Дирк Детеринг (Dirk Detering), Хироаки Накамура (Hiroaki Nakamura), Оле Хугаард (Ole Hougaard), Бхаскар Маддала (Bhaskar Maddala), Дэвид Бернар (David Bernard), Дерек Махар (Derek Mahar), Джордж Коллиас (George Kollias), Кристиан Нордал (Kristian Nordal), Нормен Мюллер (Normen Mueller), Рафаэль Феррейра (Rafael Ferreira), Бинил Томас (Binil Thomas), Джон Нильсон (John Nilsson), Хорхе Ортис (Jorge Ortiz), Маркус Шульте (Marcus Schulte), Вадим Герасимов (Vadim Gerasimov), Кэмерон Таггарт (Cameron Taggart), Джон-Андерс Тейген (Jon-Anders Teigen), Сильвестр Забала (Silvestre Zabala), Уилл Маккуин (Will McQueen) и Сэм Оуэн (Sam Owen).

Хочется также сказать спасибо тем, кто отправил сообщения о замеченных неточностях после публикации первых двух изданий. Это Феликс Зигрист (Felix Siegrist), Лотар Мейер-Лербс (Lothar Meyer-Lerbs), Диетгард Михаэлис (Diethard Michaelis), Рошан Даврани (Roshan Dawrani), Донн Стефан (Donn Stephan), Уильям Утер (William Uther), Франсиско Ревербель (Francisco Reverbel), Джим Балтер (Jim Balter), Фрик де Брюйн (Freek de Bruijn), Амброс Лэнг (Ambrose Laing), Сехар Прабхала (Sekhar Prabhala), Левон Салдамли (Levon Saldamli), Эндрю Бурсавич (Andrew Bursavich), Хьялмар Петерс (Hjalmar Peters), Томас Фер (Thomas Fehr), Ален О'Ди (Alain O'Dea), Роб Диккенс (Rob Dickens), Тим Тейлор (Tim Taylor),

Кристиан Штернагель (Christian Sternagel), Мишель Паризьен (Michel Parisien), Джоэл Нили (Joel Neely), Брайан Маккеон (Brian McKeon), Томас Фер (Thomas Fehr), Джозеф Эллиотт (Joseph Elliott), Габриэль да Силва Рибейро (Gabriel da Silva Ribeiro), Томас Фер (Thomas Fehr), Пабло Рипольес (Pablo Ripolles), Дуглас Гейлор (Douglas Gaylor), Кевин Сквайр (Kevin Squire), Гарри-Антон Талвик (Harry-Anton Talvik), Кристофер Симпкинс (Christopher Simpkins), Мартин Витман-Функ (Martin Witmann-Funk), Джим Балтер (Jim Balter), Питер Фостер (Peter Foster), Крейг Бордолон (Craig Bordelon), Хайнц-Питер Гум (Heinz-Peter Gumm), Питер Чапин (Peter Chapin), Кевин Райт (Kevin Wright), Анантан Сринивасан (Ananthan Srinivasan), Омар Килани (Omar Kilani), Дон Стефан (Donn Stephan), Гюнтер Ваффлер (Guenther Waffler).

Лекс хотел бы поблагодарить специалистов, среди которых Аарон Абрамс (Aaron Abrams), Джейсон Адамс (Jason Adams), Генри и Эмили Крутчер (Henry and Emily Crutcher), Джои Гибсон (Joey Gibson), Гунар Хиллерт (Gunnar Hillert), Мэтью Линк (Matthew Link), Тоби Рейлтс (Toby Reylts), Джейсон Снейп (Jason Snape), Джон и Мелинда Уэзерс (John and Melinda Weathers), и всех представителей Atlanta Scala Enthusiasts за множество полезных обсуждений структуры языка, его математических основ и способов представления языка Scala специалистам-практикам.

Особую благодарность хочется выразить Дэйву Брикчетти (Dave Briccetti) и Адриану Мурсу (Adriaan Moors) за рецензирование третьего издания, а также Маркони Ланна (Marconi Lanna) не только за рецензирование, но и за мотивацию выпустить третье издание, которая возникла после разговора о новинках, появившихся со времени выхода предыдущего издания.

Билл хотел бы поблагодарить нескольких специалистов за предоставленную информацию и советы по изданию книги. Его благодарность заслужили Гэри Корнелл (Gary Cornell), Грег Доенч (Greg Doench), Энди Хант (Andy Hunt), Майк Леонард (Mike Leonard), Тайлер Ортман (Tyler Ortman), Билл Поллок (Bill Pollock), Дейв Томас (Dave Thomas) и Адам Райт (Adam Wright). Билл также хотел бы поблагодарить Дика Уолла (Dick Wall) за сотрудничество над разработкой нашего курса Stairway to Scala, который большей частью основывался на материалах, вошедших в эту книгу. Наш многолетний опыт преподавания курса Stairway to Scala помог повысить его качество. И наконец, Билл хотел бы выразить благодарность Дарлин Грюндль (Darlene Gruendl) и Саманте Вулф (Samantha Woolf) за помощь в завершении третьего издания.

Наконец, мы хотели бы поблагодарить Жюльена Ричарда-Фоя (Julien Richard-Foy) за работу над обновлением четвертого издания этой книги до версии Scala 2.13, в частности за перепроектирование библиотеки коллекций.

Введение

Эта книга — руководство по Scala, созданное людьми, непосредственно занимающимися разработкой данного языка программирования. Нашей целью было научить вас всему, что необходимо для превращения в продуктивного программиста на языке Scala. Все примеры в книге компилируются с помощью Scala версии 3.0.0

Целевая аудитория

Книга в основном рассчитана на программистов, желающих научиться программировать на Scala. Если у вас есть желание создать свой следующий проект на этом языке, то наша книга вам подходит. Кроме того, она должна заинтересовать программистов, которые хотят расширить кругозор, изучив новые концепции. Если вы, к примеру, программируете на Java, то эта книга раскроет для вас множество концепций функционального программирования, а также передовых идей из сферы объектно-ориентированного программирования. Мы уверены: изучение Scala и заложенных в этот язык идей поможет вам повысить свой профессиональный уровень как программиста. Предполагается, что вы уже владеете общими знаниями в области программирования. Scala вполне подходит на роль первого изучаемого языка, однако это не та книга, которая может использоваться для обучения программированию. В то же время вам не нужно быть каким-то особенным знатоком языков программирования. Большинство людей использует Scala на платформе Java, однако наша книга не предполагает, что вы тесно знакомы с языком Java. Но все же мы ожидаем, что Java известен многим читателям, и поэтому иногда сравниваем оба языка, чтобы помочь таким читателям понять разницу.

Как пользоваться книгой

Книгу рекомендуется читать в порядке следования глав, от начала до конца. Мы очень старались в каждой главе вводить читателя в курс только одной

темы и объяснять новый материал лишь в понятиях из ранее рассмотренных тем. Поэтому если перескочить вперед, чтобы поскорее в чем-то разобраться, то можно встретить объяснения, в которых используются еще непонятные концепции. Мы считаем, что получать знания в области программирования на Scala лучше постепенно, читая главы в порядке их следования.

Встретив незнакомое понятие, можно обратиться к глоссарию. Многие читатели бегло просматривают части книги, и это вполне нормально. Но при встрече с непонятными терминами можно выяснить, что просмотр был слишком поверхностным, и вернуться к справочным материалам.

Прочитав книгу, вы можете в дальнейшем использовать ее в качестве справочника по Scala. Конечно, существует официальная спецификация языка, но в ней прослеживается стремление к точности в ущерб удобству чтения. В нашем издании не охвачены абсолютно все подробности Scala, но его особенности изложены в нем вполне обстоятельно. Так что по мере освоения программирования на этом языке издание вполне может стать доступным справочником.

Как изучать Scala

Весьма обширные познания о Scala можно получить, просто прочитав книгу от начала до конца. Но быстрее и основательнее освоить язык можно с помощью ряда дополнительных действий.

Прежде всего можно воспользоваться множеством примеров программ, включенных в эту книгу. Самостоятельно набирая их, вам придется вдумываться в каждую строку кода. Попытки его разнообразить позволят вам глубже заинтересоваться изучаемой темой и убедиться в том, что вы действительно поняли, как работает этот код.

Затем можно поучаствовать в работе множества онлайн-форумов. Это позволит вам и многим другим приверженцам Scala помочь друг другу в его освоении. Есть множество рассылок, дискуссионных форумов, чатов, вики-источников и несколько информационных каналов, которые посвящены этому языку и содержат соответствующие публикации. Уделите время поиску источников информации, более всего отвечающих вашим запросам. Вы будете гораздо быстрее решать мелкие проблемы, что позволит уделять больше времени более серьезным и глубоким вопросам.

И наконец, получив при чтении книги достаточный объем знаний, приступайте к разработке собственного проекта. Поработайте с нуля над созданием

какой-нибудь небольшой программы или разработайте дополнение к более объемной. Вы не добьетесь быстрых результатов одним только чтением.

Условные обозначения

При первом упоминании какого-либо *понятия* или *термина* его название дается курсивом. Для небольших встроенных в текст примеров кода, таких как `x + 1`, используется моноширинный шрифт. Большие примеры кода представлены в виде отдельных блоков, для которых тоже используется моноширинный шрифт:

```
def hello() =  
  println("Hello, world!")
```

Когда показывается работа с интерактивной оболочкой, ответы последней выделяются шрифтом на сером фоне:

```
scala> 3 + 4  
val res0: Int = 7
```

Структура книги

- Глава 1 «Масштабируемый язык» представляет обзор структуры языка Scala, а также ее логическое обоснование и историю.
- Глава 2 «Первые шаги в Scala» показывает, как в языке выполняется ряд основных задач программирования, не вдаваясь в подробности, касающиеся особенностей работы механизмов языка. Цель этой главы — заставить ваши пальцы набирать и запускать код на Scala.
- Глава 3 «Дальнейшие шаги в Scala» показывает ряд основных задач программирования, помогающих ускорить освоение этого языка. Изучив данную главу, вы сможете использовать Scala для автоматизации простых задач.
- Глава 4 «Классы и объекты» закладывает начало углубленного рассмотрения языка Scala, приводит описание его основных объектно-ориентированных строительных блоков и указания по выполнению компиляции и запуску приложений Scala.
- Глава 5 «Основные типы и операции» охватывает основные типы Scala, их литералы, операции, которые могут над ними проводиться,

вопросы работы уровней приоритета и ассоциативности и дает представление об обогащающих оболочках.

- Глава 6 «Функциональные объекты» углубляет представление об объектно-ориентированных свойствах Scala, используя в качестве примера функциональные (то есть неизменяемые) рациональные числа.
- Глава 7 «Встроенные управляющие конструкции» показывает способы использования таких конструкций Scala, как `if`, `while`, `for`, `try` и `match`.
- Глава 8 «Функции и замыкания» углубленно рассматривает функции как основные строительные блоки функциональных языков.
- Глава 9 «Управляющие абстракции» показывает, как усовершенствовать основные управляющие конструкции Scala с помощью определения собственных управляющих абстракций.
- Глава 10 «Композиция и наследование» рассматривает имеющуюся в Scala дополнительную поддержку объектно-ориентированного программирования. Затрагиваемые темы не столь фундаментальны, как те, что излагались в главе 4, но вопросы, которые в них рассматриваются, часто встречаются на практике.
- Глава 11 «Трейты» охватывает существующий в Scala механизм создания композиции примесей. Показана работа трейтов, описываются примеры их наиболее частого использования, и объясняется, как с помощью трейтов совершенствуется традиционное множественное наследование.
- Глава 12 «Пакеты, импорты и экспорты» рассматривает вопросы программирования в целом, включая высокоуровневые пакеты, инструкции импортирования и модификаторы управления доступом, такие как `protected` и `private`.
- Глава 13 «Сопоставление с образцом» описывает двойные конструкции, которые помогут вам при написании обычных, неинкапсулированных структур данных. Классы регистра и сопоставление с образцом особенно полезны для древовидных рекурсивных данных.
- Глава 14 «Работа со списками» подробно рассматривает списки, которые, вероятно, можно отнести к самым востребованным структурам данных в программах на Scala.
- Глава 15 «Работа с другими коллекциями» показывает способы использования основных коллекций Scala, таких как списки, массивы, кортежи, множества и отображения.

- Глава 16 «Изменяемые объекты» объясняет суть изменяемых объектов и синтаксиса для выражения этих объектов, обеспечиваемого Scala. Глава завершается практическим примером моделирования дискретного события, в котором показан ряд изменяемых объектов в действии.
- Глава 17 «Иерархия Scala» объясняет иерархию наследования языка и рассматривает универсальные методы и низшие типы.
- Глава 18 «Параметризация типов» объясняет некоторые методы сокрытия информации, представленные в главе 13, на конкретном примере: конструкции класса для чисто функциональных очередей. Глава строится на описании вариации параметров типа и того, как она взаимодействует с сокрытием информации.
- Глава 19 «Перечисления» вводит двойные конструкции, которые помогут вам при написании обычных, неинкапсулированных структур данных.
- Глава 20 «Абстрактные члены» дает описание всех видов абстрактных членов, поддерживаемых Scala, — не только методов, но и полей и типов, которые можно объявлять абстрактными.
- Глава 21 «Гивены» описывает функцию Scala, которая помогает вам работать с контекстными параметрами функций. Передача всей контекстной информации проста, но может потребовать большого количества шаблонов. Гивены позволяют вам упростить эту задачу.
- Глава 22 «Методы расширения» описывает механизм Scala, позволяющий создать впечатление, что функция определена как метод в классе, хотя на самом деле она определена вне класса.
- Глава 23 «Классы типов» (которую еще предстоит написать). В этой главе будет проиллюстрировано несколько примеров классов типов.
- Глава 24 «Углубленное изучение коллекций» предлагает углубленный обзор библиотеки коллекций.
- Глава 25 «Утверждения и тесты» демонстрирует механизм утверждения Scala и дает обзор нескольких инструментов, доступных для написания тестов на Scala, уделяя особое внимание ScalaTest.

Ресурсы

На <https://www.scala-lang.org> — официальном сайте Scala — вы найдете последнюю версию Scala и ссылки на документацию и ресурсы сообщества.

Исходный код и дополнительные материалы к книге вы найдете по адресу https://booksites.artima.com/programming_in_scala_5ed.

Исходный код

Исходный код, рассматриваемый в данной книге, выпущенный под открытой лицензией в виде ZIP-файла, можно найти на сайте книги: https://booksites.artima.com/programming_in_scala_5ed.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Масштабируемый язык

Scala означает «масштабируемый язык» (от англ. *scalable language*). Это название он получил, поскольку был спроектирован так, чтобы расти вместе с запросами своих пользователей. Язык Scala может решать широкий круг задач программирования: от написания небольших скриптов до создания больших систем¹.

Scala легко освоить. Он работает на стандартных платформах Java и JavaScript и без проблем взаимодействует с библиотеками обеих платформ. Это довольно хороший язык для написания скриптов, объединяющих существующие библиотеки. Но он может еще больше проявить себя при построении больших систем и фреймворков из компонентов многократного использования.

С технической точки зрения Scala — смесь объектно-ориентированной и функциональной концепций программирования в статически типизированном языке. Подобный сплав проявляется во многих аспектах Scala — он, вероятно, может считаться более всеобъемлющим, чем другие широко используемые языки. Когда дело доходит до масштабируемости, два стиля программирования дополняют друг друга. Используемые в Scala конструкции функционального программирования упрощают быстрое создание интересных компонентов из простых частей. Объектно-ориентированные конструкции же облегчают структурирование больших систем и их адаптацию к новым требованиям. Сочетание двух стилей в Scala позволяет создавать новые виды шаблонов программирования и абстракций компонентов. Оно также способствует выработке понятного и лаконичного стиля программирования. И благодаря такой гибкости языка программирование на Scala может принести массу удовольствия.

¹ Scala произносится как «ска́ла».

В этой вступительной главе мы отвечаем на вопрос «Почему именно Scala?». Мы даем общий обзор структуры Scala и ее обоснование. Прочитав главу, вы получите базовое представление о том, что такое Scala и с какого рода задачами он поможет справиться. Книга представляет собой руководство по языку Scala, однако данную главу нельзя считать частью этого руководства. И если вам не терпится приступить к написанию кода на Scala, то можете сразу перейти к изучению главы 2.

1.1. Язык, который растет вместе с вами

Программы различных размеров требуют, как правило, использования разных программных конструкций. Рассмотрим, к примеру, следующую небольшую программу на Scala:

```
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Эта программа устанавливает отображение стран на их столицы, модифицирует отображение, добавляя новую конструкцию ("Japan" -> "Tokyo"), и выводит название столицы, связанное со страной France¹. В этом примере используется настолько высокоуровневая система записи, что она не загромождена ненужными точками с запятыми и сигнатурами типов. И действительно возникает ощущение использования современного языка скриптов наподобие Perl, Python или Ruby. Одна из общих характеристик этих языков, применимая к данному примеру, — поддержка всеми ими в синтаксисе языка конструкции ассоциативного отображения.

Ассоциативные отображения очень полезны, поскольку помогают поддерживать понятность и краткость программ, но порой вам может не подойти их философия «на все случаи жизни», поскольку вам в своей программе нужно управлять свойствами отображений более тонко. При необходимости Scala обеспечивает точное управление, поскольку отображения в нем не являются синтаксисом языка. Это библиотечные абстракции, которые можно расширять и приспособливать под свои нужды.

В показанной ранее программе вы получите исходную реализацию отображения Map, но ее можно будет без особого труда изменить. К примеру, можно указать конкретную реализацию, такую как HashMap или TreeMap,

¹ Пожалуйста, не сердитесь на нас, если не сможете разобраться во всех тонкостях этой программы. Объяснения будут даны в двух следующих главах.

или с помощью модуля параллельных коллекций Scala вызвать метод `par` для получения отображения `ParMap`, операции в котором выполняются параллельно. Можно указать для отображения значение по умолчанию или переопределить любой другой метод созданного вами отображения. Во всех случаях для отображений вполне пригоден такой же простой синтаксис доступа, как и в приведенном примере.

В нем показано, что Scala может обеспечить вам как удобство, так и гибкость. Язык содержит набор удобных конструкций, которые помогают быстро начать работу и позволяют программировать в приятном лаконичном стиле. В то же время вы всегда сможете переключить программу под свои требования, поскольку все в ней основано на библиотечных модулях, которые можно выбрать и приспособить под свои нужды.

Растут новые типы

Эрик Рэймонд (Eric Raymond) в качестве двух метафор разработки программных продуктов ввел собор и базар [Ray99]. Под собором понимается почти идеальная разработка, создание которой требует много времени. После сборки она долго остается неизменной. Разработчики же базара, напротив, что-то адаптируют и дополняют каждый день. В книге Рэймонда базар — метафора, описывающая разработку ПО с открытым кодом. Гай Стил (Guy Steele) отметил в докладе о «растущем языке», что аналогичное различие можно применить к структуре языка программирования [Ste99]. Scala больше похож на базар, чем на собор, в том смысле, что спроектирован с расчетом на расширение и адаптацию его теми, кто на нем программирует. Вместо того чтобы предоставлять все конструкции, которые только могут пригодиться в одном всеобъемлющем языке, Scala дает вам инструменты для создания таких конструкций.

Рассмотрим пример. Многие приложения нуждаются в целочисленном типе, который при выполнении арифметических операций может становиться произвольно большим без переполнения или циклического перехода в начало. В Scala такой тип определяется в библиотеке класса `scala.math.BigInt`. Определение использующего этот тип метода, который вычисляет факториал переданного ему целочисленного значения, имеет следующий вид¹:

```
def factorial(x: BigInt): BigInt =  
  if x == 0 then 1 else x * factorial(x - 1)
```

¹ `factorial(x)`, или $x!$ в математической записи — результат вычисления $1 * 2 * \dots * x$, где для $0!$ определено значение 1.

Теперь, вызвав `factorial(30)`, вы получите:

```
265252859812191058636308480000000
```

Тип `BigInt` похож на встроенный, поскольку со значениями этого типа можно использовать целочисленные литералы и операторы наподобие `*` и `-`. Тем не менее это просто класс, определение которого задано в стандартной библиотеке Scala¹. Если бы класса не было, то любой программист на Scala мог бы запросто написать его реализацию, например создав оболочку для имеющегося в языке Java класса `java.math.BigInteger` (фактически именно так и реализован класс `BigInt` в Scala).

Конечно, класс Java можно использовать напрямую. Но результат будет не столь приятным: хоть Java и позволяет вам создавать новые типы, они не производят впечатление получающих естественную поддержку языка:

```
import java.math.BigInteger

def factorial(x: BigInteger): BigInteger =
  if x == BigInteger.ZERO then
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

Тип `BigInt` — один из многих других числовых типов: больших десятичных чисел, комплексных и рациональных чисел, доверительных интервалов, полиномов, и данный список можно продолжить. В некоторых языках программирования часть этих типов реализуется естественным образом. Например, в Lisp, Haskell и Python есть большие целые числа, в Fortran и Python — комплексные. Но любой язык, в котором пытаются одновременно реализовать все эти абстракции, разрастается до таких размеров, что становится неуправляемым. Более того, даже существуй подобный язык, нашлись бы приложения, требующие других числовых типов, которые все равно не были бы представлены. Следовательно, подход, при котором предпринимается попытка реализовать все в одном языке, не позволяет получить хорошую масштабируемость. Язык Scala, напротив, дает пользователям возможность наращивать и адаптировать его в нужных направлениях. Он делает это с помощью определения простых в использовании библиотек, которые производят *впечатление* средств, естественно реализованных в языке.

¹ Scala поставляется со стандартной библиотекой, часть которой будет рассмотрена в книге. За дополнительной информацией можно обратиться к имеющейся в библиотеке документации Scaladoc, доступной в дистрибутиве и в интернете по адресу www.scala-lang.org.

Растут новые управляющие конструкции

Такая расширяемость иллюстрируется стилем `AnyFunSuite` `ScalaTest`, популярной библиотеки тестирования для Scala. В качестве примера приведем простой тестовый класс, содержащий два теста:

```
class SetSpec extends AnyFunSuite:

  test("An empty Set should have size 0") {
    assert(Set.empty.size == 0)
  }

  test("Invoking head on an empty Set should fail") {
    assertThrows[NoSuchElementException] {
      Set.empty.head
    }
  }
}
```

Мы не ожидаем, что вы сейчас полностью поймете пример `AnyFunSuite`. Скорее, что важно в этом примере для темы масштабируемости, так это то, что ни тестовая конструкция, ни синтаксис `assertThrows` не являются встроенными операциями в Scala. Хотя обе они могут выглядеть и действовать очень похоже на встроенные управляющие конструкции, на самом деле они являются методами, определенными в библиотеке `ScalaTest`. Обе эти конструкции полностью независимы от языка программирования Scala.

Этот пример показывает, что вы можете «развивать» язык Scala в новых направлениях, даже таких специализированных, как тестирование программного обеспечения. Конечно, для этого нужны опытные архитекторы и программисты. Но важно то, что это осуществимо — вы можете разрабатывать и реализовывать абстракции в Scala, которые адресованы радикально новым доменам приложений, но при этом ощущать поддержку родного языка при использовании.

1.2. Почему язык Scala масштабируемый?

На возможность масштабирования влияет множество факторов, от особенностей синтаксиса до структуры абстрактных компонентов. Но если бы потребовалось назвать всего один аспект Scala, который способствует масштабируемости, то мы бы выбрали присущее этому языку сочетание объектно-ориентированного и функционального программирования (мы немного лукавили, на самом деле это два аспекта, но они взаимосвязаны).

Scala в объединении объектно-ориентированного и функционального программирования в однородную структуру языка пошел дальше всех остальных широко известных языков. Например, там, где в других языках объекты и функции — два разных понятия, в Scala функция по смыслу *является* объектом. Функциональные типы — это классы, которые могут наследоваться подклассами. Эти особенности могут показаться не более чем теоретическими, но имеют весьма серьезные последствия для возможностей масштабирования. Фактически ранее упомянутое понятие актора не может быть реализовано без этой унификации функций и объектов. Здесь мы рассмотрим возможные в Scala способы смешивания объектно-ориентированной и функциональной концепций.

Scala — объектно-ориентированный язык

Развитие объектно-ориентированного программирования шло весьма успешно. Появившись в языке Simula в середине 1960-х годов и Smalltalk в 1970-х, оно теперь доступно в подавляющем большинстве языков. В некоторых областях все полностью захвачено объектами. Точного определения «объектной ориентированности» нет, однако объекты явно чем-то привлекают программистов.

В принципе, мотивация для применения объектно-ориентированного программирования очень проста: все, за исключением самых простых программ, нуждается в определенной структуре. Наиболее понятный путь достижения желаемого результата заключается в помещении данных и операций в своеобразные контейнеры. Основной замысел объектно-ориентированного программирования состоит в придании этим контейнерам полной универсальности, чтобы в них могли содержаться не только операции, но и данные и чтобы сами они также были элементами, которые могли бы храниться в других контейнерах или передаваться операциям в качестве параметров. Подобные контейнеры называются объектами. Алан Кей (Alan Kay), изобретатель языка Smalltalk, заметил, что таким образом простейший объект имеет принцип построения, аналогичный полноценному компьютеру: под формализованным интерфейсом данные в нем сочетаются с операциями [Kay96]. То есть объекты имеют непосредственное отношение к масштабируемости языка: одни и те же технологии применяются к построению как малых, так и больших программ.

Хотя долгое время объектно-ориентированное программирование преобладало, немногие языки стали последователями Smalltalk по части внедрения этого принципа построения в свое логическое решение. Например, множе-

ство языков допускает использование элементов, не являющихся объектами, — можно вспомнить имеющиеся в языке Java значения примитивных типов. Или же в них допускается применение статических полей и методов, не входящих в какой-либо объект. Эти отклонения от чистой идеи объектно-ориентированного программирования на первый взгляд выглядят вполне безобидными, но имеют досадную тенденцию к усложнению и ограничению масштабирования.

В отличие от этого Scala — объектно-ориентированный язык в чистом виде: каждое значение является объектом и каждая операция — вызовом метода. Например, когда в Scala речь заходит о вычислении $1 + 2$, фактически вызывается метод по имени `+`, который определен в классе `Int`. Можно определять методы с именами, похожими на операторы, а клиенты вашего API смогут с помощью этих методов записать операторы.

Когда речь заходит о составлении объектов, Scala проявляется как более совершенный язык по сравнению с большинством других. В качестве примера приведем имеющиеся в Scala *трейты*. Они подобны интерфейсам в Java, но могут содержать также реализации методов и даже поля¹. Объекты создаются путем *композиции примесей*, при котором к членам класса добавляются члены нескольких трейтов. Таким образом, различные аспекты классов могут быть инкапсулированы в разных трейтах. Это выглядит как множественное наследование, но есть разница в конкретных деталях. В отличие от класса трейт может добавить в суперкласс новые функциональные возможности. Это придает трейтам более высокую степень подключаемости по сравнению с классами. В частности, благодаря этому удастся избежать возникновения присущих множественному наследованию классических проблем «ромбовидного» наследования, которые возникают, когда один и тот же класс наследуется по нескольким различным путям.

Scala — функциональный язык

Наряду с тем, что Scala является чистым объектно-ориентированным языком, его можно назвать и полноценным функциональным языком. Идеи функционального программирования старше электронных вычислительных систем. Их основы были заложены в лямбда-исчислении Алонзо Черча (Alonzo Church), разработанном в 1930-е годы. Первым языком функционального программирования был Lisp, появление которого датируется

¹ Начиная с Java 8, у интерфейсов могут быть реализации методов по умолчанию, но они не предлагают всех тех возможностей, которые есть у трейтов языка Scala.

концом 1950-х. К другим популярным функциональным языкам относятся Scheme, SML, Erlang, Haskell, OCaml и F#. Долгое время функциональное программирование играло второстепенные роли — будучи популярным в научных кругах, оно не столь широко использовалось в промышленности. Но в последние годы интерес к его языкам и технологиям растет.

Функциональное программирование базируется на двух основных идеях. Первая заключается в том, что функции являются значениями первого класса. В функциональных языках функция есть значение, имеющее такой же статус, как целое число или строка. Функции можно передавать в качестве аргументов другим функциям, возвращать их в качестве результатов из других функций или сохранять в переменных. Вдобавок функцию можно определять внутри другой функции точно так же, как это делается при определении внутри функции целочисленного значения. И функции можно определять, не присваивая им имен, добавляя в код функциональные литералы с такой же легкостью, как и целочисленные, наподобие 42.

Функции как значения первого класса — удобное средство абстрагирования, касающееся операций и создания новых управляющих конструкций. Эта универсальность функций обеспечивает более высокую степень выразительности, что зачастую приводит к созданию весьма разборчивых и кратких программ. Она также играет важную роль в обеспечении масштабируемости. В качестве примера библиотека тестирования `ScalaTest` предлагает конструкцию `eventually`, получающую функцию в качестве аргумента. Данная конструкция используется следующим образом:

```
val xs = 1 to 3
val it = xs.iterator
eventually { it.next() shouldBe 3 }
```

Код внутри `eventually`, являющийся утверждением, `it.next() shouldBe 3`, включает в себя функцию, передаваемую невыполненной в метод `eventually`. Через настраиваемый период времени `eventually` станет неоднократно выполнять функцию до тех пор, пока утверждение не будет успешно подтверждено.

Вторая основная идея функционального программирования заключается в том, что операции программы должны преобразовать входные значения в выходные, а не изменять данные на месте. Чтобы понять разницу, рассмотрим реализацию строк в Ruby и Java. В Ruby строка является массивом символов. Символы в строке могут быть изменены по отдельности. Например, внутри одного и того же строкового объекта символ точки с запятой в строке можно заменить точкой. А в Java и Scala строка — последовательность символов в математическом смысле. Замена символа

в строке с использованием выражения вида `s.replace(';', '.')` приводит к возникновению нового строкового объекта, отличающегося от `s`. То же самое можно сказать по-другому: в Java строки неизменяемые, а в Ruby — изменяемые. То есть, рассматривая только строки, можно прийти к выводу, что Java — функциональный язык, а Ruby — нет. Неизменяемая структура данных — один из краеугольных камней функционального программирования. В библиотеках Scala в качестве надстроек над соответствующими API Java определяется также множество других неизменяемых типов данных. Например, в Scala имеются неизменяемые списки, кортежи, отображения и множества.

Еще один способ утверждения второй идеи функционального программирования заключается в том, что у методов не должно быть никаких *побочных эффектов*. Они должны обмениваться данными со своим окружением только путем получения аргументов и возвращения результатов. Например, под это описание подпадает метод `replace`, принадлежащий Java-классу `String`. Он получает строку и два символа и выдает новую строку, где все появления одного символа заменены появлениями второго. Других эффектов от вызова `replace` нет. Методы, подобные `replace`, называются *ссылочно прозрачными*. Это значит, что для любого заданного ввода вызов функции можно заменить его результатом, при этом семантика программы остается неизменной.

Функциональные языки заставляют применять неизменяемые структуры данных и ссылочно прозрачные методы. В некоторых функциональных языках это выражено в виде категоричных требований. Scala же дает возможность выбрать. При желании можно писать программы в *императивном* стиле — так называется программирование с изменяемыми данными и побочными эффектами. Но при необходимости в большинстве случаев Scala позволяет с легкостью избежать использования императивных конструкций благодаря существованию хороших функциональных альтернатив.

1.3. Почему именно Scala

Подойдет ли вам язык Scala? Разбираться и принимать решение придется самостоятельно. Мы считаем, что, помимо хорошей масштабируемости, существует еще множество причин, по которым вам может понравиться программирование на Scala. В этом разделе будут рассмотрены четыре наиболее важных аспекта: совместимость, лаконичность, абстракции высокого уровня и расширенная статическая типизация.

Scala — совместимый язык

Scala не требует резко отходить от платформы Java, чтобы опередить на шаг этот язык. Scala позволяет повышать ценность уже существующего кода, то есть опираться на то, что у вас уже есть, поскольку он был разработан для достижения беспрепятственной совместимости с Java¹. Программы на Scala компилируются в байт-коды виртуальной машины Java (JVM). Производительность при выполнении этих кодов находится на одном уровне с производительностью программ на Java. Код Scala может вызывать методы Java, обращаться к полям этого языка, поддерживать наследование от его классов и реализовывать его интерфейсы. Для всего перечисленного не требуются ни специальный синтаксис, ни явные описания интерфейса, ни какой-либо связующий код. По сути, весь код Scala интенсивно использует библиотеки Java, зачастую даже без ведома программистов.

Еще один показатель полной совместимости — интенсивное заимствование в Scala типов данных Java. Данные типа `Int` в Scala представлены в виде имеющегося в Java примитивного целочисленного типа `int`, соответственно `Float` представлен как `float`, `Boolean` — как `boolean` и т. д. Массивы Scala отображаются на массивы Java. В Scala из Java позаимствованы и многие стандартные библиотечные типы. Например, тип строкового литерала `"abc"` в Scala фактически представлен классом `java.lang.String`, а исключение должно быть подклассом `java.lang.Throwable`.

Java-типы в Scala не только заимствованы, но и «принаряжены» для придания им привлекательности. Например, строки в Scala поддерживают такие методы, как `toInt` или `toFloat`, которые преобразуют строки в целое число или число с плавающей точкой. То есть вместо `Integer.parseInt(str)` вы можете написать `str.toInt`. Как такое возможно без нарушения совместимости? Класс `String` в Java определенно не имеет метода `toInt`! Фактически у Scala есть очень общее решение для устранения этого противоречия между передовой разработкой и функциональной совместимостью². Scala позволяет определять многофункциональные расширения, которые всегда применяются при выборе несуществующих элементов. В рассматриваемом случае при поиске метода `toInt` для работы со строковым значением компилятор

¹ Изначально существовала реализация Scala, запускаемая на платформе .NET, но она больше не используется. В последнее время все большую популярность набирает реализация Scala под названием `Scala.js`, запускаемая на JavaScript.

² В версии 3.0.0 стандартные расширения реализованы посредством неявных преобразований. В последующих версиях Scala они будут заменены методами расширения.

Scala не найдет такого элемента в классе `String`. Однако он найдет неявное преобразование, превращающее Java-класс `String` в экземпляр Scala-класса `StringOps`, в котором такой элемент определен. Затем преобразование будет автоматически применено, прежде чем будет выполнена операция `toInt`.

Код Scala также может быть вызван из кода Java. Иногда при этом следует учитывать некоторые нюансы. Scala — более утонченный язык, чем Java, поэтому некоторые расширенные функции Scala должны быть закодированы, прежде чем они смогут быть отображены на Java.

Scala — лаконичный язык

Программы на Scala, как правило, отличаются краткостью. Программисты, работающие с данным языком, отмечают сокращение количества строк почти на порядок по сравнению с Java. Но это можно считать крайним случаем. Более консервативные оценки свидетельствуют о том, что обычная программа на Scala должна уместиться в половину тех строк, которые используются для аналогичной программы на Java. Меньшее количество строк означает не только сокращение объема набираемого текста, но и экономию сил при чтении и осмыслении программ, а также уменьшение количества возможных недочетов. Свой вклад в сокращение количества строк кода вносят сразу несколько факторов.

В синтаксисе Scala не используются некоторые шаблонные элементы, отягощающие программы на Java. Например, в Scala не обязательно применять точки с запятыми. Есть и несколько других областей, где синтаксис Scala менее зашумлен. В качестве примера можно сравнить, как записывается код классов и конструкторов в Java и Scala. В Java класс с конструктором зачастую выглядит следующим образом:

```
class MyClass { // Java

    private int index;
    private String name;

    public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

А в Scala, скорее всего, будет использована такая запись:

```
class MyClass(index: Int, name: String)
```

Получив указанный код, компилятор Scala создаст класс с двумя приватными переменными экземпляра (типа `Int` по имени `index` и типа `String` по имени `name`) и конструктор, который получает исходные значения для этих переменных в виде параметров. Код данного конструктора проинициализирует две переменные экземпляра значениями, переданными в качестве параметров. Короче говоря, в итоге вы получите ту же функциональность, что и у более многословной версии кода на Java¹. Класс в Scala быстрее пишется и проще читается, а еще — и это наиболее важно — допустить ошибку при его создании значительно труднее, чем при создании класса в Java.

Еще один фактор, способствующий лаконичности, — используемый в Scala вывод типов. Повторяющуюся информацию о типе можно отбросить, и тогда программы избавятся от лишнего и их легче будет читать.

Но, вероятно, наиболее важный аспект сокращения объема кода — наличие кода, не требующего внесения в программу, поскольку это уже сделано в библиотеке. Scala предоставляет вам множество инструментальных средств для определения эффективных библиотек, позволяющих выявить и вынести за скобки общее поведение. Например, различные аспекты библиотечных классов можно выделить в трейты, которые затем можно перемешивать произвольным образом. Или же библиотечные методы могут быть параметризованы с помощью операций, позволяя вам определять конструкции, которые, по сути, являются вашими собственными управляющими конструкциями. Собранные вместе, эти конструкции позволяют определять библиотеки, сочетающие в себе высокоуровневый характер и гибкость.

Scala — высокоуровневый язык

Программисты постоянно борются со сложностью. Для продуктивного программирования нужно понимать код, над которым вы работаете. Чрезмерно сложный код был причиной краха многих программных проектов. К сожалению, важные программные продукты обычно бывают весьма сложными. Избежать сложности невозможно, но ею можно управлять.

Scala помогает управлять сложностью, позволяя повышать уровень абстракции в разрабатываемых и используемых интерфейсах. Представим, к примеру, что есть переменная `name`, имеющая тип `String`, и нужно определить,

¹ Единственное отличие заключается в том, что переменные экземпляра, полученные в случае применения Scala, будут финальными (`final`). Как сделать их не финальными, рассказывается в разделе 10.6.

наличествует ли в этой строковой переменной символ в верхнем регистре. До выхода Java 8 приходилось создавать следующий цикл:

```
boolean nameHasUpperCase = false; // Java
for (int i = 0; i < name.length(); ++i) {
    if (Character.isUpperCase(name.charAt(i))) {
        nameHasUpperCase = true;
        break;
    }
}
```

А в Scala можно написать такой код:

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Код Java считает строки низкоуровневыми элементами, требующими посимвольного перебора в цикле. Код Scala рассматривает те же самые строки как высокоуровневые последовательности символов, в отношении которых можно применять запросы с *предикатами*. Несомненно, код Scala намного короче и — для натренированного глаза — более понятен, чем код Java. Следовательно, код Scala значительно меньше влияет на общую сложность приложения. Кроме того, уменьшается вероятность допустить ошибку.

Предикат `_.isUpper` — пример используемого в Scala функционального литерала¹. В нем дается описание функции, которая получает аргумент в виде символа (представленного знаком подчеркивания) и проверяет, не является ли этот символ буквой в верхнем регистре².

В Java 8 появилась поддержка *лямбда-выражений* и *потоков* (streams), позволяющая выполнять подобные операции на Java. Вот как это могло бы выглядеть:

```
boolean nameHasUpperCase = // Java 8 или выше
    name.chars().anyMatch(
        (int ch) -> Character.isUpperCase((char) ch)
    );
```

Несмотря на существенное улучшение по сравнению с более ранними версиями Java, код Java 8 все же более многословен, чем его эквивалент на языке Scala. Излишняя тяжеловесность кода Java, а также давняя традиция использования в этом языке циклов может натолкнуть многих Java-программистов

¹ Функциональный литерал может называться предикатом, если результирующим типом будет Boolean.

² Такое использование символа подчеркивания в качестве заместителя для аргументов рассматривается в разделе 8.5.

на мысль о необходимости новых методов, подобных `exists`, позволяющих просто переписать циклы и смириться с растущей сложностью кода.

В то же время функциональные литералы в Scala действительно воспринимаются довольно легко и задействуются очень часто. По мере углубления знакомства со Scala перед вами будет открываться все больше и больше возможностей для определения и использования собственных управляющих абстракций. Вы поймете, что это поможет избежать повторений в коде, сохраняя лаконичность и чистоту программ.

Scala — статически типизированный язык

Системы со статической типизацией классифицируют переменные и выражения в соответствии с видом хранящихся и вычисляемых значений. Scala выделяется как язык своей совершенной системой статической типизации. Обладая системой вложенных типов классов, во многом похожей на имеющуюся в Java, этот язык позволяет вам проводить параметризацию типов с помощью средств *обобщенного программирования*, комбинировать типы с использованием *пересечений* и скрывать особенности типов, применяя *абстрактные типы*¹. Так формируется прочный фундамент для создания собственных типов, который дает возможность разрабатывать безопасные и в то же время гибкие в использовании интерфейсы.

Если вам нравятся динамические языки, такие как Perl, Python, Ruby или Groovy, то вы можете посчитать немного странным факт, что система статических типов в Scala упоминается как одна из его сильных сторон. Ведь отсутствие такой системы часто называют основным преимуществом динамических языков. Наиболее часто, говоря о ее недостатках, приводят такие аргументы, как присущая программам многословность, воспрепятствование свободному самовыражению программистов и невозможность применения конкретных шаблонов динамических изменений программных систем. Но зачастую эти аргументы направлены не против идеи статических типов в целом, а против конкретных систем типов, воспринимаемых как слишком многословные или недостаточно гибкие. Например, Алан Кей, автор языка Smalltalk, однажды заметил: «Я не против типов, но не знаю ни одной беспроблемной системы типов. Так что мне все еще нравится динамическая типизация»².

¹ Обобщенные типы рассматриваются в главе 18, пересечения (например, `A с B с C`) — в разделе 17.5, а абстрактные типы — в главе 20.

² *Кэу А. С.* Электронное письмо о значении объектно-ориентированного программирования [Кау03].

В этой книге мы надеемся убедить вас в том, что система типов в Scala далека от проблемной. На самом деле она вполне изящно справляется с двумя обычными опасениями, связываемыми со статической типизацией: многословия удается избежать за счет логического вывода типов, а гибкость достигается благодаря сопоставлению с образцом и ряду новых способов записи и составления типов. По мере устранения этих препятствий к классическим преимуществам систем статических типов начинают относиться намного более благосклонно. Среди наиболее важных преимуществ можно назвать верифицируемые свойства программных абстракций, безопасный рефакторинг и более качественное документирование.

Верифицируемые свойства. Системы статических типов способны подтверждать отсутствие конкретных ошибок, выявляемых в ходе выполнения программы. Это могут быть следующие правила: булевы значения никогда не складываются с целыми числами; приватные переменные недоступны за пределами своего класса; функции применяются к надлежащему количеству аргументов; в множество строк можно добавлять только строки.

Существующие в настоящее время системы статических типов не выявляют ошибки других видов. Например, обычно они не обнаруживают бесконечные функции, нарушение границ массивов или деление на ноль. Вдобавок эти системы не смогут определить несоответствие вашей программы ее спецификации (при наличии таковой!). Поэтому некоторые отказываются от них, считая не слишком полезными. Аргументация такова: если эти системы могут выявлять только простые ошибки, а модульные тесты обеспечивают более широкий охват, то зачем вообще связываться со статическими типами? Мы считаем, что в этих аргументах упущено главное. Система статических типов, конечно же, не может *заменить* собой модульное тестирование, однако может сократить количество необходимых модульных тестов, выявляя некие свойства, которые в противном случае нужно было бы протестировать. А модульное тестирование не способно заменить статическую типизацию. Ведь Эдсгер Дейкстра (Edsger Dijkstra) сказал, что тестирование позволяет убедиться лишь в наличии ошибок, но не в их отсутствии [Dij70]. Гарантии, которые обеспечиваются статической типизацией, могут быть простыми, но это реальные гарантии, не способные обеспечить никакие объемы тестирования.

Безопасный рефакторинг. Системы статических типов дают гарантии, позволяющие вам вносить изменения в основной код, будучи совершенно уверенными в благополучном исходе этого действия. Рассмотрим, к примеру, рефакторинг, при котором к методу нужно добавить еще один параметр. В статически типизированном языке вы можете внести изменения,

перекомпилировать систему и просто исправить те строки, которые вызовут ошибку типа. Сделав это, вы будете пребывать в уверенности, что были найдены все места, требовавшие изменений. То же самое справедливо для другого простого рефакторинга, например изменения имени метода или перемещения метода из одного класса в другой. Во всех случаях проверка статического типа позволяет быть вполне уверенными в том, что работоспособность новой системы осталась на уровне работоспособности старой.

Документирование. Статические типы — документация программы, проверяемой компилятором на корректность. В отличие от обычного комментария, аннотация типа никогда не станет устаревшей (по крайней мере, если содержащий ее исходный файл недавно успешно прошел компиляцию). Более того, компиляторы и интегрированные среды разработки (integrated development environments, IDE) могут использовать аннотации для выдачи более качественной контекстной справки. Например, IDE может вывести на экран все элементы, доступные для выбора, путем определения статического типа выражения, которое выбрано, и дать возможность просмотреть все элементы этого типа.

Хотя статические типы в целом полезны для документирования программы, иногда они могут вызывать раздражение тем, что засоряют ее. Обычно полезным считается документирование тех сведений, которые читателям программы самостоятельно извлечь довольно трудно. Полезно знать, что в методе, определенном так:

```
def f(x: String) = ...
```

аргументы метода `f` должны принадлежать типу `String`. В то же время может вызвать раздражение по крайней мере одна из двух аннотаций в следующем примере:

```
val x: HashMap[Int, String] = new HashMap[Int, String]()
```

Понятно, что было бы достаточно показать отношение `x` к типу `HashMap` с `Int`-типами в качестве ключей и `String`-типами в качестве значений только один раз, дважды повторять одно и то же нет смысла.

В Scala имеется весьма сложная система логического вывода типов, позволяющая опускать почти всю информацию о типах, которая обычно вызывает раздражение. В предыдущем примере вполне работоспособны и две менее раздражающие альтернативы:

```
val x = new HashMap[Int, String]()  
val x: Map[Int, String] = new HashMap()
```

Вывод типа в Scala может заходить довольно далеко. Фактически пользовательский код нередко вообще обходится без явного задания типов. Поэтому программы на Scala часто выглядят похожими на программы, написанные на динамически типизированных языках скриптов. Это, в частности, справедливо для прикладного клиентского кода, который склеивается из заранее написанных библиотечных компонентов. Но для них самих это менее характерно, поскольку в них зачастую применяются довольно сложные типы, не допускающие гибкого использования таких схем. И это вполне естественно. Ведь сигнатуры типов элементов, составляющих интерфейс повторно используемых компонентов, должны задаваться в явном виде, поскольку составляют существенную часть соглашения между компонентом и его клиентами.

1.4. Истоки Scala

На идею создания Scala повлияли многие языки программирования и идеи, выработанные на основе исследований таких языков. Фактически обновления в Scala незначительны — большинство характерных особенностей языка уже применялось в том или ином виде в других языках программирования. Инновации в Scala появляются в основном из того, как его конструкции сводятся воедино. В этом разделе будут перечислены основные факторы, оказавшие влияние на структуру языка Scala. Перечень не может быть исчерпывающим, поскольку в дизайне языков программирования так много толковых идей, что перечислить здесь их все просто невозможно.

На внешнем уровне Scala позаимствовал существенную часть синтаксиса у Java и C#, которые, в свою очередь, взяли большинство своих синтаксических соглашений у C и C++. Выражения, инструкции и блоки — в основном из Java, как, собственно, и синтаксис классов, создание пакетов и импорт¹.

¹ Главное отличие от Java касается синтаксиса для объявления типов: вместо «Тип переменная», как в Java, задействуется форма «переменная: Тип». Используемый в Scala постфиксный синтаксис типа похож на синтаксис, применяемый в Pascal, Modula-2 или Eiffel. Основная причина такого отклонения имеет отношение к логическому выводу типов, зачастую позволяющему опускать тип переменной или тип возвращаемого методом значения. Легче использовать синтаксис «переменная: Тип», поскольку двоеточие и тип можно просто не указывать. Но в стиле языка C, применяющем форму «Тип переменная», просто так не указывать тип нельзя, поскольку при этом исчезнет сам признак начала определения. Неуказанный тип в качестве заполнителя требует какое-нибудь ключевое слово (C# 3.0, в котором имеется логический вывод типов, для этой цели задействует ключевое слово `var`). Такое альтернативное ключевое слово представляется несколько более надуманным и менее привычным, чем подход, который используется в Scala.

Кроме синтаксиса, Scala позаимствовал и другие элементы Java, такие как его основные типы, библиотеки классов и модель выполнения.

Scala многим обязан и другим языкам. Его однородная модель объектов впервые появилась в Smalltalk и впоследствии была принята языком Ruby. Его идея универсальной вложенности (почти каждую конструкцию в Scala можно вложить в любую другую) реализована также в Algol, Simula, а в последнее время в Beta и gbeta. Его принцип единообразного доступа к вызову методов и выбору полей пришел из Eiffel. Его подход к функциональному программированию очень близок по духу к применяемому в семействе языков ML, видными представителями которого являются SML, OCaml и F#. Многие функции высшего порядка в стандартной библиотеке Scala присутствуют также в ML или Haskell. Толчком для появления в Scala неявных параметров стали классы типов языка Haskell — в более классическом объектно-ориентированном окружении они дают аналогичные результаты. Используемая в Scala основная библиотека многопоточного вычисления на основе акторов — Akka — создавалась под сильным влиянием особенностей языка Erlang.

Scala не первый язык, делающий упор на масштабируемость и расширяемость. Такое понятие, как расширяемые языки, которые могут охватывать различные области применения, впервые встречается в статье Питера Лэндина (Peter Landin) 1966 года (язык, описанный в этой статье, — Iswim — стоит рядом с Lisp как один из первых функциональных языков) [Lan66]. Конкретная идея рассматривать инфиксный оператор как функцию восходит к Iswim и Smalltalk. Другая важная идея — разрешить функциональный литерал (или блок) в качестве параметра, который позволяет библиотекам определять управляющие структуры. Опять же это восходит к Iswim и Smalltalk. И Smalltalk, и Lisp обладают гибким синтаксисом, который широко применялся для создания внутренних специфичных для конкретной предметной области языков. C++ — еще один масштабируемый язык, который можно адаптировать и расширить с помощью перегрузки операторов и его системы шаблонов; по сравнению со Scala он построен на более низкоуровневом, более системно — ориентированном ядре.

Кроме того, Scala не первый язык, объединяющий в себе функциональное и объектно-ориентированное программирование, хотя, вероятно, в этом направлении продвинулся гораздо дальше прочих. К числу других языков, объединивших некоторые элементы функционального программирования с объектно-ориентированным, относятся Ruby, Smalltalk и Python. Расширения Java-подобного ядра некоторыми функциональными идеями были предприняты на Java-платформе в Pizza, Nice, Multi-Java и самом Java 8.

Существуют также изначально функциональные языки, которые приобрели систему объектов. В качестве примера можно привести OCaml, F# и PLT-Scheme.

В Scala применяются также некоторые нововведения в области языков программирования. Например, его абстрактные типы — более объектно-ориентированная альтернатива обобщенным типам, его трейты позволяют выполнять гибкую сборку компонентов, а экстракторы обеспечивают независимый от представления способ сопоставления с образцом. Эти нововведения были озвучены в статьях на конференциях по языкам программирования в последние годы¹.

Резюме

Ознакомившись с текущей главой, вы получили некоторое представление о том, что такое Scala и как он может помочь программисту в работе. Разумеется, этот язык не решит все ваши проблемы и не увеличит волшебным образом вашу личную продуктивность. Следует заранее предупредить, что Scala нужно применять искусно, а для этого потребуются получить некоторые знания и практические навыки. Если вы перешли к Scala от языка Java, то одними из наиболее сложных аспектов его изучения для вас могут стать система типов Scala, которая существенно богаче, чем у Java, и его поддержка функционального стиля программирования. Цель данной книги — послужить руководством при поэтапном, от простого к сложному, изучении особенностей Scala. Полагаем, что вы приобретете весьма полезный интеллектуальный опыт, расширяющий ваш кругозор и изменяющий взгляд на проектирование программных средств. Надеемся, что вдобавок вы получите от программирования на Scala истинное удовольствие и познаете творческое вдохновение.

В следующей главе вы приступите к написанию кода Scala.

¹ Для получения дополнительной информации см. [Ode03], [Ode05] и [Emi07] в библиографии.

2

Первые шаги в Scala

Пришло время написать какой-нибудь код на Scala. Прежде чем углубиться в руководство по этому языку, мы приведем две обзорные главы по нему и, что наиболее важно, заставим вас приступить к написанию кода. Рекомендуем по мере освоения материала на практике проверить работу всех примеров кода, представленных в этой и последующей главах. Лучше всего приступить к изучению Scala, программируя на данном языке.

Запуск представленных далее примеров возможен с помощью стандартной установки Scala. Чтобы ее осуществить, перейдите по адресу www.scala-lang.org/downloads и следуйте инструкциям для вашей платформы. На этой странице описано несколько способов установки Scala. Будем считать, что вы уже установили двоичные файлы Scala и добавили их в переменную окружения `path`¹, что необходимо для выполнения шагов из этой главы.

Если вы опытный программист, но новичок в Scala, то внимательно прочитайте следующие две главы: в них приводится достаточный объем информации, позволяющий приступить к написанию полезных программ на этом языке. Если же опыт программирования у вас невелик, то часть материалов может показаться чем-то загадочным. Однако не стоит переживать. Чтобы ускорить процесс изучения, нам пришлось обойтись без некоторых подробностей. Более обстоятельные пояснения мы представим в последующих главах. Кроме того, в следующих двух главах дадим ряд ссылок с указанием разделов книги, в которых можно найти более подробные объяснения.

¹ Мы протестировали примеры из этой книги со Scala версии 3.0.0.

Шаг 1. Осваиваем Scala REPL

Самый простой способ начать работу со Scala — использовать Scala REPL¹, интерактивную оболочку для написания выражений и программ Scala. REPL, который называется `scala`, оценивает введенные вами выражения и выводит полученное значение. Чтобы его использовать, нужно набрать `scala` в командной строке²:

```
$ scala
Starting Scala REPL...
scala>
```

После того как вы наберете выражение, например `1 + 2`, и нажмете клавишу Enter:

```
scala> 1 + 2
```

REPL выведет на экран:

```
val res0: Int = 3
```

Эта строка включает:

- ключевое слово `val`, объявляющее переменную;
- автоматически сгенерированное или определенное пользователем имя для ссылки на вычисленное значение (`res0`, означающее результат 0);
- двоеточие (`:`), за которым следует тип выражения (`Int`);
- знак равенства (`=`);
- значение, полученное в результате вычисления выражения (`3`).

Тип `Int` означает класс `Int` в пакете `scala`. Пакеты в Scala аналогичны пакетам в Java — они разбивают глобальное пространство имен на части и предоставляют механизм для сокрытия данных³. Значения класса `Int` соответствуют `int`-значениям в Java. Если говорить в общем, то все примитивные типы Java имеют соответствующие классы в пакете `scala`. Например, `scala.Boolean` соответствует Java-типу `boolean`. А `scala.Float` соответствует

¹ REPL означает read, evaluate, print, loop («чтение, оценка, печать, цикл»).

² Если вы используете Windows, вам нужно будет ввести команду `scala` в оболочке командной строки.

³ Если вы не знакомы с пакетами Java, то их можно рассматривать как средство предоставления классам полных имен. `Int` входит в пакет `scala`. `Int` — простое имя класса, а `scala.Int` — полное. Подробнее о пакетах рассказывается в главе 12.

Java-типу `float`. И при компиляции вашего кода Scala в байт-код Java компилятор Scala будет по возможности использовать примитивные типы Java, чтобы обеспечить вам преимущество в производительности при работе с примитивными типами.

Идентификатор `resX` может использоваться в последующих строках. Например, поскольку ранее для `res0` было установлено значение 3, то результат выражения `res0 * 3` будет равен 9:

```
scala> res0 * 3  
val res1: Int = 9
```

Чтобы вывести на экран необходимое, но недостаточно информативное приветствие `Hello, world!`, наберите следующую команду:

```
scala> println("Hello, world!")  
Hello, world!
```

Функция `println` выводит на стандартное устройство вывода переданную ей строку, подобно тому как это делает `System.out.println` в Java.

Шаг 2. Объявляем переменные

В Scala имеются две разновидности переменных: `val`-переменные и `var`-переменные. Первые аналогичны финальным переменным в Java. После инициализации `val`-переменная уже никогда не может быть присвоена повторно. В отличие от нее `var`-переменная аналогична нефинальной переменной в Java и может быть присвоена повторно в течение своего жизненного цикла. Определение `val`-переменной выглядит так:

```
scala> val msg = "Hello, world!"  
val msg: String = Hello, world!
```

Эта инструкция вводит в употребление переменную `msg` в качестве имени для строки `"Hello, world!"`. Типом `msg` является `java.lang.String`, поскольку строки в JVM Scala реализуются Java-классом `String`.

Если вы привыкли объявлять переменные в Java, то в этом примере кода можете заметить одно существенное отличие: в `val`-определении нигде не фигурируют ни `java.lang.String`, ни `String`. Пример демонстрирует *логический вывод типов*, то есть возможность Scala определять неуказанные типы. В данном случае, поскольку вы инициализировали `msg` строковым литералом, Scala придет к выводу, что типом `msg` должен быть `String`. Когда

раздражительными, то можете поменять приветствие на просьбу оставить вас в покое:

```
scala> greeting = "Leave me alone, world!"  
greeting: String = Leave me alone, world!
```

Чтобы ввести в REPL код, который не помещается в одну строку, просто продолжайте набирать код после заполнения первой строки. Если набор кода еще не завершен, то REPL отреагирует установкой на следующей строке вертикальной черты:

```
scala> val multiline =  
    | "This is the next line."  
multiline: String = This is the next line.
```

Если вы понимаете, что набрали что-то не так, но REPL все еще ожидает ввода дополнительных данных, вы можете использовать клавиши со стрелками для перемещения вверх, вниз, влево или вправо, чтобы исправить ошибки. Если вы хотите полностью отменить ввод, вы можете выйти, дважды нажав **Enter**:

```
scala> val oops =  
    |  
    |  
You typed two blank lines. Starting a new command.  
scala>
```

Далее по тексту мы чаще всего будем опускать подсказку `scala`, верикальные линии и вывод REPL при успешном вводе, чтобы упростить чтение кода (и облегчить копирование и вставку из электронной книги PDF в REPL).

Шаг 3. Определяем функции

После работы с переменными в Scala вам, вероятно, захотелось написать какие-нибудь функции. Это делается так:

```
def max(x: Int, y: Int): Int =  
    if x > y then x  
    else y
```

Определение функции начинается с ключевого слова `def`. После имени функции, в данном случае `max`, стоит заключенный в круглые скобки перечень параметров, разделенных запятыми. За каждым параметром функции должна следовать аннотация типа, перед которой ставится двоеточие, поскольку компилятор Scala (и REPL, но с этого момента будет упоминаться только компи-

лятор) не выводит типы параметров функции. В данном примере функция по имени `max` получает два параметра, `x` и `y`, и оба они относятся к типу `Int`. После закрывающей круглой скобки перечня параметров функции `max` обнаруживается аннотация типа `: Int`. Она определяет *результующий тип* самой функции `max`¹. За типом результата функции следует знак равенства и тело функции, которое отделено отступами. В этом случае тело содержит одно выражение `if`, которое в качестве результата функции `max` выбирает либо `x`, либо `y`, в зависимости от того, что больше. Как показано здесь, выражение `if` в Scala может приводить к значению, аналогичному тернарному оператору Java. Например, в Scala выражение `if x > y then x else y` вычисляется точно так же, как выражение `(x > y) ? x : y` в Java. Знак равенства, предшествующий телу функции, дает понять, что с точки зрения функционального мира функция определяет выражение, результатом вычисления которого становится значение. Основная структура функции показана на рис. 2.1.



Рис. 2.1. Основная форма определения функции в Scala

Иногда компилятор Scala может потребовать от вас указать результирующий тип функции. Если, к примеру, функция является *рекурсивной*², то вы должны указать ее результирующий тип явно. Но в случае с функцией `max` вы можете не указывать результирующий тип функции — компилятор выведет его самостоятельно³. Кроме того, если функция состоит всего лишь

¹ В Java тип возвращаемого из метода значения является возвращаемым типом. В Scala то же самое понятие называется результирующим типом.

² Функция называется рекурсивной, если вызывает саму себя.

³ Тем не менее зачастую есть смысл указывать результирующий тип явно, даже когда компилятор этого не требует. Такая аннотация типа может упростить чтение кода, поскольку читателю не придется изучать тело функции, чтобы определить, каким будет вывод результирующего типа.

из одного оператора, вы сможете целиком написать ее в одну строку. Таким образом, у вас появляется альтернативный вариант реализации функции `max`:

```
def max(x: Int, y: Int) = if x > y then x else y
```

После того как вы определили функцию, вы можете вызвать ее по имени, например:

```
val bigger = max(3, 5) // 5
```

А вот определение функции, которая не принимает никаких параметров и не возвращает какого-либо интересного результата:

```
scala> def greet() = println("Hello, world!")  
def greet(): Unit
```

Когда определяется функция приветствия `greet()`, REPL откликается следующим приветствием: `def greet(): Unit`. Разумеется, слово `greet` — это имя функции. Пустота в скобках показывает, что функция не получает параметров. А `Unit` — результирующий тип функции `greet`. Он показывает, что функция не возвращает никакого интересного значения. Тип `Unit` в Scala подобен типу `void` в Java. Фактически каждый метод, возвращающий `void` в Java, отображается на метод, возвращающий `Unit` в Scala. Таким образом, методы с результирующим типом `Unit` выполняются только для того, чтобы проявились их побочные эффекты. В случае с `greet()` побочным эффектом будет дружеское приветствие, выведенное на стандартное устройство вывода.

При выполнении следующего шага код Scala будет помещен в файл и запущен в качестве скрипта. Если нужно выйти из REPL, то это можно сделать с помощью команды `:quit`:

```
scala> :quit  
$
```

Шаг 4. Пишем Scala-скрипты

Несмотря на то что язык Scala разработан, чтобы помочь программистам создавать очень большие масштабируемые системы, он вполне может подойти и для решения менее масштабных задач наподобие написания скриптов. Скрипт — это просто исходный файл Scala, который содержит функцию верхнего уровня, определяемую как `@main`. Поместите в файл по имени `hello.scala` следующий код:

```
@main def m() =  
  println("Hello, world, from a script!")
```

а затем запустите файл на выполнение:

```
$ scala hello.scala
```

И вы получите еще одно приветствие:

```
Hello, world, from a script!
```

В этом примере функция, отмеченная `@main`, называется `m` (от слова `main`), но это имя не имеет значения для выполнения скрипта. Чтобы скрипт сработал, вам необходимо запустить Scala и указать имя файла, содержащего функцию `main`, а не имя этой функции.

Вы можете получить доступ к аргументам командной строки, переданным вашему скрипту, приняв их в качестве параметров вашей основной функции. Например, вы можете принять строковые аргументы, взяв параметр со специальной аннотацией типа `String*`, что означает от нуля до многих повторяющихся параметров типа `String`¹. Внутри основной функции параметр будет иметь тип `Seq[String]`, то есть последовательность строк. В Scala последовательности начинаются с нуля, и чтобы получить доступ к элементу, необходимо указать его индекс в круглых скобках. Таким образом, первым элементом в последовательности Scala с именем `steps` будет `steps(0)`. Чтобы попробовать это, введите в новый файл с именем `helloarg.scala` следующее:

```
@main def m(args: String*) =  
  // поприветствуйте содержимое первого аргумента  
  println("Hello, " + args(0) + "!")
```

а затем запустите его на выполнение:

```
$ scala helloarg.scala planet
```

В данной команде `planet` передается в качестве аргумента командной строки, доступного в скрипте при использовании выражения `args(0)`. Поэтому вы должны увидеть на экране следующий текст:

```
Hello, planet!
```

Обратите внимание на наличие комментария в скрипте. Компилятор Scala проигнорирует символы между парой символов `//` и концом строки, а также все символы между сочетаниями символов `/*` и `*/`. Вдобавок в этом примере показана конкатенация `String`-значений, выполненная с помощью оператора `+`. Весь код работает вполне предсказуемо. Выражение `"Hello, " + "world!"` будет вычислено в строку `"Hello, world!"`.

¹ Повторяющиеся параметры описаны в разделе 8.8.

Шаг 5. Организуем цикл с `while` и принимаем решение с `if`

Чтобы попробовать в работе конструкцию `while`, наберите следующий код и сохраните его в файле `printargs.scala`:

```
@main def m(args: String*) =  
  var i = 0  
  while i < args.length do  
    println(args(i))  
    i += 1
```

ПРИМЕЧАНИЕ

Хотя примеры в данном разделе помогают объяснить суть циклов `while`, они не демонстрируют наилучший стиль программирования на Scala. В следующем разделе будут показаны более рациональные подходы, позволяющие избежать повторения последовательностей с помощью индексов.

Этот скрипт начинается с определения переменной, `var i = 0`. Вывод типов относит переменную `i` к типу `Int`, поскольку это тип ее начального значения `0`. Конструкция `while` на следующей строке заставляет блок (две строки кода снизу) повторно выполняться, пока булево выражение `i < args.length` будет вычисляться в `false`. Метод `args.length` вычисляет длину последовательности `args`. Блок содержит две инструкции, каждая из которых набрана с отступом в два пробела, что является рекомендуемым стилем отступов для кода на Scala. Первая инструкция, `println(args(i))`, выводит на экран `i`-й аргумент командной строки. Вторая, `i += 1`, увеличивает значение переменной `i` на единицу. Обратите внимание: Java-код `++i` и `i++` в Scala не работает. Чтобы в Scala увеличить значение переменной на единицу, нужно использовать одно из двух выражений: либо `i = i + 1`, либо `i += 1`. Запустите этот скрипт с помощью команды, показанной ниже:

```
$ scala printargs.scala Scala is fun
```

И вы увидите:

```
Scala  
is  
fun
```

Далее наберите в новом файле по имени `echoargs.scala` следующий код:

```
@main def m(args: String*) =  
  var i = 0
```

```
while i < args.length do
  if i != 0 then
    print(" ")
    print(args(i))
    i += 1
  println()
```

В целях вывода всех аргументов в одной и той же строке в этой версии вместо вызова `println` используется вызов `print`. Чтобы эту строку можно было прочитать, перед каждым аргументом, за исключением первого, благодаря использованию конструкции `if i != 0 then` вставляется пробел. При первом проходе цикла `while` выражение `i != 0` станет вычисляться в `false`, поэтому перед начальным элементом пробел выводиться не будет. В самом конце добавлена еще одна инструкция `println`, чтобы после вывода аргументов произошел переход на новую строку. Тогда у вас получится очень красивая картинка. Если запустить этот скрипт с помощью команды:

```
$ scala echoargs.scala Scala is even more fun
```

то вы увидите на экране такой текст:

```
Scala is even more fun
```

Обратите внимание, что в Scala, в отличие от Java, вам не нужно помещать логическое выражение `while` или `if` в круглые скобки. Еще одно отличие от Java состоит в том, что вы можете опустить фигурные скобки в блоке, даже если он содержит более одного оператора, при условии, что вы сделаете соответствующий отступ для каждой строки. И хотя вы не видели ни одной точки с запятой, Scala использует их для разделения операторов, как и Java, за исключением того, что в Scala эти знаки очень часто являются необязательными, что дает некоторое облегчение вашему правому мизинцу. Если бы вы были более многословны, вы могли бы написать скрипт `echoargs.scala` в стиле Java следующим образом:

```
@main def m(args: String*) = {
  var i = 0;
  while (i < args.length) {
    if (i != 0) {
      print(" ");
    }
    print(args(i));
    i += 1;
  }
  println();
}
```

Начиная со Scala 3, вместо фигурных скобок рекомендуется использовать стиль на основе отступов, называемый «тихим синтаксисом». В Scala 3 также были добавлены маркеры окончания кода, помогающие понять, где заканчиваются более крупные области с отступом. Маркеры окончания кода состоят из ключевого слова `end` и следующего за ним токена спецификатора, который является либо идентификатором, либо ключевым словом. Пример показан в листинге 10.9.

Шаг 6. Перебираем элементы с `foreach` и `for-do`

Возможно, при написании циклов `while` на предыдущем шаге вы даже не осознавали того, что программирование велось в *императивном* стиле. Обычно он применяется с такими языками, как Java, C++ и Python. При работе в этом стиле императивные команды в случае последовательного перебора элементов в цикле выдаются поочередно и зачастую изменяемое состояние совместно используется различными функциями. Scala позволяет программировать в императивном стиле, но, узнав этот язык получше, вы, скорее всего, перейдете преимущественно на *функциональный* стиль. По сути, одна из основных целей этой книги — помочь освоить работу в функциональном стиле, чтобы она стала такой же комфортной, как и работа в императивном.

Одна из основных характеристик функционального языка — то, что его функции относятся к конструкциям первого класса, и это абсолютно справедливо для языка Scala. Например, еще один, гораздо более лаконичный вариант вывода каждого аргумента командной строки выглядит так:

```
@main def m(args: String*) =  
  args.foreach(arg => println(arg))
```

В этом коде в отношении массива `args` вызывается метод `foreach`, в который передается функция. В данном случае передается *функциональный литерал* с одним параметром `arg`. Тело функции — вызов `println(arg)`. Если набрать показанный ранее код в новом файле по имени `pa.scala` и запустить этот файл на выполнение с помощью команды:

```
$ scala pa.scala Concise is nice
```

то на экране появятся строки:

```
Concise  
is  
nice
```


Поскольку вы не сможете понять всю его эффективность и выразительность, пока не доберетесь до раздела 7.3 (или не заглянете в него), здесь о нем будет дано лишь общее представление. Наберите в новом файле по имени `forargs.scala` следующий код:

```
@main def m(args: String*) =  
  for arg <- args do  
    println(arg)
```

Между `for` и `do` находится `arg <- args`¹. Справа от символа `<-` расположена уже знакомая вам последовательность `args`. Слева от `<-` указана переменная `arg`, относящаяся к `val`-, а не к `var`-переменным (так как она всегда относится к `val`-переменным, записывается только `arg`, а не `val arg`). Может показаться, что `arg` относится к `var`-переменной, поскольку она будет получать новое значение при каждой итерации, однако в действительности она относится к `val`-переменной: `arg` не может получить новое значение внутри тела выражения. Вместо этого для каждого элемента массива `args` будет создана новая `val`-переменная по имени `arg`, которая будет инициализирована значением элемента, и тело `for` будет выполнено.

Если скрипт `forargs.scala` запустить с помощью команды:

```
$ scala forargs.scala for arg in args
```

то вы увидите:

```
for  
arg  
in  
args
```

Диапазон применения выражения `for` значительно шире, но для начала этого примера достаточно. Мы расскажем вам больше о `for` в шаге 12 главы 3 и в разделе 7.3.

Резюме

В данной главе мы привели основную информацию о Scala. Надеемся, вы воспользовались возможностью создать код на этом языке. В следующей главе мы продолжим вводный обзор и рассмотрим более сложные темы.

¹ Вы можете интерпретировать символ `<-` как `in`. Следовательно, выражение `for arg <- args do` можно прочитать как `for arg in args do`.

3 Дальнейшие шаги в Scala

В этой главе продолжается введение в Scala, начатое в предыдущей главе. Здесь мы рассмотрим более сложные функциональные возможности. Когда вы усвоите материал главы, у вас будет достаточно знаний, чтобы начать создавать полезные скрипты на Scala. Мы вновь рекомендуем по мере чтения текста получать практические навыки с помощью приводимых примеров. Лучше всего осваивать Scala, начиная создавать код на данном языке.

Шаг 7. Параметризуем массивы типами

В Scala создавать объекты или экземпляры класса можно с помощью ключевого слова `new`. При создании объекта в Scala вы можете *параметризовать* его значениями и типами. Параметризация означает «конфигурирование» экземпляра при его создании. Параметризация экземпляра значениями производится путем передачи конструктору объектов в круглых скобках. Например, код Scala, который показан ниже, создает новый объект `java.math.BigInteger`, выполняя его параметризацию значением `"12345"`:

```
val big = new java.math.BigInteger("12345")
```

Параметризация экземпляра типами выполняется с помощью указания одного или нескольких типов в квадратных скобках. Пример показан в листинге 3.1. Здесь `greetStrings` — значение типа `Array[String]` («массив строк»), инициализируемое длиной 3 путем его параметризации значением 3 в первой строке кода. Если запустить код в листинге 3.1 в качестве скрипта, то вы увидите еще одно приветствие `Hello, world!`. Учтите, что при параметризации экземпляра как типом, так и значением тип стоит

первым и указывается в квадратных скобках, а за ним следует значение в круглых скобках.

Листинг 3.1. Параметризация массива типом

```
val greetStrings = new Array[String](3)
```

```
greetStrings(0) = "Hello"  
greetStrings(1) = ", "  
greetStrings(2) = "world!\n"
```

```
for i <- 0 to 2 do  
  print(greetStrings(i))
```

ПРИМЕЧАНИЕ

Хотя код в листинге 3.1 содержит важные понятия, он не показывает рекомендуемый способ создания и инициализации массива в Scala. Более рациональный способ будет показан в листинге 3.2.

Если вы склонны делать более явные указания, то тип `greetStrings` можно обозначить так:

```
val greetStrings: Array[String] = new Array[String](3)
```

С учетом имеющегося в Scala вывода типов эта строка кода семантически эквивалентна первой строке листинга 3.1. Но в данной форме показано следующее: часть параметризации, которая относится к типу (название типа в квадратных скобках), формирует часть типа экземпляра, однако часть параметризации, относящаяся к значению (значения в круглых скобках), в формировании не участвует. Типом `greetStrings` является `Array[String]`, а не `Array[String](3)`.

В следующих трех строках кода в листинге 3.1 инициализируется каждый элемент массива `greetStrings`:

```
greetStrings(0) = "Hello"  
greetStrings(1) = ", "  
greetStrings(2) = "world!\n"
```

Как уже упоминалось, доступ к массивам в Scala осуществляется за счет помещения индекса элемента в круглые, а не в квадратные скобки, как в Java. Следовательно, нулевым элементом массива будет `greetStrings(0)`, а не `greetStrings[0]`.

Эти три строки кода иллюстрируют важное понятие, помогающее осмыслить значение для Scala `val`-переменных. Когда переменная определяется

с помощью `val`, повторно присвоить значение данной переменной нельзя, но объект, на который она ссылается, потенциально может быть изменен. Следовательно, в данном случае присвоить `greetStrings` значение другого массива невозможно — переменная `greetStrings` всегда будет указывать на один и тот же экземпляр типа `Array[String]`, которым она была инициализирована. Но впоследствии в элементы типа `Array[String]` можно вносить изменения, то есть сам массив является изменяемым.

Последние две строки листинга 3.1 содержат выражение `for`, которое поочередно выводит каждый элемент массива `greetStrings`:

```
for i <- 0 to 2 do
  print(greetStrings(i))
```

В первой строке кода для этого выражения `for` показано еще одно общее правило Scala: если метод получает лишь один параметр, то его можно вызвать без точки или круглых скобок. В данном примере `to` на самом деле является методом, получающим один `Int`-аргумент. Код `0 to 2` преобразуется в вызов метода `0.to(2)`¹. Следует заметить, что этот синтаксис работает только при явном указании получателя вызова метода. Код `println 10` использовать нельзя, а код `Console.println 10` — можно.

С технической точки зрения в Scala нет перегрузки операторов, поскольку в нем фактически отсутствуют операторы в традиционном понимании. Вместо этого такие символы, как `+`, `-`, `*`, `/`, могут использоваться в качестве имен методов. Следовательно, когда при выполнении шага 1 вы набираете в интерпретаторе Scala код `1 + 2`, в действительности вы вызываете метод по имени `+` в отношении `Int`-объекта `1`, передавая ему в качестве параметра значение `2`. Как показано на рис. 3.1, вместо этого `1 + 2` можно записать с помощью традиционного синтаксиса вызова метода: `1.+(2)`.

Еще одна весьма важная идея, проиллюстрированная в этом примере, поможет понять, почему доступ к элементам массивов Scala осуществляется с помощью круглых скобок. В Scala меньше особых случаев по сравнению с Java. Массивы в Scala, как и в случае с любыми другими классами, — просто экземпляры классов. При использовании круглых скобок, окружающих одно или несколько значений переменной, Scala преобразует код в вызов метода по имени `apply` применительно к данной переменной.

¹ Этот метод `to` фактически возвращает не массив, а иную разновидность последовательности, содержащую значения 0, 1 и 2, последовательный перебор которых выполняется выражением `for`. Последовательности и другие коллекции будут рассматриваться в главе 15.

Следовательно, код `greetStrings(i)` преобразуется в код `greetStrings.apply(i)`. Получается, что элемент массива в Scala является просто вызовом обычного метода, ничем не отличающегося от любого своего собрата. Этот принцип не ограничивается массивами: любое использование объекта в отношении каких-либо аргументов в круглых скобках будет преобразовано в вызов метода `apply`. Разумеется, данный код будет скомпилирован, только если в этом типе объекта определен метод `apply`. То есть это не особый случай, а общее правило.

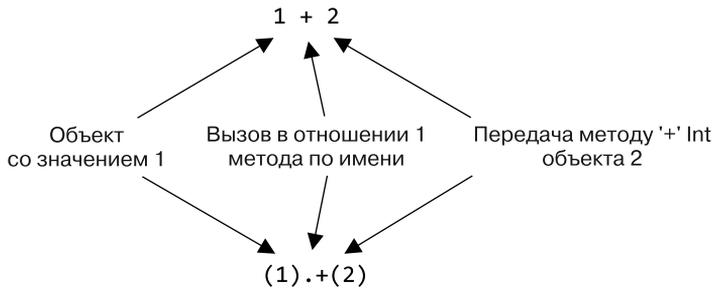


Рис. 3.1. Все операции в Scala являются вызовами методов

По аналогии с этим, когда присваивание выполняется в отношении переменной, к которой применены круглые скобки с одним или несколькими аргументами внутри, компилятор выполнит преобразование в вызов метода `update`, получающего не только аргументы в круглых скобках, но и объект, расположенный справа от знака равенства. Например, код

```
greetStrings(0) = "Hello"
```

будет преобразован в код

```
greetStrings.update(0, "Hello")
```

Таким образом, следующий код семантически эквивалентен коду листинга 3.1:

```
val greetStrings = new Array[String](3)

greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")

for i <- 0.to(2) do
  print(greetStrings.apply(i))
```

Концептуальная простота в Scala достигается за счет того, что все — от массивов до выражений — рассматривается как объекты с методами. Вам не нужно запоминать особые случаи, например такие, как существующее в Java различие между примитивными типами и соответствующими им типами-оболочками или между массивами и обычными объектами. Более того, подобное единообразие не вызывает больших потерь производительности. Компилятор Scala везде, где только возможно, использует в скомпилированном коде массивы Java, элементарные типы и чистые арифметические операции.

Рассмотренные до сих пор в этом шаге примеры компилируются и выполняются весьма неплохо, однако в Scala имеется более лаконичный способ создания и инициализации массивов, который, как правило, вы и будете использовать (см. листинг 3.2). Данный код создает новый массив длиной три элемента, инициализируемый переданными строками "zero", "one" и "two". Компилятор выводит тип массива как `Array[String]`, поскольку ему передаются строки.

Листинг 3.2. Создание и инициализация массива

```
val numNames = Array("zero", "one", "two")
```

Фактически в листинге 3.2 вызывается фабричный метод по имени `apply`, создающий и возвращающий новый массив. Метод `apply` получает переменное количество аргументов¹ и определяется в *объекте-компаньоне* `Array`. Подробнее объекты-компаньоны будут рассматриваться в разделе 4.3. Если вам приходилось программировать на Java, то можете воспринимать это как вызов статического метода по имени `apply` в отношении класса `Array`. Менее лаконичный способ вызова того же метода `apply` выглядит следующим образом:

```
val numNames2 = Array.apply("zero", "one", "two")
```

Шаг 8. Используем списки

Одна из превосходных отличительных черт функционального стиля программирования — полное отсутствие у методов побочных эффектов. Единственным действием метода должно быть вычисление и возвращение значения. Получаемые в результате применения такого подхода преимущества

¹ Списки аргументов переменной длины или повторяемые параметры рассматриваются в разделе 8.8.

закljučаются в том, что методы становятся менее запутанными, и это упрощает их чтение и повторное использование. Есть и еще одно преимущество (в статически типизированных языках): все попадающее в метод и выходящее за его пределы проходит проверку на принадлежность к определенному типу, поэтому логические ошибки, скорее всего, проявятся сами по себе в виде ошибок типов. Применять данную функциональную философию к миру объектов означает превратить эти объекты в неизменяемые.

Как вы уже видели, массив Scala — неизменяемая последовательность объектов с общим типом. Тип `Array[String]`, к примеру, содержит только строки. Изменить длину массива после создания его экземпляра невозможно, но вы можете изменять значения его элементов. Таким образом, массивы относятся к изменяемым объектам.

Для неизменяемой последовательности объектов с общим типом можно воспользоваться списком, определяемым Scala-классом `List`. Как и в случае применения массивов, в типе `List[String]` содержатся только строки. Список Scala `List` отличается от Java-типа `java.util.List` тем, что списки Scala всегда неизменяемые, а списки Java могут изменяться. В более общем смысле список Scala разработан с прицелом на использование функционального стиля программирования. Список создается очень просто, и листинг 3.3 как раз показывает это.

Листинг 3.3. Создание и инициализация списка

```
val oneTwoThree = List(1, 2, 3)
```

Код в листинге 3.3 создает новую `val`-переменную по имени `oneTwoThree`, инициализируемую новым списком `List[Int]` с целочисленными элементами 1, 2 и 3¹. Из-за своей неизменяемости списки ведут себя подобно строкам в Java: при вызове метода в отношении списка из-за имени данного метода может создаваться впечатление, что обрабатываемый список будет изменен, но вместо этого создается и возвращается новый список с новым значением. Например, в `List` для объединения списков имеется метод, обозначаемый как `::`. Используется он следующим образом:

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo :: threeFour
```

¹ Использовать запись `new List` не нужно, поскольку `List.apply()` определен в объекте-компаньоне `scala.List` как фабричный метод. Более подробно объекты-компаньоны рассматриваются в разделе 4.3.

После выполнения этого кода `oneTwoThreeFour` будет ссылаться на `List(1, 2, 3, 4)`, но `oneTwo` по-прежнему будет ссылаться на `List(1, 2)`, а `threeFour` — на `List(3, 4)`. Ни один из списков операндов не изменяется оператором конкатенации `:::`, который возвращает новый список со значением `List(1, 2, 3, 4)`. Возможно, работая со списками, вы чаще всего будете пользоваться оператором `::`, который называется `cons`. `cons` добавляет новый элемент в начало существующего списка и возвращает полученный список. Например, если вы запустите этот код:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
```

значение `oneTwoThree` будет `List(1, 2, 3)`.

ПРИМЕЧАНИЕ

В выражении `1 :: twoThree` метод `::` относится к правому операнду — списку `twoThree`. Можно заподозрить, будто с ассоциативностью метода `::` что-то не то, но есть простое мнемоническое правило: если метод используется в виде оператора, например `a * b`, то вызывается в отношении левого операнда, как в выражении `a.*(b)`, если только имя метода не заканчивается двоеточием. А если оно заканчивается двоеточием, то метод вызывается в отношении правого операнда. Поэтому в выражении `1 :: twoThree` метод `::` вызывается в отношении `twoThree` с передачей ему `1`, то есть `twoThree.::(1)`. Ассоциативность операторов более подробно будет рассматриваться в разделе 5.9.

Исходя из того, что короче всего указать пустой список с помощью `Nil`, один из способов инициализировать новые списки — связать элементы с помощью `cons`-оператора с `Nil` в качестве последнего элемента¹. Например, использование следующего способа инициализации переменной `oneTwoThree` даст ей то же значение, что и в предыдущем подходе `List(1, 2, 3)`:

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
```

Имеющийся в Scala класс `List` укомплектован весьма полезными методами, многие из которых показаны в табл. 3.1. Вся эффективность списков будет раскрыта в главе 14.

¹ Причина, по которой в конце списка нужен `Nil`, заключается в том, что метод `::` определен в классе `List`. Если попытаться просто воспользоваться кодом `1 :: 2 :: 3`, то он не пройдет компиляцию, поскольку `3` относится к типу `Int`, у которого нет метода `::`.

Тонкости добавления в списки

Класс `List` реализует операцию добавления в список с помощью команды `:+`. Подробнее об этом — в главе 24. Однако эта операция используется редко, поскольку время, необходимое для добавления элемента в список, увеличивается в соответствии с размером списка, а время при добавлении методом `::` фиксированное и не зависит от размера списка. Если вы хотите эффективно работать со списками, то добавляйте элементы в начало, а в конце вызовите `reverse`. В противном случае вы можете использовать `ListBuffer` — изменяемый список, который реализует операцию добавления, а после ее окончания вызовите `toList`. `ListBuffer` будет описан в разделе 15.1.

Таблица 3.1. Некоторые методы класса `List` и их использование

Что используется	Что этот метод делает
<code>List.empty</code> или <code>Nil</code>	Создает пустой список <code>List</code>
<code>List("Cool", "tools", "rule")</code>	Создает новый список типа <code>List[String]</code> с тремя значениями: "Cool", "tools" и "rule"
<code>val thrill = "Will" :: "fill" :: "until" :: Nil</code>	Создает новый список типа <code>List[String]</code> с тремя значениями: "Will", "fill" и "until"
<code>List("a", "b") ::: List("c", "d")</code>	Объединяет два списка (возвращает новый список типа <code>List[String]</code> со значениями "a", "b", "c" и "d")
<code>thrill(2)</code>	Возвращает элемент с индексом 2 (при начале отсчета с нуля) списка <code>thrill</code> (возвращает "until")
<code>thrill.count(s => s.length == 4)</code>	Подсчитывает количество строковых элементов в <code>thrill</code> , имеющих длину 4 (возвращает 2)
<code>thrill.drop(2)</code>	Возвращает список <code>thrill</code> без его первых двух элементов (возвращает <code>List("until")</code>)
<code>thrill.dropRight(2)</code>	Возвращает список <code>thrill</code> без двух крайних справа элементов (возвращает <code>List("Will")</code>)
<code>thrill.exists(s => s == "until")</code>	Определяет наличие в списке <code>thrill</code> строкового элемента, имеющего значение "until" (возвращает <code>true</code>)

Что используется	Что этот метод делает
<code>thrill.filter(s => s.length == 4)</code>	Возвращает список всех элементов списка <code>thrill</code> , имеющих длину 4, соблюдая порядок их следования в списке (возвращает <code>List("Will", "fill")</code>)
<code>thrill.forall(s => s.endsWith("l"))</code>	Показывает, заканчиваются ли все элементы в списке <code>thrill</code> буквой "l" (возвращает <code>true</code>)
<code>thrill.foreach(s => print(s))</code>	Выполняет инструкцию <code>print</code> в отношении каждой строки в списке <code>thrill</code> (выводит <code>"Willfilluntil"</code>)
<code>thrill.foreach(print)</code>	Делает то же самое, что и предыдущий код, но с использованием более лаконичной формы записи (также выводит <code>"Willfilluntil"</code>)
<code>thrill.head</code>	Возвращает первый элемент в списке <code>thrill</code> (возвращает <code>"Will"</code>)
<code>thrill.init</code>	Возвращает список всех элементов списка <code>thrill</code> , кроме последнего (возвращает <code>List("Will", "fill")</code>)
<code>thrill.isEmpty</code>	Показывает, не пуст ли список <code>thrill</code> (возвращает <code>false</code>)
<code>thrill.last</code>	Возвращает последний элемент в списке <code>thrill</code> (возвращает <code>"until"</code>)
<code>thrill.length</code>	Возвращает количество элементов в списке <code>thrill</code> (возвращает 3)
<code>thrill.map(s => s + "y")</code>	Возвращает список, который получается в результате добавления "y" к каждому строковому элементу в списке <code>thrill</code> (возвращает <code>List("Willy", "filly", "untily")</code>)
<code>thrill.mkString(", ")</code>	Создает строку с элементами списка (возвращает <code>"Will, fill, until"</code>)
<code>thrill.filterNot(s => s.length == 4)</code>	Возвращает список всех элементов в порядке их следования в списке <code>thrill</code> , за исключением имеющих длину 4 (возвращает <code>List("until")</code>)
<code>thrill.reverse</code>	Возвращает список, содержащий все элементы списка <code>thrill</code> , следующие в обратном порядке (возвращает <code>List("until", "fill", "Will")</code>)

Таблица 3.1 (окончание)

Что используется	Что этот метод делает
<code>thrill.sortWith((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)</code>	Возвращает список, содержащий все элементы списка <code>thrill</code> в алфавитном порядке с первым символом, преобразованным в символ нижнего регистра (возвращает <code>List("fill", "until", "will")</code>)
<code>thrill.tail</code>	Возвращает список <code>thrill</code> за исключением его первого элемента (возвращает <code>List("fill", "until")</code>)

Шаг 9. Используем кортежи

Еще один полезный объект-контейнер — *кортеж*. Как и списки, кортежи не могут быть изменены, но, в отличие от списков, могут содержать различные типы элементов. Список может быть типа `List[Int]` или `List[String]`, а кортеж может содержать одновременно как целые числа, так и строки. Кортежи находят широкое применение, например, при возвращении из метода сразу нескольких объектов. Там, где на Java для хранения нескольких возвращаемых значений зачастую приходится создавать `JavaBean`-подобный класс, в Scala можно просто вернуть кортеж. Все делается просто: чтобы создать экземпляр нового кортежа, содержащего объекты, нужно лишь заключить объекты в круглые скобки, отделив их друг от друга запятыми. Создав экземпляр кортежа, вы можете получить доступ к его элементам по отдельности с помощью нулевого индекса в круглых скобках. Пример показан в листинге 3.4.

Листинг 3.4. Создание и использование кортежа

```
val pair = (99, "Luftballons")
val num = pair(0) // тип Int, значение 99
val what = pair(1) // тип String, значение "Luftballons"
```

В первой строке листинга 3.4 создается новый кортеж, содержащий в качестве первого элемента целочисленное значение 99, а в качестве второго — строку "Luftballons". Scala выводит тип кортежа в виде `Tuple2[Int, String]`, а также присваивает этот тип паре переменных¹. Во второй строке

¹ Компилятор Scala использует синтаксический сахар для типов кортежей, который выглядит как кортеж типов. Например, `Tuple2[Int, String]` представлен как `(Int, String)`.

вы получаете доступ к первому элементу 99 по его индексу 0¹. Результатом типа `pair(0)` является `Int`. В третьей строке вы получаете доступ ко второму элементу `Luftballons` по его индексу 1. Результатом типа `pair(1)` является `String`. Это говорит о том, что кортежи отслеживают индивидуальные типы каждого из своих элементов.

Реальный тип кортежа зависит от количества содержащихся в нем элементов и от типов этих элементов. Следовательно, типом кортежа `(99, "Luftballons")` является `Tuple2[Int, String]`. А типом кортежа `('u', 'r', "the", 1, 4, "me")` — `Tuple6[Char, Char, String, Int, Int, String]`².

Шаг 10. Используем множества и отображения

Scala призван помочь вам использовать преимущества как функционального, так и объектно-ориентированного стиля, поэтому в библиотеках его коллекций особое внимание обращают на разницу между изменяемыми и неизменяемыми коллекциями. Например, массивы всегда изменяемы, а списки всегда неизменяемы. Scala также предоставляет изменяемые и неизменяемые альтернативы для множеств и отображений, но использует для обеих версий одни и те же простые имена. Для множеств и отображений Scala моделирует изменяемость в иерархии классов.

Например, в API Scala содержится основной *трейт* для множеств, где этот трейт аналогичен Java-интерфейсу (более подробно трейты рассматриваются в главе 11). Затем Scala предоставляет два трейта-наследника: один для изменяемых, а второй для неизменяемых множеств.

На рис. 3.2 показано, что для всех трех трейтов используется одно и то же простое имя `Set`. Но их полные имена отличаются друг от друга, поскольку все трейты размещаются в разных пакетах. Классы для конкретных множеств в Scala API, например `HashSet` (см. рис. 3.2), являются расширениями либо изменяемого, либо неизменяемого трейта `Set`. (В то время как в Java вы реализуете интерфейсы, в Scala расширяете (иначе говоря, подмешиваете) трейты.) Следовательно, если нужно воспользоваться `HashSet`, то в зависимости от потребностей можно выбирать между его изменяемой и неизменяемой

¹ Обратите внимание, что до Scala 3 обращение к элементам кортежа осуществлялось с помощью имен полей, начинающихся с единицы, например `_1` или `_2`.

² Как и в Scala 3, вы можете создавать кортежи любой длины.

разновидностями. Способ создания множества по умолчанию показан в листинге 3.5.

Листинг 3.5. Создание, инициализация и использование неизменяемого множества

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
val query = jetSet.contains("Cessna") // false
```

В первой строке кода листинга 3.5 определяется новая `var`-переменная по имени `jetSet`, которая инициализируется неизменяемым множеством, содержащим две строки: `"Boeing"` и `"Airbus"`. В этом примере показано, что в Scala множества можно создавать точно так же, как списки и массивы: путем вызова фабричного метода по имени `apply` в отношении объекта-компаньона `Set`. В листинге 3.5 метод `apply` вызывается в отношении объекта-компаньона для `scala.collection.immutable.Set`, возвращающего экземпляр исходного, неизменяемого класса `Set`. Компилятор Scala выводит тип переменной `jetSet`, определяя его как неизменяемый `Set[String]`.

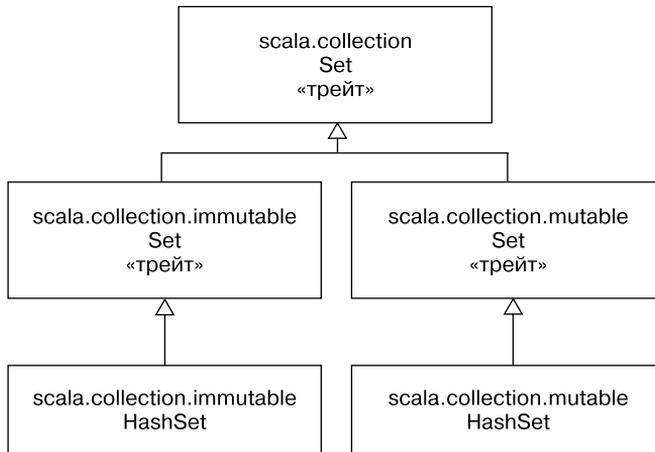


Рис. 3.2. Иерархия классов для множеств Scala

Чтобы добавить новый элемент в неизменяемое множество, в отношении последнего вызывается метод `+`, которому и передается этот элемент. Метод `+` создает и возвращает новое неизменяемое множество с добавленным элементом. Конкретный метод `+=` предоставляется исключительно для изменяемых множеств.

В данном случае вторая строка кода, `jetSet += "Lear"`, фактически является сокращенной формой записи следующего кода:

```
jetSet = jetSet + "Lear"
```

Следовательно, во второй строке кода листинга 3.5 `var`-переменной `jetSet` присваивается новое множество, содержащее "Boeing", "Airbus" и "Lear". Наконец, последняя строка листинга 3.5 определяет, содержит ли множество строку "Cessna" (как и следовало ожидать, результат — `false`).

Если нужно изменяемое множество, то следует, как показано в листинге 3.6, воспользоваться инструкцией `import`.

Листинг 3.6. Создание, инициализация и использование изменяемого множества

```
import scala.collection.mutable

val movieSet = mutable.Set("Spotlight", "Moonlight")
movieSet += "Parasite"
// movieSet теперь содержит: "Spotlight", "Moonlight", "Parasite"
```

В первой строке данного листинга выполняется импорт `scala.collection.mutable`. Инструкция `import` позволяет использовать простое имя, например `Set`, вместо длинного полного имени. В результате при указании `mutable.Set` во второй строке компилятор знает, что имеется в виду `scala.collection.mutable.Set`. В этой строке `movieSet` инициализируется новым изменяемым множеством, содержащим строки "Spotlight" и "Moonlight". В следующей строке к изменяемому множеству добавляется "Parasite", для чего в отношении множества вызывается метод `+=` с передачей ему строки "Parasite". Как уже упоминалось, `+=` — метод, определенный для изменяемых множеств. При желании можете вместо кода `movieSet += "Parasite"` воспользоваться кодом `movieSet.+=("Shrek")`¹.

Рассмотренной до сих пор исходной реализации множеств, которые выполняются изменяемыми и неизменяемыми фабричными методами `Set`, скорее всего, будет достаточно для большинства ситуаций. Однако временами может потребоваться специальный вид множества. К счастью, при

¹ Множество в листинге 3.6 изменяемое, поэтому повторно присваивать значение `movieSet` не нужно, и данная переменная может относиться к `val`-переменным. В отличие от этого, использование метода `+=` с неизменяемым множеством в листинге 3.5 требует повторного присваивания значения переменной `jetSet`, поэтому она должна быть `var`-переменной.

этом используется аналогичный синтаксис. Следует просто импортировать нужный класс и применить фабричный метод в отношении его объекта-компаньона. Например, если требуется неизменяемый `HashSet`, то можно сделать следующее:

```
import scala.collection.immutable.HashSet

val hashSet = HashSet("Tomatoes", "Chilies")
val ingredients = hashSet + "Coriander"
// ingredients содержит "Tomatoes", "Chilies", "Coriander"
```

Еще одним полезным трейтом коллекций в Scala является отображение — `Map`. Как и для множеств, Scala предоставляет изменяемые и неизменяемые версии `Map` с применением иерархии классов. Как показано на рис. 3.3, иерархия классов для отображений во многом похожа на иерархию для множеств. В пакете `scala.collection` есть основной трейт `Map` и два трейта-наследника отображения `Map`: изменяемый вариант в `scala.collection.mutable` и неизменяемый в `scala.collection.immutable`.

Реализации `Map`, например `HashMap`-реализации в иерархии классов, показанной на рис. 3.3, расширяются либо в изменяемый, либо в неизменяемый трейт. Отображения можно создавать и инициализировать, используя фабричные методы, подобные тем, что применялись для массивов, списков и множеств.

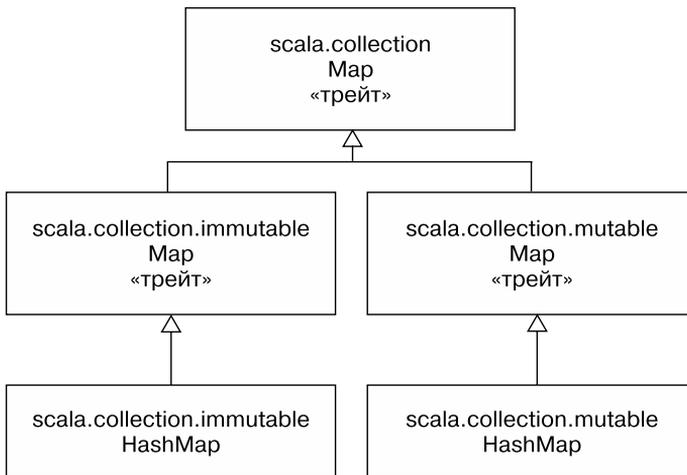


Рис. 3.3. Иерархия классов для отображений Scala

Листинг 3.7. Создание, инициализация и использование изменяемого отображения

```
import scala.collection.mutable

val treasureMap = mutable.Map.empty[Int, String]
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
val step2 = treasureMap(2) // " Find big X on ground."
```

Например, в листинге 3.7 показана работа с изменяемым отображением: в первой строке оно импортируется, затем определяется `val`-переменная `treasureMap`, которая инициализируется пустым изменяемым отображением, имеющим целочисленные ключи и строковые значения. Оно пустое, поскольку вызывается фабричный метод с именем `empty` и указывается `Int` в качестве типа ключа и `String` в качестве типа значения¹. В следующих трех строках к отображению добавляются пары «ключ — значение», для чего используются методы `->` и `+=`. Как уже было показано, компилятор Scala преобразует выражения бинарных операций вида `1 -> "Go to island."` в код `(1).->("Go to island.")`. Следовательно, когда указывается `1 -> "Go to island."`, фактически в отношении объекта `1` вызывается метод по имени `->`, которому передается строка со значением `"Go to island."`. Метод `->`, который можно вызвать в отношении любого объекта в программе Scala, возвращает двухэлементный кортеж, содержащий ключ и значение². Затем этот кортеж передается методу `+=` объекта отображения, на который ссылается `treasureMap`. И наконец, в последней строке ищется значение, соответствующее ключу `2` в `treasureMap`. После выполнения этого кода переменная `step2` будет ссылаться на `"Find big X on ground"`.

Если отдать предпочтение неизменяемому отображению, то ничего импортировать не нужно, поскольку это отображение используется по умолчанию. Пример показан в листинге 3.8.

¹ Явная параметризация типа `"[Int, String]"` требуется в листинге 3.7 из-за того, что без какого-либо значения, переданного фабричному методу, компилятор не в состоянии выполнить логический вывод типов параметров отображения. В отличие от этого компилятор может выполнить вывод типов параметров из значений, переданных фабричному методу `map`, показанному в листинге 3.8, поэтому явного указания типов параметров там не требуется.

² Механизм Scala, позволяющий вызывать такие методы, как `->` для объектов, которые не объявляют их напрямую, называется методом расширения. Он будет рассмотрен в главе 22.

Листинг 3.8. Создание, инициализация и использование неизменяемого отображения

```
val romanNumeral = Map(  
  1 -> "I", 2 -> "II", 3 -> "III", 4 -> "IV", 5 -> "V"  
)  
val four = romanNumeral(4) // "IV"
```

Учитывая отсутствие импортирования, при указании `Map` в первой строке данного листинга вы получаете используемый по умолчанию экземпляр класса `scala.collection.immutable.Map`. Фабричному методу отображения передаются пять кортежей «ключ — значение», а он возвращает неизменяемое `Map`-отображение, содержащее эти переданные пары. Если запустить код, показанный в листинге 3.8, то переменная `four` будет ссылаться на `IV`.

Шаг 11. Учимся распознавать функциональный стиль

Как упоминалось в главе 1, Scala позволяет программировать в императивном стиле, но побуждает вас переходить преимущественно к функциональному. Если к Scala вы пришли, имея опыт работы в императивном стиле, к примеру, вам приходилось программировать на Java, то одной из основных возможных сложностей станет программирование в функциональном стиле. Мы понимаем, что поначалу этот стиль может быть неизвестен, и в данной книге стараемся перевести вас из одного состояния в другое. От вас также потребуются некоторые усилия, которые мы настоятельно рекомендуем приложить. Мы уверены, что при наличии опыта работы в императивном стиле изучение программирования в функциональном позволит вам не только стать более квалифицированным программистом на Scala, но и расширит ваш кругозор, сделав вас более ценным программистом в общем смысле.

Сначала нужно усвоить разницу между двумя стилями, отражающуюся в коде. Один верный признак заключается в том, что если код содержит `var`-переменные, то он, вероятнее всего, написан в императивном стиле. Если он вообще не содержит `var`-переменных, то есть включает *только* `val`-переменные, то, вероятнее всего, он написан в функциональном стиле. Следовательно, один из способов приблизиться к последнему — попытаться обойтись в программах без `var`-переменных.

Обладая багажом императивности, то есть опытом работы с такими языками, как Java, C++ или C#, `var`-переменные можно рассматривать в качестве обычных, а `val`-переменные — в качестве переменных особого вида. В то же

время, если у вас имеется опыт работы в функциональном стиле на таких языках, как Haskell, OCaml или Erlang, `val`-переменные можно представлять как обычные, а `var`-переменные — как некое кощунственное обращение с кодом. Но с точки зрения Scala `val`- и `var`-переменные — всего лишь два разных инструмента в вашем арсенале средств и оба одинаково полезны и не отвергаемы. Scala побуждает вас к использованию `val`-переменных, но, по сути, дает возможность применять тот инструмент, который лучше подходит для решаемой задачи. И тем не менее, даже будучи согласными с подобной философией, вы поначалу можете испытывать трудности, связанные с избавлением от `var`-переменных в коде.

Рассмотрим позаимствованный из главы 2 пример цикла `while`, в котором используется `var`-переменная, означающая, что он выполнен в императивном стиле:

```
def printArgs(args: List[String]): Unit =
  var i = 0
  while i < args.length do
    println(args(i))
    i += 1
```

Вы можете преобразовать этот код — придать ему более функциональный стиль, отказавшись от использования `var`-переменной, например, так:

```
def printArgs(args: List[String]): Unit =
  for arg <- args do
    println(arg)
```

или вот так:

```
def printArgs(args: List[String]): Unit =
  args.foreach(println)
```

В этом примере демонстрируется одно из преимуществ программирования с меньшим количеством `var`-переменных. Код после рефакторинга (более функциональный) выглядит понятнее, он более лаконичен, и в нем труднее допустить какие-либо ошибки, чем в исходном (более императивном) коде. Причина навязывания в Scala функционального стиля заключается в том, что он помогает создавать более понятный код, при написании которого труднее ошибиться.

Но вы можете пойти еще дальше. Метод после рефакторинга `printArgs` нельзя отнести к *чисто* функциональным, поскольку у него имеются побочные эффекты. В данном случае такой эффект — вывод в поток стандартного устройства вывода. Признаком функции, имеющей побочные эффекты,

выступает то, что результирующим типом у нее является `Unit`. Если функция не возвращает никакого интересного значения, о чем, собственно, и свидетельствует результирующий тип `Unit`, то единственный способ внести этой функцией какое-либо изменение в окружающий мир — проявить некий побочный эффект. Более функциональным подходом будет определение метода, который форматирует передаваемые аргументы в целях их последующего вывода и, как показано в листинге 3.9, просто возвращает отформатированную строку.

Листинг 3.9. Функция без побочных эффектов или `var`-переменных

```
def formatArgs(args: List[String]) = args.mkString("\n")
```

Теперь вы действительно перешли на функциональный стиль: нет ни побочных эффектов, ни `var`-переменных. Метод `mkString`, который можно вызвать в отношении любой коллекции, допускающей последовательный перебор элементов (включая массивы, списки, множества и отображения), возвращает строку, состоящую из результата вызова метода `toString` в отношении каждого элемента, с разделителями из переданной строки. Таким образом, если `args` содержит три элемента, "zero", "one" и "two", то метод `formatArgs` возвращает "zero\none\ntwo". Разумеется, эта функция, в отличие от методов `printArgs`, ничего не выводит, но в целях выполнения данной работы ее результаты можно легко передать функции `println`:

```
println(formatArgs(args))
```

Каждая полезная программа, вероятнее всего, будет иметь какие-либо побочные эффекты. Отдавая предпочтение методам без побочных эффектов, вы будете стремиться к разработке программ, в которых такие эффекты сведены к минимуму. Одним из преимуществ такого подхода станет упрощение тестирования ваших программ.

Например, чтобы протестировать любой из трех показанных ранее в этом разделе методов `printArgs`, вам придется переопределить метод `println`, перехватить передаваемый ему вывод и убедиться в том, что он соответствует вашим ожиданиям. В отличие от этого функцию `formatArgs` можно протестировать, просто проверяя ее результат:

```
val res = formatArgs(List("zero", "one", "two"))
assert(res == "zero\none\ntwo")
```

Имеющийся в `Scala` метод `assert` проверяет переданное ему булево выражение и, если последнее вычисляется в `false`, выдает ошибку `AssertionError`. Если же переданное булево выражение вычисляется

в `true`, то метод просто молча возвращает управление вызвавшему его коду. Более подробно о тестах, проводимых с помощью `assert`, и тестировании речь пойдет в главе 25.

И все-таки нужно иметь в виду: ни `var`-переменные, ни побочные эффекты не следует рассматривать как нечто абсолютно неприемлемое. Scala не является чисто функциональным языком, заставляющим вас программировать в функциональном стиле. Scala — гибридный императивный и функциональный языки. Может оказаться, что в некоторых ситуациях для решения текущей задачи больше подойдет императивный стиль, и тогда вы должны прибегнуть к нему без всяких колебаний. Но чтобы помочь вам разобраться в программировании без использования `var`-переменных, в главе 7 мы покажем множество конкретных примеров кода с использованием `var`-переменных и рассмотрим способы их преобразования в `val`-переменные.

Сбалансированный подход Scala-программистов

Старайтесь отдавать предпочтение `val`-переменным, неизменяемым объектам и методам без побочных эффектов. Используйте `var`-переменные, изменяемые объекты и методы с побочными эффектами тогда, когда у вас есть конкретная необходимость и обоснование для их использования.

Шаг 12. Преобразование с отображениями и `for-yield`

При программировании в императивном стиле вы видоизменяете существующие структуры данных до тех пор, пока не достигнете цели алгоритма. В функциональном стиле для достижения цели вы преобразуете неизменяемые структуры данных в новые.

Важным методом, упрощающим функциональные преобразования неизменяемых коллекций, является `map`. Как и `foreach`, `map` принимает функцию в качестве параметра. Но в отличие от `foreach`, который использует переданную функцию для выполнения побочного эффекта для каждого элемента, `map` использует переданную функцию для преобразования каждого элемента в новое значение. Результатом работы `map` является новая коллекция, содержащая эти новые значения. Например, учитывая этот список строк:

```
val adjectives = List("One", "Two", "Red", "Blue")
```

вы можете преобразовать его в новый список из новых строк, например:

```
val nouns = adjectives.map(adj => adj + " Fish")
// List(One Fish, Two Fish, Red Fish, Blue Fish)
```

Другой способ выполнить преобразование — использовать выражение `for`, в котором вы вводите тело функции с ключевым словом `yield` вместо `do`:

```
val nouns =
  for adj <- adjectives yield
    adj + " Fish"
// List(One Fish, Two Fish, Red Fish, Blue Fish)
```

`For-yield` дает точно такой же результат, что и `map`, потому что компилятор преобразует выражение `for-yield` в вызов `map`¹. Поскольку список, возвращаемый `map`, содержит значения, созданные переданной функцией, тип элементов возвращаемого списка будет такой же, как и результат функции. В предыдущем примере переданная функция возвращает строку, поэтому `map` возвращает `List[String]`. Если функция, переданная `map`, приводит к другому типу, то список, возвращаемый `map`, будет содержать этот тип в качестве типа элемента. Например, ниже функция `map` преобразует строку в целое число, равное длине каждого элемента строки. Следовательно, результатом `map` является новый `List[Int]`, содержащий эти длины:

```
val lengths = nouns.map(noun => noun.length)
// List(8, 8, 8, 9)
```

Как и раньше, вы также можете использовать выражение `for-yield` для достижения того же преобразования:

```
val lengths =
  for noun <- nouns yield
    noun.length
// List(8, 8, 8, 9)
```

Метод `map` присутствует во многих типах, не только в `List`. Это позволяет использовать выражения со многими типами. Одним из примеров является `Vector` — неизменяемая последовательность, обеспечивающая «фактически фиксированное время» для всех своих операций. Поскольку `Vector` предлагает метод `map` с соответствующей сигнатурой, вы можете выполнять те же виды функциональных преобразований в `Vectors`, что и в `Lists`, либо напрямую вызывая `map`, либо используя `for-yield`. Например:

¹ Подробности того, как компилятор переписывает выражения, будут даны в разделе 7.3.

```
val ques = Vector("Who", "What", "When", "Where", "Why")

val usingMap = ques.map(q => q.toLowerCase + "?")
// Vector(who?, what?, when?, where?, why?)

val usingForYield =
  for q <- ques yield
    q.toLowerCase + "?"
// Vector(who?, what?, when?, where?, why?)
```

Обратите внимание, что при сопоставлении `List` вы получаете новый `List`. Когда вы сопоставляете `Vector`, вы получаете обратно новый `Vector`. В дальнейшем вы поймете, что этот шаблон верен для большинства типов, которые определяют метод `map`.

В качестве последнего примера рассмотрим тип `Option` в Scala. Scala использует `Option` для представления необязательного значения, избегая традиционной техники Java, использующей для этой цели `null`¹. Параметр `Option` — это либо `Some`, что указывает на то, что значение существует, либо `None`, которое указывает, что значение не существует.

В качестве примера, показывающего `Option` в действии, рассмотрим метод `find`. Все типы коллекций Scala, включая `List` и `Vector`, предлагают `find`, который ищет элемент, соответствующий заданному предикату, — функцию, которая принимает аргумент типа элемента и возвращает булево значение. Тип результата `find` — `Option[E]`, где `E` — тип элемента коллекции. Метод `find` выполняет итерации по элементам коллекции, передавая каждый из них предикату. Если функция возвращает `true`, `find` прекращает итерацию и возвращает этот элемент, заключенный в `Some`. Если `find` доходит до конца элементов без передачи предикату, он возвращает `None`. Вот несколько примеров, в которых тип результата поиска всегда `Option[String]`:

```
val startsw = ques.find(q => q.startsWith("W")) // Some(Who)
val hasLen4 = ques.find(q => q.length == 4)    // Some(What)
val hasLen5 = ques.find(q => q.length == 5)    // Some(Where)
val startsh = ques.find(q => q.startsWith("H")) // None
```

Хотя `Option` не является коллекцией, он предлагает `map`-метод². Если `Option` является `Some`, который называется «определенным» параметром, `map`

¹ В Java 8 к стандартной библиотеке был добавлен тип `Optional`, но многие существующие библиотеки Java по-прежнему используют `null` для обозначения отсутствующего необязательного значения.

² Однако `Option` можно представить как набор, который содержит либо ноль (случай `None`) элементов, либо один (случай `Some`).

возвращает новый `Option`, содержащий результат передачи исходного элемента `Some` в функцию, переданную в `map`. Вот пример преобразования `startSW` в `Some`, содержащего строку `WHO`:

```
startSW.map(word => word.toUpperCase) // Some(WHO)
```

Как и в случае с `List` и `Vector`, вы можете добиться того же преобразования в `Option` с помощью `for-yield`:

```
for word <- startSW yield word.toUpperCase // Some(WHO)
```

Если вы вызываете `map` с параметром `None` (параметр, который не определен), то вернете значение `None`. Например:

```
startSH.map(word => word.toUpperCase) // None
```

А вот такое же преобразование с использованием `for-yield`:

```
for word <- startSH yield word.toUpperCase // None
```

Вы можете преобразовать многие другие типы с помощью `map` и `for-yield`, но пока этого достаточно. Этот шаг дал вам представление о том, сколько кода Scala написано в виде функциональных преобразований неизменяемых структур данных.

Резюме

Знания, полученные в этой главе, позволят вам начать применять Scala для решения небольших задач, в особенности тех, для которых используются скрипты. В последующих главах мы глубже разберем рассмотренные темы, а также представим другие, не затронутые здесь.

4

Классы и объекты

В предыдущих двух главах вы разобрались в основах классов и объектов. В этой главе вам предстоит углубленно проработать данную тему. Здесь мы дадим дополнительные сведения о классах, полях и методах, а также общее представление о том, когда подразумевается использование точки с запятой. Кроме того, рассмотрим объекты-одиночки (singleton) и то, как с их помощью писать и запускать приложения на Scala. Если вам уже знаком язык Java, то вы увидите, что в Scala фигурируют похожие, но все же немного отличающиеся концепции. Поэтому чтение данной главы пойдет на пользу даже великим знатокам языка Java.

4.1. Классы, поля и методы

Классы — «чертежи» объектов. После определения класса из него, как по чертежу, можно создавать объекты, воспользовавшись для этого ключевым словом `new`. Например, при наличии следующего определения класса:

```
class ChecksumAccumulator:  
  // Сюда помещается определение класса с отступом
```

с помощью кода

```
new ChecksumAccumulator
```

можно создавать объекты `ChecksumAccumulator`.

Внутри определения класса помещаются поля и методы, которые в общем называются *членами* класса. Поля, которые определяются либо как `val`-, либо как `var`-переменные, являются переменными, относящимися к объектам.

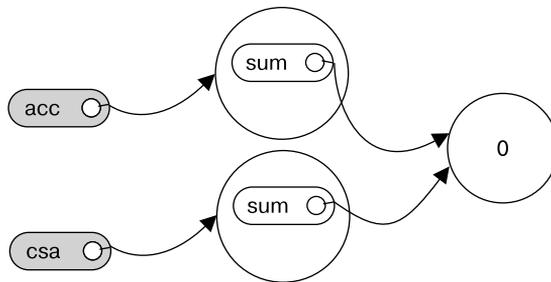
Методы, определяемые с помощью ключевого слова `def`, содержат исполняемый код. В полях хранятся состояние или данные объекта, а методы используют эти данные для выполнения в отношении объекта вычислений. При создании экземпляра класса среда выполнения приложения резервирует часть памяти для хранения образа состояния получающегося при этом объекта (то есть содержимого его полей). Например, если вы определите класс `ChecksumAccumulator` и дадите ему `var`-поле по имени `sum`:

```
class ChecksumAccumulator:  
    var sum = 0
```

а потом дважды создадите его экземпляры с помощью следующего кода:

```
val acc = new ChecksumAccumulator  
val csa = new ChecksumAccumulator
```

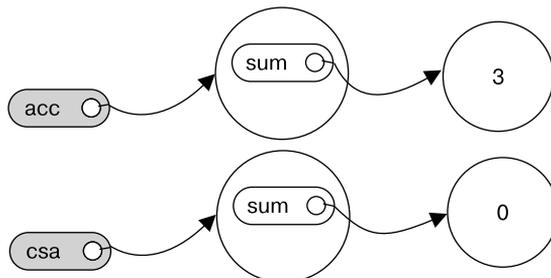
то образ объектов в памяти может выглядеть так:



Поскольку `sum` — поле, определенное внутри класса `ChecksumAccumulator`, и относится к `var`-, а не к `val`-переменным, то впоследствии ему (полю) можно заново присвоить другое `Int`-значение:

```
acc.sum = 3
```

Теперь картинка может выглядеть так:



По поводу этой картинки нужно отметить следующее: на ней показаны две переменные `sum`. Одна из них находится в объекте, на который ссылается `acc`, а другая — в объекте, на который ссылается `csa`. Поля также называют *переменными экземпляра*, поскольку каждый экземпляр получает собственный набор переменных. Все переменные экземпляра объекта составляют образ объекта в памяти. То, что здесь показано, свидетельствует не только о наличии двух переменных `sum`, но и о том, что изменение одной из них никак не отражается на другой.

В этом примере следует также отметить: у вас есть возможность изменить объект, на который ссылается `acc`, даже несмотря на то, что `acc` относится к `val`-переменным. Но с учетом того, что `acc` (или `csa`) являются `val`-, а не `var`-переменными, вы не можете присвоить им какой-нибудь другой объект. Например, попытка, показанная ниже, не будет успешной:

```
// Не пройдет компиляцию, поскольку acc является val-переменной
acc = new ChecksumAccumulator
```

Теперь вы можете рассчитывать на то, что переменная `acc` всегда будет ссылаться на тот же объект `ChecksumAccumulator`, с помощью которого вы ее инициализировали; поля же, содержащиеся внутри этого объекта, могут со временем измениться.

Один из важных способов обеспечения надежности объекта — гарантия того, что состояние этого объекта, то есть значения его переменных экземпляра, остается корректным в течение всего его жизненного цикла. Первый шаг к предотвращению непосредственного стороннего доступа к полям — создание *приватных* (`private`) полей. Доступ к приватным полям можно получить только методами, определенными в том же самом классе, поэтому весь код, который может обновить состояние, будет локализован в классе. Чтобы объявить поле приватным, перед ним нужно поставить модификатор доступа `private`:

```
class ChecksumAccumulator:
    private var sum = 0
```

С таким определением `ChecksumAccumulator` любая попытка доступа к `sum` за пределами класса будет неудачной:

```
val acc = new ChecksumAccumulator
acc.sum = 5 // Не пройдет компиляцию, поскольку поле sum
            // является приватным
```

Теперь, когда поле `sum` стало приватным, доступ к нему можно получить только из кода, определенного внутри тела самого класса. Следовательно,

класс `ChecksumAccumulator` не будет особо полезен, пока внутри него не будут определены некоторые методы:

```
class ChecksumAccumulator:

  private var sum = 0

  def add(b: Byte): Unit =
    sum += b

  def checksum(): Int =
    return ~(sum & 0xFF) + 1
```

ПРИМЕЧАНИЕ

В Scala элементы класса делают публичными, если нет явного указания какого-либо модификатора доступа. Иначе говоря, там, где в Java ставится модификатор `public`, в Scala вы обходитесь простым замалчиванием. Публичный (`public`) доступ в Scala — уровень доступа по умолчанию.

Теперь у `ChecksumAccumulator` есть два метода: `add` и `checksum`, оба они демонстрируют основную форму определения функции, показанную на рис. 2.1¹.

Внутри этого метода могут использоваться любые параметры метода. Одной из важных характеристик параметров метода в Scala является то, что они относятся к `val`-, а не к `var`-переменным². При попытке повторного присваивания значения параметру внутри метода в Scala произойдет сбой компиляции:

```
def add(b: Byte): Unit =
  b = 1 // Не пройдет компиляцию, поскольку b относится к val-переменным
  sum += b
```

Хотя методы `add` и `checksum` в данной версии `ChecksumAccumulator` реализуют желаемые функциональные свойства вполне корректно, их можно определить в более лаконичном стиле. Во-первых, в конце метода `checksum` можно избавиться от лишнего слова `return`. В отсутствие явно указанной инструкции `return` метод в Scala возвращает последнее вычисленное им значение.

¹ В методе `checksum` используются два оператора: тильда (`~`) для побитового дополнения и амперсанд (`&`) для побитового И. Оба оператора описаны в разделе 5.7.

² Причина, по которой параметры имеют значение `val`, заключается в том, что о `val` легче рассуждать. Вам не нужно ничего дополнительно изучать, как это делается с `var`, чтобы определить, переназначается ли `val`.

При написании методов рекомендуется применять стиль, исключающий явное и особенно многократное использование инструкции `return`. Каждый метод нужно рассматривать в качестве выражения, выдающего одно значение, которое и является возвращаемым. Эта философия будет побуждать вас создавать небольшие методы и разбивать слишком крупные методы на несколько мелких. В то же время выбор конструктивного решения зависит от контекста решаемых задач, и, если того требуют условия, Scala упрощает написание методов, которые имеют несколько явно указанных возвращаемых значений.

Поскольку `checksum` выполняет только вычисление значений, ему не требуется прямая инструкция `return`. Еще одним способом обобщения методов является то, что если метод вычисляет только одно результирующее выражение и оно короткое, его можно поместить в ту же строку, что и сам `def`. Для максимальной краткости вы можете не указывать тип результата, и Scala самостоятельно сделает его вывод. С этими изменениями класс `ChecksumAccumulator` выглядит так:

```
class ChecksumAccumulator:
  private var sum = 0
  def add(b: Byte) = sum += b
  def checksum() = ~(sum & 0xFF) + 1
```

Несмотря на то что компилятор Scala вполне корректно выполнит вывод результирующих типов методов `add` и `checksum`, показанных в предыдущем примере, читатели кода будут вынуждены вывести результирующие типы путем *логических умозаключений* на основе изучения тел методов. Поэтому лучше все-таки будет всегда явно указывать результирующие типы для публичных методов, объявленных в классе, даже когда компилятор может вывести их для вас самостоятельно. Применение этого стиля показано в листинге 4.1.

Листинг 4.1. Окончательная версия класса `ChecksumAccumulator`

```
// Этот код находится в файле ChecksumAccumulator.scala
class ChecksumAccumulator:
  private var sum = 0
  def add(b: Byte): Unit = sum += b
  def checksum(): Int = ~(sum & 0xFF) + 1
```

Методы с результирующим типом `Unit`, к которым относится и метод `add` класса `ChecksumAccumulator`, выполняются для получения побочного эффекта. Последний обычно определяется в виде изменения внешнего по отношению к методу состояния или в виде выполнения какой-либо операции ввода-вывода. Что касается метода `add`, то побочный эффект заключается в присваивании `sum` нового значения. Метод, который выполняется только для получения его побочного эффекта, называется *процедурой*.

4.2. Когда подразумевается использование точки с запятой

В программе на Scala точку с запятой в конце инструкции обычно можно не ставить. Если вся инструкция помещается на одной строке, то при желании можете поставить в конце данной строки точку с запятой, но это не обязательно. В то же время точка с запятой нужна, если на одной строке размещаются сразу несколько инструкций:

```
val s = "hello"; println(s)
```

Если требуется набрать инструкцию, занимающую несколько строк, то в большинстве случаев вы можете просто ее ввести, а Scala разделит инструкции в нужном месте. Например, следующий код рассматривается как одна инструкция, расположенная на четырех строках:

```
if x < 2 then  
  "too small"  
else  
  "ok"
```

Правила расстановки точек с запятой

Правила разделения операторов удивительно просты. Вкратце, точка с запятой всегда обозначает конец строки, кроме случаев, когда не выполняется одно из следующих условий.

1. Рассматриваемая строка заканчивается словом или символом, который недопустим в качестве конца оператора, например точкой или инфиксным оператором.
2. Следующая строка начинается со слова, с которого не может начинаться оператор.
3. Строка заканчивается внутри круглых (...) или квадратных [...] скобок, потому что они не могут содержать несколько операторов.

4.3. Объекты-одиночки

Как упоминалось в главе 1, один из аспектов, позволяющих Scala быть более объектно-ориентированным языком, чем Java, заключается в том, что в классах Scala не могут содержаться статические элементы. Вместо этого

в Scala есть *объекты-одиночки*, или *синглтоны*. Определение объекта-одиночки выглядит так же, как определение класса, за исключением того, что вместо ключевого слова `class` используется ключевое слово `object`. Пример показан в листинге 4.2.

Объект-одиночка в этом листинге называется `ChecksumAccumulator`, то есть носит имя, совпадающее с именем класса в предыдущем примере. Когда объект-одиночка использует общее с классом имя, то для класса он называется *объектом-компаньоном*. И класс, и его объект-компаньон нужно определять в одном и том же исходном файле. Класс по отношению к объекту-одиночке называется *классом-компаньоном*. Класс и его объект-компаньон могут обращаться к приватным элементам друг друга.

Листинг 4.2. Объект-компаньон для класса `ChecksumAccumulator`

```
// Этот код находится в файле ChecksumAccumulator.scala
import scala.collection.mutable

object ChecksumAccumulator:

  private val cache = mutable.Map.empty[String, Int]

  def calculate(s: String): Int =
    if cache.contains(s) then
      cache(s)
    else
      val acc = new ChecksumAccumulator
      for c <- s do
        acc.add((c >> 8).toByte)
        acc.add(c.toByte)
      val cs = acc.checksum()
      cache += (s -> cs)
      cs
```

Объект-одиночка `ChecksumAccumulator` располагает одним методом по имени `calculate`, который получает строку `String` и вычисляет контрольную сумму символов этой строки. Вдобавок он имеет одно приватное поле `cache`, представленное изменяемым отображением, в котором кэшируются ранее вычисленные контрольные суммы¹. В первой строке метода, "`if cache.contains(s)`

¹ Здесь `cache` используется, чтобы показать объект-одиночку с полем. Кэширование с помощью поля `cache` помогает оптимизировать производительность, сокращая за счет расхода памяти время вычисления и разменивая расход памяти на время вычисления. Как правило, использовать кэш-память таким образом целесообразно только в том случае, если с ее помощью можно решить проблемы производительности и воспользоваться отображением со слабыми ссылками, например `WeakHashMap` в `scala.collection.mutable`, чтобы записи в кэш-памяти могли попадать в сборщик мусора при наличии дефицита памяти.

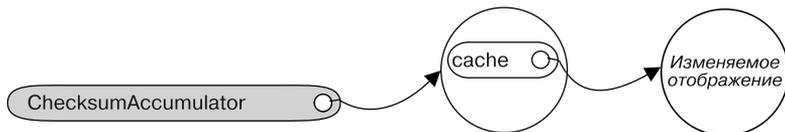
then", определяется, не содержится ли в отображении `cache` переданная строка в качестве ключа. Если да, то просто возвращается отображенное на этот ключ значение `cache(s)`. В противном случае выполняется условие `else`, которое вычисляет контрольную сумму. В первой строке условия `else` определяется `val`-переменная по имени `acc`, которая инициализируется новым экземпляром `ChecksumAccumulator`¹. В следующей строке находится выражение `for`. Оно выполняет последовательный перебор каждого символа в переданной строке, преобразует символ в значение типа `Byte`, вызывая в отношении этого символа метод `toByte`, и передает результат в метод `add` того экземпляра `ChecksumAccumulator`, на который ссылается `acc`². Когда завершится вычисление выражения `for`, в следующей строке метода в отношении `acc` будет вызван метод `checksum`, который берет контрольную сумму для переданного значения типа `String` и сохраняет ее в `val`-переменной по имени `cs`.

В следующей строке, `cache += (s -> cs)`, переданный строковый ключ отображается на целочисленное значение контрольной суммы, и эта пара «ключ — значение» добавляется в отображение `cache`. В последнем выражении метода, `cs`, обеспечивается использование контрольной суммы в качестве результата выполнения метода.

Если у вас есть опыт программирования на Java, то объекты-одиночки можно представить в качестве хранилища для любых статических методов, которые вполне могли быть написаны на Java. Методы в объектах-одиночках можно вызывать с помощью такого синтаксиса: имя объекта, точка, имя метода. Например, метод `calculate` объекта-одиночки `ChecksumAccumulator` можно вызвать следующим образом:

```
ChecksumAccumulator.calculate("Every value is an object.")
```

Но объект-одиночка не только хранилище статических методов. Он объект первого класса. Поэтому имя объекта-одиночки можно рассматривать в качестве «этикетки», прикрепленной к объекту.



¹ Поскольку ключевое слово `new` используется только для создания экземпляров классов, новый объект, созданный здесь в качестве экземпляра класса `ChecksumAccumulator`, не является одноименным объектом-одиночкой.

² Оператор `>>`, выполняющий побитовый сдвиг вправо, описан в разделе 5.7.

Определение объекта-одиночки не является определением типа на том уровне абстракции, который используется в Scala. Имея лишь определение объекта `ChecksumAccumulator`, невозможно создать одноименную переменную типа. Точнее, тип с именем `ChecksumAccumulator` определяется классом-компаньоном объекта-одиночки. Тем не менее объекты-одиночки расширяют суперкласс и могут подмешивать трейты. Учитывая то, что каждый объект-одиночка — экземпляр своего суперкласса и подмешанных в него трейтов, его методы можно вызывать через эти типы, ссылаясь на него из переменных этих типов и передавая ему методы, ожидающие использования этих типов. Примеры объектов-одиночек, являющихся наследниками классов и трейтов, показаны в главе 12.

Одно из отличий классов от объектов-одиночек состоит в том, что объекты-одиночки не могут принимать параметры, а классы — могут. Создать экземпляр объекта-одиночки с помощью ключевого слова `new` нельзя, поэтому передать ему параметры не представляется возможным. Каждый объект-одиночка реализуется как экземпляр *синтетического класса*, ссылка на который находится в статической переменной, поэтому у них и у статических классов Java одинаковая семантика инициализации¹. В частности, объект-одиночка инициализируется при первом обращении к нему какого-либо кода.

Объект-одиночка, который не имеет общего имени с классом-компаньоном, называется *самостоятельным*. Такие объекты можно применять для решения многих задач, включая сбор в одно целое родственных вспомогательных методов или определение точки входа в приложение Scala. Именно этот случай мы и рассмотрим в следующем разделе.

4.4. Case-классы

Часто при написании класса вам потребуется реализация таких методов, как `equals`, `hashCode`, `toString` — методы доступа или фабричные методы. Их написание может занять много времени и привести к ошибкам. Scala предлагает такой инструмент, как *case-классы* (классы-образцы), которые могут генерировать реализации нескольких методов на основе значений, переданных его основному конструктору. Вы создаете класс `case`, помещая модификатор `case` перед `class`, например:

```
case class Person(name: String, age: Int)
```

¹ В качестве имени синтетического класса используется имя объекта со знаком доллара. Следовательно, синтетический класс, применяемый для объекта-одиночки `ChecksumAccumulator`, называется `ChecksumAccumulator$`.

С добавлением модификатора `case` компилятор сгенерирует для вас несколько полезных методов. Во-первых, компилятор создаст объект-компаньон и поместит в него фабричный метод с именем `apply`. Таким образом, вы можете создать новый объект `Person` следующим образом:

```
val p = Person("Sally", 39)
```

Компилятор перепишет эту строку кода в вызов сгенерированного фабричного метода: `Person.apply("Sally", 39)`.

Во-вторых, компилятор будет хранить все параметры класса в полях и генерировать методы доступа с тем же именем, что и у заданного параметра¹.

Например, вы можете получить доступ к заданным в `Person` значениям имени и возраста следующим образом:

```
p.name // Sally  
p.age  // 39
```

В-третьих, компилятор предоставит вам реализацию `toString`:

```
p.toString // Person(Sally,39)
```

В-четвертых, компилятор сгенерирует реализацию `hashCode` и `equals` для вашего класса. Эти методы будут основывать свой результат на параметрах, переданных конструктору. Например, объект `Person` будет учитывать и имя, и возраст при сравнении:

```
p == Person("Sally", 21)           // false  
p.hashCode == Person("Sally", 21).hashCode // false  
p == Person("James", 39)          // false  
p.hashCode == Person("James", 39).hashCode // false  
p == Person("Sally", 39)          // true  
p.hashCode == Person("Sally", 39).hashCode // true
```

Компилятор не будет генерировать метод, который вы реализуете самостоятельно. Он будет использовать вашу реализацию. Вы также можете добавить другие поля и методы к классу и его компаньону. Вот пример, в котором вы определяете метод `apply` в сопутствующем объекте `Person` (компилятор не будет его генерировать) и добавляете метод `appendToName` в класс:

```
case class Person(name: String, age: Int):  
  def appendToName(suffix: String): Person =  
    Person(s"$name$suffix", age)
```

¹ Они называются параметрическими полями, которые будут описаны в разделе 10.6.

```
object Person:
  // Убедитесь, что непустое имя написано с заглавной буквы
  def apply(name: String, age: Int): Person =
    val capitalizedName =
      if !name.isEmpty then
        val firstChar = name.charAt(0).toUpper
        val restOfName = name.substring(1)
        s"$firstChar$restOfName"
      else throw new IllegalArgumentException("Empty name")
    new Person(capitalizedName, age)
```

Этот `apply`-метод гарантирует, что первый символ имени будет начинаться с заглавной буквы:

```
val q = Person("sally", 39) // Person(Sally,39)
```

Вы также можете вызвать метод `appendToName`, который вы определили в классе:

```
q.appendToName(" Smith") // Person(Sally Smith,39)
```

Наконец, компилятор добавляет метод `copy` в ваш класс и метод `unapply` к компаньону. Они будут описаны в главе 13.

Все эти условности облегчают работу, но с небольшой оговоркой: вам всего лишь понадобится написать модификатор `case`, а ваши классы и объекты при этом станут немного больше. Они вырастают, потому что генерируются дополнительные методы и для каждого параметра конструктора добавляется неявное поле.

4.5. Приложение на языке Scala

Чтобы запустить программу на Scala, нужно предоставить имя автономного объекта-одиночки с методом `main`, который получает один параметр с типом `Array[String]` и имеет результирующий тип `Unit`. Точкой входа в приложение может стать любой самостоятельный объект с методом `main`, имеющим надлежащую сигнатуру¹. Пример показан в листинге 4.3.

Листинг 4.3. Приложение Summer

```
// Код находится в файле Summer.scala
import ChecksumAccumulator.calculate
```

¹ Вы можете обозначить методы другими именами в качестве основных функций с помощью `@main`. Этот метод будет описан в разделе 23.3.

```
object Summer:
  def main(args: Array[String]): Unit =
    for arg <- args do
      println(arg + ": " + calculate(arg))
```

Объект-одиночка, показанный в данном листинге, называется `Summer`. Его метод `main` имеет надлежащую сигнатуру, поэтому его можно задействовать в качестве приложения. Первая инструкция в файле импортирует метод `calculate`, который определен в объекте `ChecksumAccumulator` из предыдущего примера. Инструкция `import` позволяет далее использовать в файле простое имя метода¹. Тело метода `main` всего лишь выводит на стандартное устройство каждый аргумент и контрольную сумму для аргумента, разделяя их двоеточием.

ПРИМЕЧАНИЕ

Подразумевается, что в каждый свой исходный файл Scala импортирует элементы пакетов `java.lang` и `scala`, а также элементы объекта-одиночки по имени `Predef`. В `Predef`, который находится в пакете `scala`, содержится множество полезных методов. Например, когда в исходном файле Scala встречается `println`, фактически вызывается `println` из `Predef`. (А метод `Predef.println`, в свою очередь, вызывает метод `Console.println`, который фактически и выполняет всю работу.) Когда же встречается `assert`, вызывается метод `Predef.assert`.

Чтобы запустить приложение `Summer`, поместите код из листинга 4.3 в файл `Summer.scala`. В `Summer` используется `ChecksumAccumulator`, поэтому поместите код для `ChecksumAccumulator` как для класса, показанного в листинге 4.1, так и для его объекта-компаньона, показанного в листинге 4.2, в файл `ChecksumAccumulator.scala`.

Одним из отличий Scala от Java является то, что в Java от вас требуется поместить публичный класс в файл, названный по имени класса, например, класс `SpeedRacer` — в файл `SpeedRacer.java`. А в Scala файл с расширением `.scala` можно называть как угодно независимо от того, какие классы Scala или код в них помещаются. Но обычно, когда речь идет не о скриптах, рекомендуется придерживаться стиля, при котором файлы называются по именам включенных в них классов, как это делается в Java, чтобы программистам было легче искать классы по именам их файлов. Именно этим подходом мы

¹ Наличие опыта программирования на Java позволяет сопоставить такой импорт с объявлением статического импорта, введенным в Java 5. Единственное отличие — в Scala импортировать элементы можно из любого объекта, а не только из объектов-одиночек.

и воспользовались в отношении двух файлов в данном примере. Имеются в виду файлы `Summer.scala` и `ChecksumAccumulator.scala`.

Ни `ChecksumAccumulator.scala`, ни `Summer.scala` не являются скриптами, поскольку заканчиваются определением. В отличие от этого скрипт должен заканчиваться выражением, выдающим результат. Поэтому при попытке запустить `Summer.scala` в качестве скрипта интерпретатор Scala пожалуется на то, что `Summer.scala` не заканчивается выражением, выдающим результат. (Конечно, если предположить, что вы самостоятельно не добавили какое-либо выражение после определения объекта `Summer`.) Вместо этого нужно будет скомпилировать данные файлы с помощью компилятора Scala, а затем запустить получившиеся в результате файлы классов. Для этого можно воспользоваться основным компилятором Scala по имени `scalac`:

```
$ scalac ChecksumAccumulator.scala Summer.scala
```

Эта команда скомпилирует ваши исходные файлы и приведет к созданию файлов классов Java, которые затем можно будет запускать через команду `scala` — ту же самую, с помощью которой вы вызывали интерпретатор в предыдущих примерах. Однако вместо того, чтобы указывать ему имя файла с расширением `.scala`, содержащим код Scala для интерпретации (как вы делали в каждом предыдущем примере)¹, вы дадите ему имя отдельного объекта, содержащего метод `main` с соответствующей сигнатурой. Следовательно, приложение `Summer` можно запустить, набрав команду:

```
$ scala Summer of love
```

Вы сможете увидеть контрольные суммы, выведенные для двух аргументов командной строки:

```
of: -213  
love: -182
```

Резюме

В этой главе мы рассмотрели основы классов и объектов в Scala и показали приемы компиляции и запуска приложений. В следующей главе рассмотрим основные типы данных и варианты их использования.

¹ Фактический механизм, который программа Scala использует для «интерпретации» исходного файла Scala, заключается в том, что она компилирует исходный код Scala в байт-коды Java, немедленно загружает их через загрузчик классов и выполняет их.

5

Основные типы и операции

После того как были рассмотрены в действии классы и объекты, самое время поглубже изучить имеющиеся в Scala основные типы и операции. Если вы хорошо знакомы с Java, то вас может обрадовать тот факт, что в Scala и в Java основные типы и операторы имеют тот же смысл. И все же есть интересные различия, ради которых с этой главой стоит ознакомиться даже тем, кто считает себя опытным разработчиком Java-приложений. Некоторые аспекты Scala, рассматриваемые в данной главе, в основном такие же, как и в Java, поэтому мы указываем, какие разделы Java-разработчики могут пропустить.

В текущей главе мы представим обзор основных типов Scala, включая строки типа `String` и типы значений `Int`, `Long`, `Short`, `Byte`, `Float`, `Double`, `Char` и `Boolean`. Кроме того, рассмотрим операции, которые могут выполняться с этими типами, и вопросы соблюдения приоритета операторов в выражениях Scala. Поговорим мы и о том, как Scala «обогащает» варианты основных типов, позволяя выполнять дополнительные операции вдобавок к тем, что поддерживаются в Java.

5.1. Некоторые основные типы

В табл. 5.1 показан ряд основных типов, используемых в Scala, а также диапазоны значений, которые могут принимать их экземпляры. В совокупности типы `Byte`, `Short`, `Int`, `Long` и `Char` называются *целочисленными*. Целочисленные типы плюс `Float` и `Double` называются *числовыми*.

Таблица 5.1. Некоторые основные типы

Основной тип	Диапазон
Byte	8-битовое знаковое целое число в дополнительном коде (от -2^7 до $2^7 - 1$ включительно)
Short	16-битовое знаковое целое число в дополнительном коде (от -2^{15} до $2^{15} - 1$ включительно)
Int	32-битовое знаковое целое число в дополнительном коде (от -2^{31} до $2^{31} - 1$ включительно)
Long	64-битовое знаковое целое число в дополнительном коде (от -2^{63} до $2^{63} - 1$ включительно)
Char	16-битовый беззнаковый Unicode-символ (от 0 до $2^{16} - 1$ включительно)
String	Последовательность из Char
Float	32-битовое число с плавающей точкой одинарной точности, которое соответствует стандарту IEEE 754
Double	64-битовое число с плавающей точкой двойной точности, которое соответствует стандарту IEEE 754
Boolean	true или false

За исключением типа `String`, который находится в пакете `java.lang`, все типы, показанные в данной таблице, входят в пакет `scala`¹. Например, полное имя типа `Int` обозначается `scala.Int`. Но, учитывая, что все элементы пакета `scala` и `java.lang` автоматически импортируются в каждый исходный файл Scala, можно повсеместно использовать только простые имена, то есть имена вида `Boolean`, `Char` или `String`.

Опытные Java-разработчики заметят, что основные типы Scala имеют в точности такие же диапазоны, как и соответствующие им типы в Java. Это позволяет компилятору Scala в создаваемом им байт-коде преобразовывать экземпляры *типов значений* Scala, например `Int` или `Double`, в примитивные типы Java.

5.2. Литералы

Все основные типы, перечисленные в табл. 5.1, можно записать с помощью *литералов*. Литерал представляет собой способ записи постоянного значения непосредственно в коде.

¹ Пакеты, кратко рассмотренные в шаге 1 главы 2, более подробно рассматриваются в главе 12.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ ДЛЯ JAVA-ПРОГРАММИСТОВ

Синтаксис большинства литералов, показанных в данном разделе, совпадает с синтаксисом, применяемым в Java, поэтому знатоки Java могут спокойно пропустить практически весь раздел. Отдельные различия, о которых стоит прочитать, касаются используемых в Scala неформатированных строк (рассматриваются в подразделе «Строковые литералы»), а также интерполяции строк. Кроме того, в Scala не поддерживаются восьмеричные литералы, а целочисленные, начинающиеся с нуля, например 031, не проходят компиляцию.

Целочисленные литералы

Целочисленные литералы для типов `Int`, `Long`, `Short` и `Byte` используются в двух видах: десятичном и шестнадцатеричном. Способ, применяемый для начала записи целочисленного литерала, показывает основание числа. Если число начинается с `0x` или `0X`, то оно шестнадцатеричное (по основанию 16) и может содержать цифры от 0 до 9, а также буквы от А до F в верхнем или нижнем регистре. Вы можете использовать символы подчеркивания (`_`), чтобы улучшить читаемость больших значений, например:

```
val hex = 0x5           // 5: Int
val hex2 = 0x00FF      // 255: Int
val magic = 0xcafebabe // -889275714: Int
val billion = 1_000_000_000 // 1000000000: Int
```

Обратите внимание на то, что оболочка Scala REPL всегда выводит целочисленные значения в десятичном виде независимо от формы литерала, которую вы могли задействовать для инициализации этих значений. Таким образом, REPL показывает значение переменной `hex2`, которая была инициализирована с помощью литерала `0x00FF`, как десятичное число 255. (Разумеется, не нужно все принимать на веру. Хорошим способом начать осваивать язык станет практическая работа с этими инструкциями в интерпретаторе по мере чтения данной главы.) Если цифра, с которой начинается число, не ноль и не имеет никаких других знаков отличия, значит, число десятичное (по основанию 10), например:

```
val dec1 = 31 // 31: Int
val dec2 = 255 // 255: Int
val dec3 = 20 // 20: Int
```

Если целочисленный литерал заканчивается на `L` или `l`, значит, показывает число типа `Long`, в противном случае это число относится к типу `Int`. Посмотрите на примеры целочисленных литералов `Long`:

```
val prog = 0XCAFEBABEL // 3405691582: Long
val tower = 35L        // 35: Long
val of = 311           // 31: Long
```

Если `Int`-литерал присваивается переменной типа `Short` или `Byte`, то рассматривается как принадлежащий к типу `Short` или `Byte`, если, конечно, его значение находится внутри диапазона, допустимого для данного типа, например:

```
val little: Short = 367 // 367: Short
val littler: Byte = 38  // 38: Byte
```

Литералы чисел с плавающей точкой

Литералы чисел с плавающей точкой состоят из десятичных цифр, которые также могут содержать необязательный символ десятичной точки, и после них может стоять необязательный символ `E` или `e` и экспонента. Посмотрите на примеры литералов чисел с плавающей точкой:

```
val big = 1.2345           // 1.2345: Double
val bigger = 1.2345e1     // 12.345: Double
val biggerStill = 123E45  // 1.23E47: Double
val trillion = 1_000_000_000e3 // 1.0E12: Double
```

Обратите внимание: экспонента означает степень числа `10`, на которую умножается остальная часть числа. Следовательно, `1.2345e1` равняется числу `1,2345`, *умноженному* на `10`, то есть получается число `12,345`. Если литерал числа с плавающей точкой заканчивается на `F` или `f`, значит, число относится к типу `Float`, в противном случае оно относится к типу `Double`. Дополнительно литералы чисел с плавающей точкой могут заканчиваться на `D` или `d`. Посмотрите на примеры литералов чисел с плавающей точкой:

```
val little = 1.2345F      // 1.2345: Float
val littleBigger = 3e5f  // 300000.0: Float
```

Последнее значение, выраженное как тип `Double`, может также принимать иную форму:

```
val anotherDouble = 3e5 // 300000.0: Double
val yetAnother = 3e5D  // 300000.0: Double
```

Большие числовые литералы

В Scala 3 добавлена экспериментальная функция, которая устраняет ограничения на размер числовых литералов и позволяет использовать их для

инициализации произвольных типов. Вы можете включить эту функцию с помощью импорта этого языка:

```
import scala.language.experimental.genericNumberLiterals
```

Вот два примера из стандартной библиотеки:

```
val invoice: BigInt = 1_000_000_000_000_000_000_000
val pi: BigDecimal = 3.1415926535897932384626433833
```

Символьные литералы

Символьные литералы состоят из любого Unicode-символа, заключенного в одинарные кавычки:

```
scala> val a = 'A'
val a: Char = A
```

Помимо того что символ представляется в одинарных кавычках в явном виде, его можно указывать с помощью кода из таблицы символов Unicode. Для этого нужно записать `\u`, после чего указать четыре шестнадцатеричные цифры кода:

```
scala> val d = '\u0041'
val d: Char = A
scala> val f = '\u0044'
val f: Char = D
```

Такие символы в кодировке Unicode могут появляться в любом месте программы на языке Scala. Например, вы можете набрать следующий идентификатор:

```
scala> val B\u0041\u0044 = 1
val BAD: Int = 1
```

Он рассматривается точно так же, как идентификатор `BAD`, являющийся результатом раскрытия символов в кодировке Unicode в показанном ранее коде. По сути, в именовании идентификаторов подобным образом нет ничего хорошего, поскольку их трудно прочесть. Иногда с помощью этого синтаксиса исходные файлы Scala, которые содержат отсутствующие в таблице ASCII символы из таблицы Unicode, можно представить в кодировке ASCII.

И наконец, нужно упомянуть о нескольких символьных литералах, представленных специальными управляющими последовательностями (escape sequences), показанными в табл. 5.2, например:

```
scala> val backslash = '\\'
val backslash: Char = \
```

Таблица 5.2. Управляющие последовательности специальных символьных литералов

Литерал	Предназначение
<code>\n</code>	Перевод строки (<code>\u000A</code>)
<code>\b</code>	Возврат на одну позицию (<code>\u0008</code>)
<code>\t</code>	Табуляция (<code>\u0009</code>)
<code>\f</code>	Перевод страницы (<code>\u000C</code>)
<code>\r</code>	Возврат каретки (<code>\u000D</code>)
<code>\"</code>	Двойные кавычки (<code>\u0022</code>)
<code>\'</code>	Одинарная кавычка (<code>\u0027</code>)
<code>\\</code>	Обратный слеш (<code>\u005C</code>)

Строковые литералы

Строковый литерал состоит из символов, заключенных в двойные кавычки:

```
scala> val hello = "hello"
val hello: String = hello
```

Синтаксис символов внутри кавычек такой же, как и в символьных литералах, например:

```
scala> val escapes = "\\\"\'"
val escapes: String = "\"'
```

Данный синтаксис неудобен для строк, в которых содержится множество управляющих последовательностей, или для строк, не уместяющихся в одну строку текста, поэтому для *неформатированных строк* в Scala включен специальный синтаксис. Неформатированная строка начинается и заканчивается тремя идущими подряд двойными кавычками ("""). Внутри нее могут содержаться любые символы, включая символы новой строки, кавычки и специальные символы, за исключением, разумеется, трех кавычек подряд. Например, следующая программа выводит сообщение, используя неформатированную строку:

```
println("""Welcome to Ultamix 3000.
        Type "HELP" for help.""")
```

Но при запуске этого кода получается не совсем то, что хотелось:

```
Welcome to Ultamix 3000.
        Type "HELP" for help.
```

Проблема во включении в строку пробелов перед второй строкой текста! Чтобы справиться с этой весьма часто возникающей ситуацией, вы можете вызывать в отношении строк метод `stripMargin`. Чтобы им воспользоваться, поставьте символ вертикальной черты (|) перед каждой строкой текста, а затем в отношении всей строки вызовите метод `stripMargin`:

```
println("""|Welcome to Ultamix 3000.  
         |Type "HELP" for help.""").stripMargin)
```

Вот теперь код ведет себя подобающим образом:

```
Welcome to Ultamix 3000.  
Type "HELP" for help.
```

Булевы литералы

У типа `Boolean` имеется два литерала, `true` и `false`:

```
val bool = true // true: Boolean  
val fool = false // false: Boolean
```

Вот, собственно, и все. Теперь вы буквально (или литерально) стали большим специалистом по Scala.

5.3. Интерполяция строк

В Scala включен довольно гибкий механизм для интерполяции строк, позволяющий вставлять выражения в строковые литералы. В самом распространенном случае использования этот механизм предоставляет лаконичную и удобочитаемую альтернативу конкатенации строк. Рассмотрим пример:

```
val name = "reader"  
println(s"Hello, $name!")
```

Выражение `s"Hello, $name!"` — *обрабатываемый* строковый литерал. Поскольку за буквой `s` стоят открывающие кавычки, то Scala для обработки литерала воспользуется *интерполятором строк* `s`. Он станет вычислять каждое встроенное выражение, вызывая в отношении каждого результата метод `toString` и заменяя встроенные выражения в литерале этими результатами. Таким образом, из `s"Hello, $name!"` получится `"Hello, reader!"`,

то есть точно такой же результат, как при использовании кода "Hello, " + name + "!".

После знака доллара (\$) в обрабатываемом строковом литерале можно указать любое выражение. Для выражений с одной переменной зачастую можно просто поместить после знака доллара имя этой переменной. Все символы, вплоть до первого символа, не относящегося к идентификатору, Scala будет интерпретировать как выражение. Если в него включены символы, не являющиеся идентификаторами, то это выражение следует заключить в фигурные скобки, а открывающая фигурная скобка должна ставиться сразу же после знака доллара, например:

```
scala> s"The answer is ${6 * 7}."
val res0: String = The answer is 42.
```

Scala содержит еще два интерполятора строк: `raw` и `f`. Интерполятор строк `raw` ведет себя практически так же, как и `s`, за исключением того, что не распознает управляющие последовательности символьных литералов (те самые, которые показаны в табл. 5.2). Например, следующая инструкция выводит четыре, а не два обратных слеша:

```
println(raw"No\\\\\escape!") // выводит: No\\\\\escape!
```

Интерполятор строк `f` позволяет прикреплять к встроенным выражениям инструкции форматирования в стиле функции `printf`. Инструкции ставятся после выражения и начинаются со знака процента (%), при этом используется синтаксис, заданный классом `java.util.Formatter`. Например, вот как можно было бы отформатировать число π :

```
scala> f"${math.Pi}%.5f"
val res1: String = 3.14159
```

Если для встроенного выражения не указать никаких инструкций форматирования, то интерполятор строк `f` по умолчанию превратится в `%s`, что означает подстановку значения, полученного в результате выполнения метода `toString`, точно так же, как это делает интерполятор строк `s`, например:

```
scala> val pi = "Pi"
val pi: String = Pi

scala> f"$pi is approximately ${math.Pi}%.8f."
val res2: String = Pi is approximately 3.14159265.
```

В Scala интерполяция строк реализуется перезаписью кода в ходе компиляции. Компилятор в качестве выражения интерполятора строк будет

рассматривать любое выражение, состоящее из идентификатора, за которым сразу же стоят открывающие двойные кавычки строкового литерала. Интерполяторы строк `s`, `f` и `raw` реализуются с помощью этого общего механизма. Библиотеки и пользователи могут определять другие интерполяторы строк, применяемые в иных целях.

5.4. Все операторы являются методами

Для основных типов Scala предоставляет весьма богатый набор операторов. Как упоминалось в предыдущих главах, эти операторы — всего лишь приятный синтаксис для обычных вызовов методов. Например, `1 + 2` означает то же самое, что и `1.+(2)`. Иными словами, в классе `Int` имеется метод по имени `+`, который получает `Int`-значение и возвращает `Int`-результат. Он вызывается при сложении двух `Int`-значений:

```
val sum = 1 + 2 // Scala вызывает 1.+(2)
```

Чтобы убедиться в этом, можете набрать выражение, в точности соответствующее вызову метода:

```
scala> val sumMore = 1.+(2)
val sumMore: Int = 3
```

Фактически в классе `Int` содержится несколько *перегруженных* методов `+`, получающих различные типы параметров¹. Например, у `Int` есть еще один метод, тоже по имени `+`, который получает и возвращает значения типа `Long`. При сложении `Long` и `Int` будет вызван именно этот альтернативный метод:

```
scala> val longSum = 1 + 2L // Scala вызывает 1.+(2L)
val longSum: Long = 3
```

Символ `+` — оператор, точнее, инфиксный оператор. Форма записи операторов не ограничивается методами, подобными `+`, которые в других языках выглядят как операторы. *Любой* метод может использоваться в нотации операторов, если он принимает только один параметр². Например, в классе `String` есть метод `indexOf`, получающий один параметр типа `Char`. Метод `indexOf` ведет поиск первого появления в строке указанного символа и воз-

¹ Перегруженные методы имеют точно такие же имена, но используют другие типы аргументов. Более подробно перегрузка методов рассматривается в разделе 6.11.

² В будущих версиях Scala методы с несимвольными именами будут разрешены в качестве операторов только в том случае, если они объявлены с модификатором `infix`.

вращает его индекс или `-1`, если символ найден не будет. Метод `indexOf` можно использовать как оператор:

```
scala> val s = "Hello, world!"  
val s: String = Hello, world!  
  
scala> s indexOf 'o' // Scala вызывает s.indexOf('o')  
val res0: Int = 4
```

Любой однопараметрический метод может быть оператором

В Scala операторы не относятся к специальному синтаксису языка. Любой метод, который содержит один параметр, может быть оператором. Однопараметрический метод становится оператором в зависимости от того, как вы его *используете*. Если вы напишете `s.indexOf('o')`, то `indexOf` не будет являться оператором, но станет им, если запись будет иметь вид формы оператора — `s indexOf 'o'`.

До сих пор рассматривались только примеры *инфиксной* формы записи операторов, означающей, что вызываемый метод находится между объектом и параметром или параметрами, которые нужно передать методу, как в выражении `7 + 2`. В Scala также имеются две другие формы записи операторов: префиксная и постфиксная. В префиксной форме записи имя метода ставится перед объектом, в отношении которого вызывается этот метод (например, `-` в выражении `-7`). В постфиксной форме имя метода ставится после объекта (например, `toLong` в выражении `7 toLong`).

В отличие от инфиксной формы записи, в которой операторы получают два операнда (один слева, другой справа), префиксные и постфиксные операторы являются *унарными* — получают только один операнд. В префиксной форме записи операнд размещается справа от оператора. В качестве примеров можно привести выражения `-2.0`, `!found` и `~0xFF`. Как и в случае использования инфиксных операторов, эти префиксные операторы являются сокращенной формой вызова методов. Но в данном случае перед символом оператора в имени метода ставится приставка `unary_`. Например, Scala превратит выражение `-2.0` в вызов метода `(2.0).unary_-`. Вы можете убедиться в этом, набрав вызов метода как с использованием формы записи операторов, так и в явном виде:

```
scala> -2.0 // Scala вызывает (2.0).unary_-  
val res2: Double = -2.0  
  
scala> (2.0).unary_-  
val res3: Double = -2.0
```

Идентификаторами, которые могут служить в качестве префиксных операторов, являются только +, -, ! и ~. Следовательно, если вы определите метод по имени unary_!, то сможете вызвать его в отношении значения или переменной подходящего типа, прибегнув к префиксной форме записи операторов, например !p. Но, определив метод по имени unary_*, вы не сможете использовать префиксную форму записи операторов, поскольку * не входит в число четырех идентификаторов, которые могут использоваться в качестве префиксных операторов. Метод можно вызвать обычным способом как p.unary_*, но при попытке вызвать его в виде *p Scala воспримет код так, словно он записан в виде *.p, что, вероятно, совершенно не совпадает с задуманным¹!

Постфиксные операторы, будучи вызванными без точки или круглых скобок, являются методами, не получающими аргументов. В Scala при вызове метода пустые круглые скобки можно не ставить. Соглашение гласит, что круглые скобки ставятся, если метод имеет побочные эффекты, как в случае с методом println(). Но их можно не ставить, если метод не имеет побочных эффектов, как в случае с методом toLowerCase, вызываемым в отношении значения типа String:

```
scala> val s = "Hello, world!"  
val s: String = Hello, world!  
  
scala> s.toLowerCase  
val res4: String = hello, world!
```

В последнем случае, где методу не требуются аргументы, можно при желании не ставить точку и воспользоваться постфиксной формой записи операторов. Однако компилятор потребует, чтобы вы импортировали scala.language.postfixOps, прежде чем вызывать метод:

```
scala> import scala.language.postfixOps  
  
scala> s toLowerCase  
val res5: String = hello, world!
```

Здесь метод toLowerCase используется в качестве постфиксного оператора в отношении операнда s.

Чтобы понять, какие операторы можно использовать с основными типами Scala, нужно посмотреть на методы, объявленные в классах типов, в документации по Scala API. Но данная книга — пособие по языку Scala, поэтому в нескольких следующих разделах будет представлен краткий обзор большинства этих методов.

¹ Однако не обязательно все будет потеряно. Есть весьма незначительная вероятность того, что программа с кодом *p может скомпилироваться как код C++.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ ДЛЯ JAVA-ПРОГРАММИСТОВ

Многие аспекты Scala, рассматриваемые в оставшейся части главы, совпадают с аналогичными Java-аспектами. Если вы хорошо разбираетесь в Java и у вас мало времени, то можете спокойно перейти к разделу 5.8, в котором рассматриваются отличия Scala от Java в области равенства объектов.

5.5. Арифметические операции

Арифметические методы при работе с любыми числовыми типами можно вызвать в инфиксной форме для сложения (+), вычитания (-), умножения (*), деления (/) и получения остатка от деления (%). Вот несколько примеров:

```
1.2 + 2.3    // 3.5: Double
3 - 1        // 2: Int
'b' - 'a'    // 1: Int
2L * 3L      // 6: Long
11 / 4       // 2: Int
11 % 4       // 3: Int
11.0f / 4.0f // 2.75: Float
11.0 % 4.0   // 3.0: Double
```

Когда целочисленными типами являются как правый, так и левый операнды (Int, Long, Byte, Short или Char), оператор / выведет всю числовую часть результата деления, исключая остаток. Оператор % показывает остаток от предполагаемого целочисленного деления.

Остаток от деления числа с плавающей точкой, полученный с помощью метода %, не определен в стандарте IEEE 754. Что касается операции вычисления остатка, в этом стандарте используется деление с округлением, а не деление с отбрасыванием остатка. Поэтому данная операция сильно отличается от операции вычисления остатка от целочисленного деления. Если все-таки нужно получить остаток по стандарту IEEE 754, то можно вызвать метод IEEEremainder из scala.math:

```
math.IEEEremainder(11.0, 4.0) // -1.0: Double
```

Числовые типы также предлагают прибегнуть к унарным префиксным операторам + (метод unary_+) и - (метод unary_-), позволяющим показать положительное или отрицательное значение числового литерала, как в -3 или +4.0. Если не указать унарный + или -, то литерал интерпретируется как положительный. Унарный + существует исключительно для симметрии с унарным -, однако не производит никаких действий. Унарный - может

также использоваться для смены знака переменной. Вот несколько примеров:

```
val neg = 1 + -3 // -2 : Neg
val y = +3      // 3: Int
-neg           // 2: Int
```

5.6. Отношения и логические операции

Числовые типы можно сравнивать с помощью методов отношений «больше» (>), «меньше» (<), «больше или равно» (>=) и «меньше или равно» (<=), которые выдают в качестве результата булево значение. Дополнительно, чтобы инвертировать булево значение, можно использовать унарный оператор ! (метод `unary_!`). Вот несколько примеров:

```
1 > 2           // false: Boolean
1 < 2           // true: Boolean
1.0 <= 1.0     // true: Boolean
3.5f >= 3.6f   // false: Boolean
'a' >= 'A'     // true: Boolean
val untrue = !true // false: Boolean
```

Методы «логическое И» (&& и &) и «логическое ИЛИ» (|| и |) получают операнды типа `Boolean` в инфиксной нотации и выдают результат в виде `Boolean`-значения, например:

```
val toBe = true           // true: Boolean
val question = toBe || !toBe // true: Boolean
val paradox = toBe && !toBe // false: Boolean
```

Операции && и ||, как и в Java, — *сокращенно вычисляемые*: выражения, построенные с помощью этих операторов, вычисляются, только когда это нужно для определения результата. Иными словами, правая часть выражений с использованием && и || не будет вычисляться, если результат уже определен при вычислении левой части. Например, если левая часть выражения с методом && вычисляется в `false`, то результатом выражения, несомненно, будет `false`, поэтому правая часть не вычисляется. Аналогично этому если левая часть выражения с методом || вычисляется в `true`, то результатом выражения конечно же будет `true`, поэтому правая часть не вычисляется:

```
scala> def salt() = { println("salt"); false }
def salt(): Boolean

scala> def pepper() = { println("pepper"); true }
def pepper(): Boolean
```

```
scala> pepper() && salt()
pepper
salt
val res21: Boolean = false
```

```
scala> salt() && pepper()
salt
val res22: Boolean = false
```

В первом выражении вызываются `pepper` и `salt`, но во втором вызывается только `salt`. Поскольку `salt` возвращает `false`, то необходимость в вызове `pepper` отпадает.

Если правую часть нужно вычислить при любых условиях, то вместо показанных выше методов следует обратиться к методам `&` и `|`. Первый выполняет логическую операцию И, а второй — операцию ИЛИ, но при этом они не прибегают к сокращенному вычислению, как это делают методы `&&` и `||`. Вот как выглядит пример их использования:

```
scala> salt() & pepper()
salt
pepper
val res23: Boolean = false
```

ПРИМЕЧАНИЕ

Возникает вопрос: как сокращенное вычисление может работать, если операторы — всего лишь методы? Обычно все аргументы вычисляются перед входом в метод, тогда каким же образом метод может избежать вычисления своего второго аргумента? Дело в том, что у всех методов Scala есть средство для задержки вычисления его аргументов или даже полной его отмены. Оно называется «параметр, передаваемый по имени» и будет рассмотрено в разделе 9.5.

5.7. Поразрядные операции

Scala позволяет выполнять операции над отдельными разрядами целочисленных типов, используя несколько поразрядных методов. К таким методам относятся поразрядное И (`&`), поразрядное ИЛИ (`|`) и поразрядное исключающее ИЛИ (`^`)¹. Унарный поразрядный оператор дополнения (`~`, метод `unary_~`) инвертирует каждый разряд в своем операнде, например:

¹ Метод поразрядного исключающего ИЛИ выполняет соответствующую операцию в отношении своих операндов. Из одинаковых разрядов получается 0, а из разных -1. Следовательно, выражение `0011 ^ 0101` вычисляется в `0110`.

```

1 & 2 // 0: Int
1 | 2 // 3: Int
1 ^ 3 // 2: Int
~1    // -2: Int

```

В первом выражении, `1 & 2`, выполняется поразрядное И над каждым разрядом чисел `1 (0001)` и `2 (0010)` и выдается результат `0 (0000)`. Во втором выражении, `1 | 2`, выполняется поразрядное ИЛИ над теми же операндами и выдается результат `3 (0011)`. В третьем выражении, `1 ^ 3`, выполняется поразрядное исключающее ИЛИ над каждым разрядом `1 (0001)` и `3 (0011)` и выдается результат `2 (0010)`. В последнем выражении, `~1`, инвертируется каждый разряд в `1 (0001)` и выдается результат `-2`, который в двоичной форме выглядит как `11111111111111111111111111111110`.

Целочисленные типы Scala также предлагают три метода сдвига: влево (`<<`), вправо (`>>`) и беззнаковый сдвиг вправо (`>>>`). Методы сдвига, примененные в инфиксной форме записи операторов, сдвигают разряды целочисленного значения, указанные слева от оператора, на количество разрядов, указанное в целочисленном значении справа от оператора. При сдвиге влево и беззнаковом сдвиге вправо разряды по мере сдвига заполняются нулями. При сдвиге вправо разряды указанного слева значения по мере сдвига заполняются значением самого старшего разряда (разряда знака). Вот несколько примеров:

```

-1 >> 31 // -1: Int
-1 >>> 31 // 1: Int
1 << 2    // 4: Int

```

Число `-1` в двоичном виде выглядит как `11111111111111111111111111111111`. В первом примере, `-1 >> 31`, в числе `-1` происходит сдвиг вправо на 31 разрядную позицию. В значении типа `Int` содержатся 32 разряда, поэтому данная операция, по сути, перемещает самый левый разряд до тех пор, пока тот не станет самым правым¹. Поскольку метод `>>` выполняет заполнение по мере сдвига единицами ввиду того, что самый левый разряд числа `-1` — это `1`, результат получается идентичным исходному левому операнду и состоит из 32 единичных разрядов, или равняется `-1`. Во втором примере, `-1 >>> 31`, самый левый разряд опять сдвигается вправо в самую правую позицию, однако на сей раз освобождающиеся разряды заполняются нулями. Поэтому результат в двоичном виде получается `00000000000000000000000000000001`, или `1`. В последнем примере, `1 << 2`, левый операнд, `1`, сдвигается влево на две позиции (освобождающиеся позиции заполняются нулями), в результате

¹ Самый левый разряд в целочисленном типе является знаковым. Если самый левый разряд установлен в `1`, значит, число отрицательное, если в `0`, то положительное.

содержимого. Например, вот как сравниваются две строки, в которых по пять одинаковых букв:

```
("he" + "llo") == "hello" // true: Boolean
```

Различия операторов == в Scala и Java

В Java оператор == может использоваться для сравнения как примитивных, так и ссылочных типов. В отношении примитивных типов оператор == в Java проверяет равенство значений, как и в Scala. Но в отношении ссылочных типов оператор == в Java проверяет *равенство ссылок*. Это значит, две переменные указывают на один и тот же объект в куче, принадлежащей JVM. Scala также предоставляет средство eq для сравнения равенства ссылок. Но метод eq и его противоположность, метод ne, применяются только к объектам, которые непосредственно отображаются на объекты Java. Исчерпывающие подробности о eq и ne приводятся в разделах 17.1 и 17.2. Кроме того, в главе 8 показано, как создавать хорошие методы equals.

5.9. Приоритет и ассоциативность операторов

Приоритет операторов определяет, какая часть выражения вычисляется самой первой. Например, выражение $2 + 2 * 7$ вычисляется в 16, а не в 28, поскольку оператор * имеет более высокий приоритет, чем оператор +. Поэтому та часть выражения, в которой требуется перемножить числа, вычисляется до того, как будет выполнена часть, в которой числа складываются. Разумеется, чтобы уточнить в выражении порядок вычисления или переопределить приоритеты, можно воспользоваться круглыми скобками. Например, если вы действительно хотите, чтобы результат вычисления ранее показанного выражения был 28, то можете набрать следующее выражение:

```
(2 + 2) * 7
```

Если учесть, что в Scala, по сути, нет операторов, а есть только способ применения методов в форме записи операторов, то возникает вопрос: а как тогда работает приоритет операторов? Scala принимает решение о приоритете на основе первого символа метода, использованного в форме записи операторов (из этого правила есть одно исключение, рассматриваемое ниже). Если имя метода начинается, к примеру, с *, то он получит более высокий приоритет,

чем метод, чье имя начинается на +. Следовательно, выражение $2 + 2 * 7$ будет вычислено как $2 + (2 * 7)$. Аналогично этому выражение $a +++ b *** c$, в котором a , b и c — переменные, $a +++$ и $b ***$ — методы, будет вычислено как $a +++ (b *** c)$, поскольку метод $***$ обладает более высоким уровнем приоритета, чем метод $+++$.

В табл. 5.3 показан приоритет применительно к первому символу метода в убывающем порядке, где символы, показанные на одной строке, определяют одинаковый уровень приоритета. Чем выше символ в списке, тем выше приоритет начинающегося с него метода. Вот пример, показывающий влияние приоритета:

```
2 << 2 + 2 // 32: Int
```

Таблица 5.3. Приоритет операторов

(Все специальные символы)
* / %
+ -
:
= !
< >
&
^
(Все буквы)
(Все операторы присваивания)

Имя метода $<<$ начинается с символа $<$, который появляется в приведенном списке ниже символа $+$ — первого и единственного символа метода $+$. Следовательно, $<<$ будет иметь более низкий уровень приоритета, чем $+$, и выражение будет вычислено путем вызова сначала метода $+$, а затем метода $<<$, как в выражении $2 << (2 + 2)$. При сложении $2 + 2$ в результате математического действия получается 4, а вычисление выражения $2 << 4$ дает результат 32. Если поменять операторы местами, то будет получен другой результат:

```
2 + 2 << 2 // 16: Int
```

Поскольку первые символы, по сравнению с предыдущим примером, не изменились, то методы будут вызваны в том же порядке: $+$, а затем $<<$. Следовательно, $2 + 2$ опять будет равен 4, а $4 << 2$ даст результат 16.

Единственное исключение из правил, о существовании которого уже говорилось, относится к *операторам присваивания*, заканчивающимся знаком равенства. Если оператор заканчивается знаком равенства (=) и не относится к одному из операторов сравнения <=, >=, == и !=, то приоритет оператора имеет такой же уровень, что и простое присваивание (=). То есть он ниже приоритета любого другого оператора. Например:

```
x *= y + 1
```

означает то же самое, что и

```
x *= (y + 1)
```

поскольку оператор *= классифицируется как оператор присваивания, приоритет которого ниже, чем у +, даже притом что первым символом оператора выступает знак *, который обозначил бы приоритет выше, чем у +.

Если в выражении рядом появляются операторы с одинаковым уровнем приоритета, то способ группировки операторов определяется их *ассоциативностью*. Ассоциативность оператора в Scala определяется по его последнему символу. Как уже упоминалось в главе 3, любой метод, имя которого заканчивается символом :, вызывается в отношении своего правого операнда с передачей ему левого. Методы, в окончании имени которых используются любые другие символы, действуют наоборот: они вызываются в отношении своего левого операнда с передачей себе правого. То есть из выражения `a * b` получается `a.*(b)`, но из `a ::: b` получается `b.:::(a)`.

Но независимо от того, какова ассоциативность оператора, его операнды всегда вычисляются слева направо. Следовательно, если `a` — выражение, не являющееся простой ссылкой на неизменяемое значение, то выражение `a ::: b` при более точном рассмотрении представляется следующим блоком:

```
{ val x = a; b.:::(x) }
```

В этом блоке `a` по-прежнему вычисляется раньше `b`, а затем результат данного вычисления передается в качестве операнда принадлежащему `b` методу `:::`.

Это правило ассоциативности играет роль также при появлении в одном выражении рядом сразу нескольких операторов с одинаковым уровнем приоритета. Если имена методов заканчиваются на :, они группируются справа налево, в противном случае — слева направо. Например, `a ::: b ::: c` рассматривается как `a ::: (b ::: c)`. Но `a * b * c`, в отличие от этого, рассматривается как `(a * b) * c`.

Правила приоритета операторов — часть языка Scala, и вам не следует бояться применять ими. При этом, чтобы прояснить первоочередность использования операторов, в некоторых выражениях все же лучше прибегнуть к круглым скобкам. Пожалуй, единственное, на что можно реально рассчитывать в отношении знания порядка приоритета другими программистами, — то, что мультипликативные операторы `*`, `/` и `%` имеют более высокий уровень приоритета, чем аддитивные `+` и `-`. Таким образом, даже если выражение `a + b << c` выдает нужный результат и без круглых скобок, стоит внести дополнительную ясность с помощью записи `(a + b) << c`. Это снизит количество нелестных отзывов ваших коллег по поводу использованной вами формы записи операторов, которое выражается, к примеру, в недовольном восклицании вроде «Опять в его коде невозможно разобраться!» и отправке вам сообщения наподобие `bills !*%^~ code!`¹.

5.10. Обогащающие операции

В отношении основных типов Scala можно вызвать намного больше методов, чем рассмотрено в предыдущих разделах. Некоторые примеры показаны в табл. 5.4. Начиная со Scala 3, эти методы доступны через неявные преобразования — устаревшую технику, которая в конечном итоге будет заменена методами расширения, подробно описанными в главе 22. А пока вам нужно знать лишь то, что для каждого основного типа, рассмотренного в текущей

Таблица 5.4. Некоторые обогащающие операции

Код	Результат
<code>0 max 5</code>	<code>5</code>
<code>0 min 5</code>	<code>0</code>
<code>-2.7 abs</code>	<code>2.7</code>
<code>-2.7 round</code>	<code>-3L</code>
<code>1.5 isInfinity</code>	<code>False</code>
<code>(1.0 / 0) isInfinity</code>	<code>True</code>
<code>4 to 6</code>	<code>Range(4, 5, 6)</code>
<code>"bob" capitalize</code>	<code>"Bob"</code>
<code>"robert" drop 2</code>	<code>"bert"</code>

¹ Теперь вы уже знаете, что, получив такой код, компилятор Scala создаст вызов `(bills !*%^~(code)).!`.

главе, существует обогащающая оболочка, которая предоставляет ряд дополнительных методов. Поэтому увидеть все доступные методы, применяемые в отношении основных типов, можно, обратившись к документации по API, которая касается обогащающей оболочки для каждого основного типа. Эти классы перечислены в табл. 5.5.

Таблица 5.5. Классы обогащающих оболочек

Основной тип	Обогащающая оболочка
Byte	<code>scala.runtime.RichByte</code>
Short	<code>scala.runtime.RichShort</code>
Int	<code>scala.runtime.RichInt</code>
Long	<code>scala.runtime.RichLong</code>
Char	<code>scala.runtime.RichChar</code>
Float	<code>scala.runtime.RichFloat</code>
Double	<code>scala.runtime.RichDouble</code>
Boolean	<code>scala.runtime.RichBoolean</code>
String	<code>scala.collection.immutable.StringOps</code>

Резюме

Основное, что следует усвоить, прочитав данную главу, — операторы в Scala являются вызовами методов и для основных типов Scala существуют неявные преобразования в обогащенные варианты, которые добавляют дополнительные полезные методы. В главе 6 мы покажем, что означает конструирование объектов в функциональном стиле, обеспечивающее новые реализации некоторых операторов, рассмотренных в настоящей главе.

6

Функциональные объекты

Усвоив основы, рассмотренные в предыдущих главах, вы готовы разработать больше полнофункциональных классов Scala. В этой главе основное внимание мы уделим классам, определяющим функциональные объекты или объекты, не имеющие никакого изменяемого состояния. Запуская примеры, мы создадим несколько вариантов класса, моделирующего рациональные числа в виде неизменяемых объектов. Попутно будут показаны дополнительные аспекты объектно-ориентированного программирования на Scala: параметры класса и конструкторы, методы и операторы, приватные члены, переопределение, проверка соблюдения предварительных условий, перегрузка и рекурсивные ссылки.

6.1. Спецификация класса Rational

Рациональным называется число, которое может быть выражено соотношением n / d , где n и d представлены целыми числами, за исключением того, что d не может быть нулем. Здесь n называется *числителем*, а d — *знаменателем*. Примерами рациональных чисел могут послужить $1/2$, $2/3$, $112/239$ и $2/1$. В сравнении с числами с плавающей точкой рациональные числа имеют то преимущество, что дроби представлены точно, без округлений или приближений.

Разрабатываемый в этой главе класс должен моделировать поведение рациональных чисел, позволяя производить над ними арифметические действия по сложению, вычитанию, умножению и делению. Для сложения двух рациональных чисел сначала нужно получить общий знаменатель, после чего сложить два числителя. Например, чтобы выполнить сложение $1/2 + 2/3$,

обе части левого операнда умножаются на 3, а обе части правого операнда — на 2, в результате чего получается $3/6 + 4/6$. Сложение двух числителей дает результат $7/6$. Для перемножения двух рациональных чисел можно просто перемножить их числители, а затем знаменатели. Таким образом, $1/2 \cdot 2/5$ дает число $2/10$, которое можно представить более кратко в нормализованном виде как $1/5$. Деление выполняется путем перестановки местами числителя и знаменателя правого операнда с последующим перемножением чисел. Например, $1/2 / 3/5$ — то же самое, что и $1/2 \cdot 5/3$, в результате получается число $5/6$.

Одно, возможно, очевидное наблюдение заключается в том, что в математике рациональные числа не имеют изменяемого состояния. Можно сложить два рациональных числа, и результатом будет новое рациональное число. Исходные числа не будут изменены. Неизменяемый класс `Rational`, разрабатываемый в данной главе, будет иметь такое же свойство. Каждое рациональное число будет представлено одним объектом `Rational`. При сложении двух объектов `Rational` для хранения суммы будет создаваться новый объект `Rational`.

В этой главе мы представим некоторые допустимые в Scala способы написания библиотек, которые создают впечатление, будто используется поддержка, присущая непосредственно самому языку программирования. Например, в конце этой главы вы сможете сделать с классом `Rational` следующее:

```
scala> val oneHalf = Rational(1, 2)
val oneHalf: Rational = 1/2

scala> val twoThirds = Rational(2, 3)
val twoThirds: Rational = 2/3

scala> (oneHalf / 7) + (1 - twoThirds)
val res0: Rational = 17/42
```

6.2. Конструирование класса `Rational`

Конструирование класса `Rational` неплохо начать с рассмотрения того, как клиенты-программисты будут создавать новый объект `Rational`. Было решено создавать объекты `Rational` неизменяемыми, и потому мы потребуем, чтобы эти клиенты при создании экземпляра предоставляли все необходимые ему данные (в нашем случае числитель и знаменатель). Поэтому начнем конструирование со следующего кода:

```
class Rational(n: Int, d: Int)
```

По поводу этой строки кода в первую очередь следует заметить: если у класса нет тела, то вам не нужно ставить пустые фигурные скобки, а также нет необ-

ходимости завершать строку двоеточием. Идентификаторы `n` и `d`, указанные в круглых скобках после имени класса, `Rational`, называются *параметрами класса*. Компилятор Scala подберет эти два параметра и создаст *первичный конструктор*, получающий их же.

Плюсы и минусы неизменяемого объекта

Неизменяемые объекты имеют ряд преимуществ над изменяемыми и один потенциальный недостаток. Во-первых, о неизменяемых объектах проще говорить, чем об изменяемых, поскольку у них нет изменяемых со временем сложных областей состояния. Во-вторых, неизменяемые объекты можно совершенно свободно куда-нибудь передавать, а перед передачей изменяемых объектов в другой код порой приходится делать страховочные копии. В-третьих, если объект правильно сконструирован, то при одновременном обращении к неизменяемому объекту из двух потоков повредить его состояние невозможно, поскольку никакой поток не может изменить состояние неизменяемого объекта. В-четвертых, неизменяемые объекты обеспечивают безопасность ключей хеш-таблиц. Если, к примеру, изменяемый объект изменился после помещения в `HashSet`, то в следующий раз при поиске там его можно не найти.

Главный недостаток неизменяемых объектов — им иногда требуется копирование больших графов объектов, тогда как вместо этого можно было бы сделать обновление. В некоторых случаях это может быть сложно выразить, а также могут выявиться узкие места в производительности. В результате в библиотеки нередко включают изменяемые альтернативы неизменяемым классам. Например, класс `StringBuilder` — изменяемая альтернатива неизменяемому классу `String`. Дополнительная информация о конструировании изменяемых объектов в Scala будет дана в главе 16.

ПРИМЕЧАНИЕ

Исходный пример с `Rational` подчеркивает разницу между Java и Scala. В Java классы имеют конструкторы, которые могут принимать параметры, а в Scala классы могут принимать параметры напрямую. Система записи в Scala куда более лаконична — параметры класса могут использоваться напрямую в теле, нет никакой необходимости определять поля и записывать присваивания, копирующие параметры конструктора в поля. Это может привести к дополнительной экономии на шаблонном коде, особенно когда дело касается небольших классов.

Компилятор Scala скомпилирует любой код, помещенный в тело класса и не являющийся частью поля или определения метода, в первичный конструктор. Например, можно вывести такое отладочное сообщение:

```
class Rational(n: Int, d: Int):  
  println("Created " + n + "/" + d)
```

Получив данный код, компилятор Scala поместит вызов `println` в первичный конструктор класса `Rational`. Поэтому при создании нового экземпляра `Rational` вызов `println` приведет к выводу отладочного сообщения:

```
scala> new Rational(1, 2)  
Created 1/2  
val res0: Rational = Rational@6121a7dd
```

При создании экземпляров классов, таких как `Rational`, вы можете при желании опустить ключевое слово `new`. Такое выражение использует так называемый универсальный метод применения. Вот пример:

```
scala> Rational(1, 2)  
Created 1/2  
val res1: Rational = Rational@5dc7841c
```

6.3. Переопределение метода `toString`

При создании экземпляра `Rational` в предыдущем примере REPL вывел `Rational@5dc7841c`. Эта странная строка получилась ввиду вызова в отношении объекта `Rational` метода `toString`. По умолчанию класс `Rational` наследует реализацию `toString`, определенную в классе `java.lang.Object`, которая просто выводит имя класса, символ `@` и шестнадцатеричное число. Предполагалось, что результат выполнения `toString` поможет программистам, предоставив информацию, которую можно использовать в отладочных инструкциях вывода информации, для ведения логов, в отчетах о сбоях тестов, а также для просмотра выходной информации REPL и отладчика. Результат, выдаваемый на данный момент методом `toString`, не приносит особой пользы, поскольку не дает никакой информации относительно значения рационального числа. Более полезная реализация `toString` будет выводить значения числителя и знаменателя объекта `Rational`. *Переопределить* исходную реализацию можно, добавив метод `toString` к классу `Rational`:

```
class Rational(n: Int, d: Int):  
  override def toString = s"$n/$d"
```

Модификатор `override` перед определением метода показывает, что предыдущее определение метода переопределяется (более подробно этот вопрос рассматривается в главе 10). Поскольку отныне числа типа `Rational` будут выводиться совершенно отчетливо, мы удаляем отладочную инструкцию `println`, помещенную в тело предыдущей версии класса `Rational`. Теперь новое поведение `Rational` можно протестировать в REPL:

```
scala> val x = Rational(1, 3)
x: Rational = 1/3
```

```
scala> val y = Rational(5, 7)
y: Rational = 5/7
```

6.4. Проверка соблюдения предварительных условий

В качестве следующего шага переключим внимание на проблему, связанную с текущим поведением первичного конструктора. Как упоминалось в начале главы, рациональные числа не должны содержать ноль в знаменателе. Но пока первичный конструктор может принимать ноль, передаваемый в качестве параметра `d`:

```
scala> new Rational(5, 0) // 5/0
val res1: Rational = 5/0
```

Одно из преимуществ объектно-ориентированного программирования — возможность инкапсуляции данных внутри объектов, чтобы можно было гарантировать, что данные корректны в течение всей жизни объекта. В данном случае для такого неизменяемого объекта, как `Rational`, это значит, что вы должны гарантировать корректность данных на этапе конструирования объекта при условии, что нулевой знаменатель — недопустимое состояние числа типа `Rational` и такое число не должно создаваться, если в качестве параметра `d` передается ноль.

Лучше всего решить эту проблему, определив для первичного конструктора *предусловие*, согласно которому `d` должен иметь ненулевое значение. Предусловие — ограничение, накладываемое на значения, передаваемые в метод или конструктор, то есть требование, которое должно выполняться вызывающим кодом. Один из способов решить задачу — использовать метод `require`¹:

¹ Метод `require` определен в самостоятельном объекте `Predef`. Как упоминалось в разделе 4.5, элементы класса `Predef` автоматически импортируются в каждый исходный файл Scala.

```
class Rational(n: Int, d: Int):
  require(d != 0)
  override def toString = s"$n/$d"
```

Метод `require` получает один булев параметр. Если переданное значение приведет к вычислению в `true`, то из метода `require` произойдет нормальный выход. В противном случае объект не создастся и будет выдано исключение `IllegalArgumentException`.

6.5. Добавление полей

Теперь, когда первичный конструктор выдвигает нужные предусловия, мы переключимся на поддержку сложения. Для этого определим в классе `Rational` публичный метод `add`, получающий в качестве параметра еще одно значение типа `Rational`. Чтобы сохранить неизменяемость класса `Rational`, метод `add` не должен прибавлять переданное рациональное число к объекту, в отношении которого он вызван. Ему нужно создать и вернуть новый объект `Rational`, содержащий сумму. Можно подумать, что метод `add` создается следующим образом:

```
class Rational(n: Int, d: Int): // Этот код не будет скомпилирован
  require(d != 0)
  override def toString = s"$n/$d"
  def add(that: Rational): Rational =
    Rational(n * that.d + that.n * d, d * that.d)
```

Но, получив этот код, компилятор выдаст свои возражения:

```
5 | Rational(n * that.d + that.n * d, d * that.d)
  |           ^^^^^^
  | value n in class Rational cannot be accessed as a member
  | of (that : Rational) from class Rational.
5 | Rational(n * that.d + that.n * d, d * that.d)
  |           ^^^^^^
  | value d in class Rational cannot be accessed as a member
  | of (that : Rational) from class Rational.
5 | Rational(n * that.d + that.n * d, d * that.d)
  |           ^^^^^^
  | value d in class Rational cannot be accessed as a member
  | of (that : Rational) from class Rational.
```

Хотя параметры `n` и `d` класса находятся в области видимости кода вашего метода `add`, получить доступ к их значениям можно только в объекте, в отношении которого вызван данный метод. Следовательно, когда в реализации последнего указывается `n` или `d`, компилятор рад предоставить вам значения

для этих параметров класса. Но он не может позволить указать `that.n` или `that.d`, поскольку они не ссылаются на объект `Rational`, в отношении которого был вызван метод `add`¹. Чтобы получить доступ к числителю и знаменателю, вам нужно превратить их в поля. В листинге 6.1 показано, как можно добавить эти поля в класс `Rational`².

Листинг 6.1. Класс `Rational` с полями

```
class Rational(n: Int, d: Int):
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  override def toString = s"$numer/$denom"
  def add(that: Rational): Rational =
    Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
```

В версии `Rational`, показанной в листинге, добавлены два поля с именами `numer` и `denom`, которые были проинициализированы значениями параметров `n` и `d` данного класса³. Вдобавок были внесены изменения в реализацию методов `toString` и `add`, позволяющие им использовать поля, а не параметры класса. Эта версия класса `Rational` проходит компиляцию. Ее можно протестировать путем сложения рациональных чисел:

```
val oneHalf = Rational(1, 2)    // 1/2
val twoThirds = Rational(2, 3)  // 2/3
oneHalf.add(twoThirds)         // 7/6
```

Теперь вы уже можете сделать то, чего не могли сделать раньше, а именно получить доступ к значениям числителя и знаменателя из-за пределов объекта. Для этого нужно просто обратиться к публичным полям `numer` и `denom`:

¹ Фактически объект `Rational` можно сложить с самим собой, тогда ссылка будет на тот же объект, в отношении которого был вызван метод `add`. Но поскольку данному методу можно передать любой объект `Rational`, то компилятор все же не позволит вам воспользоваться кодом `that.n`.

² В разделе 10.6 вы узнаете о параметрических полях, которые позволяют сделать тот же самый код короче.

³ Несмотря на то что `n` и `d` используются в теле класса и учитывая, что они применяются только внутри конструкторов, компилятор Scala не станет выделять под них поля. Таким образом, получив этот код, компилятор Scala создаст класс с двумя полями типа `Int`: одним для `numer`, другим для `denom`.

```
val r = Rational(1, 2) // 1/2
r.numer              // 1
r.denom              // 2
```

6.6. Собственные ссылки

Ключевое слово `this` позволяет сослаться на экземпляр объекта, в отношении которого был вызван выполняемый в данный момент метод, или, если оно использовалось в конструкторе, — на создаваемый экземпляр объекта. Рассмотрим в качестве примера добавление метода `lessThan`. Он проверяет, не имеет ли объект `Rational`, в отношении которого он вызван, значение меньше значения параметра:

```
def lessThan(that: Rational) =
  this.numer * that.denom < that.numer * this.denom
```

Здесь выражение `this.numer` ссылается на числительное объекта, в отношении которого вызван метод `lessThan`. Можно также не указывать префикс `this` и написать просто `numer`, обе записи будут равнозначны.

В качестве примера случаев, когда вам не обойтись без `this`, рассмотрим добавление к классу `Rational` метода `max`, возвращающего наибольшее число из заданного рационального числа и переданного аргумента:

```
def max(that: Rational) =
  if this.lessThan(that) then that else this
```

Здесь первое ключевое слово `this` избыточно. Можно его не указывать и написать `lessThan(that)`. Но второе ключевое слово `this` представляет результат метода в том случае, если тест вернет `false`, и если вы его не укажете, то возвращать будет просто нечего!

6.7. Вспомогательные конструкторы

Иногда нужно, чтобы в классе было несколько конструкторов. В Scala все конструкторы, кроме первичного, называются *вспомогательными*. Например, рациональное число со знаменателем 1 можно кратко записать просто в виде числителя. Вместо `5/1`, например, можно просто указать `5`. Поэтому было бы неплохо, чтобы вместо записи `Rational(5, 1)` программисты могли написать просто `Rational(5)`. Для этого потребуется добавить к классу `Rational` вспомогательный конструктор, который получает только один аргумент — числитель, а в качестве знаменателя имеет predetermined значение 1. Как может выглядеть соответствующий код, показано в листинге 6.2.

Листинг 6.2. Класс `Rational` со вспомогательным конструктором

```
class Rational(n: Int, d: Int):
  require(d != 0)

  val numer: Int = n
  val denom: Int = d

  def this(n: Int) = this(n, 1) // вспомогательный конструктор

  override def toString = s"$numer/$denom"

  def add(that: Rational): Rational =
    Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
```

Определения вспомогательных конструкторов в Scala начинаются с `def this(...)`. Тело вспомогательного конструктора класса `Rational` просто вызывает первичный конструктор, передавая дальше свой единственный аргумент `n` в качестве числителя и `1` — в качестве знаменателя. Увидеть вспомогательный конструктор в действии можно, набрав в REPL следующий код:

```
val y = Rational(3) // 3/1
```

В Scala каждый вспомогательный конструктор в качестве первого действия должен вызывать еще один конструктор того же класса. Иными словами, первой инструкции в каждом вспомогательном конструкторе каждого класса Scala следует иметь вид `this(...)`. Вызываемым должен быть либо первичный конструктор (как в примере с классом `Rational`), либо другой вспомогательный конструктор, который появляется в тексте программы перед вызывающим его конструктором. Конечный результат применения данного правила заключается в том, что каждый вызов конструктора в Scala должен в конце концов завершаться вызовом первичного конструктора класса. Первичный конструктор, таким образом, — единственная точка входа в класс.

ПРИМЕЧАНИЕ

Знатоков Java может удивить то, что в Scala правила в отношении конструкторов более строгие, чем в Java. Ведь в Java первым действием конструктора должен быть либо вызов другого конструктора того же класса, либо вызов конструктора суперкласса напрямую. В классе Scala конструктор суперкласса может быть вызван только первичным конструктором. Более сильные ограничения в Scala фактически являются компромиссом дизайнера — платой за большую лаконичность и простоту конструкторов Scala по сравнению с конструкторами Java. Суперклассы и подробности вызова конструкторов и наследования будут рассмотрены в главе 10.

6.8. Приватные поля и методы

В предыдущей версии класса `Rational` мы просто инициализировали `numer` значением `n`, а `denom` — значением `d`. Из-за этого числитель и знаменатель `Rational` могут превышать необходимые значения. Например, дробь $66/42$ можно сократить и привести к виду $11/7$, но первичный конструктор класса `Rational` пока этого не делает:

```
Rational(66, 42) // 66/42
```

Чтобы выполнить такое сокращение, нужно разделить числитель и знаменатель на их *наибольший общий делитель*. Например, таковым для 66 и 42 будет число 6. (Иными словами, 6 — наибольшее целое число, на которое без остатка делится как 66, так и 42.) Деление и числителя, и знаменателя числа $66/42$ на 6 приводит к получению сокращенной формы $11/7$. Один из способов решения данной задачи показан в листинге 6.3.

В данной версии класса `Rational` было добавлено приватное поле `g` и изменены инициализаторы для полей `numer` и `denom`. (*Инициализатором* называется код, инициализирующий переменную, например `n / g`, который инициализирует поле `numer`.) Поле `g` является приватным, поэтому доступ к нему может быть выполнен изнутри, но не снаружи тела класса. Кроме того, был добавлен приватный метод по имени `gcd`, вычисляющий наибольший общий делитель двух переданных ему значений `Int`. Например, вызов `gcd(12, 8)` дает результат 4. Как было показано в разделе 4.1, чтобы сделать поле или метод приватным, следует просто поставить перед его определением ключевое слово `private`. Назначение приватного «вспомогательного метода» `gcd` — обособление кода, необходимого для остальных частей класса, в данном случае для первичного конструктора. Чтобы обеспечить постоянное положительное значение поля `g`, методу передаются абсолютные значения параметров `n` и `d`, которые вызов получает в отношении этих параметров метода `abs`. Последний может вызываться в отношении любого `Int`-объекта в целях получения его абсолютного значения.

Листинг 6.3. Класс `Rational` с приватным полем и методом

```
class Rational(n: Int, d: Int):
  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)
```

```

def add(that: Rational): Rational =
  Rational(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )

override def toString = s"$numer/$denom"

private def gcd(a: Int, b: Int): Int =
  if b == 0 then a else gcd(b, a % b)

```

Компилятор Scala поместит коды инициализаторов трех полей класса `Rational` в первичный конструктор в порядке их следования в исходном коде. Таким образом, инициализатор поля `g`, имеющий код `gcd(n.abs, d.abs)`, будет выполнен до выполнения двух других инициализаторов, поскольку в исходном коде появляется первым. Поле `g` будет инициализировано результатом — наибольшим общим делителем абсолютных значений параметров `n` и `d` класса. Затем поле `g` будет использовано в инициализаторах полей `numer` и `denom`. Разделив `n` и `d` на их наибольший общий делитель `g`, каждый объект `Rational` можно сконструировать в нормализованной форме:

```
Rational(66, 42) // 11/7
```

6.9. Определение операторов

Текущая реализация класса `Rational` нас вполне устраивает, но ее можно сделать гораздо более удобной в использовании. Вы можете спросить, почему допустима запись

```
x + y
```

если `x` и `y` — целые числа или числа с плавающей точкой. Но когда это рациональные числа, приходится пользоваться записью

```
x.add(y)
```

или в крайнем случае

```
x add y
```

Такое положение дел ничем не оправдано. Рациональные числа совершенно неотличимы от остальных чисел. В математическом смысле они гораздо более естественны, чем, скажем, числа с плавающей точкой. Так почему бы не воспользоваться во время работы с ними естественными математическими

операторами? И в Scala есть такая возможность. Она будет показана в оставшейся части главы.

Сначала нужно заменить `add` обычным математическим символом. Сделать это нетрудно, поскольку знак `+` является в Scala вполне допустимым идентификатором. Можно просто определить метод с именем `+`. Если уж на то пошло, то можно определить и метод `*`, выполняющий умножение. Результат показан в листинге 6.4.

Листинг 6.4. Класс `Rational` с методами-операторами

```
class Rational(n: Int, d: Int):

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def * (that: Rational): Rational =
    Rational(numer * that.numer, denom * that.denom)

  override def toString = s"$numer/$denom"

  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
```

После такого определения класса `Rational` можно будет воспользоваться следующим кодом:

```
val x = Rational(1, 2) // 1/2
val y = Rational(2, 3) // 2/3
x + y                 // 7/6
```

Как всегда, синтаксис оператора в последней строке ввода — эквивалент вызова метода. Можно также использовать следующий код:

```
x.+(y) // 7/6
```

но читать его будет намного труднее.

Следует также заметить, что из-за действующих в Scala правил приоритета операторов, рассмотренных в разделе 5.9, метод `*` будет привязан к объектам `Rational` сильнее метода `+`. Иными словами, выражения, в которых к объектам `Rational` применяются операции `+` и `*`, будут вести себя вполне ожидаемым образом. Например, `x + x * y` будет выполняться как `x + (x * y)`, а не как `(x + x) * y`:

```
x + x * y // 5/6
(x + x) * y // 2/3
x + (x * y) // 5/6
```

6.10. Идентификаторы в Scala

Вам уже встречались два наиболее важных способа составления идентификаторов в Scala — из буквенно-цифровых символов и из операторов. В Scala используются весьма гибкие правила формирования идентификаторов. Кроме двух уже встречавшихся форм, существует еще две. Все четыре формы составления идентификаторов рассматриваются в этом разделе.

Буквенно-цифровые идентификаторы начинаются с буквы или знака подчеркивания, за которыми могут следовать другие буквы, цифры или знаки подчеркивания. Символ `$` также считается буквой, но зарезервирован для идентификаторов, создаваемых компилятором Scala. Идентификаторы в пользовательских программах не должны содержать символы `$`, несмотря на возможность успешно пройти компиляцию: если это произойдет, то могут возникнуть конфликты имен с теми идентификаторами, которые будут созданы компилятором Scala.

В Scala соблюдается соглашение, принятое в Java относительно применения идентификаторов в смешанном регистре¹, таких как `toString` и `HashSet`. Хотя использование знаков подчеркивания в идентификаторах вполне допустимо, в программах на Scala они встречаются довольно редко — отчасти в целях соблюдения совместимости с Java, а также из-за того, что знаки подчеркивания в коде Scala активно применяются не только для идентификаторов. Поэтому лучше избегать таких идентификаторов, как, например, `to_string`, `__init__` или `name_`. Имена полей, параметры методов, имена локальных переменных и имена функций в смешанном регистре должны начинаться с буквы в нижнем регистре, например: `length`, `flatMap` и `s`. Имена классов и трейтов

¹ Этот стиль именования идентификаторов называется верблюжьим, поскольку у идентификаторов Имеются Горбы, состоящие из символов в верхнем регистре.

в смешанном регистре должны начинаться с буквы в верхнем регистре, например: `BigInt`, `List` и `UnbalancedTreeMap`¹.

ПРИМЕЧАНИЕ

Одним из последствий использования в идентификаторе замыкающего знака подчеркивания при попытке, к примеру, написания объявления `val name_: Int = 1` может стать ошибка компиляции. Компилятор подумает, что вы пытаетесь объявить `val`-переменную по имени `name_:`. Чтобы такой идентификатор прошел компиляцию, перед двоеточием нужно поставить дополнительный пробел, как в коде `val name_ : Int = 1`.

Один из примеров отступления Scala от соглашений, принятых в Java, касается имен констант. В Scala слово «константа» означает не только `val`-переменную. Даже притом что `val`-переменная остается неизменной после инициализации, она не перестает быть переменной. Например, параметры метода относятся к `val`-переменным, но при каждом вызове метода в этих `val`-переменных содержатся разные значения. Константа обладает более выраженным постоянством. Например, `scala.math.Pi` определяется как значение с двойной точностью, наиболее близкое к реальному значению числа π — отношению длины окружности к ее диаметру. Это значение вряд ли когда-либо изменится, поэтому со всей очевидностью можно сказать, что `Pi` — константа. Константы можно использовать также для присваивания имен значениям, которые иначе были бы в вашем коде *магическими числами* — буквальными значениями без объяснений, которые в худшем случае появлялись бы в коде в нескольких местах. Вдобавок может понадобиться определить константы для использования при сопоставлении с образцом (подобный случай будет рассматриваться в разделе 13.2). В соответствии с соглашением, принятым в Java, константам присваиваются имена, в которых используются символы в верхнем регистре, где знак подчеркивания является разделителем слов, например `MAX_VALUE` или `PI`. В Scala соглашение требует, чтобы в верхнем регистре была только первая буква. Таким образом, константы, названные в стиле Java, например `X_OFFSET`, будут работать в Scala в качестве констант, но в соответствии с соглашением, принятым в Scala, для имен констант применяется смешанный регистр, например `XOffset`.

Идентификатор оператора состоит из одного или нескольких символов операторов. Таковыми являются выводимые на печать ASCII-символы, такие

¹ В разделе 14.5 вы увидите, что иногда может возникнуть желание придать классу особый вид, как у `case`-класса, чье имя состоит только из символов оператора. Например, в API Scala имеется класс по имени `::`, облегчающий сопоставление с образцом для объектов `List`.

как `+`, `:`, `?`, `~` или `#`¹. Ниже показаны некоторые примеры идентификаторов операторов:

```
+ ++ ::: <?> :->
```

Компилятор Scala на внутреннем уровне перерабатывает идентификаторы операторов, чтобы превратить их в допустимые Java-идентификаторы со встроенными символами `$`. Например, идентификатор `:->` будет представлен как `$colon$minus$greater`. Если вам когда-либо захочется получить доступ к этому идентификатору из кода Java, то потребуется использовать данное внутреннее представление.

Поскольку идентификаторы операторов в Scala могут принимать произвольную длину, то между Java и Scala есть небольшая разница в этом вопросе. В Java введенный код `x<-y` будет разобран на четыре лексических символа, в результате чего станет эквивалентен `x < - y`. В Scala оператор `<-` будет рассмотрен как единый идентификатор, в результате чего получится `x <- y`. Если нужно получить первую интерпретацию, то следует разделить символы `<` и `-` друг от друга пробелом. На практике это вряд ли станет проблемой, так как немногие станут писать на Java `x<-y`, не вставляя пробелы или круглые скобки между операторами.

Смешанный идентификатор состоит из буквенно-цифрового идентификатора, за которым стоят знак подчеркивания и идентификатор оператора. Например, `unary_+`, использованный как имя метода, определяет унарный оператор `+`. А `myvar_ =`, использованный как имя метода, определяет оператор присваивания. Кроме того, смешанный идентификатор вида `myvar_ =` генерируется компилятором Scala в целях поддержки *свойств* (более подробно этот вопрос рассматривается в главе 16).

Литеральный идентификатор представляет собой произвольную строку, заключенную в обратные кавычки (``...``). Примеры литеральных идентификаторов выглядят следующим образом:

```
`x` `<clinit>` `yield`
```

Замысел состоит в том, что между обратными кавычками можно поместить любую строку, которую среда выполнения станет воспринимать в качестве

¹ Точнее, символ оператора принадлежит к математическим символам (Sm) или прочим символам (So) стандарта Unicode либо к семибитным ASCII-символам, не являющимся буквами, цифрами, круглыми, квадратными и фигурными скобками, одинарными или двойными кавычками или знаками подчеркивания, точки, точки с запятой, запятой или обратных кавычек.

идентификатора. В результате всегда будет получаться идентификатор Scala. Это работает даже в том случае, если имя, заключенное в обратные кавычки, является в Scala зарезервированным словом. Обычно такие идентификаторы используются при обращении к статическому методу `yield` в Java-классе `Thread`. Вы не можете прибегнуть к коду `Thread.yield()`, поскольку в Scala `yield` является зарезервированным словом. Но имя метода все же можно применить, если заключить его в обратные кавычки, например `Thread.`yield`()`.

6.11. Перегрузка методов

Вернемся к классу `Rational`. После внесения последних изменений появилась возможность применять операции сложения и умножения рациональных чисел в их естественном виде. Но мы все же упустили из виду смешанную арифметику. Например, вы не можете умножить рациональное число на целое, поскольку операнды у оператора `*` всегда должны быть объектами `Rational`. Следовательно, для рационального числа `r` вы не можете написать код `r * 2`. Вам нужно написать `r * Rational(2)`, а это имеет неприглядный вид.

Чтобы сделать класс `Rational` еще более удобным в использовании, добавим к нему новые методы, выполняющие смешанное сложение и умножение рациональных и целых чисел. А заодно добавим методы вычитания и деления. Результат показан в листинге 6.5.

Теперь здесь две версии каждого арифметического метода: одна в качестве аргумента получает рациональное число, вторая — целое. Иными словами, все эти методы называются *перегруженными*, поскольку каждое имя теперь используется несколькими методами. Например, имя `+` применяется и методом, получающим объект `Rational`, и методом, получающим объект `Int`. При вызове метода компилятор выбирает версию перегруженного метода, которая в точности соответствует типу аргументов. Например, если аргумент `y` в вызове `x.+(y)` является объектом `Rational`, то компилятор выберет метод `+`, получающий в качестве параметра объект `Rational`. Но если аргумент — целое число, то компилятор выберет метод `+`, получающий в качестве параметра объект `Int`. Если испытать код в действии:

```
Val r = Rational(2, 3) // 2/3
r * r                 // 4/9
r * 2                 // 4/3
```

станет понятно, что вызываемый метод `*` определяется каждый раз по типу его правого операнда.

ПРИМЕЧАНИЕ

В Scala процесс анализа при выборе перегруженного метода очень похож на аналогичный процесс в Java. В любом случае выбирается перегруженная версия, которая лучше подходит к статическим типам аргументов. Иногда случается, что одной такой версии нет, и тогда компилятор выдаст ошибку, связанную с неоднозначной ссылкой, — *ambiguous reference*.

Листинг 6.5. Класс Rational с перегруженными методами

```
class Rational(n: Int, d: Int):
  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def + (i: Int): Rational =
    Rational(numer + i * denom, denom)

  def - (that: Rational): Rational =
    Rational(
      numer * that.denom - that.numer * denom,
      denom * that.denom
    )

  def - (i: Int): Rational =
    Rational(numer - i * denom, denom)

  def * (that: Rational): Rational =
    Rational(numer * that.numer, denom * that.denom)

  def * (i: Int): Rational =
    Rational(numer * i, denom)

  def / (that: Rational): Rational =
    Rational(numer * that.denom, denom * that.numer)

  def / (i: Int): Rational =
    Rational(numer, denom * i)

  override def toString = s"$numer/$denom"

  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
```

6.12. Методы расширения

Теперь, когда можно воспользоваться кодом `r * 2`, вы также можете захотеть поменять операнды местами, задействовав код `2 * r`. К сожалению, пока этот код работать не будет:

```
scala> 2 * r
1 | 2 * r
  | ^^
  | None of the overloaded alternatives of method * in
  | class Int with types
  | (x: Double): Double
  | (x: Float): Float
  | (x: Long): Long
  | (x: Int): Int
  | (x: Char): Int
  | (x: Short): Int
  | (x: Byte): Int
  | match arguments ((r : Rational))
```

Проблема в том, что эквивалент выражения `2 * r` — выражение `2.*(r)`, то есть вызов метода в отношении числа 2, которое является целым. Но в классе `Int` не содержится метода умножения, получающего в качестве аргумента объект `Rational`, его там и не может быть, поскольку он не входит в состав стандартных классов библиотеки `Scala`.

Но в `Scala` есть другой способ решения этой проблемы. Вы можете создавать методы расширения для `Int`, которые содержат рациональные числа. Попробуйте добавить эти строки в `REPL`:

```
extension (x: Int)
  def + (y: Rational) = Rational(x) + y
  def - (y: Rational) = Rational(x) - y
  def * (y: Rational) = Rational(x) * y
  def / (y: Rational) = Rational(x) / y
```

Это определяет четыре метода расширения для `Int`, каждый из которых использует `Rational`. Компилятор может использовать их автоматически в ряде ситуаций. С определенными методами расширения теперь вы можете повторить пример, который раньше не удался:

```
val r = Rational(2,3) // 2/3
2 * r                // 4/3
```

Чтобы метод расширения работал, он должен находиться в области видимости. Если вы поместите определение метода расширения внутри класса

`Rational`, он не попадет в область действия REPL, поэтому вам необходимо определить его непосредственно в REPL.

Как видно из этого примера, методы расширения — это очень эффективная техника, позволяющая сделать библиотеки более гибкими и удобными в использовании. Однако ее чрезмерное использование может и навредить. В главе 22 вы узнаете больше о методах расширения, в том числе о способах включения их в область видимости там, где они необходимы.

6.13. Предостережение

Создание методов с именами операторов и определение методов расширения, продемонстрированные в этой главе, призваны помочь в проектировании библиотек, для которых код клиента будет лаконичным и понятным. Scala предоставляет широкие возможности для разработки таких весьма доступных для использования библиотек. Но, пожалуйста, имейте в виду: реализуя возможности, не стоит забывать об ответственности.

При неумелом использовании и методы-операторы, и методы расширения могут сделать клиентский код таким, что его станет трудно читать и понимать. Выполнение компилятором методов расширения никак внешне не проявляется и не записывается в явном виде в исходный код. Поэтому программистам на клиенте может быть невдомек, что именно оно и применяется в вашем коде. И хотя методы-операторы обычно делают клиентский код более лаконичным и читабельным, таким он становится только для наиболее сведущих программистов-клиентов, способных запомнить и распознать значение каждого оператора.

При проектировании библиотек всегда нужно стремиться сделать клиентский код не просто лаконичным, но и легко читаемым и понятным. Читабельность в значительной степени может быть обусловлена лаконичностью, которая способна заходить очень далеко. Проектируя библиотеки, позволяющие добиваться изысканной лаконичности, и в то же время создавая понятный клиентский код, вы можете существенно повысить продуктивность работы программистов, которые используют эти библиотеки.

Резюме

В этой главе мы рассмотрели многие аспекты классов Scala. Вы увидели способы добавления к классу параметров, определили несколько конструк-

торов, операторы и методы и настроили классы таким образом, чтобы их применение приобрело более естественный вид. Важнее всего, вероятно, было показать вам, что определение и использование неизменяющихся объектов — вполне естественный способ программирования на Scala.

Хотя показанная здесь финальная версия класса `Rational` соответствует всем требованиям, обозначенным в начале главы, ее можно усовершенствовать. Позже мы вернемся к этому примеру. В частности, в главе 8 будет рассмотрено переопределение методов `equals` и `hashCode`, которое позволяет объектам `Rational` улучшить свое поведение в момент, когда их сравнивают с помощью оператора `==` или помещают в хеш-таблицы. В главе 22 поговорим о том, как помещать методы расширения в объекты-компаньоны класса `Rational`, которые упрощают для программистов-клиентов помещение в область видимости объектов типа `Rational`.

7

Встроенные управляющие конструкции

В Scala имеется весьма незначительное количество встроенных управляющих конструкций. К ним относятся `if`, `while`, `for`, `try`, `match` и вызовы функций. Их в Scala немного, поскольку с момента создания данного языка в него были включены функциональные литералы. Вместо накопления в базовом синтаксисе одной за другой высокоуровневых управляющих конструкций Scala собирает их в библиотеках. Как именно это делается, мы покажем в главе 9. А здесь рассмотрим имеющиеся в Scala немногочисленные встроенные управляющие конструкции.

Следует учесть, что почти все управляющие конструкции Scala приводят к какому-либо значению. Такой подход принят в функциональных языках, где программы рассматриваются в качестве вычислителей значений, стало быть, компоненты программы тоже должны вычислять значения. Можно рассматривать данное обстоятельство как логическое завершение тенденции, уже присутствующей в императивных языках. В них вызовы функций могут возвращать значение, даже когда наряду с этим будет происходить обновление вызываемой функцией выходной переменной, переданной в качестве аргумента. Кроме того, в императивных языках зачастую имеется тернарный оператор (такой как оператор `?:` в C, C++ и Java), который ведет себя полностью аналогично `if`, но при этом возвращает значение. Scala позаимствовал эту модель тернарного оператора, но назвал ее `if`. Иными словами, используемый в Scala оператор `if` может выдавать значение. Затем эта тенденция в Scala получила развитие: `for`, `try` и `match` тоже стали выдавать значения.

Программисты могут использовать полученное в результате значение, чтобы упростить свой код, применяя те же приемы, что и для значений, возвращаемых функциями. Не будь этой особенности, программистам пришлось бы создавать временные переменные, просто чтобы хранить результаты, вычисленные

внутри управляющей конструкции. Отказ от таких переменных немного упрощает код, а также избавляет от многих ошибок, возникающих, когда в одном ответвлении переменная создается, а в другом о ее создании забывают.

В целом, основные управляющие конструкции Scala в минимальном составе обеспечивают все, что нужно было взять из императивных языков. При этом они позволяют сделать код более лаконичным за счет неизменного наличия значений, получаемых в результате их применения. Чтобы показать все это в работе, далее более подробно рассмотрим основные управляющие конструкции Scala.

7.1. Выражения `if`

Выражение `if` в Scala работает практически так же, как во многих других языках. Оно проверяет условие, а затем выполняет одну из двух ветвей кода в зависимости от того, вычисляется ли условие в `true`. Простой пример, написанный в императивном стиле, выглядит следующим образом:

```
var filename = "default.txt"
if !args.isEmpty then
  filename = args(0)
```

В этом коде объявляется переменная по имени `filename`, которая инициализируется значением по умолчанию. Затем в нем используется выражение `if` с целью проверить, предоставлены ли программе какие-либо аргументы. Если да, то в переменную вносят изменения, чтобы в ней содержалось значение, указанное в списке аргументов. Если нет, то выражение оставляет значение переменной, установленное по умолчанию.

Этот код можно сделать гораздо более выразительным, поскольку, как упоминалось в шаге 3 главы 2, выражение `if` в Scala возвращает значение. В листинге 7.1 показано, как можно выполнить те же самые действия, что и в предыдущем примере, не прибегая к использованию `var`-переменных.

Листинг 7.1. Особый стиль Scala, применяемый для условной инициализации

```
val filename =
  if !args.isEmpty then args(0)
  else "default.txt"
```

На этот раз у `if` имеются два ответвления. Если массив `args` непустой, то выбирается его начальный элемент `args(0)`. В противном случае выбирается значение по умолчанию. Выражение `if` выдает результат в виде выбранного

значения, которым инициализируется переменная `filename`. Данный код немного короче предыдущего. Но гораздо более существенно то, что в нем используется `val`-, а не `var`-переменная. Это соответствует функциональному стилю и помогает вам примерно так же, как применение финальной (`final`) переменной в Java. Она сообщает читателям кода, что переменная никогда не изменится, избавляя их от необходимости просматривать весь код в области видимости переменной, чтобы понять, изменяется ли она где-нибудь.

Второе преимущество использования `var`-переменной вместо `val`-переменной заключается в том, что она лучше поддерживает выводы, которые делаются с помощью *эквациональных рассуждений* (*equational reasoning*). Введенная переменная *равна* вычисляющему выражению при условии, что у него нет побочных эффектов. Таким образом, всякий раз, собираясь написать имя переменной, вы можете вместо него написать выражение. Вместо `println(filename)`, к примеру, можно просто написать следующий код:

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

Выбор за вами. Вы можете прибегнуть к любому из вариантов. Использование `val`-переменных помогает совершенно безопасно проводить подобный рефакторинг кода по мере его развития.

Всегда ищите возможности для применения `val`-переменных. Они смогут упростить не только чтение вашего кода, но и его рефакторинг.

7.2. Циклы while

Используемые в Scala циклы `while` ведут себя точно так же, как и в других языках. В них имеются условие и тело, которое выполняется снова и снова, пока условие вычисляется в `true`. Пример показан в листинге 7.2.

Листинг 7.2. Вычисление наибольшего общего делителя с применением цикла `while`

```
def gcdLoop(x: Long, y: Long): Long =
  var a = x
  var b = y
  while a != 0 do
    val temp = a
    a = b % a
    b = temp
  b
```

Конструкция `while` называется циклом, а не выражением, потому что она не возвращает значение. Типом результата является `Unit`. Получается так, что фактически существует только одно значение, имеющее тип `Unit`. Оно называется `unit`-значением и записывается как `()`. Существование `()` отличает имеющийся в Scala класс `Unit` от используемого в Java типа `void`. Попробуйте сделать это в REPL:

```
scala> def greet() = println("hi")
def greet(): Unit

scala> val iAmUnit = greet() == ()
hi
val iAmUnit: Boolean = true
```

Поскольку тип выражения `println("hi")` определяется как `Unit`, то `greet` определяется как процедура с результирующим типом `Unit`. Поэтому `greet` возвращает `()`. Это подтверждается в следующей строке, где переменная `iAmUnit` возвращает `true`, потому что результат `greet` равен `()`.

Начиная с третьего выпуска, Scala больше не предлагает цикл `do-while`, в котором условие проверялось после тела цикла, а не до него. Вместо этого вы можете поместить операторы тела цикла первыми после `while`, закончить логическим условием и затем поставить `do()`. В листинге 7.3 показан Scala-скрипт, использующий этот подход для отображения строк, считанных из стандартного ввода, до тех пор, пока не будет введена пустая строка.

Листинг 7.3. Выполнение тела цикла хотя бы один раз без `do-while`

```
import scala.io.StdIn.readLine
while
  val line = readLine()
  println(s"Read: $line")
  line != ""
do ()
```

Еще одна уместная здесь конструкция, приводящая к `unit`-значению, — это переназначение `var`. Например, если вы попытаетесь прочитать строки в Scala, используя следующую идиому цикла `while` из Java (а также C и C++), вы столкнетесь с проблемой:

```
var line = "" // Этот код не скомпилируется!
while (line = scala.io.StdIn.readLine()) != "" do
  println(s"Read: $line")
```

Если вы попытаетесь скомпилировать этот код, Scala выдаст вам ошибку о том, что вы не можете сравнивать значения типа `Unit` и `String`, используя `!=`. В то время как в Java присваивание приводит к присваиваемому

значению (в данном случае это строка из стандартного ввода), в Scala присваивание всегда приводит к `Unit`-значению `()`. Таким образом, значение присваивания `"line = readLine()"` всегда будет `()` и никогда не будет `" "`. В результате условие этого цикла `while` никогда не было бы ложным и, следовательно, цикл никогда не завершился бы.

В результате цикла `while` никакое значение не возвращается, поэтому зачастую его не включают в чисто функциональные языки. В таких языках имеются выражения, а не циклы. И тем не менее цикл `while` включен в Scala, поскольку иногда императивные решения бывает легче читать, особенно тем программистам, которые много работали с императивными языками. Например, если нужно закодировать алгоритм, повторяющий процесс до тех пор, пока не изменятся какие-либо условия, то цикл `while` позволяет выразить это напрямую, в то время как функциональную альтернативу наподобие использования рекурсии читателям кода распознавать оказывается сложнее.

Например, в листинге 7.4 показан альтернативный способ определения наибольшего общего знаменателя двух чисел¹. При условии присваивания `x` и `y` в функции `gcd` таких же значений, как и в функции `gcdLoop`, показанной в листинге 7.2, будет выдан точно такой же результат. Разница между этими двумя подходами состоит в том, что функция `gcdLoop` написана в императивном стиле с использованием `var`-переменных и цикла `while`, а функция `gcd` — в более функциональном стиле с применением рекурсии (`gcd` вызывает себя), для чего не нужны `var`-переменные.

Листинг 7.4. Вычисление наибольшего общего делителя с применением рекурсии

```
def gcd(x: Long, y: Long): Long =
  if y == 0 then x else gcd(y, x % y)
```

В целом мы рекомендуем относиться к циклам `while` в своем коде с оглядкой, как и к использованию в нем `var`-переменных. Фактически циклы `while` и `var`-переменные зачастую идут рука об руку. Поскольку циклы не дают результата в виде значения, то для внесения в программу каких-либо изменений цикл `while` обычно будет нуждаться либо в обновлении `var`-переменных, либо в выполнении ввода-вывода. В этом можно убедиться, посмотрев на работу показанного ранее примера `gcdLoop`. По мере выполнения своей задачи цикл `while` обновляет значения `var`-переменных `a` и `b`. Поэтому мы советуем

¹ Здесь в функции `gcd` используется точно такой же подход, который применялся в одноименной функции в листинге 6.3 в целях вычисления наибольших общих делителей для класса `Rational`. Основное отличие заключается в том, что вместо `Int`-значений `gcd` код работает с `Long`-значениями.

проявлять особую осмотрительность при использовании в коде циклов `while`. Если нет достаточных оснований для применения цикла `while` или `do-while`, то попробуйте найти способ сделать то же самое без их участия.

7.3. Выражения `for`

Используемые в Scala выражения `for` являются для итераций чем-то наподобие швейцарского армейского ножа. Чтобы можно было реализовать широкий спектр итераций, эти выражения позволяют различными способами составлять комбинации из довольно простых ингредиентов. Простое применение дает возможность решать простые задачи вроде поэлементного обхода последовательности целых чисел. Более сложные выражения могут выполнять обход элементов нескольких коллекций различных видов, фильтровать элементы на основе произвольных условий и создавать новые коллекции.

Обход элементов коллекций

Самое простое, что можно сделать с выражением `for`, — это выполнить обход всех элементов коллекции. Например, в листинге 7.5 показан код, который выводит имена всех файлов, содержащихся в текущем каталоге. Ввод-вывод выполняется с помощью API Java. Сначала в текущем каталоге, ".", создается объект `java.io.File` и вызывается его метод `listFiles`. Последний возвращает массив объектов `File`, по одному на каталог или файл, содержащийся в текущем каталоге. Получившийся в результате массив сохраняется в переменной `filesHere`.

Листинг 7.5. Получение списка файлов в каталоге с применением выражения `for`

```
val filesHere = (new java.io.File(".")).listFiles
```

```
for file <- filesHere do  
  println(file)
```

С помощью синтаксиса `file <- filesHere`, называемого *генератором*, выполняется обход элементов массива `filesHere`. При каждой итерации значением элемента инициализируется новая `val`-переменная по имени `file`. Компилятор приходит к выводу, что типом `file` является `File`, поскольку `filesHere` имеет тип `Array[File]`. Для каждой итерации выполняется тело выражения `for`, имеющее код `println(file)`. Метод `toString`, определенный в классе `File`, выдает имя файла или каталога, поэтому будут выведены имена всех файлов и каталогов текущего каталога.

Синтаксис выражения `for` работает не только с массивами, но и с коллекциями любого типа¹. Особый и весьма удобный случай — применение типа `Range`, который был упомянут в табл. 5.4. Можно создавать объекты `Range`, используя синтаксис вида `1 to 5`, и выполнять их обход с помощью `for`. Простой пример имеет следующий вид:

```
scala> for i <- 1 to 4 do
  println(s"Iteration $i")
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Если не нужно включать верхнюю границу диапазона в перечисляемые значения, то вместо `to` используется `until`:

```
scala> for i <- 1 until 4 do
  println(s"Iteration $i")
Iteration 1
Iteration 2
Iteration 3
```

Перебор целых чисел наподобие этого встречается в Scala довольно часто, но намного реже, чем в других языках, где данным свойством можно воспользоваться для перебора элементов массива:

```
// В Scala такой код встречается довольно редко...
for i <- 0 to filesHere.length - 1 do
  println(filesHere(i))
```

В это выражение `for` введена переменная `i`, которой по очереди присваивается каждое целое число от `0` до `filesHere.length - 1`, и для каждой установки `i` выполняется тело выражения. Для каждого значения `i` из массива `filesHere` извлекается и обрабатывается `i`-й элемент.

Такого вида итерации меньше распространены в Scala потому, что есть возможность выполнить непосредственный обход элементов коллекции. При этом код становится короче и исключаются многие ошибки смещения на единицу, которые могут возникнуть при обходе элементов массива. С чего нужно начинать — с `0` или `1`? Что нужно прибавлять к завершающему

¹ Точнее, выражение справа от символа `<-` в выражении `for` может быть любого типа, имеющего определенные методы (в данном случае `foreach`) с соответствующими сигнатурами. Подробная информация о том, как компилятор Scala обрабатывает выражения `for`, дана в главе 2.

индексу, -1 , $+1$ или вообще ничего? На подобные вопросы ответить несложно, но также просто дать и неверный ответ. Безопаснее их вообще исключить.

Фильтрация

Иногда перебирать коллекцию целиком не нужно, а требуется отфильтровать ее в некое подмножество. В выражении `for` это можно сделать путем добавления *фильтра* в виде условия `if`, указанного в выражении `for` внутри круглых скобок. Например, код, показанный в листинге 7.6, выводит список только тех файлов текущего каталога, имена которых заканчиваются на `.scala`.

Листинг 7.6. Поиск файлов с расширением `.scala` с помощью `for` с фильтром

```
val filesHere = (new java.io.File(".")).listFiles
for file <- filesHere if file.getName.endsWith(".scala") do
  println(file)
```

Для достижения той же цели можно применить альтернативный вариант:

```
for file <- filesHere do
  if file.getName.endsWith(".scala") then
    println(file)
```

Этот код выдает на выходе то же самое, что и предыдущий, и выглядит, вероятно, более привычно для программистов с опытом работы на императивных языках. Но императивная форма — только вариант, поскольку данное выражение `for` выполняется в целях получения побочных эффектов, выражающихся в выводе данных, и выдает результат в виде `Unit`-значения (`()`). Чуть позже в этом разделе будет показано, что `for` называется выражением, так как по итогу его выполнения получается представляющий интерес результат, то есть коллекция, чей тип определяется компонентами `<-` выражения `for`.

Если потребуется, то в выражение можно включить еще больше фильтров. В него просто нужно добавлять условия `if`. Например, чтобы обеспечить безопасность, код в листинге 7.7 выводит только файлы, исключая каталоги. Для этого добавляется фильтр, который проверяет имеющийся у файла метод `isFile`.

Листинг 7.7. Использование в выражении `for` нескольких фильтров

```
for
  file <- filesHere
  if file.isFile
  if file.getName.endsWith(".scala")
do println(file)
```

Вложенные итерации

Если добавить несколько операторов `<-`, то будут получены вложенные циклы. Например, выражение `for`, показанное в листинге 7.8, имеет два таких цикла. Внешний перебирает элементы массива `filesHere`, а внутренний — элементы `fileLines(file)` для каждого файла, имя которого заканчивается на `.scala`.

Листинг 7.8. Использование в выражении `for` нескольких генераторов

```
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines().toArray

def grep(pattern: String) =
  for
    file <- filesHere if file.getName.endsWith(".scala")
    line <- fileLines(file)
    if line.trim.matches(pattern)
  do println(s"$file: ${line.trim}")

grep(".*gcd.*")
```

Привязки промежуточных переменных

Обратите внимание на повторение в предыдущем коде выражения `line.trim`. Данное вычисление довольно сложное, и потому выполнить его лучше один раз. Сделать это позволяет привязка результата к новой переменной с помощью знака равенства (`=`). Связанная переменная вводится и используется точно так же, как и `val`-переменная, но ключевое слово `val` не ставится. Пример показан в листинге 7.9.

Листинг 7.9. Промежуточное присваивание в выражении `for`

```
def grep(pattern: String) =
  for
    file <- filesHere if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(pattern)
  do println(s"$file: $trimmed")

grep(".*gcd.*")
```

В листинге 7.9 переменная по имени `trimmed` вводится в ходе выполнения выражения `for`. Ее инициализирует результат вызова метода `line.trim`. Затем остальная часть кода выражения `for` использует новую переменную в двух местах: в выражении `if` и в методе `println`.

Создание новой коллекции

Хотя во всех рассмотренных до сих пор примерах вы работали со значениями, получаемыми при обходе элементов, после чего о них уже не вспоминали, у вас есть возможность создать значение, чтобы запомнить результат каждой итерации.

Для этого нужно, как описано в шаге 12 главы 3, перед телом выражения `for` поставить ключевое слово `yield`. Рассмотрим, к примеру, функцию, которая определяет файлы с расширением `.scala` и сохраняет их имена в массиве:

```
def scalaFiles =  
  for  
    file <- filesHere  
    if file.getName.endsWith(".scala")  
  yield file
```

При каждом выполнении тела выражения `for` создается одно значение, в данном случае это просто `file`. Когда выполнение выражения `for` завершится, результат будет включать все выданные значения, содержащиеся в единой коллекции. Тип получающейся коллекции задается на основе вида коллекции, обрабатываемой операторами итерации. В данном случае результат будет иметь тип `Array[File]`, поскольку `filesHere` является массивом, а выдаваемые выражением значения относятся к типу `File`.

В качестве другого примера выражение `for`, показанное в листинге 7.10, сначала преобразует объект типа `Array[File]` по имени `filesHere`, в котором содержатся имена всех файлов, которые есть в текущем каталоге, в объект, содержащий только имена файлов с расширением `.scala`. Для каждого из элементов создается объект типа `Array[String]`, являющийся результатом выполнения метода `fileLines`, определение которого показано в листинге 7.8. Каждый элемент этого объекта `Array[String]` содержит одну строку из текущего обрабатываемого файла. Данный объект превращается в другой объект типа `Array[String]`, содержащий только те строки, обработанные методом `trim`, которые включают подстроку `"for"`. И наконец, для каждого из них выдается целочисленное значение длины. Результатом этого выражения `for` становится объект, имеющий тип `Array[Int]` и содержащий эти значения длины.

Листинг 7.10. Преобразование объекта типа `Array[File]` в объект типа `Array[Int]` с помощью выражения `for`

```
val forLineLengths =  
  for  
    file <- filesHere  
    if file.getName.endsWith(".scala")
```

```
line <- fileLines(file)
trimmed = line.trim
if trimmed.matches(".*for.*")
yield trimmed.length
```

Итак, основные свойства выражения `for`, применяемого в Scala, рассмотрены, но мы прошли по ним слишком поверхностно.

7.4. Обработка исключений с помощью выражений try

Исключения в Scala ведут себя практически так же, как во многих других языках. Вместо того чтобы возвращать значение (как это обычно происходит), метод может прервать работу с генерацией исключения. Код, вызвавший метод, может либо перехватить и обработать данное исключение, либо прекратить собственное выполнение, при этом передав исключение тому коду, который его вызвал. Исключение распространяется подобным образом, раскручивая стек вызова, до тех пор, пока не встретится обрабатывающий его метод или вообще не останется ни одного метода.

Генерация исключений

Генерация исключений в Scala выглядит так же, как в Java. Создается объект исключения, который затем бросается с помощью ключевого слова `throw`:

```
throw new IllegalArgumentException
```

Как бы парадоксально это ни звучало, в Scala `throw` — это выражение, у которого есть результирующий тип. Рассмотрим пример, где этот тип играет важную роль:

```
def half(n: Int) =
  if n % 2 == 0 then
    n / 2
  else
    throw new RuntimeException("n must be even")
```

Здесь получается, что если `n` — четное число, то переменная `half` вернет половину от `n`. Если нечетное, то исключение сгенерируется еще до того, как переменная `half` вернет какое-либо значение. Поэтому генерируемое исключение можно без малейших опасений рассматривать как абсолютно любое

значение. Любой контекст, который пытается воспользоваться значением, возвращаемым из `throw`, никогда не сможет этого сделать, и потому никакого вреда ожидать не приходится.

С технической точки зрения сгенерированное исключение имеет тип `Nothing`. Генерацией исключения можно воспользоваться, даже если оно никогда и ни во что не будет вычислено. Подобные технические нюансы могут показаться несколько странными, но в случаях, подобных предыдущему примеру, зачастую могут пригодиться. Одно ответвление от `if` вычисляет значение, а другое выдает исключение и вычисляется в `Nothing`. Тогда типом всего выражения `if` является тип, вычисленный в той ветви, в которой проводилось вычисление. Тип `Nothing` дополнительно будет рассмотрен в разделе 17.3.

Перехват исключений

Перехват исключений выполняется с применением синтаксиса, показанного в листинге 7.11. Для выражений `catch` был выбран синтаксис с прицелом на совместимость с весьма важной частью Scala — *сопоставлением с образцом*. Этот механизм — весьма эффективное средство, которое вкратце рассматривается в данной главе, а более подробно — в главе 13.

Листинг 7.11. Применение в Scala конструкции `try-catch`

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

try
  val f = new FileReader("input.txt")
  // использование и закрытие файла
catch
  case ex: FileNotFoundException => // обработка ошибки отсутствия файла
  case ex: IOException => // обработка других ошибок ввода-вывода
```

Поведение данного выражения `try-catch` ничем не отличается от его поведения в других языках, использующих исключения. Если при выполнении тела генерируется исключение, то по очереди предпринимается попытка выполнить каждый вариант `case`. Если в данном примере исключение имеет тип `FileNotFoundException`, то будет выполнено первое условие, если тип `IOException` — то второе. Если исключение не относится ни к одному из этих типов, то выражение `try-catch` прервет свое выполнение и исключение будет распространено далее.

ПРИМЕЧАНИЕ

Одно из отличий Scala от Java, которое довольно просто заметить, заключается в том, что язык Scala не требует от вас перехватывать проверяемые исключения или их объявления в условии генерации исключений. При необходимости условие генерации исключений можно объявить с помощью аннотации `@throws`, но делать это не обязательно. Дополнительную информацию о `@throws` можно найти в разделе 9.2.

Условие finally

Если нужно, чтобы некий код выполнялся независимо от того, как именно завершилось выполнение выражения, то можно воспользоваться условием `finally`, заключив в него этот код. Например, может понадобиться гарантированное закрытие открытого файла, даже если выход из метода произошел с генерацией исключения. Пример показан в листинге 7.12¹.

Листинг 7.12. Применение в Scala условия `try-finally`

```
import java.io.FileReader

val file = new FileReader("input.txt")
try
  println(file.read()) // использование файла
finally
  file.close() // гарантированное закрытие файла
```

ПРИМЕЧАНИЕ

В листинге 7.12 показан характерный для языка способ гарантированного закрытия ресурса, не имеющего отношения к оперативной памяти, например файла, сокета или подключения к базе данных. Сначала вы получаете ресурс. Затем запускается на выполнение блок `try`, в котором используется этот ресурс. И наконец, вы закрываете ресурс в блоке `finally`. В качестве альтернативного варианта достичь той же цели более лаконичным способом в Scala можно с помощью технологии под названием «шаблон временного пользования» (`loan pattern`). Он будет рассмотрен в разделе 9.4.

Выдача значения

Как и большинство других управляющих конструкций Scala, `try-catch-finally` выдает значение. Например, в листинге 7.13 показано, как можно

¹ Хотя инструкции оператора `catch` всегда нужно окружать фигурными скобками или делать отступы в блоке, `try` и `finally` не требуют использования фигурных скобок, если в них содержится только одно выражение. Например, можно написать: `try t() catch { case e: Exception => ... } finally f()`.

попытаться разобрать URL, но при этом воспользоваться значением по умолчанию в случае плохого формирования URL. Результат получается при выполнении условия `try`, если не генерируется исключение, или же при выполнении связанного с ним условия `catch`, если исключение генерируется и перехватывается. Значение, вычисленное в условии `finally`, при наличии такого, отбрасывается. Как правило, условия `finally` выполняют какую-либо подчистку, например закрытие файла. Обычно они не должны изменять значение, вычисленное в основном теле или в `catch`-условии, связанном с `try`.

Листинг 7.13. Условие `catch`, выдающее значение

```
import java.net.URL
import java.net.MalformedURLException

def urlFor(path: String) =
  try new URL(path)
  catch case e: MalformedURLException =>
    new URL("http://www.scala-lang.org")
```

Если вы знакомы с Java, то стоит отметить, что поведение Scala отличается от поведения Java только тем, что используемая в Java конструкция `try-finally` не возвращает в результате никакое значение. Как и в Java, если в условии `finally` включена в явном виде инструкция возвращения значения `return` или же в нем генерируется исключение, то это возвращаемое значение или исключение будут перевешивать все ранее выданное `try` или одним из его условий `catch`. Например, если взять вот такое несколько надуманное определение функции:

```
def f(): Int = try return 1 finally return 2
```

то при вызове `f()` будет получен результат 2. Для сравнения, если взять определение

```
def g(): Int = try 1 finally 2
```

то при вызове `g()` будет получен результат 1. Обе функции демонстрируют поведение, которое может удивить большинство программистов, поэтому все же лучше обойтись без значений, возвращаемых из условий `finally`. Условие `finally` более предпочтительно считать способом, который гарантирует выполнение какого-либо побочного эффекта, например закрытие открытого файла.

7.5. Выражения `match`

Используемое в Scala выражение сопоставления `match` позволяет выбрать из нескольких *альтернатив* (вариантов), как это делается в других языках с по-

мощью инструкции `switch`. В общем, выражение `match` позволяет задействовать произвольные *шаблоны*, которые будут рассмотрены в главе 13. Общая форма может подождать. А пока нужно просто рассматривать использование `match` для выбора среди ряда альтернатив.

В качестве примера скрипт, показанный в листинге 7.14, считывает из списка аргументов название пищевого продукта и выводит пару к нему. Это выражение `match` анализирует значение переменной `firstArg`, которое установлено на первый аргумент, извлеченный из списка аргументов. Если это строковое значение `"salt"` (соль), то оно выводит `"pepper"` (перец), а если это `"chips"` (чипсы), то `"salsa"` (острый соус) и т. д. Вариант по умолчанию указывается с помощью знака подчеркивания (`_`), который является подстановочным символом, часто используемым в Scala в качестве заместителя для неизвестного значения.

Листинг 7.14. Выражение сопоставления с побочными эффектами

```
val firstArg = if !args.isEmpty then args(0) else ""

firstArg match
  case "salt" => println("pepper")
  case "chips" => println("salsa")
  case "eggs" => println("bacon")
  case _ => println("huh?")
```

Есть несколько важных отличий от используемой в Java инструкции `switch`. Одно из них заключается в том, что в `case`-инструкциях Scala наряду с прочим могут применяться любые разновидности констант, а не только константы целочисленного типа, перечисления или строковые константы, как в `case`-инструкциях Java. В представленном выше листинге в качестве альтернатив используются строки. Еще одно отличие заключается в том, что в конце каждой альтернативы нет инструкции `break`. Она присутствует неявно, и нет «выпадения» (fall through) с одной альтернативы на следующую. Общий случай — без «выпадения» — становится короче, а частых ошибок удастся избежать, поскольку программисты теперь не «выпадают» нечаянно.

Но, возможно, наиболее существенным отличием от `switch`-инструкции является то, что выражения сопоставления дают значение. В предыдущем примере в каждой альтернативе в выражении сопоставления на стандартное устройство выводится значение. Как показано в листинге 7.15, данный вариант будет работать так же хорошо и выдавать значение вместо того, чтобы выводить его на устройство. Значение, получаемое из этого выражения сопоставления, сохраняется в переменной `friend`. Кроме того, что код становится короче (по крайней мере на несколько символов), теперь он выполняет две

отдельные задачи: сначала выбирает продукт питания, а затем выводит его на устройство.

Листинг 7.15. Выражение сопоставления, выдающее значение

```
val firstArg = if !args.isEmpty then args(0) else ""

val friend =
  firstArg match
    case "salt" => "pepper"
    case "chips" => "salsa"
    case "eggs" => "bacon"
    case _ => "huh?"

println(friend)
```

7.6. Программирование без `break` и `continue`

Вероятно, вы заметили, что здесь не упоминались ни `break`, ни `continue`. Из Scala эти инструкции исключены, поскольку плохо сочетаются с функциональными литералами, которые описываются в следующей главе. Назначение инструкции `continue` в цикле `while` понятно, но что она будет означать внутри функционального литерала? Так как в Scala поддерживаются оба стиля программирования — императивный, и функциональный, — в данном случае из-за упрощения языка прослеживается небольшой перекосяк в сторону функционального программирования. Но волноваться не стоит. Существует множество способов писать программы, не прибегая к `break` и `continue`, и если воспользоваться функциональными литералами, то варианты с ними зачастую могут быть короче первоначального кода.

Простейший подход заключается в замене каждой инструкции `continue` условием `if`, а каждой инструкции `break` — булевой переменной. Последняя показывает, должен ли продолжаться охватывающий цикл `while`. Предположим, ведется поиск в списке аргументов строки, которая заканчивается на `.scala`, но не начинается с дефиса. В Java можно, отдавая предпочтение циклам `while`, а также инструкциям `break` и `continue`, написать следующий код:

```
int i = 0;    // Это код Java
boolean foundIt = false;
while (i < args.length) {
  if (args[i].startsWith("-")) {
    i = i + 1;
    continue;
  }
  if (args[i].endsWith(".scala")) {
```

```

        foundIt = true;
        break;
    }
    i = i + 1;
}

```

Данный фрагмент на Java можно перекодировать непосредственно в код Scala. Для этого вместо того, чтобы использовать условие `if` с последующей инструкцией `continue`, можно написать условие `if`, охватывающее всю оставшуюся часть цикла `while`. Чтобы избавиться от `break`, обычно добавляют булеву переменную, которая указывает на необходимость продолжения, но в данном случае можно задействовать уже существующую переменную `foundIt`. При использовании этих двух приемов код приобретает вид, показанный в листинге 7.16.

Листинг 7.16. Выполнение цикла без break или continue

```

var i = 0
var foundIt = false

while i < args.length && !foundIt do
  if !args(i).startsWith("-") then
    if args(i).endsWith(".scala") then
      foundIt = true
    else
      i = i + 1
  else
    i = i + 1

```

Код Scala, показанный в листинге 7.16, очень похож на первоначальный код Java. Основные части остались на месте и располагаются в том же порядке. Используются две переназначаемые переменные и цикл `while`. Внутри цикла выполняются проверки того, что `i` меньше `args.length`, а также наличия `"-"` и `".scala"`.

Если в коде листинга 7.16 нужно избавиться от `var`-переменных, то можно попробовать применить один из подходов, заключающийся в переписывании цикла в рекурсивную функцию. Можно, к примеру, определить функцию `searchFrom`, которая получает на входе целочисленное значение, выполняет поиск с указанной им позиции, а затем возвращает индекс желаемого аргумента. При использовании данного приема код приобретет вид, показанный в листинге 7.17.

Листинг 7.17. Рекурсивная альтернатива циклу с применением var-переменных

```

def searchFrom(i: Int): Int =
  if i >= args.length then -1

```

```
else if args(i).startsWith("-") then searchFrom(i + 1)
else if args(i).endsWith(".scala") then i
else searchFrom(i + 1)
```

```
val i = searchFrom(0)
```

В версии, показанной в данном листинге, в имени функции отображено ее назначение, изложенное в понятной человеку форме, а в качестве замены цикла применяется рекурсия. Каждая инструкция `continue` заменена рекурсивным вызовом, в котором в качестве аргумента используется выражение `i + 1`, позволяющее эффективно переходить к следующему целочисленному значению. Многие программисты, привыкшие к рекурсиям, считают этот стиль программирования более наглядным.

ПРИМЕЧАНИЕ

В действительности компилятор Scala не будет выдавать для кода, показанного в листинге 7.17, рекурсивную функцию. Поскольку все рекурсивные вызовы находятся в хвостовой позиции, то компилятор создаст код, похожий на цикл `while`. Каждый рекурсивный вызов будет реализован как возврат к началу функции. Оптимизация хвостовых вызовов рассматривается в разделе 8.10.

7.7. Область видимости переменных

Теперь, когда стала понятна суть встроенных управляющих конструкций Scala, мы воспользуемся ими, чтобы объяснить, как в этом языке работает область видимости.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ ДЛЯ JAVA-ПРОГРАММИСТОВ

Если вы программировали на Java, то увидите, что правила области видимости в Scala почти идентичны правилам, которые действуют в Java. Единственное различие между языками состоит в том, что в Scala позволяет определять переменные с одинаковыми именами во вложенных областях видимости. Поэтому Java-программистам может быть интересно по крайней мере бегло просмотреть данный раздел.

Область видимости — это область программы в Scala, в пределах которой идентификатор некоторой переменной продолжает быть связанным с этой переменной и возвращать ее значение. Самый распространенный пример определения области видимости — применение отступа. Все, что находится на данном уровне отступа, теряет видимость за его пределами. Рассмотрим в качестве иллюстрации функцию, показанную в листинге 7.18.

Листинг 7.18. Область видимости переменных при выводе таблицы умножения

```
def printMultiTable() =
  var i = 1
  // видима только i

  while i <= 10 do

    var j = 1
    // видимы i и j

    while j <= 10 do

      val prod = (i * j).toString
      // видимы i, j и prod

      var k = prod.length
      // видимы i, j, prod и k

      while k < 4 do
        print(" ")
        k += 1

      print(prod)
      j += 1

    // i и j все еще видимы; prod и k – нет

    println()
    i += 1

  // i все еще видима; j, prod и k – нет
```

Показанная здесь функция `printMultiTable` выводит таблицу умножения¹. В первой инструкции этой функции вводится переменная `i`, которая инициализируется целым числом 1. Затем имя `i` можно использовать во всей остальной части функции.

Следующая инструкция в `printMultiTable` является циклом `while`:

```
while i <= 10 do
  var j = 1
  ...
```

Переменная `i` может использоваться здесь, поскольку по-прежнему находится в области видимости. В первой инструкции внутри цикла `while` вводится

¹ Функция `printMultiTable`, показанная в листинге 7.18, написана в императивном стиле. В следующем разделе мы преобразуем его в функциональный стиль.

еще одна переменная, которой дается имя `j`, и она также инициализируется значением `1`. Так как переменная `j` была определена внутри отступов цикла `while`, она может использоваться только внутри данного цикла `while`. При попытке что-либо сделать с `j` в конце цикла `while` после комментария, сообщающего, что `j`, `prod` и `k` уже вне области видимости, ваша программа не будет скомпилирована.

Все переменные, определенные в этом примере: `i`, `j`, `prod` и `k` — локальные. Они локальны по отношению к функциям, в которых определены. При каждом вызове функции используется новый набор локальных переменных.

После определения переменной определить новую переменную с таким же именем в той же области видимости уже нельзя. Например, следующий скрипт с двумя переменными по имени `a` в одной и той же области видимости скомпилирован не будет:

```
val a = 1
val a = 2 // Не скомпилируется
println(a)
```

В то же время во внешней области видимости вполне возможно определить переменную с точно таким же именем, что и во внутренней области видимости. Следующий скрипт будет скомпилирован и сможет быть запущен:

```
val a = 1;
if a == 1 then
  val a = 2 // Компилируется без проблем
  println(a)
println(a)
```

Данный скрипт при выполнении выведет `2`, а затем `1`, поскольку переменная `a`, определенная внутри выражения `if`, — это уже другая переменная, область видимости которой распространяется только до конца блока с отступами¹. Следует отметить одно различие между Scala и Java. Оно состоит в том, что Java не позволит создать во внутренней области видимости переменную, имя которой совпадает с именем переменной во внешней области видимости. В программе на Scala внутренняя переменная, как говорят, *перекрывает* внешнюю переменную с точно таким же именем, поскольку внешняя переменная становится невидимой во внутренней области видимости.

¹ Кстати, в данном случае после первого определения нужно поставить точку с запятой, поскольку в противном случае действующий в Scala механизм, который подразумевает их использование, не сработает.

Вы уже, вероятно, замечали что-либо подобное эффекту перекрытия в REPL:

```
scala> val a = 1
a: Int = 1

scala> val a = 2
a: Int = 2

scala> println(a)
2
```

Там имена переменных можно использовать повторно как вам угодно. Среди прочего это позволяет вам передумать, если при первом определении переменной в REPL была допущена ошибка. Подобная возможность появляется благодаря тому, что REPL концептуально создает для каждой введенной вами инструкции новую вложенную область видимости.

Следует помнить: отслеживание может сильно запутать читателей, поскольку имена переменных приобретают во вложенных областях видимости совершенно новый смысл. Зачастую вместо того, чтобы перекрывать внешнюю переменную, лучше выбрать для переменной новое узнаваемое имя.

7.8. Рефакторинг кода, написанного в императивном стиле

Чтобы помочь вам вникнуть в функциональный стиль, в данном разделе мы проведем рефакторинг императивного подхода к выводу таблицы умножения, показанной в листинге 7.18. Наша функциональная альтернатива представлена в листинге 7.19.

Листинг 7.19. Функциональный способ создания таблицы умножения

```
// возвращение строки в виде последовательности
def makeRowSeq(row: Int) =
  for col <- 1 to 10 yield
    val prod = (row * col).toString
    val padding = " " * (4 - prod.length)
    padding + prod

// возвращение строки в виде строкового значения
def makeRow(row: Int) = makeRowSeq(row).mkString

// возвращение таблицы в виде строковых значений, по одному значению
// на каждую строчку
```

```
def multiTable() =  
    val tableSeq = // последовательность строк из строчек таблицы  
        for row <- 1 to 10  
        yield makeRow(row)  
  
    tableSeq.mkString("\n")
```

На наличие в листинге 7.18 императивного стиля указывают два момента. Первый — побочный эффект от вызова `printMultiTable` — вывод таблицы умножения на стандартное устройство. В листинге 7.19 функция реорганизована таким образом, чтобы возвращать таблицу умножения в виде строкового значения. Поскольку функция больше не занимается выводом на стандартное устройство, то переименована в `multiTable`. Как уже упоминалось, одно из преимуществ функций, не имеющих побочных эффектов, — упрощение их модульного тестирования. Для тестирования `printMultiTable` понадобилось бы каким-то образом переопределять `print` и `println`, чтобы можно было проверить вывод на корректность. А протестировать `multiTable` гораздо проще — проверив ее строковой результат.

Второй момент, служащий верным признаком императивного стиля в функции `printMultiTable` — ее цикл `while` и `var`-переменные. В отличие от этого, в функции `multiTable` для выражений, вспомогательных *функций* и вызовов `mkString` используются `val`-переменные.

Чтобы облегчить чтение кода, мы выделили две вспомогательные функции: `makeRowSeq` и `makeRow`. Первая использует выражение `for`, генератор которого перебирает номера столбцов от 1 до 10. В теле этого выражения вычисляется произведение значения строки на значение столбца, определяется отступ, необходимый для произведения, выдается результат объединения строк отступа и произведения. Результатом выражения `for` будет последовательность (один из подклассов `Seq`), содержащая выданные строки в качестве элементов. Вторая вспомогательная функция, `makeRow`, просто вызывает метод `mkString` в отношении результата, возвращенного функцией `makeRowSeq`. Этот метод объединяет имеющиеся в последовательности строки, возвращая их в виде одной строки.

Метод `multiTable` сначала инициализирует `tableSeq` результатом выполнения выражения `for`, генератор которого перебирает числа от 1 до 10, чтобы для каждого вызова `makeRow` получалось строковое значение для данной строки таблицы. Именно эта строка и выдается, вследствие чего результатом выполнения данного выражения `for` будет последовательность строковых значений, представляющих строки таблицы. Остается лишь преобразовать последовательность строк в одну строку. Для выполнения этой задачи вы-

зывается метод `mkString`, и, поскольку ему передается значение `"\n"`, мы получаем символ конца строки, вставленный после каждой строки. Передав строку, возвращенную `multiTable`, функции `println`, вы увидите, что выводится точно такая же таблица, как и при вызове функции `printMultiTable`:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

7.9. Резюме

Перечень встроенных в Scala управляющих конструкций минимален, но они вполне справляются со своими задачами. Их работа похожа на действия их императивных эквивалентов, но, поскольку им свойственно выдавать значение, они поддерживают и функциональный стиль. Не менее важно и то, что управляющие конструкции оставляют поле деятельности для одной из самых эффективных возможностей Scala — функциональных литералов, которые рассматриваются в следующей главе.

8

Функции и замыкания

По мере роста программ появляется необходимость разбивать их на небольшие части, которыми удобнее управлять. Разделение потока управления в Scala реализуется с помощью подхода, знакомого всем опытным программистам: разделение кода на функции. Фактически в Scala предлагается несколько способов определения функций, которых нет в Java. Кроме методов, представляющих собой функции, являющиеся частью объектов, есть также функции, вложенные в другие функции, функциональные литералы и функциональные значения. В данной главе вам предстоит познакомиться со всеми этими разновидностями функций, имеющимися в Scala.

8.1. Методы

Наиболее распространенный способ определения функций — включение их в состав объекта. Такая функция называется *методом*. В качестве примера в листинге 8.1 показаны два метода, которые вместе считывают данные из файла с заданным именем и выводят строки, длина которых превышает заданную. Перед каждой выведенной строкой указывается имя файла, в котором она появляется.

Листинг 8.1. LongLines с приватным методом processLine

object Padding:

```
def padLines(text: String, minWidth: Int): String =
  val paddedLines =
    for line <- text.linesIterator yield
      padLine(line, minWidth)
  paddedLines.mkString("\n")
```

```
private def padLine(line: String, minWidth: Int): String =  
  if line.length >= minWidth then line  
  else line + " " * (minWidth - line.length)
```

Метод `padLines` принимает в качестве параметров `text` и `minWidth`. Он вызывает `linesIterator` для `text`, который возвращает итератор строк в типе данных `string`, исключая любые символы окончания строки. Выражение `for` обрабатывает каждую из этих строк, вызывая вспомогательный метод `padLine`. Метод `padLine` принимает два параметра: `minWidth` и `line`. Он сравнивает длину строки с заданной шириной и, если длина меньше, добавляет соответствующее количество пробелов в конец строки, чтобы их уравнять. Пока что это очень похоже на то, что вы делаете в любом объектно-ориентированном языке. Однако понятие функции в `Scala` является более общим, чем метод. Другие способы выражения функций в `Scala` будут объяснены в следующих разделах.

8.2. Локальные функции

Конструкция метода `padLines` из предыдущего раздела показала важность принципов разработки, присущих функциональному стилю программирования: программы должны быть разбиты на множество небольших функций с четко определенными задачами. Зачастую отдельно взятая функция весьма невелика. Преимущество подобного стиля — в предоставлении программисту множества строительных блоков, позволяющих составлять гибкие композиции для решения более сложных задач. Такие строительные блоки должны быть довольно простыми, чтобы с ними было легче разобратся по отдельности.

Подобный подход выявляет одну проблему: имена всех вспомогательных функций могут засорять пространство имен программы. В `REPL` это проявляется не так ярко, но по мере упаковки функций в многократно применяемые классы и объекты желательно будет скрыть вспомогательные функции от пользователей класса. Зачастую, будучи отдельно взятыми, такие функции не имеют особого смысла, и возникает желание сохранять достаточную степень гибкости, чтобы можно было удалить вспомогательные функции, если позже класс будет переписан.

В `Java` основной инструмент для этого — приватный метод. Как было показано в листинге 8.1, точно такой же подход с использованием приватного метода работает и в `Scala`, но в этом языке предлагается и еще один: можно определить функцию внутри другой функции. Подобно локальным

переменным, такие локальные функции видны только в пределах своего приватного блока. Рассмотрим пример:

```
def padLines(text: String, minWidth: Int): String =
  def padLine(line: String, minWidth: Int): String =
    if line.length >= minWidth then line
    else line + " " * (minWidth - line.length)
  val paddedLines =
    for line <- text.linesIterator yield
      padLine(line, minWidth)
  paddedLines.mkString("\n")
```

Здесь реорганизована исходная версия `Padding`, показанная в листинге 8.1: приватный метод `padLine` был превращен в локальную функцию для `padLines`. Для этого был удален модификатор `private`, который мог быть применен (и нужен) только для членов класса, а определение функции `padLine` было помещено внутрь определения функции `padLines`. В качестве локальной функция `padLine` находится в области видимости внутри функции `padLines`, за пределами которой она недоступна.

Но теперь, когда функция `padLine` определена внутри функции `padLines`, появилась возможность улучшить кое-что еще. Вы заметили, что `minWidth` передается вспомогательной функции, как и прежде? В этом нет никакой необходимости, поскольку локальные функции могут получать доступ к параметрам охватывающей их функции. Как показано в листинге 8.2, можно просто воспользоваться параметрами внешней функции `padLines`.

Листинг 8.2. `LongLines` с локальной функцией `processLine`
object `Padding`:

```
def padLines(text: String, minWidth: Int): String =

  def padLine(line: String): String =
    if line.length >= minWidth then line
    else line + " " * (minWidth - line.length)

  val paddedLines =
    for line <- text.linesIterator yield
      padLine(line)

  paddedLines.mkString("\n")
```

Заметили, как упростился код? Такое использование параметров охватывающей функции — широко распространенный и весьма полезный пример универсальной вложенности, предоставляемой Scala. Вложенность и области видимости применительно ко всем конструкциям данного языка, включая

функции, рассматриваются в разделе 7.7. Это довольно простой, но весьма эффективный принцип.

8.3. Функции первого класса

В Scala есть *функции первого класса*. Вы можете не только определить их и вызвать, но и записать в виде безымянных *литералов*, после чего передать их в качестве *значений*. Понятие функциональных литералов было введено в главе 2, а их основной синтаксис показан на рис. 2.2.

Функциональный литерал компилируется в дескрипторе методов Java, который при создании экземпляра во время выполнения программы становится *функциональным значением*¹. Таким образом, разница между функциональными литералами и значениями состоит в том, что первые существуют в исходном коде, а вторые — в виде объектов во время выполнения программы. Эта разница во многом похожа на разницу между классами (исходным кодом) и объектами (создаваемыми во время выполнения программы).

Простой функциональный литерал, прибавляющий к числу единицу, имеет следующий вид:

```
(x: Int) => x + 1
```

Сочетание символов `=>` указывает на то, что эта функция превращает стоящий слева от данного сочетания параметр (любое целочисленное значение `x`) в результат вычисления выражения `(x + 1)`. Таким образом, данная функция отображает на любую целочисленную переменную `x` значение `x + 1`.

Функциональные значения — это объекты, следовательно, при желании их можно хранить в переменных. Они также являются функциями, следовательно, вы можете вызывать их, используя обычную форму записи вызова функций с применением круглых скобок. Вот как выглядят примеры обоих действий:

```
val increase = (x: Int) => x + 1
increase(10) // 11
```

¹ Каждое функциональное значение является экземпляром какого-нибудь класса, который представляет собой расширение одного из нескольких трейтов `FunctionN` в пакете `Scala`, например, `Function0` для функций без параметров, `Function1` для функций с одним параметром и т. д. В каждом трейте `FunctionN` имеется метод `apply`, используемый для вызова функции.

Если нужно, чтобы в функциональном литерале использовалось более одной инструкции, то следует заключить его тело в фигурные скобки и поместить каждую инструкцию на отдельной строке, сформировав блок. Как и в случае создания метода, когда вызывается функциональное значение, будут выполнены все инструкции и значением, возвращаемым из функции, станет значение, получаемое при вычислении последнего выражения:

```
val addTwo = (x: Int) =>
  val increment = 2
  x + increment
addTwo(10) // 12
```

Итак, вы увидели все основные составляющие функциональных литералов и функциональных значений. Возможности их применения обеспечиваются многими библиотеками Scala. Например, методом `foreach`, доступным для всех коллекций¹. Он получает функцию в качестве аргумента и вызывает ее в отношении каждого элемента своей коллекции. А вот как его можно использовать для вывода всех элементов списка:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
val someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10
```

В другом примере также используется имеющийся у типов коллекций метод `filter`. Он выбирает те элементы коллекции, которые проходят выполняемую пользователем проверку с применением функции. Например, фильтрацию можно осуществить с помощью функции `(x: Int) => x > 0`. Она отображает положительные целые числа в `true`, а все остальные числа — в `false`. Метод `filter` можно задействовать следующим образом:

```
scala> someNumbers.filter((x: Int) => x > 0)
val res4: List[Int] = List(5, 10)
```

Более подробно о методах, подобных `foreach` и `filter`, поговорим позже. В главе 14 рассматривается их использование в классе `List`, а в главе 15 — применение с другими типами коллекций.

¹ Метод `foreach` определен в трейте `Iterable`, который является супертрейтом для `List`, `Set`, `Array` и `Map`. Подробности — в главе 15.

8.4. Краткие формы функциональных литералов

В Scala есть несколько способов избавления от избыточной информации, позволяющих записывать функциональные литералы более кратко. Не упускайте такой возможности, поскольку она позволяет вам убрать из своего кода ненужный хлам. Один из способов писать функциональные литералы более лаконично заключается в отбрасывании типов параметров. При этом предыдущий пример с фильтром может быть написан следующим образом:

```
scala> someNumbers.filter((x) => x > 0)
val res5: List[Int] = List(5, 10)
```

Компилятор Scala знает, что переменная *x* должна относиться к целым числам, поскольку видит, что вы сразу же применяете функцию для фильтрации списка целых чисел, на который ссылается объект `someNumbers`. Это называется *целевой типизацией*, поскольку целевому использованию выражения — в данном случае в качестве аргумента для `someNumbers.filter()` — разрешено влиять на типизацию выражения — в данном случае на определение типа параметра *x*. Подробности целевой типизации нам сейчас неважны. Вы можете просто приступить к написанию функционального литерала без типа аргумента и, если компилятор не сможет в нем разобраться, добавить тип. Со временем вы начнете понимать, в каких ситуациях компилятор сможет решить эту загадку, а в каких — нет.

Второй способ избавления от избыточных символов заключается в отказе от круглых скобок вокруг параметра, тип которого будет выведен автоматически. В предыдущем примере круглые скобки вокруг *x* совершенно излишни:

```
scala> someNumbers.filter(x => x > 0)
val res6: List[Int] = List(5, 10)
```

8.5. Синтаксис заместителя

Можно сделать функциональный литерал еще короче, воспользовавшись знаком подчеркивания в качестве заместителя для одного или нескольких параметров при условии, что каждый параметр появляется внутри функционального литерала только один раз. Например, `_ > 0` — очень краткая форма записи для функции, проверяющей, что значение больше нуля:

```
scala> someNumbers.filter(_ > 0)
val res7: List[Int] = List(5, 10)
```

Знак подчеркивания можно рассматривать как бланк, который следует заполнить. Он будет заполнен аргументом функции при каждом ее вызове. Например, при условии, что переменная `someNumbers` была здесь инициализирована значением `List(-11, -10, -5, 0, 5, 10)`, метод `filter` заменит бланк в `_ > 0` сначала значением `-11`, получив `-11 > 0`, затем значением `-10`, получив `-10 > 0`, затем значением `-5`, получив `-5 > 0`, и так далее до конца списка `List`. Таким образом, функциональный литерал `_ > 0` является, как здесь показано, эквивалентом немного более пространныго литерала `x => x > 0`:

```
scala> someNumbers.filter(x => x > 0)
val res8: List[Int] = List(5, 10)
```

Иногда при использовании знаков подчеркивания в качестве заместителей параметров у компилятора может оказаться недостаточно информации для вывода неуказанных типов параметров. Предположим, к примеру, что вы сами написали `_ + _`:

```
scala> val f = _ + _
           error: missing parameter type for expanded function
           ((x$1: <error>, x$2) => x$1.$plus(x$2))
```

В таких случаях нужно указать типы, используя двоеточие:

```
scala> val f = (_: Int) + (_: Int)
val f: (Int, Int) => Int = $$Lambda$1075/1481958694@289ffff3c
scala> f(5, 10)
val res9: Int = 15
```

Следует заметить, что `_ + _` расширяется в литерал для функции, получающей два параметра. Поэтому сокращенную форму можно использовать, только если каждый параметр применяется в функциональном литерале не более одного раза. Несколько знаков подчеркивания означают наличие нескольких параметров, а не многократное использование одного и того же параметра. Первый знак подчеркивания представляет первый параметр, второй знак — второй параметр, третий знак — третий параметр и т. д.

8.6. Частично примененные функции

В Scala, когда при вызове функции в нее передаются любые необходимые аргументы, вы *применяете* эту функцию к этим аргументам. Например, если есть следующая функция:

```
def sum(a: Int, b: Int, c: Int) = a + b + c
```

то функцию `sum` можно применить к аргументам 1, 2 и 3 таким образом:

```
sum(1, 2, 3) // 6
```

Когда вы используете синтаксис заполнителя, в котором каждое подчеркивание используется для пересылки параметра в метод, вы пишете частично применяемую функцию. Частично примененная функция — выражение, в котором не содержатся все аргументы, необходимые функции. Вместо этого в ней есть лишь некоторые из них или вообще нет никаких необходимых аргументов. Например, чтобы создать выражение частично применяемой функции, включающее сумму, в которой вы не указываете ни один из трех обязательных аргументов, вы можете использовать подчеркивание для каждого параметра. Полученную функцию затем можно сохранить в переменной. Вот пример:

```
val a = sum(_, _, _) // имеет тип (Int, Int, Int) => Int
```

Если данный код есть, то компилятор Scala создает экземпляр функционального значения, который получает три целочисленных параметра, не указанных в выражении частично примененной функции, `sum (_, _, _)`, и присваивает ссылку на это новое функциональное значение переменной `a`. Когда к этому новому значению применяются три аргумента, оно развернется и вызовет `sum`, передав в нее те же три аргумента:

```
a(1, 2, 3) // 6
```

Происходит следующее: переменная по имени `a` ссылается на объект функционального значения. Это функциональное значение является экземпляром класса, сгенерированного автоматически компилятором Scala из `sum (_, _, _)` — выражения частично примененной функции. Класс, сгенерированный компилятором, имеет метод `apply`, получающий три аргумента¹. Имеющийся метод `apply` получает три аргумента, поскольку это и есть количество аргументов, отсутствующих в выражении `sum (_, _, _)`. Компилятор Scala транслирует выражение `a(1, 2, 3)` в вызов метода `apply`, принадлежащего объекту функционального значения, передавая ему три аргумента: 1, 2 и 3. Таким образом, `a(1, 2, 3)` — краткая форма следующего кода:

```
a.apply(1, 2, 3) // 6
```

Этот метод `apply`, который определен в автоматически генерируемом компилятором Scala классе из выражения `sum (_, _, _)`, просто передает дальше

¹ Сгенерированный класс является расширением трейта `Function3`, в котором объявлен метод `apply`, предусматривающий использование трех аргументов.

эти три отсутствовавших параметра функции `sum` и возвращает результат. В данном случае метод `apply` вызывает `sum(1, 2, 3)` и возвращает то, что возвращает функция `sum`, то есть число 6.

Данный вид выражений, в которых знак подчеркивания используется для представления всего списка параметров, можно представить себе и по-другому — в качестве способа преобразования `def` в функциональное значение. Например, если имеется локальная функция, скажем, `sum(a: Int, b: Int, c: Int): Int`, то ее можно завернуть в функциональное значение, чей метод `apply` имеет точно такие же типы списка параметров и результата. Это функциональное значение, будучи примененным к неким аргументам, в свою очередь, применяет `sum` для тех же самых аргументов и возвращает результат. Вы не можете присвоить переменной метод или вложенную функцию или передать их в качестве аргументов другой функции. Однако все это можно сделать, если завернуть метод или вложенную функцию в функциональное значение, поместив знак подчеркивания.

Теперь, несмотря на то что `sum (_, _, _)` действительно является частично примененной функцией, вам может быть не вполне понятно, почему она называется именно так. Это потому, что она применяется не ко всем своим аргументам. Что касается `sum (_, _, _)`, то она не применяется *ни к одному* из своих аргументов. Но вы также можете выразить частично примененную функцию, предоставив ей только *некоторые* из требуемых аргументов. Рассмотрим пример:

```
val b = sum(1, _, 3) // b имеет тип Int => Int
```

В данном случае функции `sum` предоставлены первый и последний аргументы, а средний аргумент не указан. Поскольку пропущен только один аргумент, то компилятор Scala сгенерирует новый функциональный класс, чей метод `apply` получает один аргумент. При вызове с этим одним аргументом метод `apply` сгенерированной функции вызывает функцию `sum`, передавая ей 1, затем аргумент, переданный функции, и, наконец, 3. Рассмотрим два примера:

```
b(2) // 6  
b(5) // 9
```

В первом случае `b.apply` вызывает `sum(1, 2, 3)`, а во втором случае `b.apply` вызывает `sum(1, 5, 3)`.

Если функция требуется в конкретном месте кода, то при написании выражения частично примененной функции, в котором не указан ни один параметр,

например `sum (_, _, _)`, его можно записать более кратко для всего списка параметров. Вот пример:

```
val c = sum // c имеет тип (Int, Int, Int) => Int
```

Поскольку `sum` — это имя метода, а не переменной, которая ссылается на значение, компилятор создаст значение функции с той же сигнатурой, что и метод, заключающий в себе его вызов. Этот процесс называется *ета-расширением*. Другими словами, `sum` — это более краткий способ записи `sum (_, _, _)`. Вот пример вызова функции:

```
c(10, 20, 30) // 60
```

8.7. Замыкания

Все рассмотренные до сих пор в этой главе примеры функциональных литералов ссылались только на передаваемые параметры. Так, в выражении `(x: Int) => x > 0` в теле функции `x > 0` использовалась только одна переменная, `x`, которая объявлена как параметр функции. Но вы можете ссылаться на переменные, объявленные и в других местах:

```
(x: Int) => x + more // На сколько больше?
```

Эта функция прибавляет значение переменной `more` к своему аргументу, но что такое `more`? С точки зрения данной функции `more` — *свободная переменная*, поскольку в самом функциональном литерале значение ей не присваивается. В отличие от нее переменная `x` является *связанной*, поскольку в контексте функции имеет значение: определена как единственный параметр функции, имеющий тип `Int`. Если попытаться воспользоваться этим функциональным литералом в чистом виде, без каких-либо определений в его области видимости, то компилятор выразит недовольство:

```
scala> (x: Int) => x + more
1 | (x: Int) => x + more
  |                ^^^^^
  |                Not found: more
```

С другой стороны, тот же функциональный литерал будет нормально работать, пока будет доступно нечто с именем `more`:

```
var more = 1
val addMore = (x: Int) => x + more
addMore(10) // 11
```

Функциональное значение (объект), создаваемое во время выполнения программы из этого функционального литерала, называется *замыканием*. Данное название появилось из-за «замыкания» функционального литерала путем «захвата» привязок его свободных переменных. Функциональный литерал, не имеющий свободных переменных, например $(x: \text{Int}) \Rightarrow x + 1$, называется *замкнутым термом*, где *терм* — это фрагмент исходного кода. Таким образом, функциональное значение, созданное во время выполнения программы из этого функционального литерала, строго говоря, не является замыканием, поскольку функциональный литерал $(x: \text{Int}) \Rightarrow x + 1$ всегда замкнут уже по факту его написания. Но любой функциональный литерал со свободными переменными, например $(x: \text{Int}) \Rightarrow x + \text{more}$, является *открытым термом*. Поэтому любое функциональное значение, созданное во время выполнения программы из $(x: \text{Int}) \Rightarrow x + \text{more}$, будет по определению требовать, чтобы привязка его свободной переменной, `more`, была захвачена. Получившееся функциональное значение, в котором может содержаться ссылка на захваченную переменную `more`, называется замыканием, поскольку функциональное значение — конечный продукт замыкания открытого терма, $(x: \text{Int}) \Rightarrow x + \text{more}$.

Этот пример вызывает вопрос: что случится, если значение `more` изменится после создания замыкания? В Scala можно ответить, что замыкание видит изменение, например:

```
more = 9999
addMore(10) // 10009
```

На интуитивном уровне понятно, что замыкания в Scala перехватывают сами переменные, а не значения, на которые те ссылаются¹. Как показано в предыдущем примере, замыкание, созданное для $(x: \text{Int}) \Rightarrow x + \text{more}$, видит изменение `more` за пределами замыкания. То же самое справедливо и в обратном направлении. Изменения, вносимые в захваченную переменную, видимы за пределами замыкания. Рассмотрим пример:

```
val someNumbers = List(-11, -10, -5, 0, 5, 10)
var sum = 0
someNumbers.foreach(sum += _)
sum // -11
```

¹ Для сравнения: лямбда-выражения Java не позволяют вам обращаться к локальным переменным в окружающих областях, если они не являются окончательными или фактически окончательными, поэтому нет никакой разницы между захватом переменной и захватом ее текущего значения.

Здесь используется обходной способ сложения чисел в списке типа `List`. Переменная `sum` находится в области видимости, охватывающей функциональный литерал `sum += _`, который прибавляет числа к `sum`. Несмотря на то что замыкание модифицирует `sum` во время выполнения программы, получающийся конечный результат `-11` по-прежнему виден за пределами замыкания.

А что, если замыкание обращается к некоей переменной, у которой во время выполнения программы есть несколько копий? Например, что, если замыкание использует локальную переменную некоей функции и последняя вызывается множество раз? Какой из экземпляров этой переменной будет задействован при каждом обращении?

С остальной частью языка согласуется только один ответ: задействуется тот экземпляр, который был активен на момент создания замыкания. Рассмотрим, к примеру, функцию, создающую и возвращающую замыкания прироста значения:

```
def makeIncreaser(more: Int) = (x: Int) => x + more
```

При каждом вызове эта функция будет создавать новое замыкание. Каждое замыкание будет обращаться к той переменной `more`, которая была активна при создании замыкания.

```
val inc1 = makeIncreaser(1)
val inc9999 = makeIncreaser(9999)
```

При вызове `makeIncreaser(1)` создается и возвращается замыкание, захватывающее в качестве привязки к `more` значение `1`. По аналогии с этим при вызове `makeIncreaser(9999)` возвращается замыкание, захватывающее для `more` значение `9999`. Когда эти замыкания применяются к аргументам (в данном случае имеется только один передаваемый аргумент, `x`), получаемый результат зависит от того, как переменная `more` была определена в момент создания замыкания:

```
inc1(10)    // 11
inc9999(10) // 10009
```

И неважно, что `more` в данном случае — параметр вызова метода, из которого уже произошел возврат. В подобных случаях компилятор Scala осуществляет реорганизацию, которая дает возможность захваченным параметрам продолжать существовать в динамической памяти (куче), а не в стеке и пережить таким образом создавший их метод. Данная реорганизация происходит

в автоматическом режиме, поэтому вам о ней не стоит беспокоиться. Захватывайте какую угодно переменную: `val` или `var` или любой параметр¹.

8.8. Специальные формы вызова функций

Большинство встречающихся вам функций и вызовов функций будут аналогичны уже увиденным в этой главе. У функции будет фиксированное число параметров, у вызова будет точно такое же количество аргументов, и аргументы будут указаны в точно таких же порядке и количестве, что и параметры.

Но поскольку вызовы функций в программировании на Scala весьма важны, то для обеспечения некоторых специальных потребностей в язык было добавлено несколько специальных форм определений и вызовов функций. В Scala поддерживаются повторяющиеся параметры, именованные аргументы и аргументы со значениями по умолчанию.

Повторяющиеся параметры

В Scala допускается указание на то, что последний параметр функции может повторяться. Это позволяет клиентам передавать функции список аргументов переменной длины. Чтобы обозначить повторяющийся параметр, поставьте после типа параметра знак звездочки, например:

```
scala> def echo(args: String*) =  
    for arg <- args do println(arg)  
def echo(args: String*): Unit
```

Определенная таким образом функция `echo` может быть вызвана с нулем и большим количеством аргументов типа `String`:

```
scala> echo()  
  
scala> echo("one")  
one  
  
scala> echo("hello", "world!")  
hello  
world!
```

¹ С другой стороны, в функциональном программировании вы будете учитывать только `vals`. Кроме того, в императивном программировании при параллельной разработке захват `vars` может привести к ошибкам параллелизма из-за несинхронизированного доступа к общему изменяемому состоянию.

Внутри функции типом повторяющегося параметра является `Seq` из элементов объявленного типа параметра. Таким образом, типом переменной `args` внутри функции `echo`, которая объявлена как тип `String*`, фактически является `Seq[String]`. Несмотря на это, если у вас имеется массив подходящего типа, при попытке передать его в качестве повторяющегося параметра будет получена ошибка компиляции:

```
scala> val seq = Seq("What's", "up", "doc?")
val seq: Seq[String] = List(What's, up, doc?)
```

```
scala> echo(seq)
1 |echo(seq)
  |      ^^
  |      Found: (seq : Seq[String])
  |      Required: String
```

Чтобы добиться успеха, нужно после аргумента-массива поставить двоеточие, знак подчеркивания и знак звездочки:

```
scala> echo(seq*)
What's
up
doc?
```

Эта форма записи заставит компилятор передать в `echo` каждый элемент массива `seq` в виде самостоятельного аргумента, а не передавать его целиком в виде одного аргумента.

Именованные аргументы

При обычном вызове функции аргументы в вызове поочередно сопоставляются в указанном порядке с параметрами вызываемой функции:

```
def speed(distance: Float, time: Float) = distance / time
speed(100, 10) // 10.0
```

В данном вызове `100` сопоставляется с `distance`, а `10` — с `time`. Сопоставление `100` и `10` производится в том же порядке, в котором перечислены формальные параметры.

Именованные аргументы позволяют передавать аргументы функции в ином порядке. Синтаксис просто предусматривает, что перед каждым аргументом указывается имя параметра и знак равенства. Например, следующий вызов функции `speed` эквивалентен вызову `speed(100, 10)`:

```
speed(distance = 100, time = 10) // 10.0
```

При вызове с именованными аргументами эти аргументы можно поменять местами, не изменяя их значения:

```
speed(time = 10, distance = 100) // 10.0
```

Можно также смешивать позиционные и именованные аргументы. В этом случае сначала указываются позиционные аргументы. Именованные чаще всего используются в сочетании со значениями параметров по умолчанию.

Значения параметров по умолчанию

Scala позволяет указать для параметров функции значения по умолчанию. Аргумент для такого параметра может быть произвольно опущен из вызова функции, в таком случае соответствующий аргумент будет заполнен значением по умолчанию.

Например, если вам нужно создать объект-компаньон для класса `Rational`, показанного в листинге 6.5, вы можете определить фабричный метод `apply`, как показано в листинге 8.3. Функция `apply` имеет два параметра: `denom`, для которого значение по умолчанию равно 1, и `numer`.

Если вызвать функцию `Rational(42)`, то есть без указания аргумента, используемого для `denom`, то для этого параметра будет установлено его значение по умолчанию 1. Можно также вызвать функцию с явно указанным знаменателем. Например, установите знаменатель равным 83, вызвав функцию как `Rational(42, 83)`¹.

Листинг 8.3. Параметр со значением по умолчанию

```
// в том же исходнике, что и класс Rational
object Rational:
  def apply(numer: Int, denom: Int = 1) =
    new Rational(numer, denom)
```

Параметры по умолчанию особенно полезны, когда применяются в сочетании с именованными параметрами. В листинге 8.4 у функции `point` имеются два необязательных параметра: `x` и `y`, значение которых по умолчанию равно 0.

¹ Значение по умолчанию, равное 1, можно было бы также использовать для параметра `d` класса `Rational` в листинге 6.5, как в `class Rational(n: Int, d: Int = 1)`, вместо использования вспомогательного конструктора, который заполняет 1 для `d`.

Листинг 8.4. Функция с двумя параметрами, у которых имеются значения по умолчанию

```
def point(x: Int = 0, y: Int = 0) = (x, y)
```

Чтобы оба параметра заполнялись значениями по умолчанию, функцию `point` можно вызывать как `point()`. Используя именованные аргументы, можно указать любой из параметров, оставив при этом другой по умолчанию. Когда необходимо указать `x`, оставив `y`:

```
point(x = 42)
```

И когда требуется указать `y`, оставив `x`:

```
point(y = 1000)
```

8.9. Тип SAM

Лямбда-выражение в Java можно использовать везде, где требуется экземпляр класса или интерфейса только с одним абстрактным методом (SAM). Java `ActionListener` является таким интерфейсом, потому что он содержит SAM — `actionPerformed`. Таким образом, лямбда-выражение может использоваться для регистрации действий слушателя `Listener` на кнопке `Swing`. Вот пример:

```
JButton button = new JButton(); // Java
button.addActionListener(
    event -> System.out.println("pressed!")
);
```

В той же ситуации при помощи `Scala` вы можете применить экземпляр анонимного внутреннего класса, однако лучше предпочесть функциональный литерал, например:

```
val button = new JButton
button.addActionListener(
    _ => println("pressed!")
)
```

`Scala`, как и `Java`, позволяет применять функциональный тип там, где требуется экземпляр класса или трейта, объявляющий SAM. Это будет работать с любым SAM. Например, вы можете определить признак `Increaser` с помощью одного абстрактного метода `Increase`:

```
trait Increaser:
  def increase(i: Int): Int
```

Затем вы можете определить метод, который примет `Increaser`:

```
def increaseOne(increaser: Increaser): Int =  
    increaser.increase(1)
```

Чтобы вызвать ваш новый метод, необходимо передать анонимный экземпляр типажа `Increaser`, например:

```
increaseOne(  
    new Increaser:  
        def increase(i: Int): Int = i + 7  
    )
```

Однако, начиная с версии 2.12 и выше, в Scala можно просто использовать функциональный литерал, потому что `Increaser` относится к типу SAM:

```
increaseOne(i => i + 7) // Scala
```

8.10. Хвостовая рекурсия

В разделе 7.2 упоминалось, что для преобразования цикла `while`, который обновляет значение `var`-переменных в код более функционального стиля, использующий только `val`-переменные, обычно нужно прибегнуть к рекурсии. Рассмотрим пример рекурсивной функции, которая вычисляет приближенное значение, повторяя уточнение приближительного расчета, пока не будет получен приемлемый результат:

```
def approximate(guess: Double): Double =  
    if isGoodEnough(guess) then guess  
    else approximate(improve(guess))
```

С соответствующими реализациями `isGoodEnough` и `improve` подобные функции часто используются при решении задач поиска. Если нужно, чтобы функция `approximate` выполнялась быстрее, может возникнуть желание написать ее с циклом `while`:

```
def approximateLoop(initialGuess: Double): Double =  
    var guess = initialGuess  
    while !isGoodEnough(guess) do  
        guess = improve(guess)  
    guess
```

Какая из двух версий `approximate` более предпочтительна? Если требуется лаконичность и нужно избавиться от использования `var`-переменных, то выигрывает первый, функциональный вариант. Но, может быть, более эффектив-

ным окажется императивный подход? На самом деле, если измерить время выполнения, окажется, что они практически одинаковы!

Этот результат может показаться неожиданным, поскольку рекурсивный вызов выглядит намного более затратным, чем простой переход из конца цикла в его начало. Но в показанном ранее вычислении приблизительного значения компилятор Scala может применить очень важную оптимизацию. Обратите внимание на то, что в вычислении тела функции `approximate` рекурсивный вызов стоит в самом конце. Функции наподобие `approximate`, которые в качестве последнего действия вызывают сами себя, называются *функциями с хвостовой рекурсией*. Компилятор Scala обнаруживает хвостовую рекурсию и заменяет ее переходом к началу функции после обновления параметров функции новыми значениями.

Из этого можно сделать вывод, что избегать использования рекурсивных алгоритмов для решения ваших задач не стоит. Зачастую рекурсивное решение выглядит более элегантно и лаконично, чем решение на основе цикла. Если в решении задействована хвостовая рекурсия, то расплачиваться за него издержками производительности во время выполнения программы не придется.

Трассировка функций с хвостовой рекурсией

Для каждого вызова функции с хвостовой рекурсией новый фрейм стека создаваться не будет, все вызовы станут выполняться с использованием одного и того же фрейма. Это обстоятельство может вызвать удивление у программиста, который исследует трассировку стека программы, давшей сбой. Например, данная функция вызывает себя несколько раз и генерирует исключение:

```
def boom(x: Int): Int =
  if x == 0 then throw new Exception("boom!")
  else boom(x - 1) + 1
```

Эта функция не относится к функциям с хвостовой рекурсией, поскольку после рекурсивного вызова выполняет операцию инкремента. При ее запуске будет получен вполне ожидаемый результат:

```
scala> boom(3)
java.lang.Exception: boom!
  at .boom(<console>:5)
  at .boom(<console>:6)
  at .boom(<console>:6)
```

```

at .boom(<console>:6)
at .<init>(<console>:6)
...

```

Если теперь внести изменения в `boom` так, чтобы в ней появилась хвостовая рекурсия:

```

def bang(x: Int): Int =
  if x == 0 then throw new Exception("bang!")
  else bang(x - 1)

```

то получится следующий результат:

```

scala> bang(5)
java.lang.Exception: bang!
  at .bang(<console>:5)
  at .<init>(<console>:6) ...

```

На сей раз вы видите только фрейм стека для `bang`. Можно подумать, `bang` дает сбой перед своим собственным вызовом, но это не так.

Оптимизация хвостового вызова

Код, скомпилированный для `approximate`, по сути, такой же, как и код, скомпилированный для `approximateLoop`. Обе функции компилируются в одни и те же 13 инструкций байт-кода Java. Если просмотреть байт-коды, сгенерированные компилятором Scala для метода с хвостовой рекурсией `approximate`, то можно увидеть, что, хотя и `isGoodEnough`, и `improve` вызываются в теле метода, `approximate` там не вызывается. При оптимизации компилятор Scala убирает рекурсивный вызов:

```

public double approximate(double);
Code:
  0:  aload_0
  1:  astore_3
  2:  aload_0
  3:  dload_1
  4:  invokevirtual #24; //метод isGoodEnough:(D)Z
  7:  ifeq     12
 10:  dload_1
 11:  dreturn
 12:  aload_0
 13:  dload_1
 14:  invokevirtual #27; //метод improve:(D)D
 17:  dstore_1
 18:  goto    2

```

Обычно вы добавляете аннотацию `scala.annotation.tailrec` к методу, который должен быть хвостовой рекурсией, например, когда вы ожидаете, что рекурсия может зайти очень далеко. Чтобы убедиться, что компилятор Scala выполняет оптимизацию хвостовой рекурсии, вы можете добавить `@tailrec` перед определением метода. В случае невозможности оптимизации компилятор выдаст ошибку и объяснение, почему она возникла.

Ограничения хвостовой рекурсии

Использование хвостовой рекурсии в Scala строго ограничено, поскольку набор инструкций виртуальной машины Java (JVM) существенно затрудняет реализацию более сложных форм хвостовых рекурсий. Оптимизация в Scala касается только непосредственных рекурсивных вызовов той же самой функции, из которой выполняется вызов. Если рекурсия косвенная, как в следующем примере, где применяются две взаимно рекурсивные функции, то ее оптимизация невозможна:

```
def isEven(x: Int): Boolean =
  if x == 0 then true else isOdd(x - 1)
def isOdd(x: Int): Boolean =
  if x == 0 then false else isEven(x - 1)
```

Получить оптимизацию хвостового вызова невозможно и в том случае, если завершающий вызов делается в отношении функционального значения. Рассмотрим, к примеру, такой рекурсивный код:

```
val funValue = nestedFun
def nestedFun(x: Int): Unit =
  if x != 0 then
    println(x)
    funValue(x - 1)
```

Переменная `funValue` ссылается на функциональное значение, которое, по сути, включает в себе вызов функции `nestedFun`. В момент применения функционального значения к аргументу все изменяется и `nestedFun` применяется к тому же самому аргументу, возвращая результат. Поэтому вы можете понадеяться на то, что компилятор Scala выполнит оптимизацию хвостового вызова, но в данном случае этого не произойдет. Оптимизация хвостовых вызовов ограничивается ситуациями, когда метод или вложенная функция вызывают сами себя непосредственно в качестве своей последней операции, не обращаясь к функциональному значению или через какого-то другого посредника. (Если вы еще не усвоили, что такое хвостовая рекурсия, то перечитайте раздел 8.10.)

Резюме

В данной главе мы представили довольно подробный обзор использования функций в Scala. Кроме методов, этот язык предоставляет локальные функции, функциональные литералы и функциональные значения. В дополнение к обычным вызовам функций в Scala используются частично примененные функции и функции с повторяющимися параметрами. При благоприятной возможности вызовы функций реализуются в виде оптимизированных хвостовых вызовов, благодаря чему многие привлекательные рекурсивные функции выполняются практически так же быстро, как и оптимизированные вручную версии, использующие циклы `while`. В следующей главе на основе этих положений мы покажем, как имеющаяся в Scala расширенная поддержка функций помогает абстрагировать процессы управления.

9

Управляющие абстракции

В главе 7 мы отметили, что встроенных управляющих абстракций в Scala не так уж много, поскольку этот язык позволяет вам создавать собственные управляющие абстракции. В предыдущей главе мы рассмотрели функциональные значения. В этой покажем способы применения функциональных значений в целях создания новых управляющих абстракций. Попутно рассмотрим карринг и передачу параметров по имени.

9.1. Сокращение повторяемости кода

Каждую функцию можно разделить на общую часть, одинаковую для всех вызовов функции, и особую часть, которая может варьироваться от одного вызова функции к другому. Общая часть находится в теле функции, а особая должна предоставляться через аргументы. Когда в качестве аргумента используется функциональное значение, особая часть алгоритма сама по себе является еще одним алгоритмом! При каждом вызове такой функции ей можно передавать в качестве аргумента другое функциональное значение, и вызванная функция в этом случае будет вызывать переданное функциональное значение. Такие *функции высшего порядка*, то есть функции, которые получают функции в качестве параметров, обеспечивают вам дополнительные возможности по сокращению и упрощению кода.

Одним из преимуществ функций высшего порядка является то, что они предоставляют вам возможность создавать управляющие абстракции, которые позволяют избавиться от повторяющихся фрагментов кода. Предположим, вы создаете браузер файлов и должны разработать API, разрешающий пользователям искать файлы, соответствующие какому-либо критерию. Сначала

вы добавляете средство поиска тех файлов, чьи имена заканчиваются конкретной строкой. Это даст пользователям возможность найти, к примеру, все файлы с расширением `.scala`. Такой API можно создать путем определения публичного метода `filesEnding` внутри следующего объекта-одиночки:

```
object FileMatcher:
  private def filesHere = (new java.io.File(".")).listFiles

  def filesEnding(query: String) =
    for file <- filesHere if file.getName.endsWith(query)
    yield file
```

Метод `filesEnding` получает список всех файлов, находящихся в текущем каталоге, применяя приватный вспомогательный метод `filesHere`, затем фильтрует этот список по признаку, завершается ли имя файла тем содержимым, которое указано в пользовательском запросе. Поскольку `filesHere` является приватным методом, то метод `filesEnding` — единственный доступный метод, определенный в `FileMatcher`, то есть в API, который вы предлагаете своим пользователям.

Пока все идет неплохо — повторяющегося кода нет. Но чуть позже вы хотите разрешить пользователям искать по любой части имени файла. Такой поиск пригодится, когда пользователи не смогут вспомнить, как именно они назвали файл, `phb-important.doc`, `joyful-phb-report.doc`, `may2020salesdoc.phb` или совершенно иначе, и единственное, в чем они уверены, — что где-то в имени фигурирует `phb`. Вы возвращаетесь к работе и к `FileMatcher` API добавляете соответствующую функцию:

```
def filesContaining(query: String) =
  for file <- filesHere if file.getName.contains(query)
  yield file
```

Данная функция работает точно так же, как и `filesEnding`. Она ищет текущие файлы с помощью `filesHere`, проверяет имя и возвращает файл, если его имя соответствует критерию поиска. Единственное отличие — функция использует метод `contains` вместо метода `endsWith`. Проходит несколько месяцев, и программа набирает популярность. Со временем вы уступаете просьбам некоторых активных пользователей, желающих вести поиск с помощью регулярных выражений. У этих нерадивых пользователей образовались огромные каталоги с тысячами файлов, и им хочется получить возможность искать все `pdf`-файлы, в названии которых имеется сочетание `oopsla`. Чтобы позволить им сделать это, вы создаете следующую функцию:

```
def filesRegex(query: String) =
  for file <- filesHere if file.getName.matches(query)
  yield file
```

Опытные программисты могут обратить внимание на все допущенные повторения и удивиться тому, что они не были сведены в общую вспомогательную функцию. Но если подходить к решению этой задачи в лоб, то ничего не получится. Можно было бы придумать следующее:

```
def filesMatching(query: String, method) =
  for file <- filesHere if file.getName.method(query)
  yield file
```

В некоторых динамичных языках такой подход сработал бы, но в Scala не разрешается вставлять подобный код во время выполнения. Что же делать?

Ответ дают функциональные значения. Передавать имя метода в качестве значения нельзя, но точно такой же эффект можно получить, если передать функциональное значение, вызывающее для вас этот метод. В этом случае к методу, единственной задачей которого будет проверка соответствия имени файла запросу, добавляется параметр `matcher`:

```
def filesMatching(query: String,
  matcher: (String, String) => Boolean) =
  for file <- filesHere if matcher(file.getName, query)
  yield file
```

В данной версии метода условие `if` теперь использует параметр `matcher` для проверки соответствия имени файла запросу. Что именно проверяется, зависит от того, что указано в качестве `matcher`. А теперь посмотрите на тип самого этого параметра. Это функция, вследствие чего в типе имеется обозначение `=>`. Функция получает два строковых аргумента, имя файла и запрос, и возвращает булево значение, следовательно, типом этой функции является `(String, String) => Boolean`.

Располагая новым вспомогательным методом по имени `filesMatching`, можно упростить три поисковых метода, заставив их вызывать вспомогательный метод, передавая в него соответствующую функцию:

```
def filesEnding(query: String) =
  filesMatching(query, _.endsWith(_))

def filesContaining(query: String) =
  filesMatching(query, _.contains(_))

def filesRegex(query: String) =
  filesMatching(query, _.matches(_))
```

Функциональные литералы, показанные в данном примере, задействуют синтаксис заместителя, рассмотренный в предыдущей главе, который может

быть вам еще не совсем привычен. Поэтому поясним, как применяются заместители: функциональный литерал `_.endsWith(_)`, используемый в методе `filesEnding`, означает то же самое, что и следующий код:

```
(fileName: String, query: String) => fileName.endsWith(query)
```

Поскольку `filesMatching` получает функцию, требующую два `String`-аргумента, то типы аргументов указывать не нужно — можно просто воспользоваться кодом `(filename, query) => filename.endsWith(query)`. А так как в теле функции каждый из параметров используется только раз (то есть первый параметр, `filename`, применяется в теле первым, второй параметр, `query`, — вторым), можно прибегнуть к синтаксису заместителей — `_.endsWith(_)`. Первый знак подчеркивания станет заместителем для первого параметра — имени файла, а второй — заместителем для второго параметра — строки запроса.

Этот код уже упрощен, но может стать еще короче. Обратите внимание: аргумент `query` передается `filesMatching`, но данная функция ничего с ним не делает, за исключением того, что передает его обратно переданной функции `matcher`. Такая передача туда-сюда необязательна, поскольку вызывающий код с самого начала знает о `query`! Это позволяет удалить параметр `query` как из `filesMatching`, так и из `matcher`, упростив код до состояния, показанного в листинге 9.1.

Листинг 9.1. Использование замыканий для сокращения повторяемости кода

```
object FileMatcher:
  private def filesHere = (new java.io.File(".")).listFiles

  private def filesMatching(matcher: String => Boolean) =
    for file <- filesHere if matcher(file.getName)
    yield file

  def filesEnding(query: String) =
    filesMatching(_.endsWith(query))

  def filesContaining(query: String) =
    filesMatching(_.contains(query))

  def filesRegex(query: String) =
    filesMatching(_.matches(query))
```

В этом примере показан способ, позволяющий функциям первого класса помочь вам избавиться от дублирующегося кода, что без них сделать было бы очень трудно. В Java, к примеру, можно создать интерфейс, который содержит метод, получающий одно `String`-значение и возвращающий значение

типа `Boolean`, а затем создать и передать функции `filesMatching` экземпляры анонимного внутреннего класса, реализующие этот интерфейс. Такой подход позволит избавиться от дублирующегося кода, чего, собственно, вы и добивались, но в то же время приведет к добавлению чуть ли не большего количества нового кода. Стало быть, цена вопроса сведет на нет все преимущества и лучше будет, вероятно, смириться с повторяемостью кода.

Кроме того, данный пример показывает, как замыкания способны помочь сократить повторяемость кода. Функциональные литералы, использованные в предыдущем примере, такие как `_.endsWith(_)` и `_.contains(_)`, во время выполнения программы становятся экземплярами функциональных значений, которые *не* являются замыканиями, поскольку не захватывают никаких свободных переменных. К примеру, обе переменные, использованные в выражении `_.endsWith(_)`, представлены в виде знаков подчеркивания, следовательно, берутся из аргументов функции. Таким образом, в `_.endsWith(_)` задействуются не свободные, а две связанные переменные. В отличие от этого, в функциональном литерале `_.endsWith(query)`, использованном в самом последнем примере, содержатся одна связанная переменная, а именно аргумент, представленный знаком подчеркивания, и одна свободная переменная по имени `query`. Возможность убрать параметр `query` из `filesMatching` в этом примере, тем самым еще больше упростив код, появилась у вас только потому, что в `Scala` поддерживаются замыкания.

9.2. Упрощение клиентского кода

В предыдущем примере было показано, что применение функций высшего порядка способствует сокращению повторяемости кода по мере реализации API. Еще один важный способ использовать функции высшего порядка — поместить их в сам API с целью повысить лаконичность клиентского кода. Хорошим примером могут послужить методы организации циклов специального назначения, принадлежащие имеющимся в `Scala` типам коллекций¹. Многие из них перечислены в табл. 3.1, но сейчас, чтобы понять, почему эти методы настолько полезны, внимательно рассмотрите только один пример.

Рассмотрим `exists` — метод, определяющий факт наличия переданного значения в коллекции. Разумеется, искать элемент можно, инициализировав `var`-переменную значением `false` и выполнив перебор элементов коллекции,

¹ Эти специализированные методы циклической обработки определены в трейте `Iterable`, который является расширением классов `List`, `Set` и `Map`. Более подробно данный вопрос рассматривается в главе 15.

проверяя каждый из них и присваивая `var`-переменной значение `true`, если будет найден предмет поиска. Метод, в котором такой подход используется с целью определить, имеется ли в переданном списке `List` отрицательное число, выглядит следующим образом:

```
def containsNeg(nums: List[Int]): Boolean =
  var exists = false
  for num <- nums do
    if num < 0 then
      exists = true
  exists
```

Если определить этот метод в REPL, то его можно вызвать следующими командами:

```
containsNeg(List(1, 2, 3, 4)) // false
containsNeg(List(1, 2, -3, 4)) // true
```

Но более лаконичный способ определения метода предусматривает вызов в отношении списка `List` функции высшего порядка `exists`:

```
def containsNeg(nums: List[Int]) = nums.exists(_ < 0)
```

Эта версия `containsNeg` выдает те же результаты, что и предыдущая:

```
containsNeg(Nil) // false
containsNeg(List(0, -1, -2)) // true
```

Метод `exists` представляет собой управляющую абстракцию. Это специализированная циклическая конструкция, которая не встроена в язык `Scala`, как `while` или `for`, а предоставляется библиотекой `Scala`. В предыдущем разделе функция высшего порядка `filesMatching` позволила сократить повторяемость кода в реализации объекта `FileMatcher`. Метод `exists` обеспечивает такое же преимущество, но поскольку это публичный метод в API коллекций `Scala`, то сокращение повторяемости относится к клиентскому коду этого API. Если бы метода `exists` не было и потребовалось бы написать метод выявления наличия в списке четных чисел `containsOdd`, то это можно было бы сделать так:

```
def containsOdd(nums: List[Int]): Boolean =
  var exists = false
  for num <- nums do
    if num % 2 == 1 then
      exists = true
  exists
```

Сравнивая тело метода `containsNeg` с телом метода `containsOdd`, можно заметить повторяемость во всем, за исключением условия проверки в выражении

expression. С помощью метода `exists` вместо этого можно воспользоваться следующим кодом:

```
def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

Тело кода в этой версии также практически идентично телу соответствующего метода `containsNeg` (той его версии, в которой используется `exists`), за исключением того, что условие, по которому выполняется поиск, иное. И тем не менее объем повторяющегося кода значительно уменьшился, поскольку вся инфраструктура организации цикла убрана в метод `exists`.

В стандартной библиотеке Scala имеется множество других методов для организации цикла. Как и `exists`, они зачастую могут сократить объем вашего кода, если появится возможность их применения.

9.3. Карринг

В главе 1 говорилось, что Scala позволяет создавать новые управляющие абстракции, которые воспринимаются как естественная языковая поддержка. Хотя показанные до сих пор примеры фактически и были управляющими абстракциями, вряд ли кто-то смог бы воспринять их как естественную поддержку со стороны языка. Чтобы понять, как создаются управляющие абстракции, больше похожие на расширения языка, сначала нужно разобраться с приемом функционального программирования, который называется *каррингом*.

Каррированная функция применяется не к одному, а к нескольким спискам аргументов. В листинге 9.2 показана обычная, некаррированная функция, складывающая два `Int`-параметра, `x` и `y`.

Листинг 9.2. Определение и вызов обычной функции

```
def plainOldSum(x: Int, y: Int) = x + y
plainOldSum(1, 2) // 3
```

В листинге 9.3 показана аналогичная, но уже каррированная функция. Вместо списка из двух параметров типа `Int` эта функция применяется к двум спискам, в каждом из которых содержится по одному параметру типа `Int`.

Листинг 9.3. Определение и вызов каррированной функции

```
def curriedSum(x: Int)(y: Int) = x + y
curriedSum(1)(2) // 3
```

Здесь при вызове `curriedSum` вы фактически получаете два обычных вызова функции, следующих непосредственно друг за другом. Первый получает единственный параметр `Int` по имени `x` и возвращает функциональное значение для второй функции. А та получает `Int`-параметр `y`. Здесь действие функции по имени `first` соответствует тому, что должно было происходить при вызове первой традиционной функции `curriedSum`:

```
def first(x: Int) = (y: Int) => x + y
```

Применение первой функции к числу 1, иными словами, вызов первой функции и передача ей значения 1, образует вторую функцию:

```
val second = first(1) // second имеет тип Int => Int
```

Применение второй функции к числу 2 дает результат:

```
second(2) //3
```

Функции `first` и `second` всего лишь показывают процесс карринга. Они не связаны непосредственно с функцией `curriedSum`. И тем не менее это способ получить фактическую ссылку на вторую функцию из `curriedSum`. Чтобы воспользоваться `curriedSum` в выражении частично примененной функцией, можно обратиться к форме записи с заместителем:

```
val onePlus = curriedSum(1) // onePlus имеет тип Int => Int
```

Знак подчеркивания в `curriedSum(1)_` является заместителем для второго списка, используемого в качестве `B`-параметра. В результате получается ссылка на функцию, при вызове которой единица прибавляется к ее единственному `Int`-аргументу, и возвращается результат:

```
onePlus(2) //3
```

А вот как можно получить функцию, прибавляющую число 2 к ее единственному `Int`-аргументу:

```
val twoPlus = curriedSum(2)
twoPlus(2) // 4
```

9.4. Создание новых управляющих конструкций

В языках, использующих функции первого класса, даже если синтаксис языка устоялся, есть возможность эффективно создавать новые управляющие

конструкции. Нужно лишь создать методы, получающие функции в виде аргументов.

Например, во фрагменте кода ниже показана удваивающая управляющая конструкция — она повторяет операцию два раза и возвращает результат:

```
def twice(op: Double => Double, x: Double) = op(op(x))
twice(_ + 1, 5) // 7.0
```

Типом `op` в данном примере является `Double => Double`. Это значит, что функция получает одно `Double`-значение в качестве аргумента и возвращает другое `Double`-значение.

Каждый раз, замечая шаблон управления, повторяющийся в разных частях вашего кода, вы должны задуматься о его реализации в виде новой управляющей конструкции. Ранее в этой главе был показан `filesMatching`, узкоспециализированный шаблон управления. Теперь рассмотрим более широко применяющийся шаблон программирования: открытие ресурса, работу с ним, а затем закрытие ресурса. Все это можно собрать в управляющую абстракцию, прибегнув к методу, показанному ниже:

```
def withPrintWriter(file: File, op: PrintWriter => Unit) =
  val writer = new PrintWriter(file)
  try op(writer)
  finally writer.close()
```

При наличии такого метода им можно воспользоваться так:

```
withPrintWriter(
  new File("date.txt"),
  writer => writer.println(new java.util.Date)
)
```

Преимущества применения этого метода состоят в том, что закрытие файла в конце работы гарантируется `withPrintWriter`, а не пользовательским кодом. Поэтому забыть закрыть файл просто невозможно. Данная технология называется *шаблоном временного пользования* (loan pattern), поскольку функция управляющей абстракции, такая как `withPrintWriter`, открывает ресурс и отдает его функции во временное пользование. Так, в предыдущем примере `withPrintWriter` отдает во временное пользование `PrintWriter` функции `op`. Когда функция завершает работу, она сигнализирует, что ей уже не нужен одолженный ресурс. Затем в блоке `finally` ресурс закрывается; это гарантирует его безусловное закрытие независимо от того, как завершилась работа функции — успешно или с генерацией исключения.

Один из способов придать клиентскому коду вид, который делает его похожим на встроенную управляющую конструкцию, предусматривает заключение списка аргументов в фигурные, а не в круглые скобки. Если в Scala при каждом вызове метода ему передается строго один аргумент, то можно заключить его не в круглые, а в фигурные скобки. Например, вместо

```
val s = "Hello, world!"
s.charAt(1) // 'e'
```

можно написать:

```
s.charAt { 1 } // 'e'
```

Во втором примере аргумент для `charAt` вместо круглых скобок заключен в фигурные. Но такой прием использования фигурных скобок будет работать только при передаче одного аргумента. Попытка нарушить это правило приводит к следующему результату:

```
s.substring { 7, 9 }
1 |s.substring { 7, 9 }
  |                ^
  |                end of statement expected but ',' found
1 |s.substring { 7, 9 }
  |                ^
  |                ';' expected, but integer literal found
```

Поскольку была предпринята попытка передать функции `substring` два аргумента, то при их заключении в фигурные скобки выдается ошибка. Вместо фигурных в данном случае нужно использовать круглые скобки:

```
s.substring(7, 9) // "wo"
```

Назначение такой возможности заменить круглые скобки фигурными при передаче одного аргумента — позволить программистам-клиентам записать в фигурных скобках функциональный литерал. Тем самым можно сделать вызов метода похожим на управляющую абстракцию. В качестве примера можно взять определенный ранее метод `withPrintWriter`. В своем самом последнем виде метод `withPrintWriter` получает два аргумента, поэтому использовать фигурные скобки нельзя. Тем не менее, поскольку функция, переданная `withPrintWriter`, является последним аргументом в списке, можно воспользоваться каррингом, чтобы переместить первый аргумент типа `File` в отдельный список аргументов. Тогда функция останется единственным параметром второго списка параметров. Способ переопределения `withPrintWriter` показан в листинге 9.4.

Листинг 9.4. Применение шаблона временного пользования для записи в файл

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) =
  val writer = new PrintWriter(file)
  try op(writer)
  finally writer.close()
```

Новая версия отличается от старой всего лишь тем, что теперь есть два списка параметров, по одному параметру в каждом, а не один список из двух параметров. Загляните между двумя параметрами. В показанной здесь прежней версии `withPrintWriter` вы видите `...File, op...`. Но в этой версии вы видите `...File)(op...`. Благодаря определению, приведенному ранее, метод можно вызвать с помощью более привлекательного синтаксиса:

```
val file = new File("date.txt")

withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

В этом примере первый список аргументов, в котором содержится один аргумент типа `File`, заключен в круглые скобки. А второй список аргументов, содержащий функциональный аргумент, заключен в фигурные скобки.

9.5. Передача параметров по имени

Метод `withPrintWriter`, рассмотренный в предыдущем разделе, отличается от встроенных управляющих конструкций языка, таких как `if` и `while`, тем, что тело управляющей абстракции (код между фигурными скобками) получает аргумент. Функция, переданная `withPrintWriter`, требует одного аргумента типа `PrintWriter`. Этот аргумент показан в следующем коде как `writer =>`:

```
withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

А что нужно сделать, если понадобится реализовать нечто больше похожее на `if` или `while`, где в теле нет значения для передачи в код? Помочь справиться с подобными ситуациями могут имеющиеся в Scala *параметры, передаваемые по имени* (by-name parameters).

В качестве конкретного примера представим, будто нужно реализовать конструкцию утверждения под названием `myAssert`¹. Функция `myAssert` будет получать в качестве ввода функциональное значение и обращаться к флагу, чтобы решить, что делать. Если флаг установлен, то `myAssert` вызовет переданную функцию и проверит, что она возвращает `true`. Если сброшен, то `myAssert` будет молча бездействовать.

Не прибегая к использованию параметров, передаваемых по имени, конструкцию `myAssert` можно создать следующим образом:

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if assertionsEnabled && !predicate() then
    throw new AssertionError
```

С определением все в порядке, но пользоваться им неудобно:

```
myAssert(() => 5 > 3)
```

Конечно, лучше было бы обойтись в функциональном литерале без пустого списка параметров и обозначения `=>` и создать следующий код:

```
myAssert(5 > 3) // Не будет работать из-за отсутствия () =>
```

Именно для воплощения задуманного и существуют параметры, передаваемые по имени. Чтобы создать такой параметр, задавать тип параметра нужно с обозначения `=>`, а не с `() =>`. Например, можно заменить в `myAssert` параметр `predicate` параметром, передаваемым по имени, изменив его тип `() => Boolean` на `=> Boolean`. Как это должно выглядеть, показано в листинге 9.5.

Листинг 9.5. Использование параметра, передаваемого по имени

```
def byNameAssert(predicate: => Boolean) =
  if assertionsEnabled && !predicate then
    throw new AssertionError
```

Теперь в свойстве, по поводу которого нужно высказать утверждение, можно избавиться от пустого параметра. В результате этого использование `byNameAssert` выглядит абсолютно так же, как встроенная управляющая конструкция:

```
byNameAssert(5 > 3)
```

¹ Здесь используется название `myAssert`, а не `assert`, поскольку имя `assert` предоставляется самим языком Scala. Соответствующее описание будет дано в разделе 25.1.

Тип «по имени» (*by-name*), в котором отбрасывается пустой список параметров `()`, допустимо использовать только в отношении параметров. Никаких *by-name*-переменных или *by-name*-полей не существует.

Можно, конечно, удивиться, почему нельзя просто написать функцию `myAssert`, воспользовавшись для ее параметров старым добрым типом `Boolean` и создав следующий код:

```
def boolAssert(predicate: Boolean) =  
  if assertionsEnabled && !predicate then  
    throw new AssertionError
```

Разумеется, такая формулировка тоже будет работать и код, использующий эту версию `boolAssert`, будет выглядеть точно так же, как и прежде:

```
boolAssert(5 > 3)
```

И все же эти два подхода различаются весьма значительным образом. Для параметра `boolAssert` используется тип `Boolean`, и потому выражение внутри круглых скобок в `boolAssert(5 > 3)` вычисляется до вызова `boolAssert`. Выражение `5 > 3` выдает значение `true`, которое передается в `boolAssert`. В отличие от этого, поскольку типом параметра `predicate` функции `byNameAssert` является `=> Boolean`, выражение внутри круглых скобок в `byNameAssert(5 > 3)` до вызова `byNameAssert` не вычисляется. Вместо этого будет создано функциональное значение, чей метод `apply` станет вычислять `5 > 3`, и это функциональное значение будет передано функции `byNameAssert`.

Таким образом, разница между двумя подходами состоит в том, что при отключении утверждений вам будут видны любые побочные эффекты, которые могут быть в выражении внутри круглых скобок в `boolAssert`, но `byNameAssert` это не касается. Например, если утверждения отключены, то попытки утверждать, что `x / 0 == 0`, в случае использования `boolAssert` приведут к генерации исключения:

```
val x = 5  
  
assertionsEnabled = false  
boolAssert(x / 0 == 0)  
java.lang.ArithmeticException: / by zero  
... 27 elided
```

Но попытки утверждать это на основе того же самого кода в случае использования `byNameAssert` не приведут к генерации исключения:

```
byNameAssert(x / 0 == 0) // Возвращается нормально
```

Резюме

В этой главе мы показали, как с помощью богатого инструментария функций Scala строить управляющие абстракции. Функции внутри вашего кода можно применять для избавления от распространенных шаблонов управления, а чтобы повторно задействовать шаблоны управления, часто встречающиеся в вашем программном коде, можно прибегнуть к функциям высшего порядка из библиотеки Scala. Кроме того, мы обсудили приемы использования карринга и параметров, передаваемых по имени, которые позволяют применять в весьма лаконичном синтаксисе собственные функции высшего порядка.

В двух последних главах мы рассмотрели довольно много всего, что относится к функциям. В следующих нескольких главах вернемся к рассмотрению дополнительных объектно-ориентированных средств языка.

10

Композиция и наследование

В главе 6 мы представили часть основных объектно-ориентированных аспектов Scala. В текущей продолжим рассматривать эту тему с того места, где остановились в главе 6, и дадим углубленное и гораздо более полное представление об имеющейся в Scala поддержке объектно-ориентированного программирования.

Нам предстоит сравнить два основных вида взаимоотношений между классами: композицию и наследование. Композиция означает, что один класс содержит ссылку на другой и использует класс, на который ссылается, в качестве вспомогательного средства для выполнения своей миссии. Наследование — это отношения «суперкласс/подкласс» («родительский/дочерний класс»).

Помимо этих тем, мы рассмотрим абстрактные классы, методы без параметров, расширение классов, переопределение методов и полей, параметрические поля, вызов конструкторов суперкласса, полиморфизм и динамическое связывание, финальные члены и классы, а также фабричные объекты и методы.

10.1. Библиотека двумерной разметки

В качестве рабочего примера в этой главе мы создадим библиотеку для построения и вывода на экран двумерных элементов разметки. Каждый элемент будет представлен прямоугольником, заполненным текстом. Для удобства библиотека будет предоставлять фабричные методы по имени `elem`, которые создают новые элементы из переданных данных. Например, вы сможете создать элемент разметки, содержащий строку, используя фабричный метод со следующей сигнатурой:

```
elem(s: String): Element
```

Как видите, элементы будут моделироваться с помощью типа данных по имени `Element`. Чтобы получить новый элемент, объединяющий два элемента, вы можете вызвать в отношении элемента операторы `above` или `beside`, передавая им второй элемент. Например, выражение, показанное ниже, создаст более крупный элемент, содержащий два столбца, каждый высотой два элемента:

```
val column1 = elem("hello") above elem("****")
val column2 = elem("****") above elem("world")
column1 beside column2
```

Вывод результата этого выражения даст следующий результат:

```
hello ***
*** world
```

Элементы разметки — хороший пример системы, в которой объекты могут создаваться из простых частей с помощью операторов композиции. В данной главе будут определены классы, позволяющие создавать объекты элементов из векторов, рядов и прямоугольников. Эти объекты базовых элементов будут простыми деталями. Вдобавок будут определены операторы композиции `above` и `beside`. Такие операторы зачастую называют *комбинаторами*, поскольку они комбинируют элементы некой области в новые элементы.

Подходить к проектированию библиотеки лучше всего в понятиях комбинаторов: они позволяют осмыслить основные способы конструирования объектов в прикладной области. Что представляют собой простые объекты? Какими способами из простых объектов могут создаваться более интересные объекты? Как комбинаторы должны сочетаться друг с другом? Что должны представлять собой наиболее общие комбинации? Удовлетворяют ли они всем выдвигаемым правилам? Если у вас есть верные ответы на все эти вопросы, то вы на правильном пути.

10.2. Абстрактные классы

Нашей первой задачей будет определить тип `Element`, представляющий элементы разметки. Элементы — двумерные прямоугольники из символов, поэтому имеет смысл включить в класс метод по имени `contents`, ссылающийся на содержимое элемента разметки. Данный метод может быть представлен в виде векторов строк, где каждая строка обозначает ряд. Отсюда типом возвращаемого `contents` значения должен быть `Vector[String]`. Как он будет выглядеть, показано в листинге 10.1.

Листинг 10.1. Определение абстрактного метода и класса

```
abstract class Element:
  def contents: Vector[String]
```

В этом классе `contents` объявляется в качестве метода, у которого нет реализации. Иными словами, метод является *абстрактным* членом класса `Element`. Класс с абстрактными членами сам по себе должен быть объявлен абстрактным; это можно сделать, указав модификатор `abstract` перед ключевым словом `class`:

```
abstract class Element ...
```

Модификатор `abstract` указывает на то, что у класса могут быть абстрактные члены, не имеющие реализации. Поэтому создать экземпляр абстрактного класса невозможно. При попытке сделать это будет получена ошибка компиляции:

```
scala> new Element
1 | new Element
  |          ^^^^^^^
  |          Element is abstract; it cannot be instantiated
```

Чуть позже в этой главе будет показан способ создания подклассов класса `Element`, экземпляры которых можно будет создавать, поскольку они заполняют отсутствующее определение метода `contents`.

Обратите внимание на то, что у метода `contents` класса `Element` нет модификатора `abstract`. Метод абстрактный, если у него нет реализации (то есть знака равенства и тела). В отличие от Java здесь при объявлении методов не нужны (не разрешены) модификаторы `abstract`. Методы, имеющие реализацию, называются *конкретными*.

И еще одно различие в терминологии между *объявлениями* и *определениями*. Класс `Element` *объявляет* абстрактный метод `contents`, но пока *не определяет* никаких конкретных методов. Но в следующем разделе класс `Element` будет усилен определением нескольких конкретных методов.

10.3. Определяем методы без параметров

В качестве следующего шага к `Element` будут добавлены методы, показывающие его ширину (`width`) и высоту (`height`) (листинг 10.2). Метод `height` возвращает количество рядов в содержимом. Метод `width` возвращает длину первого ряда или, при отсутствии рядов в элементе, ноль. Это

значит, что нельзя определить элемент с нулевой высотой и ненулевой шириной.

Листинг 10.2. Определение не имеющих параметров методов `width` и `height`

```
abstract class Element:
  def contents: Vector[String]
  def height: Int = contents.length
  def width: Int = if height == 0 then 0 else contents(0).length
```

Обратите внимание: ни в одном из трех методов класса `Element` нет списка параметров, даже пустого. Например, вместо

```
def width(): Int
```

метод определен без круглых скобок:

```
def width: Int
```

Такие *методы без параметров* встречаются в Scala довольно часто. В отличие от них методы, определенные с пустыми круглыми скобками, например `def height(): Int`, называются *методами с пустыми скобками*. Согласно имеющимся рекомендациям методы без параметров следует использовать, когда параметры отсутствуют и метод обращается к изменяемому состоянию только для чтения полей содержащего его объекта (при этом он не модифицирует изменяемое состояние). По этому соглашению поддерживается *принцип единообразного доступа* [Meu00], который гласит, что на клиентский код не должен влиять способ реализации атрибута в виде поля или метода.

Например, можно реализовать `width` и `height` в виде полей, а не методов, просто заменив `def` в каждом определении на `val`:

```
abstract class Element:
  def contents: Vector[String]
  val height = contents.length
  val width = if height == 0 then 0 else contents(0).length
```

С точки зрения клиента две пары определений абсолютно эквивалентны. Единственное различие заключается в том, что доступ к полю может осуществляться немного быстрее вызова метода, поскольку значения полей предварительно вычислены при инициализации класса, а не вычисляются при каждом вызове метода. В то же время полям в каждом объекте `Element` требуется дополнительное пространство памяти. Поэтому вопрос о том, как лучше представить атрибут, в виде поля или в виде метода, зависит от способа использования класса клиентами, который со временем может изме-

ниться. Главное, чтобы на клиенты класса `Element` никак не воздействовали изменения, вносимые во внутреннюю реализацию класса.

В частности, клиент класса `Element` не должен испытывать необходимости в перезаписи кода, если поле данного класса было переделано в функцию доступа, при условии, что это *чистая* функция доступа (то есть не имеет никаких побочных эффектов и не зависит от изменяемого состояния). Как бы то ни было, клиент не должен решать какие-либо проблемы.

Пока у нас все получается. Но есть небольшое осложнение, связанное с методами работы Java и Scala 2. Дело в том, что в данных языках нет полноценной реализации принципа единообразного доступа. Например, `string.length()` в Java — не то же самое, что `string.length`, и даже `array.length` — не то же самое, что `array.length()`. Это может привести к путанице.

Преодолеть это препятствие языку Scala 3 помогает то, что он весьма мягко относится к смешиванию методов, определенных в Java или Scala 2. В частности, можно заменить метод без параметров методом с пустыми круглыми скобками и *наоборот*, если родительский класс был написан на Java или Scala 2. Можно также не ставить пустые круглые скобки при вызове любой функции, не получающей аргументов. Например, в Scala 3 одинаково допустимо применение двух следующих строк кода:

```
Array(1, 2, 3).toString  
"abc".length
```

В принципе, в вызовах функций, определенных в Java или Scala 2, можно вообще не ставить пустые круглые скобки. Но их все же рекомендуется использовать, когда вызываемый метод представляет нечто большее, чем свойство своего объекта-получателя. Например, пустые круглые скобки уместны, если метод выполняет ввод-вывод, записывает переназначаемые переменные (`var`-переменные) или считывает `var`-переменные, не являющиеся полями объекта-получателя, как непосредственно, так и косвенно, используя изменяемые объекты. Таким образом, список параметров служит визуальным признаком того, что вызов инициирует некие примечательные вычисления, например:

```
"hello".length // нет (), поскольку побочные эффекты отсутствуют  
println()     // лучше () не отбрасывать
```

Подводя итоги, следует отметить, что в Scala приветствуется определение методов, не получающих параметров и не имеющих побочных эффектов, в виде методов без параметров, то есть без пустых круглых скобок. В то же время никогда не нужно определять метод, имеющий побочные эффекты, без

круглых скобок, поскольку вызов этого метода будет выглядеть как выбор поля. Следовательно, ваши клиенты могут быть удивлены, столкнувшись с побочными эффектами.

По аналогии с этим при вызове функции, имеющей побочные эффекты, не забудьте при написании вызова поставить пустые круглые скобки. Иначе говоря, если вызываемая функция выполняет операцию, то используйте круглые скобки, даже если компилятор этого не требует. Но если она просто предоставляет доступ к свойству, то круглые скобки следует отбросить.

10.4. Расширяем классы

Нам по-прежнему нужна возможность создавать объекты-элементы. Вы уже видели, что новый класс `Element` не приспособлен для этого в силу своей абстрактности. Поэтому для получения экземпляра элемента необходимо создать подкласс, расширяющий класс `Element` и реализующий абстрактный метод `contents`. Как это делается, показано в листинге 10.3.

Листинг 10.3. Определение класса `ArrayElement` в качестве подкласса класса `Element`

```
class VectorElement(cons: Vector[String]) extends Element:  
  def contents: Vector[String] = cons
```

Класс `ArrayElement` определен в целях *расширения* класса `Element`. Как и в Java, для выражения данного обстоятельства после имени класса указывается уточнение `extends`:

```
... extends Element ...
```

Использование уточнения `extends` заставляет класс `VectorElement` *унаследовать* у класса `Element` все его неprivate элементы и превращает тип `VectorElement` в *подтип* типа `Element`. Поскольку `VectorElement` расширяет `Element`, то класс `VectorElement` называется *подклассом* класса `Element`. В свою очередь, `Element` — *суперкласс* `VectorElement`. Если не указать уточнение `extends`, то компилятор Scala, безусловно, предположит, что ваш класс является расширением класса `scala.AnyRef`, который на платформе Java будет соответствовать классу `java.lang.Object`. Получается, класс `Element` неявно расширяет класс `AnyRef`. Эти отношения наследования показаны на рис. 10.1.

Наследование означает, что все элементы суперкласса являются также элементами подкласса, но с двумя исключениями. Первое: private элементы

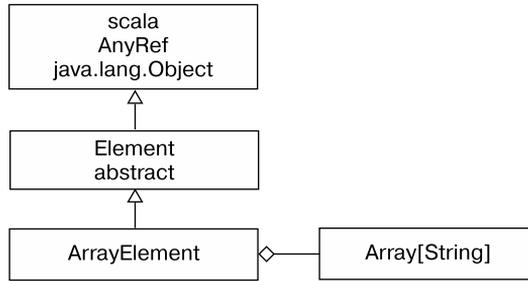


Рис. 10.1. Схема классов для VectorElement

суперкласса не наследуются подклассом. Второе: элемент суперкласса не наследуется, если элемент с такими же именем и параметрами уже реализован в подклассе. В таком случае говорится, что элемент подкласса *переопределяет* элемент суперкласса. Если элемент в подклассе является конкретным, а в суперклассе абстрактным, также говорится, что конкретный элемент — это *реализация* абстрактного элемента.

Например, метод `contents` в `VectorElement` переопределяет (или, в ином толковании, реализует) абстрактный метод `contents` класса `Element`¹. В отличие от этого класс `VectorElement` наследует у класса `Element` методы `width` и `height`. Например, располагая `VectorElement`-объектом `ae`, можно запросить его ширину, используя выражение `ae.width`, как будто метод `width` был определен в классе `VectorElement`²:

```
val ve = VectorElement(Vector("hello", "world"))
ve.width // 5
```

Создание подтипов означает, что значение подкласса может быть использовано там, где требуется значение суперкласса. Например:

```
val e: Element = VectorElement(Vector("hello"))
```

Переменная `e` определена как принадлежащая типу `Element`, следовательно, значение, используемое для ее инициализации, также должно быть типа `Element`. А фактически типом этого значения является класс `VectorElement`.

¹ Один из недостатков данной конструкции заключается в том, что мы пока не гарантируем одинаковой длины массива `contents` каждого `String`-элемента. Задачу можно решить, проверяя соблюдение предварительного условия в первичном конструкторе и генерируя исключения при его нарушении.

² Как упоминалось в разделе 6.2, при создании экземпляров классов, принимающих параметры, таких как `VectorElement`, вы можете не использовать ключевое слово `new`.

Это нормально, поскольку он расширяет класс `Element` и в результате тип `VectorElement` совместим с типом `Element`¹.

На рис. 10.1 также показано отношение *композиции* между `VectorElement` и `Vector[String]`. Оно так называется, поскольку класс `VectorElement` состоит из `Vector[String]`, то есть компилятор Scala помещает в генерируемый им для `VectorElement` двоичный класс поле, содержащее ссылку на переданный массив `conts`.

Некоторые моменты, касающиеся композиции и наследования, будут рассмотрены чуть позже — в разделе 10.11.

10.5. Переопределяем методы и поля

Принцип единообразного доступа является одним из тех аспектов, где Scala подходит к полям и методам единообразно — не так, как Java. Еще одно отличие заключается в том, что в Scala поля и методы принадлежат одному и тому же пространству имен. Этот позволяет полю переопределить метод без параметров. Например, можно, как показано в листинге 10.4, изменить реализацию `contents` в классе `VectorElement` из метода в поле, не модифицируя определение абстрактного метода `contents` в классе `Element`.

Листинг 10.4. Переопределение метода без параметров в поле

```
class VectorElement(conts: Vector[String]) extends Element:  
  val contents: Vector[String] = conts
```

Поле `contents` (определенное с ключевым словом `val`) в этой версии `VectorElement` — вполне подходящая реализация метода без параметров `contents` (объявленное с ключевым словом `def`) в классе `Element`. В то же время в Scala запрещено в одном и том же классе определять поле и метод с одинаковыми именами, а в Java это разрешено.

Например, этот класс в Java пройдет компиляцию вполне успешно:

```
// Это код Java  
class CompilesFine {  
  private int f = 0;  
  public int f() {  
    return 1;  
  }  
}
```

¹ Чтобы получить более четкое представление о разнице между подклассом и подтипом, обратитесь к статье глоссария о подтипах.

Но соответствующий класс в Scala не скомпилируется:

```
class WontCompile:
  private var f = 0 // Не пройдет компиляцию, поскольку поле
  def f = 1        // и метод имеют одинаковые имена
```

В принципе, в Scala вместо четырех имеющихся в Java пространств имен для определений имеются только два пространства. Четыре пространства имен в Java — это поля, методы, типы и пакеты. Напротив, двумя пространствами имен в Scala являются:

- значения (поля, методы, пакеты и объекты-одиночки);
- типы (имена классов и трейтов).

Причина, по которой поля и методы в Scala помещаются в одно и то же пространство имен, заключается в предоставлении возможности переопределить методы без параметров в `val`-поля, чего нельзя сделать в Java¹.

10.6. Определяем параметрические поля

Рассмотрим еще раз определение класса `VectorElement`, показанное в предыдущем разделе. В нем имеется параметр `conts`, единственное предназначение которого — его копирование в поле `contents`. Имя `conts` было выбрано для параметра, чтобы походило на имя поля `contents`, но не вступало с ним в конфликт имен. Это «код с душком» — признак того, что в вашем коде может быть некая совершенно ненужная избыточность и повторяемость.

От этого кода сомнительного качества можно избавиться, скомбинировав параметр и поле в едином определении *параметрического поля*, что и показано в листинге 10.5.

Листинг 10.5. Определение `contents` в качестве параметрического поля

```
// Расширенный элемент, показанный в листинге 10.2.
class VectorElement(
  val contents: Vector[String]
) extends Element
```

¹ Причиной того, что пакеты в Scala используют общее с полями и методами пространство имен, является стремление предоставить вам возможность получать доступ к импорту пакетов (а не только к именам типов), а также к полям и методам объектов-одиночек. Это тоже входит в перечень того, что невозможно сделать в Java. Подробности будут рассмотрены в разделе 12.3.

Обратите внимание: параметр `contents` имеет префикс `val`. Это сокращенная форма записи, определяющая одновременно параметр и поле с одним и тем же именем. Если выразиться более конкретно, то класс `VectorElement` теперь имеет поле `contents` (непереназначаемое), доступ к которому может быть получен за пределами класса. Поле инициализировано значением параметра. Похоже на то, будто бы класс был написан следующим образом:

```
class VectorElement(x123: Vector[String]) extends Element:  
  val contents: Vector[String] = x123
```

где `x123` — произвольное имя для параметра.

Кроме того, можно поставить перед параметром класса префикс `var`, и тогда соответствующее поле станет переназначаемым. И наконец, подобным параметризованным полям, как и любым другим членам класса, можно добавлять такие модификаторы, как `private`, `protected`¹ или `override`. Рассмотрим, к примеру, следующие определения классов:

```
class Cat:  
  val dangerous = false  
class Tiger(  
  override val dangerous: Boolean,  
  private var age: Int  
) extends Cat
```

Определение класса `Tiger` — сокращенная форма для следующего альтернативного определения класса с переопределяемым элементом `dangerous` и приватным элементом `age`:

```
class Tiger(param1: Boolean, param2: Int) extends Cat:  
  override val dangerous = param1  
  private var age = param2
```

Оба элемента инициализируются соответствующими параметрами. Имена для этих параметров, `param1` и `param2`, были выбраны произвольно. Главное, чтобы они не конфликтовали с какими-либо другими именами в пространстве имен.

10.7. Вызываем конструктор суперкласса

Теперь вы располагаете полноценной системой из двух классов: абстрактного класса `Element`, который расширяется конкретным классом `VectorElement`.

¹ Модификатор `protected`, предоставляющий доступ к подклассам, будет подробно рассмотрен в главе 12.

Можно также наметить иные способы выражения элемента. Например, клиенту может понадобиться создать элемент разметки, содержащий один ряд, задаваемый строкой. Объектно-ориентированное программирование упрощает расширение системы новыми вариантами данных. Можно просто добавить подклассы. Например, в листинге 10.6 показан класс `LineElement`, расширяющий класс `VectorElement`.

Листинг 10.6. Вызов конструктора суперкласса

```
// Расширенный элемент VectorElement, показанный в листинге 10.5.
class LineElement(s: String) extends VectorElement(Vector(s)):
  override def width = s.length
  override def height = 1
```

Поскольку `LineElement` расширяет `VectorElement`, а конструктор `VectorElement` получает параметр (`Vector[String]`), то `LineElement` нужно передать аргумент первичному конструктору своего суперкласса. В целях вызова конструктора суперкласса аргумент или аргументы, которые нужно передать, просто помещаются в круглые скобки, стоящие за именем суперкласса. Например, класс `LineElement` передает аргумент `Vector(s)` первичному конструктору класса `VectorElement`, поместив его в круглые скобки и указав после имени суперкласса `VectorElement`:

```
... extends VectorElement(Vector(s)) ...
```

С появлением нового подкласса иерархия наследования для элементов разметки приобретает вид, показанный на рис. 10.2.

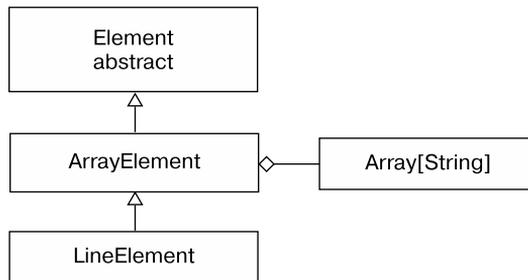


Рис. 10.2. Схема классов для `LineElement`

10.8. Используем модификатор `override`

Обратите внимание: определения `width` и `height` в `LineElement` имеют модификатор `override`. В разделе 6.3 он встречался в определении метода

`toString`. В Scala такой модификатор требуется для всех элементов, переопределяющих конкретный член родительского класса. Если элемент является реализацией абстрактного элемента с тем же именем, то модификатор указывать не обязательно. Применять модификатор запрещено, если член не переопределяется или не является реализацией какого-либо другого члена базового класса. Поскольку `height` и `width` в классе `LineElement` переопределяют конкретные определения в классе `Element`, то модификатор `override` указывать обязательно.

Соблюдение этого правила дает полезную информацию для компилятора, которая помогает избежать некоторых трудно отлавливаемых ошибок и сделать развитие системы более безопасным. Например, если вы опечатались в названии метода или случайно указали для него не тот список параметров, то компилятор тут же отреагирует, выдав сообщение об ошибке:

```
$ scalac LineElement.scala
-- [E037] Declaration Error: LineElement.scala:3:15 --
3 | override def hight = 1
  |                ^
  |                method hight overrides nothing
```

Соглашение о применении модификатора `override` приобретает еще большее значение, когда дело доходит до развития системы. Скажем, вы определили библиотеку методов рисования двумерных фигур, сделали ее общедоступной, и она стала использоваться довольно широко. Посмотрев на эту библиотеку позже, вы захотели добавить к основному классу `Shape` новый метод с такой сигнатурой:

```
def hidden(): Boolean
```

Новый метод будут использовать различные средства рисования, чтобы определить необходимости отрисовки той или иной фигуры. Тем самым можно существенно ускорить работу, но сделать это, не рискуя вывести из строя клиентский код, невозможно. Кроме всего прочего, у клиента может быть определен подкласс класса `Shape` с другой реализацией `hidden`. Возможно, клиентский метод фактически заставит объект-получатель просто исчезнуть вместо того, чтобы проверять, не является ли он невидимым. Две версии `hidden` переопределяют друг друга, поэтому ваши методы рисования в итоге заставят объекты исчезать, что совершенно не соответствует задуманному!

Эти «случайные переопределения» — наиболее распространенное проявление проблемы так называемого хрупкого базового класса. Проблема в том, что, если вы добавите новые члены в базовые классы (которые мы обычно называем суперклассами), вы рискуете вывести из строя клиентский код.

Полностью разрешить проблему хрупкого базового класса в Scala невозможно, но по сравнению с Java ситуация несколько лучше¹. Если библиотека рисования и ее клиентский код созданы в Scala, то у клиентской исходной реализации `hidden` не мог использоваться модификатор `override`, поскольку на момент его применения не могло быть другого метода с таким же именем.

Когда вы добавите метод `hidden` во вторую версию вашего класса фигур, при повторной компиляции кода клиента будет выдана следующая ошибка:

```
-- Error: Circle.scala:3:6 -----
3 | def hidden(): Boolean =
  |     ^
  |     error overriding method hidden in class Shape
  |       of type (): Boolean; method hidden of type
  |       (): Boolean needs `override` modifier
```

То есть вместо неверного поведения ваш клиент получит ошибку в ходе компиляции, и такой исход обычно более предпочтителен.

10.9. Полиморфизм и динамическое связывание

В разделе 10.4 было показано, что переменная типа `Element` может ссылаться на объект типа `VectorElement`. Этот феномен называется *полиморфизмом*, что означает «множество форм». В данном случае объекты типа `Element` могут иметь множество форм².

Ранее вам уже попадались две такие формы: `VectorElement` и `LineElement`. Можно создать еще больше форм `Element`, определив новые подклассы класса `Element`. Например, можно определить новую форму `Element` с заданными шириной и высотой (`width` и `height`) и полностью заполненную заданным символом:

```
// Расширенный элемент, показанный в листинге 10.2
class UniformElement(
  ch: Char,
```

¹ В Java есть аннотация `@Override`, работающая аналогично имеющемуся в Scala модификатору `override`, но в отличие от Scala-модификатора `override` применять ее не обязательно.

² Этот вид полиморфизма называется полиморфизмом подтипов. Другие виды полиморфизма в Scala обсуждаются в последующих главах, универсальный полиморфизм — в главе 18, а специальный полиморфизм — в главах 21 и 23.

```

    override val width: Int,
    override val height: Int
  ) extends Element:
    private val line = ch.toString * width
    def contents = Vector.fill(height)(line)

```

Иерархия наследования для класса `Element` теперь приобретает вид, показанный на рис. 10.3. В результате Scala примет все следующие присваивания, поскольку тип выражения присваивания соответствует типу определяемой переменной:

```

val e1: Element = VectorElement(Vector("hello", "world"))
val ve: VectorElement = LineElement("hello")
val e2: Element = ve
val e3: Element = UniformElement('x', 2, 3)

```

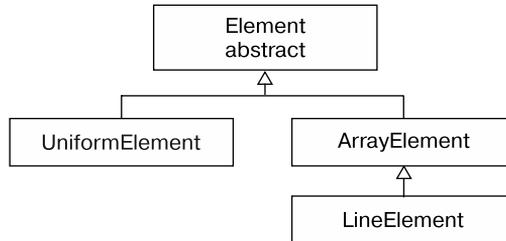


Рис. 10.3. Иерархия классов элементов разметки

Если изучить иерархию наследования, то окажется, что в каждом из этих четырех `val`-определений тип выражения справа от знака равенства принадлежит типу `val`-переменной, инициализируемой слева от знака равенства.

Есть еще одна сторона вопроса: вызовы методов с переменными и выражениями *динамически связаны*. Это значит, текущая реализация вызываемого метода определяется во время выполнения программы на основе класса объекта, а не типа переменной или выражения. Чтобы продемонстрировать данное поведение, мы временно уберем все существующие члены из наших классов `Element` и добавим к `Element` метод по имени `demo`. Затем переопределим `demo` в `VectorElement` и `LineElement`, но не в `UniformElement`:

```

abstract class Element:
  def demo = "Element's implementation invoked"

class VectorElement extends Element:
  override def demo = "VectorElement's implementation invoked"

```

```
class LineElement extends VectorElement:  
  override def demo = "LineElement's implementation invoked"
```

```
// UniformElement наследует demo из Element  
class UniformElement extends Element
```

Если ввести данный код в REPL, то можно будет определить метод, который получает объект типа `Element` и вызывает в отношении него метод `demo`:

```
def invokeDemo(e: Element) = e.demo
```

Если передать методу `invokeDemo` объект типа `VectorElement`, то будет показано сообщение, свидетельствующее о вызове реализации `demo` из класса `VectorElement`, даже если типом переменной `e`, в отношении которой был вызван метод `demo`, являлся `Element`:

```
invokeDemo(new VectorElement)  
// Вызвана реализация, определенная в VectorElement
```

Аналогично, если передать `invokeDemo` объект типа `LineElement`, будет показано сообщение, свидетельствующее о вызове той реализации `demo`, которая была определена в классе `LineElement`:

```
invokeDemo(new LineElement)  
// Вызвана реализация, определенная в LineElement
```

Поведение при передаче объекта типа `UniformElement` может на первый взгляд показаться неожиданным, но оно вполне корректно:

```
invokeDemo(new UniformElement)  
// Вызвана реализация, определенная в Element
```

В классе `UniformElement` метод `demo` не переопределяется, поэтому в нем наследуется реализация `demo` из его суперкласса `Element`. Таким образом, реализация, определенная в классе `Element`, — это правильная реализация вызова `demo`, когда классом объекта является `UniformElement`.

10.10. Объявляем финальные элементы

Иногда при проектировании иерархии наследования нужно обеспечить невозможность переопределения элемента подклассом. В Scala, как и в Java, это делается путем добавления к элементу модификатора `final`. Как показано в листинге 10.7, модификатор `final` можно указать для метода `demo` класса `VectorElement`.

Листинг 10.7. Объявление финального метода

```
class VectorElement extends Element:
  final override def demo =
    "VectorElement's implementation invoked"
```

При наличии данной версии в классе `VectorElement` попытка переопределить `demo` в его подклассе `LineElement` не пройдет компиляцию:

```
-- Error: LineElement.scala:2:15 -----
2 | override def demo =
  | ^
  | error overriding method demo in class VectorElement
  | of type => String; method demo of type => String
  | cannot override final member method demo in class
  | VectorElement
```

Порой вы должны быть уверены, что создать подкласс для класса в целом невозможно. Для этого нужно просто сделать весь класс финальным, добавив к объявлению класса модификатор `final`. Например, в листинге 10.8 показано, как должен быть объявлен финальный класс `VectorElement`.

Листинг 10.8. Объявление финального класса

```
final class VectorElement extends Element:
  override def demo = "VectorElement's implementation invoked"
```

При наличии данной версии класса `VectorElement` любая попытка определить подкласс не пройдет компиляцию:

```
-- [E093] Syntax Error: LineElement.scala:1:6 -----
1 | class LineElement extends VectorElement:
  | ^
  | class LineElement cannot extend final class
  | VectorElement
```

Теперь мы удалим модификаторы `final` и методы `demo` и вернемся к прежней реализации семейства классов `Element`. Далее в главе мы сконцентрируемся на завершении создания работоспособной версии библиотеки разметки.

10.11. Используем композицию и наследование

Композиция и наследование — два способа определить новый класс в понятиях другого, уже существующего класса. Если вы ориентируетесь преимущественно на повторное использование кода, то, как правило,

предпочтение нужно отдавать композиции, а не наследованию. Только ему свойственна проблема хрупкого базового класса, вследствие которой можно ненароком сделать неработоспособными подклассы, внося изменения в суперкласс.

Насчет отношения наследования нужно задаться лишь одним вопросом: не моделируется ли им взаимоотношение типа *is-a* (является) [Meу91]. Например, нетрудно будет заметить, что класс `VectorElement` является разновидностью `Element`. Можно задаться еще одним вопросом: придется ли клиентам использовать тип подкласса в качестве типа суперкласса [Еск98]. Применительно к классу `VectorElement` не вызывает никаких сомнений, что клиентам потребуется задействовать объекты типа `VectorElement` в качестве объектов типа `Element`.

Если задаться этими вопросами относительно отношений наследования, показанных на рис. 10.3, то не покажутся ли вам какие-либо из них подозрительными? В частности, насколько для вас очевидно, что `LineElement` является (*is-a*) `VectorElement`? Как вы думаете, понадобится ли когда-нибудь клиентам воспользоваться типом `LineElement` в качестве типа `VectorElement`?

Фактически класс `LineElement` был определен как подкласс класса `VectorElement` преимущественно для повторного использования имеющегося в `VectorElement` определения `contents`. Поэтому, возможно, будет лучше определить `LineElement` в качестве прямого подкласса класса `Element`:

```
class LineElement(s: String) extends Element:  
  val contents = Vector(s)  
  override def width = s.length  
  override def height = 1
```

В предыдущей версии `LineElement` состоял в отношении наследования с `VectorElement`, откуда наследовал метод `contents`. Теперь же у него отношение композиции с классом `Vector`: в нем содержится ссылка на строковый массив из его собственного поля `contents`¹. При наличии этой реализации класса `LineElement` иерархия наследования для `Element` приобретет вид, показанный на рис. 10.4.

¹ Класс `VectorElement` также имеет отношение композиции с классом `Vector`, поскольку его параметрическое поле `contents` содержит ссылку на строковый массив. Код для `VectorElement` показан в листинге 10.5. Его отношение композиции представлено в схеме классов в виде ромба, к примеру на рис. 10.1.

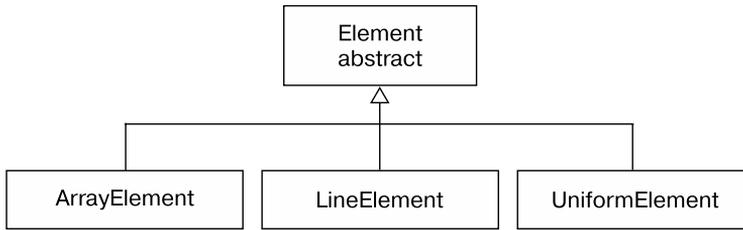


Рис. 10.4. Иерархия класса с пересмотренным определением подкласса `LineElement`

10.12. Реализуем методы `above`, `beside` и `toString`

В качестве следующего шага в классе `Element` будет реализован метод `above`. Поместить один элемент выше другого с помощью метода `above` означает объединить два значения содержимого элементов, представленного `contents`. Поэтому первый, черновой вариант метода `above` может иметь следующий вид:

```
def above(that: Element): Element =
  VectorElement(this.contents ++ that.contents)
```

Операция `++` объединяет два вектора. Некоторые другие векторные методы будут объяснены в этой главе, а более подробное обсуждение будет дано в главе 15.

Показанный ранее код нельзя считать достаточным, поскольку он не позволяет помещать друг на друга элементы разной ширины. Но, чтобы в этом разделе ничего не усложнять, оставим все как есть и станем передавать в метод `above` только элементы одинаковой длины. В разделе 10.14 мы усовершенствуем его, чтобы клиенты могли с его помощью объединять элементы разной ширины.

Следующим будет реализован метод `beside`. Чтобы поставить элементы рядом друг с другом, создадим новый элемент, в котором каждый ряд будет получен путем объединения соответствующих рядов двух элементов. Как и прежде, во избежание усложнений начнем с предположения, что высота двух элементов одинакова. Тогда структура метода `beside` приобретет такой вид:

```
def beside(that: Element): Element =
  val newContents = new Array[String](this.contents.length)
  for i <- 0 until this.contents.length do
    newContents(i) = this.contents(i) + that.contents(i)
  VectorElement(newContents.toVector)
```

Метод `beside` сначала выделяет массив `newContents` и заполняет его объединением соответствующих векторных элементов `this.contents` и `that.contents`. В итоге получается новый `VectorElement`, имеющий новое содержимое `toVector`.

Хотя эта реализация `beside` вполне работоспособна, в ней используется императивный стиль программирования, о чем явно свидетельствует наличие цикла, обходящего векторы по индексам. В альтернативном варианте метод можно сократить до одного выражения:

```
VectorElement(  
  for (line1, line2) <- this.contents.zip(that.contents)  
  yield line1 + line2)
```

Здесь с помощью оператора `zip` векторы `this.contents` и `that.contents` преобразуются в вектор пар (так называемый `Tuple2`). Оператор `zip` выбирает соответствующие элементы двух своих операндов и формирует вектор пар. Например, выражение

```
Vector(1, 2, 3).zip(Vector("a", "b"))
```

будет вычисляться как

```
Vector((1, "a"), (2, "b"))
```

Если один из двух векторов-операндов длиннее другого, то оставшиеся элементы оператор `zip` просто отбрасывает. В показанном ранее выражении третий элемент левого операнда, `3`, не формирует пару результата, поскольку для него не находится соответствующий элемент в правом операнде.

Затем вектор, подвергшийся `zip`, итерируется с помощью выражения `for`. Здесь синтаксис `for ((line1, line2) <- ...)` позволяет указать имена обоих элементов пары в одном *паттерне* (то есть теперь `line1` обозначает первый элемент пары, а `line2` — второй). Имеющаяся в Scala система сопоставления с образцом (паттерном) будет подробнее рассмотрена в главе 13. А сейчас все это можно представить себе как способ определения для каждого шага итерации двух `val`-переменных: `line1` и `line2`.

У выражения `for` есть часть `yield`, и поэтому оно выдает (`yields`) результат. Данный результат того же вида, что и перебираемый выражением объект (то есть данный вектор). Каждый элемент — результат объединения соответствующих рядов: `line1` и `line2`. Следовательно, конечный результат выполнения этого кода получается таким же, как и результат выполнения первой версии `beside`, но поскольку в нем удалось обойтись без явной индексации векторов, результат достигается способом, при котором допускается меньше ошибок.

Но вам все еще нужен способ отображения элементов. Как обычно, отображение выполняется с помощью определения метода `toString`, возвращающего элемент, отформатированный в виде строки. Его определение выглядит так:

```
override def toString = contents.mkString("\n")
```

В реализации `toString` используется метод `mkString`, который определен для всех последовательностей, включая векторы. В разделе 7.8 было показано выражение `vec.mkString sep`, которое возвращает строку, состоящую из всех векторов `vec`. Каждый элемент отображается на строку путем вызова его метода `toString`. Между последовательными элементами строк вставляется разделитель `sep`. Следовательно, выражение `contents.mkString ("\n")` форматирует содержимое векторов как строку, где каждый вектор появляется в собственном ряду.

Листинг 10.9. Класс `Element` с методами `above`, `beside` и `toString`

```
abstract class Element:  
  
  def contents: Vector[String]  
  
  def width: Int =  
    if height == 0 then 0 else contents(0).length  
  
  def height: Int = contents.length  
  
  def above(that: Element): Element =  
    VectorElement(this.contents ++ that.contents)  
  
  def beside(that: Element): Element =  
    VectorElement(  
      for (line1, line2) <- this.contents.zip(that.contents)  
      yield line1 + line2  
    )  
  
  override def toString = contents.mkString("\n")  
  
end Element
```

Обратите внимание: метод `toString` не требует указания пустого списка параметров. Это соответствует рекомендациям по соблюдению принципа единообразного доступа, поскольку `toString` — чистый метод, не получающий никаких параметров. После добавления этих трех методов класс `Element` приобретет вид, показанный в листинге 10.9.

10.13. Определяем фабричный объект

Теперь у вас есть иерархия классов для элементов разметки. Можно предоставить ее вашим клиентам как есть или выбрать технологию сокрытия иерархии за фабричным объектом.

В нем содержатся методы, с помощью которых клиенты смогут создавать объекты вместо того, чтобы делать это непосредственно через их конструкторы. Преимущество такого подхода заключается в возможности централизации создания объектов и в сокрытии способа представления объектов с помощью классов. Такое сокрытие сделает вашу библиотеку понятнее для клиентов, поскольку в открытом виде будет предоставлено меньше подробностей. Вдобавок оно обеспечит вам больше возможностей вносить последующие изменения реализации библиотеки, не нарушая работу клиентского кода.

Первая задача при конструировании фабрики для элементов разметки — выбор места, в котором должны располагаться фабричные методы. Чьими элементами они должны быть — объекта-одиночки или класса? Как должен быть назван содержащий их объект или класс? Существует множество возможностей. Самое простое решение — создать объект-компаньон класса `Element` и превратить его в фабричный объект для элементов разметки. Таким образом, клиентам нужно предоставить только комбинацию «класс — объект `Element`», а реализацию трех классов, `VectorElement`, `LineElement` и `UniformElement`, можно скрыть.

В листинге 10.10 представлена структура объекта `Element`, соответствующего этой схеме. В объекте `Element` содержатся три переопределяемых варианта метода `elem`, которые конструируют различный вид объекта разметки.

Листинг 10.10. Фабричный объект с фабричными методами

object `Element`:

```
def elem(contents: Vector[String]): Element =
  VectorElement(contents)

def elem(chr: Char, width: Int, height: Int): Element =
  UniformElement(chr, width, height)

def elem(line: String): Element =
  LineElement(line)
```

С появлением этих фабричных методов наметился смысл изменить реализацию класса `Element` таким образом, чтобы в нем вместо явного создания новых экземпляров `VectorElement` выполнялись фабричные методы `elem`. Чтобы вызвать фабричные методы, не указывая с ними имя объекта-одиночки

`Element`, мы импортируем в верхней части кода исходный файл `Element.elem`. Иными словами, вместо вызова фабричных методов с помощью указания `Element.elem` внутри класса `Element` мы импортируем `Element.elem`, чтобы можно было просто вызвать фабричные методы по имени `elem`. Код класса `Element` после внесения изменений показан в листинге 10.11.

Листинг 10.11. Класс `Element`, реорганизованный для использования фабричных методов

```
import Element.elem

abstract class Element:

  def contents: Vector[String]

  def width: Int =
    if height == 0 then 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    elem(this.contents ++ that.contents)

  def beside(that: Element): Element =
    elem(
      for (line1, line2) <- this.contents.zip(that.contents)
      yield line1 + line2
    )

  override def toString = contents.mkString("\n")

end Element
```

Кроме того, благодаря наличию фабричных методов теперь подклассы `VectorElement`, `LineElement` и `UniformElement` могут стать приватными, поскольку отпадет надобность непосредственного обращения к ним со стороны клиентов. В Scala классы и объекты-одиночки можно определять внутри других классов и объектов-одиночек. Один из способов превратить подклассы класса `Element` в приватные — поместить их внутрь объекта-одиночки `Element` и объявить их там приватными. Классы по-прежнему будут доступны трем фабричным методам `elem` там, где в них есть надобность. Как это будет выглядеть, показано в листинге 10.12.

10.14. Методы `heighten` и `widen`

Нам нужно внести еще одно, последнее усовершенствование. Версия `Element`, показанная в листинге 10.11, не может всецело нас устроить, поскольку не

позволяет клиентам помещать друг на друга элементы разной ширины или помещать рядом друг с другом элементы разной высоты.

Например, вычисление следующего выражения не будет работать корректно, так как второй ряд в объединенном элементе длиннее первого (см. листинг 10.12).

Листинг 10.12. Соккрытие реализации с помощью использования частных классов

```
elem(Vector("hello")) above elem(Vector("world!"))
```

object Element:

```
private class VectorElement(
  val contents: Vector[String]
) extends Element

private class LineElement(s: String) extends Element:
  val contents = Vector(s)
  override def width = s.length
  override def height = 1

private class UniformElement(
  ch: Char,
  override val width: Int,
  override val height: Int
) extends Element:
  private val line = ch.toString * width

def contents = Vector.fill(height)(line)

def elem(contents: Vector[String]): Element =
  VectorElement(contents)

def elem(chr: Char, width: Int, height: Int): Element =
  UniformElement(chr, width, height)

def elem(line: String): Element =
  LineElement(line)
```

end Element

Аналогично этому вычисление следующего выражения не будет работать правильно из-за того, что высота первого элемента `VectorElement` составляет два ряда, а второго — только один:

```
elem(Vector("one", "two")) beside
elem(Vector("one"))
```

В листинге 10.13 показан приватный вспомогательный метод по имени `widen`, который получает ширину и возвращает объект `Element` указанной ширины. Результат включает в себя содержимое этого объекта, которое для достижения нужной ширины отцентрировано за счет создания отступов справа и слева с помощью любого нужного для этого количества пробелов. В листинге также показан похожий метод `heighten`, выполняющий то же в вертикальном направлении. Метод `widen` вызывается методом `above`, чтобы обеспечить одинаковую ширину элементов, которые помещаются друг над другом. Аналогично этому метод `heighten` вызывается методом `beside`, чтобы обеспечить одинаковую высоту элементов, помещаемых рядом друг с другом. После внесения этих изменений библиотека разметки будет готова к использованию.

Листинг 10.13. Класс `Element` с методами `widen` и `heighten`

```
import Element.elem

abstract class Element:
  def contents: Vector[String]

  def width: Int =
    if height == 0 then 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    val this1 = this.widen(that.width)
    val that1 = that.widen(this.width)
    elem(this1.contents ++ that1.contents)

  def beside(that: Element): Element =
    val this1 = this.heighten(that.height)
    val that1 = that.heighten(this.height)
    elem(
      for (line1, line2) <- this1.contents.zip(that1.contents)
      yield line1 + line2
    )

  def widen(w: Int): Element =
    if w <= width then this
    else
      val left = elem(' ', (w - width) / 2, height)
      val right = elem(' ', w - width - left.width, height)
      left beside this beside right

  def heighten(h: Int): Element =
    if h <= height then this
    else
```

```

    val top = elem(' ', width, (h - height) / 2)
    val bot = elem(' ', width, h - height - top.height)
    top above this above bot

    override def toString = contents.mkString("\n")
end Element

```

10.15. Собираем все вместе

Интересным способом применения почти всех элементов библиотеки разметки будет написание программы, рисующей спираль с заданным количеством ребер. Ее созданием займется программа `Spiral`, показанная в листинге 10.14.

Листинг 10.14. Приложение `Spiral`

```

import Element.elem

object Spiral:

    val space = elem(" ")
    val corner = elem("+")

    def spiral(nEdges: Int, direction: Int): Element =
        if nEdges == 1 then
            elem("+")
        else
            val sp = spiral(nEdges - 1, (direction + 3) % 4)
            def verticalBar = elem('|', 1, sp.height)
            def horizontalBar = elem('-', sp.width, 1)
            if direction == 0 then
                (corner beside horizontalBar) above (sp beside space)
            else if direction == 1 then
                (sp above space) beside (corner above verticalBar)
            else if direction == 2 then
                (space beside sp) above (horizontalBar beside corner)
            else
                (verticalBar above corner) beside (space above sp)

    def main(args: Array[String]) =
        val nSides = args(0).toInt
        println(spiral(nSides, 0))
end Spiral

```

Поскольку `Spiral` является самостоятельным объектом с методом `main`, имеющим надлежащую сигнатуру, этот код можно считать приложением,

написанным на Scala. `Spiral` получает один аргумент командной строки в виде целого числа и рисует спираль с указанным количеством граней. Например, можно нарисовать шестигранную спираль, как показано слева, и более крупную спираль, как показано справа.

```
$ scala Spiral 6
```

```
+-----
|
|  +-+
|  + |
|  | |
+-----
```

```
$ scala Spiral 11
```

```
+-----+
|
|  +-----+
|  |         | |
|  |  +-+   |
|  |  |     |
|  |  ++    |
|  |  |     |
|  |  +-----+
+-----+
```

```
$ scala Spiral 17
```

```
+-----+
|
|  +-----+
|  |         |
|  |  +-----+
|  |  |         |
|  |  |  +-----+
|  |  |  |         | |
|  |  |  |  ++    |
|  |  |  |  |     |
|  |  |  |  |     |
|  |  |  |  |     |
|  |  |  |  |     |
|  |  |  |  |     |
|  |  |  |  |     |
|  |  |  |  |     |
|  |  |  |  |     |
|  |  |  |  |     |
+-----+
```

Резюме

В этой главе мы рассмотрели дополнительные концепции объектно-ориентированного программирования на языке Scala. Среди них — абстрактные классы, наследование и создание подтипов, иерархии классов, параметрические поля и переопределение методов. У вас должно было выработаться понимание способов создания в Scala оригинальных иерархий классов. А к работе с библиотекой раскладки мы еще вернемся в главе 25.

11

Трейты

Трейты в Scala являются фундаментальными повторно используемыми блоками кода. В трейте инкапсулируются определения тех методов и полей, которые затем могут повторно использоваться путем их примешивания в классы. В отличие от наследования классов, в котором каждый класс должен быть наследником только одного суперкласса, в класс может примешиваться любое количество трейтов. В этой главе мы покажем, как работают трейты. Далее рассмотрим два наиболее распространенных способа их применения: расширение «тонких» интерфейсов и превращение их в «толстые», а также определение наращиваемых модификаций. Здесь мы покажем, как используется трейт `Ordered`, и сравним механизм трейтов с множественным наследованием, имеющимся в других языках.

11.1. Как работают трейты

Определение трейта похоже на определение класса, за исключением того, что в нем используется ключевое слово `trait`. Пример показан в листинге 11.1.

Листинг 11.1. Определение трейта `Philosophical`

```
trait Philosophical:  
  def philosophize = "На меня тратится память, следовательно, я существую!"
```

Данный трейт называется `Philosophical`. В нем не объявлен суперкласс, следовательно, как и у класса, у него есть суперкласс по умолчанию — `AnyRef`. В нем определяется один конкретный метод по имени `philosophize`. Это простой трейт, и его вполне достаточно, чтобы показать, как работают трейты.

После того как трейт определен, он может быть *примешан* в класс либо с помощью ключевого слова `extends` или `with`, либо с помощью запятой. Программисты, работающие со Scala, примешивают трейты, а не наследуют их, поскольку примешивание трейта весьма отличается от множественного наследования, встречающегося во многих других языках. Этот вопрос рассматривается в разделе 11.4. Например, в листинге 11.2 показан класс, в который с помощью ключевого слова `extends` примешивается трейт `Philosophical`.

Листинг 11.2. Примешивание трейта с использованием ключевого слова `extends`

```
class Frog extends Philosophical:  
  override def toString = "зеленая"
```

Примешивать трейт можно с помощью ключевого слова `extends`, в таком случае происходит неявное наследование суперкласса трейта. Например, в листинге 11.2 класс `Frog` (лягушка) становится подклассом `AnyRef` (это суперкласс для трейта `Philosophical`) и примешивает в себя трейт `Philosophical`. Методы, унаследованные от трейта, могут использоваться точно так же, как и методы, унаследованные от суперкласса. Рассмотрим пример:

```
val frog = new Frog  
frog.philosophize() // На меня тратится память, следовательно,  
я существую!
```

Трейт также определяет тип. Рассмотрим пример, в котором `Philosophical` используется как тип:

```
val phil: Philosophical = frog  
phil.philosophize // На меня тратится память, следовательно,  
я существую!
```

Типом `phil` является `Philosophical`, то есть трейт. Таким образом, переменная `phil` может быть инициализирована любым объектом, в чей класс примешан трейт `Philosophical`.

Если нужно примешать трейт в класс, который явно расширяет суперкласс, то ключевое слово `extends` используется для указания суперкласса, а для примешивания трейта — запятая (или ключевое слово `with`). Пример показан в листинге 11.3. Если нужно примешать сразу несколько трейтов, то дополнительные трейты указываются с помощью ключевого слова `with`. Например, располагая трейтом `HasLegs`, вы, как показано в листинге 11.4, можете примешать в класс `Frog` как трейт `Philosophical`, так и трейт `HasLegs`.

Листинг 11.3. Примешивание трейта с использованием запятой

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "зеленая"
}
```

Листинг 11.4. Примешивание нескольких трейтов

```
class Animal
trait HasLegs

class Frog extends Animal, Philosophical, HasLegs:
  override def toString = "зеленая"
```

В показанных ранее примерах класс `Frog` наследовал реализацию метода `philosophize` из трейта `Philosophical`. В качестве альтернативного варианта метод `philosophize` в классе `Frog` может быть переопределен. Синтаксис выглядит точно так же, как и при переопределении метода, объявленного в суперклассе. Рассмотрим пример:

```
class Animal

class Frog extends Animal, Philosophical:
  override def toString = "зеленая"
  override def philosophize = s"Мне живется нелегко, потому что я $this!"
```

В новое определение класса `Frog` по-прежнему примешивается трейт `Philosophical`, поэтому его, как и раньше, можно использовать из переменной данного типа. Но так как во `Frog` переопределено определение метода `philosophize`, которое было дано в трейте `Philosophical`, при вызове будет получено новое поведение:

```
val phrog: Philosophical = new Frog
phrog.philosophize // Мне живется нелегко, потому что я зеленая!
```

Теперь можно прийти к философскому умозаключению, что трейты подобны Java-интерфейсам со стандартными методами, но фактически их возможности гораздо шире. Так, в трейтах можно объявлять поля и сохранять состояние. Фактически в определении трейта можно делать то же самое, что и в определении класса, и синтаксис выглядит почти так же.

Ключевое отличие классов от трейтов заключается в том, что в классах вызовы `super` имеют статическую привязку, а в трейтах — динамическую. Если в классе воспользоваться кодом `super.toString`, то вы будете точно знать, какая именно реализация метода будет вызвана. Но когда точно такой

же код применяется в трейте, то вызываемая с помощью `super` реализация метода при определении трейта еще не определена. Вызываемая реализация станет определяться заново при каждом примешивании трейта в конкретный класс. Такое своеобразное поведение `super` является ключевым фактором, позволяющим трейтам работать в качестве *наращиваемых модификаций*, и рассматривается в разделе 11.3. А правила разрешения вызовов `super` будут изложены в разделе 11.4.

11.2. Сравнение «тонких» и «толстых» интерфейсов

Чаще всего трейты используются для автоматического добавления к классу методов в дополнение к тем методам, которые в нем уже имеются. То есть трейты способны расширить *«тонкий»* интерфейс, превратив его в *«толстый»*.

Противопоставление «тонких» интерфейсов «толстым» представляет собой компромисс, который довольно часто встречается в объектно-ориентированном проектировании. Это компромисс между теми, кто реализует интерфейс, и теми, кто им пользуется. В «толстых» интерфейсах имеется множество методов, обеспечивающих удобство применения для тех, кто их вызывает. Клиенты могут выбрать метод, целиком отвечающий их функциональным запросам. В то же время «тонкий» интерфейс имеет незначительное количество методов и поэтому проще обходится тем, кто их реализует. Но клиентам, обращающимся к «тонким» интерфейсам, приходится создавать больше собственного кода. При более скудном выборе доступных для вызова методов им приходится выбирать то, что хотя бы в какой-то мере отвечает их потребностям, а чтобы использовать выбранный метод, им требуется создавать дополнительный код.

Добавление в трейт конкретного метода уводит компромисс «тонкий — толстый» в сторону более «толстых» интерфейсов — это однократное действие. Вам нужно единожды реализовать конкретный метод, сделав это в самом трейте, вместо того чтобы возиться с его повторной реализацией для каждого класса, в который примешивается трейт. Таким образом, создание «толстых» интерфейсов в Scala требует меньше работы, чем в языках без трейтов.

Чтобы расширить интерфейс с помощью трейтов, просто определите трейт с небольшим количеством абстрактных методов — «тонкую» часть интерфейса трейта — и с потенциально большим количеством конкретных методов, реализованных в терминах абстрактных методов. Затем можно будет при-

мешать расширяющий трейт в класс, реализовав «тонкую» часть интерфейса, и получить в результате класс, позволяющий обеспечить доступ ко всему доступному «толстому» интерфейсу.

Хорошим примером области, в которой «толстый» интерфейс удобен, является сравнение.

Сравнивая два упорядочиваемых объекта, было бы рационально воспользоваться вызовом одного метода, чтобы выяснить результаты желаемого сравнения. Если нужно использовать сравнение «меньше», то предпочтительнее было бы вызвать `<`, а если требуется применить сравнение «меньше или равно» — вызвать `<=`. С «тонким» интерфейсом можно располагать только методом `<`, и тогда временами приходилось бы создавать код наподобие `(x < y) || (x == y)`. «Толстый» интерфейс предоставит вам все привычные операторы сравнения, позволяя напрямую создавать код вроде `x <= y`.

Прежде чем посмотреть на трейт `Ordered`, представим, что можно сделать без него. Предположим, вы взяли класс `Rational` из главы 6 и добавили к нему операции сравнения. У вас должен получиться примерно такой код¹:

```
class Rational(n: Int, d: Int):
  // ...
  def < (that: Rational) =
    this.numer * that.denom < that.numer * this.denom
  def > (that: Rational) = that < this
  def <= (that: Rational) = (this < that) || (this == that)
  def >= (that: Rational) = (this > that) || (this == that)
```

В данном классе определяются четыре оператора сравнения (`<`, `>`, `<=` и `>=`), и это классическая демонстрация стоимости определения «толстого» интерфейса. Сначала обратите внимание на то, что три оператора сравнения определены на основе первого оператора. Например, оператор `>` определен как противоположность оператора `<`, а оператор `<=` определен буквально как «меньше или равно». Затем обратите внимание на то, что все три этих метода будут такими же для любого другого класса, объекты которого могут сравниваться друг с другом. В отношении оператора `<=` для рациональных чисел не прослеживается никаких особенностей. В контексте сравнения оператор `<=` *всегда* используется для обозначения «меньше или равно». В общем, в этом классе есть довольно много шаблонного кода, который будет точно таким же в любом другом классе, реализующем операции сравнения.

¹ Этот пример основан на использовании класса `Rational`, показанного в листинге 6.5, с методами `equals`, `hashCode` и модификациями, гарантирующими положительный `denom`.

Данная проблема встречается настолько часто, что в Scala предоставляется трейт, помогающий справиться с ее решением. Этот трейт называется `Ordered`. Чтобы воспользоваться им, нужно заменить все отдельные методы сравнений одним методом `compare`. Затем на основе одного этого метода определить в трейте `Ordered` методы `<`, `>`, `<=` и `>=`. Таким образом, трейт `Ordered` позволит вам расширить класс методами сравнений с помощью реализации всего одного метода по имени `compare`.

Если определить операции сравнения в `Rational` путем использования трейта `Ordered`, то код будет иметь следующий вид:

```
class Rational(n: Int, d: Int) extends Ordered[Rational]:
  // ...
  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)
```

Нужно выполнить две задачи. Начнем с того, что в `Rational` примешивается трейт `Ordered`. В отличие от трейтов, которые встречались до сих пор, `Ordered` требует от вас при примешивании указать *параметр типа*. Параметры типов до главы 18 подробно рассматриваться не будут, а пока все, что нужно знать при примешивании `Ordered`, сводится к следующему: фактически следует выполнять примешивание `Ordered[C]`, где `C` обозначает класс, элементы которого сравниваются. В данном случае в `Rational` примешивается `Ordered[Rational]`.

Вторая задача, требующая выполнения, заключается в определении метода `compare` для сравнения двух объектов. Этот метод должен сравнивать получатель `this` с объектом, переданным методу в качестве аргумента. Возвращать он должен целочисленное значение, которое равно нулю, если объекты одинаковы, отрицательное число, если получатель меньше аргумента, и положительное, если получатель больше аргумента.

В данном случае метод сравнения класса `Rational` использует формулу, основанную на приведении чисел к общему знаменателю с последующим вычитанием получившихся числителей. Теперь при наличии этого примешивания и определения метода `compare` класс `Rational` имеет все четыре метода сравнения:

```
val half = new Rational(1, 2)
val third = new Rational(1, 3)
half < third // false
half > third // true
```

При каждой реализации класса с какой-либо возможностью упорядоченности путем сравнения нужно рассматривать вариант примешивания в него

трейта `Ordered`. Если выполнить это примешивание, то пользователи класса получат богатый набор методов сравнений.

Имейте в виду, что трейт `Ordered` не определяет за вас метод `equals`, поскольку не способен на это. Дело в том, что реализация `equals` в терминах `compare` требует проверки типа переданного объекта, а из-за удаления типов сам `Ordered` не может ее выполнить. Поэтому определять `equals` вам придется самим, даже если вы примешиваете трейт `Ordered`.

11.3. Трейты как наращиваемые модификации

Основное применение трейтов — превращение «тонкого» интерфейса в «толстый» — вы уже видели. Перейдем теперь ко второму по значимости способу использования трейтов — предоставлению классам наращиваемых модификаций. Трейты дают возможность *изменять* методы класса, позволяя вам *наращивать* их друг на друге.

Рассмотрим в качестве примера наращиваемые модификации применительно к очереди целых чисел. В очереди будут две операции: `put`, помещающая целые числа в очередь, и `get`, извлекающая их из очереди. Очереди работают по принципу «первым пришел, первым ушел», поэтому целые числа извлекаются из очереди в том же порядке, в котором были туда помещены.

Располагая классом, реализующим такую очередь, можно определить трейты для выполнения следующих модификаций:

- удваивания — удваиваются все целые числа, помещенные в очередь;
- увеличения на единицу — увеличиваются все целые числа, помещенные в очередь;
- фильтрации — из очереди отфильтровываются все отрицательные целые числа.

Эти три трейта представляют *модификации*, поскольку модифицируют поведение соответствующего класса очереди, а не определяют полный класс очереди. Все три трейта также являются *наращиваемыми*. Можно выбрать любые из трех трейтов, примешать их в класс и получить новый класс, обладающий всеми выбранными модификациями.

В листинге 11.5 показан абстрактный класс `IntQueue`. В `IntQueue` имеются метод `put`, добавляющий к очереди новые целые числа, и метод `get`, который

возвращает целые числа и удаляет их из очереди. Основная реализация `IntQueue`, которая использует `ArrayBuffer`, показана в листинге 11.6.

Листинг 11.5. Абстрактный класс `IntQueue`

```
abstract class IntQueue:
  def get(): Int
  def put(x: Int): Unit
```

Листинг 11.6. Реализация класса `BasicIntQueue` с использованием `ArrayBuffer`

```
import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue:
  private val buf = ArrayBuffer.empty[Int]
  def get() = buf.remove(0)
  def put(x: Int) = buf += x
```

В классе `BasicIntQueue` имеется приватное поле, содержащее буфер в виде массива. Метод `get` удаляет запись с одного конца буфера, а метод `put` добавляет элементы к другому его концу. Пример использования данной реализации выглядит следующим образом:

```
val queue = new BasicIntQueue
queue.put(10)
queue.put(20)
queue.get() // 10
queue.get() // 20
```

Пока все вроде бы в порядке. Теперь посмотрим на использование трейтов для модификации данного поведения. В листинге 11.7 показан трейт, который удваивает целые числа по мере их помещения в очередь. Трейт `Doubling` имеет две интересные особенности. Первая заключается в том, что в нем объявляется суперкласс `IntQueue`. Это объявление означает, что трейт может примешиваться только в класс, который также расширяет `IntQueue`. То есть `Doubling` можно примешивать в `BasicIntQueue`, но не в `Rational`.

Листинг 11.7. Трейт наращиваемых модификаций `Doubling`

```
trait Doubling extends IntQueue:
  abstract override def put(x: Int) = super.put(2 * x)
```

Вторая интересная особенность заключается в том, что у трейта имеется вызов `super` в отношении метода, объявленного абстрактным. Для обычных классов такие вызовы применять запрещено, поскольку во время выполнения они гарантированно дадут сбой. Но для трейта такой вызов может действительно пройти успешно. В трейте вызовы `super` динамически связаны, поэтому вызов `super` в трейте `Doubling` будет работать при условии, что трейт

примешан *после* другого трейта или класса, в котором дается конкретное определение метода.

Трейтам, которые реализуют наращиваемые модификации, зачастую нужен именно такой порядок. Чтобы сообщить компилятору, что это делается намеренно, подобные методы следует помечать модификаторами `abstract override`. Это сочетание модификаторов позволительно только для членов трейтов, но не классов и означает, что трейт должен быть примешан в некий класс, имеющий конкретное определение рассматриваемого метода.

Применение трейта выглядит следующим образом:

```
class MyQueue extends BasicIntQueue, Doubling
val queue = new MyQueue
queue.put(10)
queue.get() // 20
```

В первой строке этого примера определяется класс `MyQueue`, который расширяет класс `BasicIntQueue` и примешивая в него трейт `Doubling`. Затем мы создаем новый `MyQueue` и помещаем в него число `10`, но в результате примешивания трейта `Doubling` оно удваивается. При извлечении целого числа из очереди оно уже имеет значение `20`.

Обратите внимание: в `MyQueue` не определяется никакой новый код — просто объявляется класс и примешивается трейт. В подобной ситуации вместо определения именованного класса код `BasicIntQueue with Doubling` может быть предоставлен непосредственно с ключевым словом `new`. Работа такого кода показана в листинге 11.8¹.

Листинг 11.8. Примешивание трейта при создании экземпляра с помощью ключевого слова `new`

```
val queue = new BasicIntQueue with Doubling
queue.put(10)
queue.get() // 20
```

Чтобы посмотреть, как нарастить модификации, нужно определить еще два модифицирующих трейта: `Incrementing` и `Filtering`. Реализация этих трейтов показана в листинге 11.9.

Листинг 11.9. Трейты наращиваемых модификаций `Incrementing` и `Filtering`

```
trait Incrementing extends IntQueue:
  abstract override def put(x: Int) = super.put(x + 1)
```

¹ Вы должны использовать `with`, а не запятые, примешивая трейты в анонимный класс.

```
trait Filtering extends IntQueue:
  abstract override def put(x: Int) =
    if x >= 0 then super.put(x)
```

Теперь, располагая модифицирующими трейтами, можно выбрать, какой из них вам понадобится для той или иной очереди. Например, ниже показана очередь, в которой не только отфильтровываются отрицательные числа, но и ко всем сохраняемым числам прибавляется единица:

```
val queue = new BasicIntQueue with Incrementing with Filtering
queue.put(-1)
queue.put(0)
queue.put(1)
queue.get() // 1
queue.get() // 2
```

Порядок примешивания играет существенную роль¹. Конкретные правила даны в следующем разделе, но, грубо говоря, трейт, находящийся правее, вступает в силу первым. Когда метод вызывается в отношении экземпляра класса с примешанными трейтами, первым вызывается тот метод, который определен в самом правом трейте. Если этот метод выполняет вызов `super`, то вызывается метод, который определен в следующем трейте левее данного трейта, и т. д. В предыдущем примере сначала вызывается метод `put` трейта `Filtering`, следовательно, все начинается с того, что он удаляет отрицательные целые числа. Вторым вызывается метод `put` трейта `Incrementing`, следовательно, к оставшимся целым числам прибавляется единица.

Если расположить трейты в обратном порядке, то сначала к целым числам будет прибавляться единица и только *потом* те целые числа, которые все же останутся отрицательными, будут удалены:

```
val queue = new BasicIntQueue with
  Filtering with Incrementing
queue.put(-1)
queue.put(0)
queue.put(1)
queue.get() // 0
queue.get() // 1
queue.get() // 2
```

В общем, код, создаваемый в данном стиле, открывает перед вами широкие возможности для проявления гибкости. Примешивая эти три трейта в разных сочетаниях и разном порядке следования, можно определить 16 различных классов. Весьма впечатляющая гибкость для столь незначительного

¹ После того как трейт примешан к классу, вы также можете назвать его миксином.

объема кода, поэтому постарайтесь не проглядеть возможности организации кода в целях получения наращиваемых модификаций.

11.4. Почему не используется множественное наследование

Трейты позволяют наследовать из множества похожих на классы конструкций, но имеют весьма важные отличия от множественного наследования, имеющегося во многих языках программирования.

Одно из отличий, интерпретация `super`, играет особенно важную роль. При использовании множественного наследования метод, вызванный с помощью вызова `super`, может быть определен прямо там, где появляется этот вызов. При использовании трейтов вызываемый метод определяется путем *линеаризации*, то есть выстраивания в ряд классов и трейтов, примешанных в класс. Это то самое рассмотренное в предыдущем разделе отличие, которое позволяет выполнять наращивание модификаций.

Прежде чем рассмотреть линеаризацию, немного отвлечемся на то, как наращиваемые модификации выполняются в языке с традиционным множественным наследованием. Представим следующий код, однако на этот раз интерпретируемый не как примешивание трейтов, а как множественное наследование:

```
// Мысленный эксперимент с множественным наследованием
val q = new BasicIntQueue with Incrementing with Doubling
q.put(42) // Который из методов put будет вызван?
```

Сразу возникает вопрос: который из методов `put` будет задействован в этом вызове? Возможно, вступят в силу правила, согласно которым победу одержит самый последний суперкласс. В таком случае будет вызван метод из `Doubling`. В данном методе будет удвоен его аргумент, сделан вызов `super.put`, и на этом все. Не произойдет никакого увеличения на единицу! Кроме того, если бы действовало правило, при котором побеждал бы первый суперкласс, то в получающейся очереди целые числа увеличивались бы на единицу, но не удваивались. То есть не сбавывало бы никакое упорядочение.

Можно подумать и о том, как предоставить программистам возможность точно указывать при использовании вызова `super`, из какого именно суперкласса им нужен метод. На самом деле вы можете сделать это в Scala, указав суперкласс в квадратных скобках после `super`. Вот пример, в котором реализации `put` явно вызываются и для `Incrementing`, и для `Doubling`:

```
// Мысленный эксперимент с множественным наследованием
trait MyQueue extends BasicIntQueue,
    Incrementing, Doubling:

  def put(x: Int) =
    super[Incrementing].put(x) // (используется редко,
    super[Doubling].put(x)    // но допускается в Scala)
```

Если бы это был единственный подход Scala, то он породил бы новые проблемы (самой малой из которых будет многословие). В таком случае получается, что метод `put` базового класса вызывается *дважды*: один раз со значением, увеличенным на единицу, и один раз с удвоенным значением, но никогда с увеличенным и удвоенным значением.

Сравнение с методами Java по умолчанию

Начиная с Java 8, вы можете включать методы по умолчанию в интерфейсы. Хотя они и напоминают конкретные методы в трейтах Scala, но сильно отличаются, потому что Java не выполняет линеаризацию. Поскольку интерфейс не может указывать сегменты или расширять суперкласс, отличающийся от `Object`, метод по умолчанию может получить доступ к состоянию объекта только путем вызова методов интерфейса, реализованных подклассом. Напротив, конкретные методы в трейте Scala могут получить доступ к состоянию объекта через сегменты, объявленные в трейте, или путем вызова методов с `super`, которые получают доступ к сегментам супертрейтов или суперклассов. Кроме того, если вы прописываете класс Java, который наследует методы по умолчанию с одинаковыми подписями из двух разных суперинтерфейсов, Java потребует, чтобы вы самостоятельно реализовали этот метод в классе. Ваше внедрение может вызывать один или оба метода по умолчанию, указывая имя интерфейса перед `super`, например `"Doubling.super.put(x)"`. Для сравнения: Scala гарантирует, что ваш класс наследует ближайшую реализацию в линеаризации.

В Java методы по умолчанию направлены на то, чтобы разработчики библиотек могли добавлять методы к существующим интерфейсам. До Java 8 это было нецелесообразно, поскольку нарушало бинарную (двоичную) совместимость любого класса, реализующего интерфейс. Теперь же Java может использовать реализацию по умолчанию, если класс не предоставляет ее и даже если класс не был перекомпилирован с момента добавления нового метода в интерфейс.

При использовании множественного наследования данная задача просто не имеет правильного решения. Придется опять возвращаться к проектированию и реорганизовывать код. В отличие от этого с решением на основе применения трейтов в Scala все предельно понятно. Вы просто примешиваете трейты `Incrementing` и `Doubling`, и имеющееся в Scala особое правило, которое касается применения `super` в трейтах, позволяет добиться всего, чего вы хотели. Нечто здесь очевидно отличается от традиционного множественного наследования, но что именно?

Согласно уже данной подсказке ответом будет линеаризация. Когда с помощью ключевого слова `new` создается экземпляр класса, Scala берет класс со всеми его унаследованными классами и трейтами и располагает их в едином *линейном порядке*. Затем при любом вызове `super` внутри одного из таких классов вызывается тот метод, который идет следующим по порядку. Если во всех методах, кроме последнего, присутствует вызов `super`, то получается наращивание.

Описание конкретного порядка линеаризации дается в спецификации языка. Он сложноват, но вам нужно знать лишь главное: при любой линеаризации класс всегда следует впереди *всех* своих суперклассов и примешанных трейтов. Таким образом, при написании метода, содержащего вызов `super`, этот метод изменяет поведение суперкласса и примешанных трейтов, а не наоборот.

ПРИМЕЧАНИЕ

Далее в разделе описываются подробности линеаризации. Вы можете пропустить этот материал, если сейчас он вам неинтересен.

Главные свойства выполняющейся в Scala линеаризации показаны в следующем примере. Предположим, у вас есть класс `Cat` (Кот) — наследник класса `Animal` (Животное) — и два супертрейта: `Furry` (Пушистый) и `FourLegged` (Четырехлапый). Трейт `FourLegged` расширяет еще один трейт — `HasLegs` (С лапами):

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal, Furry, FourLegged
```

Иерархия наследования и линеаризация класса `Cat` показаны на рис. 11.1. Наследование изображено с помощью традиционной нотации UML [Rum04]: стрелки, на концах которых белые треугольники, обозначают наследование,

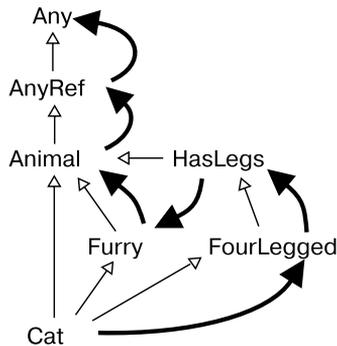


Рис. 11.1. Иерархия наследования и линейзации класса `Cat`

указывая на супертип. Стрелки с затемненными нетреугольными концами показывают линейзацию. Они указывают направление, в котором будут разрешаться вызовы `super`.

Линейзация `Cat` вычисляется от конца к началу следующим образом. Последняя часть линейзации `Cat` — линейзация его суперкласса `Animal`. Эта линейзация копируется без каких-либо изменений. (Линейзация каждого из этих типов показана в табл. 11.1.) Поскольку класс `Animal` не расширяет явным образом какой-либо суперкласс и в него не примешаны никакие супертрейты, то по умолчанию он расширяет класс `AnyRef`, который расширяет класс `Any`. Поэтому линейзация класса `Animal` имеет следующий вид:

`Animal` → `AnyRef` → `Any`

Предпоследней является линейзация первой примеси, трейта `Furry`, но все классы, которые уже присутствуют в линейзации класса `Animal`, теперь не учитываются, поэтому каждый класс в линейзации `Cat` появляется только раз. Результат выглядит следующим образом:

`Furry` → `Animal` → `AnyRef` → `Any`

Всему этому предшествует линейзация `FourLegged`, в которой также не учитываются любые классы, которые уже были скопированы в линейзациях суперкласса или первого примешанного трейта:

`FourLegged` → `HasLegs` → `Furry` → `Animal` → `AnyRef` → `Any`

И наконец, первым в линейзации класса `Cat` фигурирует сам этот класс:

`Cat` → `FourLegged` → `HasLegs` → `Furry` → `Animal` → `AnyRef` → `Any`

Когда любой из этих классов и трейтов вызывает метод через вызов `super`, вызываться будет первая реализация, которая в линеаризации расположена справа от него.

Таблица 11.1. Линеаризация типов в иерархии класса `Cat`

Тип	Линеаризация
<code>Animal</code>	<code>Animal, AnyRef, Any</code>
<code>Furry</code>	<code>Furry, Animal, AnyRef, Any</code>
<code>FourLegged</code>	<code>FourLegged, HasLegs, Animal, AnyRef, Any</code>
<code>HasLegs</code>	<code>HasLegs, Animal, AnyRef, Any</code>
<code>Cat</code>	<code>Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any</code>

11.5. Параметры трейтов

Начиная со Scala 3, трейты могут принимать параметры значений. Вы определяете их так же, как и классы: помещая их список через запятую рядом с именем трейта. Например, вы можете передать философское высказывание в качестве параметра трейту `Philosophical`, показанному в листинге 11.10.

Листинг 11.10. Определение параметра трейта

```
trait Philosophical(message: String):
  def philosophize = message
```

Теперь, когда трейт `Philosophical` принимает параметр, каждый подкласс должен передать свое собственное философское сообщение в качестве параметра для трейта, например, таким образом:

```
class Frog extends Animal,
  Philosophical("Я квакаю, значит, я существую!")

class Duck extends Animal,
  Philosophical("Я крякаю, значит, я существую!")
```

Если сказать кратко, вы должны указать значение параметра трейта при определении класса, который примешивается к трейту. Философия каждого класса `Philosophical` теперь будет определяться переданным параметром `message`:

```
val frog = new Frog
frog.philosophize // Я квакаю, значит, я существую!
val duck = new Duck
duck.philosophize // Я крякаю, значит, я существую!
```

Параметры трейта оцениваются непосредственно перед его инициализацией¹ и по умолчанию² доступны только его телу. Поэтому, чтобы использовать параметр сообщения в классе, реализующем трейт, необходимо захватить параметр, сделав его доступным из поля. Это поле гарантированно будет инициализировано и доступно для реализующего класса во время инициализации класса.

При использовании параметризованных трейтов вы можете заметить, что правила для параметров трейтов и классов немного отличаются. В обоих случаях вы можете выполнить инициализацию только один раз. Однако, хотя каждый класс может быть расширен только одним подклассом в иерархии, трейт может быть смешан несколькими подклассами. В этом случае вы должны его инициализировать при определении класса, который смешивается с трейтом, находящимся на самом высоком уровне в иерархии. Для примера рассмотрим суперкласс для любого «сознательного» животного, показанный в листинге 11.11.

Листинг 11.11. Указание параметра трейта

```
class ProfoundAnimal extends Animal,
    Philosophical("В начале было дело.")
```

Если суперкласс класса сам по себе не расширяет трейт, вы должны указать параметр трейта при определении класса. Суперклассом `ProfoundAnimal` является `Animal`, а `Animal` не расширяет `Philosophical`. Поэтому вы должны указать параметр трейта при определении `ProfoundAnimal`.

С другой стороны, если суперкласс класса также расширяет трейт, то вам больше не нужно указывать параметр трейта при определении класса. Это показано в листинге 11.12.

Листинг 11.12. Без указания параметра трейта

```
class Frog extends ProfoundAnimal, Philosophical
```

¹ Параметры трейта в Scala 3 заменяют ранние инициализаторы в Scala 2.

² Как и в случае с параметрами класса, вы можете использовать параметрическое поле для определения общедоступного поля, инициализированного переданным параметром трейта. Это будет продемонстрировано в разделе 20.5.

Суперкласс `Frog` (Лягушка) `ProfoundAnimal` расширяет трейт `Philosophical` и предоставляет свой параметр сообщения `message`. При определении `Frog` вы больше не можете указывать сообщение в качестве параметра, поскольку он уже был заполнен `ProfoundAnimal`. Таким образом, эта лягушка будет демонстрировать поведение, вытекающее в результате инициализации `ProfoundAnimal` в `Philosophical`:

```
val frog = new Frog
frog.philosophize // В начале было дело.
```

И наконец, трейты не могут передавать параметры своим родительским трейтам. Например, рассмотрим трейт `PhilosophicalAnimal`, который расширяет трейт `Philosophical`:

```
trait PhilosophicalAnimal extends Animal with Philosophical
```

Вам может показаться, что думающая лягушка определяется следующим образом:

```
// Не компилируется
class Frog extends PhilosophicalAnimal(
    "Я квакаю, значит, я существую!")
```

Однако это не работает. Вместо этого вы должны явно указать сообщение для `Philosophical` при определении класса `Frog`, например, так:

```
class Frog extends
    Philosophical("Я квакаю, значит, я существую!"),
    PhilosophicalAnimal
```

Или так:

```
class Frog extends PhilosophicalAnimal,
    Philosophical("Я квакаю, значит, я существую!")
```

Резюме

В этой главе мы показали работу трейтов и порядок их применения в нескольких часто встречающихся идиомах. Вы увидели, что трейты похожи на множественное наследование. Но благодаря тому, что в трейтах вызовы `super` интерпретируются с помощью линейаризации, удается не только избавиться от некоторых трудностей традиционного множественного наследования, но и воспользоваться наращиванием модификаций поведения программы.

Вдобавок мы рассмотрели трейт `Ordered` и изучили порядок создания собственных расширяющих трейтов.

А теперь, усвоив все эти аспекты, нам предстоит вернуться немного назад и посмотреть на трейты в целом с другого ракурса. Трейты не просто поддерживают средства выражения, рассмотренные в данной главе, — они являются базовыми блоками кода, допускающими их повторное использование с помощью механизма наследования. Благодаря этому многие опытные программисты, которые работают на Scala, на ранних стадиях реализации начинают с трейтов. Любой трейт не может охватить всю концепцию, ограничиваясь лишь ее фрагментом. По мере того как конструкция приобретает все более четкие очертания, фрагменты путем примешивания трейтов могут объединяться в более полноценные концепции.

12 Пакеты, импорты и экспорты

В ходе работы над программой, особенно объемной, важно свести к минимуму *сцепление*, то есть степень взаимозависимости различных частей программы. Низкое сцепление (low coupling) сокращает риск того, что незначительное, казалось бы, безобидное изменение в одной части программы вызовет разрушительные последствия для другой части. Один из способов сведения сцепления к минимуму — написание программы в модульном стиле. Программа разбивается на несколько меньших по размеру модулей, у каждого из которых есть внутренняя и внешняя части. При работе над внутренней частью модуля, то есть над его *реализацией*, нужно согласовывать действия только с другими программистами, работающими над созданием того же самого модуля. И лишь в случае необходимости внести изменения в его внешнюю часть, то есть в *интерфейс*, приходится координировать свои действия с разработчиками других модулей.

В этой главе мы покажем конструкции, помогающие программировать в модульном стиле. Мы рассмотрим вопросы помещения кода в пакеты, создания имен, видимых при импортировании, и управления видимостью определений с помощью модификаторов доступа. Эти конструкции сходны по духу с конструкциями, имеющимися в Java, за исключением некоторых различий, которые, как правило, выражаются в их более последовательном характере. Поэтому данную главу стоит прочитать даже тем, кто хорошо разбирается в Java.

12.1. Помещение кода в пакеты

Код Scala размещается в глобальной иерархии пакетов платформы Java. Все показанные до сих пор в этой книге примеры кода размещались в *безымянных* пакетах. Поместить код в именованные пакеты в Scala можно двумя

способами. Первый — поместить содержимое всего файла в пакет, указав директиву `package` в самом начале файла (листинг 12.1).

Листинг 12.1. Помещение в пакет всего содержимого файла

```
package bobsrockets.navigation
class Navigator
```

Указание директивы `package` в листинге 12.1 приводит к тому, что класс `Navigator` помещается в пакет по имени `bobsrockets.navigation`. По-видимому, это программные средства навигации, разработанные корпорацией Bob's Rockets.

ПРИМЕЧАНИЕ

Поскольку код Scala — часть экосистемы Java, то тем пакетам Scala, которые выпускаются открытыми, рекомендуется соответствовать принятому в Java соглашению по присваиванию имени с обратным порядком следования доменных имен. Поэтому наиболее подходящим именем пакета для класса `Navigator` может быть `com.bobsrockets.navigation`. Но в данной главе мы отбросим `com.`, чтобы было легче разобраться в примерах.

Другой способ, позволяющий в Scala помещать код в пакеты, больше похож на использование пространств имен C#. За пакетом следует двоеточие и раздел кода с отступом, содержащий определения, которые входят в пакет. Такой синтаксис называется *пакетированием*. Показанное в листинге 12.2 пакетирование имеет тот же эффект, что и код в листинге 12.1.

Листинг 12.2. Длинная форма простого объявления пакетирования

```
package bobsrockets.navigation:
  class Navigator
```

Кроме того, для таких простых примеров можно воспользоваться «синтаксическим сахаром», показанным в листинге 12.1. Но лучше все же реализовать один из вариантов более универсальной системы записи, позволяющей разместить разные части файла в разных пакетах. Например, если вы хотите отправить по электронной почте или опубликовать на дискуссионном форуме фрагмент кода Scala, включающий несколько пакетов, вы можете использовать пакетирование, как показано в листинге 12.3.

Листинг 12.3. Несколько пакетов в одном и том же файле

```
package bobsrockets:
  package navigation:
    // в пакете bobsrockets.navigation
    class Navigator
```

```
package launch:  
  
  // в пакете bobsrockets.navigation.launch  
  class Booster
```

12.2. Краткая форма доступа к родственному коду

Представление кода в виде иерархии пакетов не только помогает просматривать код, но и сообщает компилятору, что части кода в одном и том же пакете как-то связаны между собой. В Scala эта родственность позволяет при доступе к коду, находящемуся в одном и том же пакете, применять краткие имена.

В листинге 12.4 приведено три примера. В первом, согласно ожиданиям, к классу можно обращаться из его собственного пакета, не указывая префикс. Именно поэтому `new StarMap` проходит компиляцию. Класс `StarMap` находится в том же самом пакете по имени `bobsrockets.navigation`, что и выражение `new`, которое к нему обращается, поэтому указывать префикс в виде имени пакета не нужно.

Листинг 12.4. Краткая форма обращения к классам и пакетам

```
package bobsrockets:  
  package navigation:  
    class Navigator:  
      // Указывать bobsrockets.navigation.StarMap не нужно  
      val map = new StarMap  
    class StarMap  
  
  class Ship:  
    // Указывать bobsrockets.navigation.Navigator не нужно  
    val nav = new navigation.Navigator  
  
  package fleets:  
    class Fleet:  
      // Указывать bobsrockets.Ship не нужно  
      def addShip = new Ship
```

Во втором примере обращение к самому пакету может производиться из того же пакета, в котором он находится, без указания префикса. В листинге 12.4 показано, как создается экземпляр класса `Navigator`. Выражение `new` появляется в пакете `bobsrockets`, который, в свою очередь, содержится в пакете `bobsrockets.navigation`. Поэтому обращение к последнему можно указывать просто как `navigation`.

В третьем примере показано, что при использовании синтаксиса вложенного пакетирования все имена, доступные в пространстве имен вне пакета, доступны также и внутри него. Это обстоятельство позволяет в листинге 12.4 в `addShip()` создать новый экземпляр класса, воспользовавшись выражением `new Ship`. Метод определен внутри двух пакетов: внешнего `bobsrockets` и внутреннего `bobsrockets.fleets`. Поскольку к объекту `Ship` доступ можно получить во внешнем пакете, то из `addShip()` можно воспользоваться ссылкой на него.

Следует заметить, что такая форма доступа применима только в том случае, если пакеты вложены друг в друга явным образом. Если в каждый файл будет помещаться лишь один пакет, то, как и в Java, доступны будут только те имена, которые определены в текущем пакете. В листинге 12.5 пакет `bobsrockets.fleets` был перемещен на самый верхний уровень. Он больше не заключен в пакет `bobsrockets`, поэтому имена из `bobsrockets` в его пространстве имен отсутствуют. В результате использование выражения `new Ship` вызовет ошибку компиляции.

Листинг 12.5. Обозначения, заключенные в пакеты, не доступны автоматически

```
package bobsrockets:
    class Ship

package bobsrockets.fleets:
    class Fleet:
        // Не пройдет компиляцию! Ship вне области видимости.
        def addShip = new Ship
```

Если обозначение вложенности пакетов с отступами приводит к неудобному для вас сдвигу кода вправо, то можно воспользоваться несколькими указаниями директивы `package` без отступа¹. Например, в показанном далее коде класс `Fleet` определяется в двух вложенных пакетах, `bobsrockets` и `fleets`, точно так же, как это было сделано в листинге 12.4:

```
package bobsrockets
package fleets
class Fleet:
    // Указывать bobsrockets.Ship не нужно
    def addShip = new Ship
```

¹ Этот стиль, в котором используются несколько директив `package` без фигурных скобок, называется объявлением цепочки пакетов.

Важно знать еще об одной, последней особенности. Иногда в области видимости оказывается слишком много всего и имена пакетов скрывают друг друга. В листинге 12.6 пространство имен класса `MissionControl` включает три отдельных пакета по имени `launch`! Один пакет `launch` находится в `bobsrockets.navigation`, один — в `bobsrockets` и еще один — на верхнем уровне. А как тогда ссылаться на `Booster1`, `Booster2` и `Booster3`?

Листинг 12.6. Доступ к скрытым именам пакетов

```
// в файле launch.scala
package launch:
  class Booster3

// в файле bobsrockets.scala
package bobsrockets:

  package launch:
    class Booster2

  package navigation:
    package launch:
      class Booster1

  class MissionControl:
    val booster1 = new launch.Booster1
    val booster2 = new bobsrockets.launch.Booster2
    val booster3 = new _root_.launch.Booster3
```

Проще всего обратиться к первому из них. Ссылка на само имя `launch` приведет вас к пакету `bobsrockets.navigation.launch`, поскольку этот пакет `launch` определен в ближайшей области видимости. Поэтому к первому из `booster`-классов можно обратиться просто как к `launch.Booster1`. Ссылка на второй подобный класс также указывается без каких-либо особенных приемов. Можно указать `bobsrockets.launch.Booster2` и не оставить ни малейших сомнений о том, к какому из трех классов происходит обращение. Открытым остается лишь вопрос по поводу третьего класса `booster`: как обратиться к `Booster3` при условии, что пакет на самом верхнем уровне перекрывается вложенными пакетами?

Чтобы помочь справиться с подобной ситуацией, Scala предоставляет имя пакета `_root_`, являющегося внешним по отношению к любым другим создаваемым пользователем пакетам. Иначе говоря, каждый пакет верхнего уровня, который может быть создан, рассматривается как члены пакета `_root_`. Например, и `launch`, и `bobsrockets` в листинге 12.6 являются членами пакета `_root_`. В результате этого `_root_.launch` позволяет обратиться к пакету `launch` самого верхнего уровня, а `_root_.launch.Booster3` обозначает внешний класс `booster`.

12.3. Импортирование кода

В Scala пакеты и их члены могут импортироваться с использованием директивы `import`. Затем ко всему, что было импортировано, можно получить доступ, указав простое имя, такое как `File`, не используя такое развернутое имя, как `java.io.File`. Рассмотрим, к примеру, код, показанный в листинге 12.7.

Листинг 12.7. Превосходные фрукты от Боба, готовые к импорту

```
package bobsdelights

abstract class Fruit(
  val name: String,
  val color: String
)

object Fruits:
  object Apple extends Fruit("apple", "red")
  object Orange extends Fruit("orange", "orange")
  object Pear extends Fruit("pear", "yellowish")
  val menu = List(Apple, Orange, Pear)
```

Указание директивы `import` делает члены пакета или объект доступными по их именам, исключая необходимость ставить перед ними префикс с именем пакета или объекта. Рассмотрим ряд простых примеров:

```
// простая форма доступа к Fruit
import bobsdelights.Fruit

// простая форма доступа ко всем членам bobsdelights
import bobsdelights.*

// простая форма доступа ко всем членам Fruits
import bobsdelights.Fruits.*
```

Первый пример относится к импортированию отдельно взятого Java-типа, а во втором показан Java-импорт *до востребования* (on-demand). Если в Scala 2 импорты до востребования записывались с замыкающим знаком подчеркивания (`_`), то в Scala 3 он был заменен знаком звездочки (`*`), чтобы соответствовать другим языкам. Третья из показанных директив `import` относится к Java-импорту статических полей класса.

Эти три директивы `import` дают представление о том, что можно делать с помощью импортирования, но в Scala импортирование носит более универсальный характер. В частности, оно может быть где угодно, а не только в начале компилируемого модуля. К тому же при импортировании можно

ссылаться на произвольные значения. Например, возможен импорт, показанный в листинге 12.8.

Листинг 12.8. Импортирование членов обычного объекта (не одиночки)

```
def showFruit(fruit: Fruit) =
  import fruit.*
  s"${name}s are $color"
```

Метод `showFruit` импортирует все члены параметра `fruit`, относящегося к типу `Fruit`. Следующая инструкция `println` может непосредственно ссылаться на `name` и `color`. Эти две ссылки — эквиваленты ссылок `fruit.name` и `fruit.color`. Такой синтаксис пригодится, в частности, при использовании объектов в качестве модулей. Соответствующее описание будет дано в главе 7.

Гибкость импортирования в Scala

Директива `import` работает в Scala намного более гибко, чем в Java. Эта гибкость характеризуется тремя принципиальными отличиями. Импорт кода в Scala:

- может появляться где угодно;
- позволяет, помимо пакетов, ссылаться на объекты (одиночки или обычные);
- позволяет изменять имена или скрывать некоторые из импортированных членов.

Еще один из факторов гибкости импорта кода в Scala заключается в возможности импортировать пакеты как таковые, без их непакетированного наполнения. Смысл в этом будет только в том случае, если предполагается, что в пакете заключены другие пакеты. Например, в листинге 12.9 импортируется пакет `java.util.regex`. Благодаря этому `regex` можно использовать с указанием его простого имени. Чтобы обратиться к объекту-одиночке `Pattern` из пакета `java.util.regex`, можно, как показано в данном листинге, просто воспользоваться идентификатором `regex.Pattern`.

Листинг 12.9. Импортирование имени пакета

```
import java.util.regex

class AStarB:
  // обращение к java.util.regex.Pattern
  val pat = regex.Pattern.compile("a*b")
```

При импортировании кода Scala позволяет также переименовывать или скрывать члены. Для этого *импорт заключается* в фигурные скобки с указанием перед получившимся в результате блоком того объекта, из которого импортируются его члены. Рассмотрим несколько примеров:

```
import Fruits.{Apple, Orange}
```

Здесь из объекта `Fruits` импортируются только его члены `Apple` и `Orange`.

```
import Fruits.{Apple as McIntosh, Orange}
```

Здесь из объекта `Fruits` импортируются два члена, `Apple` и `Orange`. Но объект `Apple` переименовывается в `McIntosh`, поэтому к нему можно обращаться либо `Fruits.Apple`, либо `McIntosh`. Директива переименования всегда имеет вид `<исходное_имя> as <новое_имя>`. Если вам нужно импортировать и переименовать только одно имя, фигурные скобки можно не ставить:

```
import java.sql.Date as SDate
```

Здесь под именем `SDate` импортируется класс данных SQL, чтобы можно было в то же время импортировать обычный класс для работы с датами Java просто как `Date`.

```
import java.sql as S
```

Здесь под именем `S` импортируется пакет `java.sql`, чтобы можно было воспользоваться кодом вида `S.Date`.

```
import Fruits.*
```

Здесь импортируются все члены объекта `Fruits`. Это означает то же самое, что и `import Fruits.*`.

```
import Fruits.{Apple as McIntosh, *}
```

Здесь импортируются все члены объекта `Fruits`, но `Apple` переименовывается в `McIntosh`.

```
import Fruits.{Pear as _, *}
```

Здесь импортируются все члены объекта `Fruits`, за исключением `Pear`. Директива вида `<исходное_имя> => _` исключает `<исходное_имя>` из импортируемых имен. В определенном смысле переименование чего-либо в `_` говорит о полном сокрытии переименованного члена. Это помогает избегать неоднозначностей. Предположим, имеется два пакета, `Fruits` и `Laptops`, и в каждом

из них определен класс `Apple`. Если нужно получить только ноутбук под названием `Apple`, а не фрукт, то можно воспользоваться двумя импортами по запросу:

```
import Laptops.*
import Fruits.{Apple as _, *}
```

Будут импортированы все члены `Laptops` и все члены `Fruits`, за исключением `Apple`.

Эти примеры демонстрируют поразительную гибкость, которую предлагает Scala в вопросах избирательного импортирования членов, возможно, даже под другими именами. Таким образом, директива `import` может состоять из следующих селекторов:

- простого имени x , которое включается в набор импортируемых имен;
- директивы переименования x as y . Член по имени x будет виден под именем y ;
- директивы сокрытия x as `_`. Имя x исключается из набора импортируемых имен;
- элемента «поймать все» (catch-all) `*`. Импортируются все члены, за исключением тех, которые были упомянуты в предыдущей директиве. Если указан элемент «поймать все», то в списке селекторов импортирования он должен стоять последним.

Самые простые `import`-директивы, показанные в начале данного раздела, могут рассматриваться как специальные сокращения директив `import` с селекторами. Например, `import p.*` — эквивалент `import p.{*}`, а `import p.n` — эквивалент `import p.{n}`.

12.4. Неявное импортирование

Scala неявно добавляет импортируемый код в каждую программу. По сути, происходит то, что произошло бы при добавлении в самое начало каждого исходного файла с расширением `.scala` следующих трех директив `import`:

```
import java.lang.* // все из пакета java.lang
import scala.*     // все из пакета scala
import Predef.*    // все из объекта Predef
```

В пакете `java.lang` содержатся стандартные классы Java. Он всегда неявно импортируется в исходные файлы Scala¹. Неявное импортирование `java.lang` позволяет вам, например, использовать вместо `java.lang.Thread` просто идентификатор `Thread`.

Теперь уже вряд ли приходится сомневаться в том, что в пакете `scala` находится стандартная библиотека Scala, в которой содержатся многие самые востребованные классы и объекты. Поскольку пакет `scala` импортируется неявно, то можно, к примеру, вместо `scala.Int` указать просто `Int`.

В объекте `Predef` содержится множество определений псевдонимов типов, методов и преобразований, которые обычно используются в программах на Scala. Например, `Predef` импортируется неявно, поэтому можно вместо `Predef.assert` задействовать просто идентификатор `assert`.

Эти три директивы `import` трактуются особым образом, позволяющим тому импорту, который указан позже, перекрывать указанный ранее. К примеру, класс `StringBuilder` определен в обоих пакетах `scala` и `java.lang`. Импорт `scala` перекрывает импорт `java.lang`, поэтому простое имя `StringBuilder` будет ссылаться на `scala.StringBuilder`, а не на `java.lang.StringBuilder`.

12.5. Модификаторы доступа

Члены пакетов, классов или объектов могут быть помечены модификаторами доступа `private` и `protected`. Они ограничивают доступ к членам, позволяя обращаться к ним только из определенных областей кода. Трактовка модификаторов доступа Scala примерно соответствует принятой в Java, но при этом имеет ряд весьма важных отличий, которые рассматриваются в данном разделе.

Приватные члены

Приватные члены трактуются в Scala точно так же, как и в Java. Член с пометкой `private` виден только внутри класса или объекта, в котором содержится его определение. В Scala это правило распространяется и на внутренние классы. Данная трактовка более последовательна, но отличается от принятой в Java. Рассмотрим пример, показанный в листинге 12.10.

¹ Изначально имелась также реализация Scala на платформе .NET, где вместо этого импортировалось пространство имен `System`, .NET-аналог пакета `java.lang`.

Листинг 12.10. Отличие приватного доступа в Scala от такого же доступа в Java

```
class Outer:

  class Inner:
    private def f = "f"
    class InnerMost:
      f // OK

  (new Inner).f // ошибка: нет доступа к f
```

В Scala обращение `(new Inner).f` недопустимо, поскольку приватное объявление `f` сделано в классе `Inner`, а попытка обращения делается не из данного класса. В отличие от этого первое обращение к `f` в классе `InnerMost` вполне допустимо, поскольку содержится в теле класса `Inner`. В Java допустимы оба обращения, так как в данном языке разрешается обращение из внешнего класса к приватным членам его внутренних классов.

Защищенные члены

Доступ к защищенным членам в Scala также менее свободен, чем в Java. В Scala обратиться к защищенному члену можно только из подклассов того класса, в котором был определен этот член. В Java обращение возможно и из других классов того же самого пакета. В Scala есть еще один способ достижения того же самого эффекта¹, поэтому модификатор `protected` можно оставить без изменений. Защищенные виды доступа показаны в листинге 12.11.

Листинг 12.11. Отличие защищенного доступа в Scala от такого же доступа в Java

```
package p:

  class Super:
    protected def f = "f"

  class Sub extends Super:
    f

  class Other:
    (new Super).f // ошибка: нет доступа к f
```

В листинге 12.11 обращение к `f` в классе `Sub` вполне допустимо, поскольку объявление `f` было сделано с модификатором `protected` в `Super`, а `Sub` —

¹ Используя спецификаторы, рассматриваемые ниже, в подразделе «Область защиты».

подкласс `Super`. В отличие от этого обращение к `f` в `Other` недопустимо, поскольку `Other` не является наследником `Super`. В Java последнее обращение все равно будет разрешено, так как `Other` находится в том же самом пакете, что и `Super`.

Публичные члены

В Scala нет явного модификатора для публичных членов: любой член, не помеченный как `private` или `protected`, является публичным. К публичным членам можно обращаться откуда угодно.

Область защиты

Модификаторы доступа в Scala могут дополняться спецификаторами. Модификатор вида `private[X]` или `protected[X]` означает, что доступ закрыт или защищен вплоть до `X`, где `X` определяет некий внешний пакет, класс или объект-одиночку.

Специфицированные модификаторы доступа дают возможность весьма четко обозначить границы управления видимостью. В частности, они позволяют выразить понятия доступности, имеющиеся в Java, такие как приватность пакета, защищенность пакета или закрытость вплоть до самого внешнего класса, которые невозможно выразить напрямую с помощью простых модификаторов, используемых в Scala. Но помимо этого, они позволяют выразить правила доступности, которые не могут быть выражены в Java.

В листинге 12.12 представлен пример с использованием множества спецификаторов доступа. Здесь класс `Navigator` помечен как `private[bobsrockets]`. Это значит, он имеет область видимости, охватывающую все классы и объекты, которые содержатся в пакете `bobsrockets`. В частности, доступ к `Navigator` разрешен в объекте `Vehicle`, поскольку `Vehicle` содержится в пакете `launch`, который, в свою очередь, содержится в пакете `bobsrockets`. В то же время весь код, находящийся за пределами пакета `bobsrockets`, не может получить доступ к классу `Navigator`.

Листинг 12.12. Придание гибкости областям защиты с помощью спецификаторов доступа

```
package bobsrockets

package navigation:
  private[bobsrockets] class Navigator:
```

```

protected[navigation] def useStarChart() = {}
class LegOfJourney:
  private[Navigator] val distance = 100

package launch:
  import navigation.*
  object Vehicle:
    private[launch] val guide = new Navigator

```

Этот прием особенно полезен при разработке крупных проектов, содержащих несколько пакетов. Он позволяет определять элементы, видимость которых распространяется на несколько подчиненных пакетов проекта, оставляя их невидимыми для клиентов, являющихся внешними по отношению к данному проекту¹.

Разумеется, действие спецификатора `private` может распространяться и на непосредственно окружающий пакет. В листинге 12.12 показан пример модификатора доступа `guide` в объекте `Vehicle`. Такой модификатор доступа эквивалентен имеющемуся в Java доступу, ограниченному пределами одного пакета.

Все спецификаторы также могут применяться к модификатору `protected` со значениями, аналогичными тем, с которыми они применяются к модификатору `private`. То есть модификатор `protected[X]` в классе `C` позволяет получить доступ к определению с подобной пометкой во всех подклассах `C`, а также во внешнем пакете, классе или объекте с названием `X`. Например, метод `useStarChart` в приведенном выше листинге 12.12 доступен из всех подклассов `Navigator`, а также из всего кода, содержащегося во внешнем пакете `navigation`. В результате получается точное соответствие значению модификатора `protected` в Java.

Спецификаторы модификатора `private` могут также ссылаться на окружающий (внешний) класс или объект. Например, показанная в листинге 12.12 переменная `distance` в классе `LegOfJourney` имеет пометку `private[Navigator]`, следовательно, видима из любого места в классе `Navigator`. Тем самым ей придаются такие же возможности видимости, как и приватным членам внутренних классов в Java. Модификатор `private[C]`, где `C` — самый внешний класс, аналогичен простому модификатору `private` в Java.

В качестве резюме в табл. 12.1 приведен список действий спецификаторов модификатора `private`. В каждой строке показан модификатор `private`

¹ Эта техника возможна в Java благодаря системе модулей, представленной в JDK 9.

со спецификатором и раскрыто его значение при применении в отношении переменной `distance`, объявленной в классе `LegOfJourney` в листинге 12.12.

Таблица 12.1. Действия спецификаторов `private` в отношении `LegOfJourney.distance`

Спецификатор	Действие
Без указания модификатора доступа	Открытый доступ
<code>private[bobsrockets]</code>	Доступ в пределах внешнего пакета
<code>private[navigation]</code>	Аналог имеющейся в Java видимости в пределах пакета
<code>private[Navigator]</code>	Аналог имеющегося в Java модификатора <code>private</code>
<code>private[LegOfJourney]</code>	Аналог имеющегося в Scala модификатора <code>private</code>

Видимость и объекты-компаньоны

В Java статические члены и члены экземпляра принадлежат одному и тому же классу, поэтому модификаторы доступа применяются к ним одинаково. Вы уже видели, что в Scala статических членов нет, вместо них может быть объект-компаньон, содержащий члены, существующие в единственном экземпляре. Например, в листинге 12.13 объект `Rocket` — компаньон класса `Rocket`.

Листинг 12.13. Обращение к приватным членам класса- и объекта-компаньона

```
class Rocket:
  import Rocket.fuel
  private def canGoHomeAgain = fuel > 20

object Rocket:
  private def fuel = 10
  def chooseStrategy(rocket: Rocket) =
    if rocket.canGoHomeAgain then
      goHome()
    else
      pickAStar()

  def goHome() = {}
  def pickAStar() = {}
```

Что касается приватного или защищенного доступа, то в правилах доступа, действующих в Scala, объектам- и классам-компаньонам даются особые

привилегии. Класс делится всеми своими правами доступа со своим объектом-компаньоном, и *наоборот*. В частности, объект может обращаться ко всем приватным членам своего класса-компаньона точно так же, как класс может обращаться ко всем приватным членам своего объекта-компаньона.

Например, в показанном выше листинге 12.13 класс `Rocket` может обращаться к методу `fuel`, который объявлен приватным в объекте `Rocket`. Аналогично этому объект `Rocket` может обращаться к приватному методу `canGoHomeAgain` в классе `Rocket`.

Одно из исключений, которое нарушает аналогию между Scala и Java, касается защищенных статических членов. Защищенный статический член Java-класса `C` может быть доступен во всех подклассах `C`. В отличие от этого в наличии защищенного члена в объекте-компаньоне нет никакого смысла, поскольку у объектов-одиночек нет никаких подклассов.

12.6. Определения верхнего уровня

До сих пор единственным встречающимся вам кодом, добавляемым к пакетам, были классы, трейты и одиночные объекты. Они, несомненно, являются наиболее распространенными определениями, помещаемыми на самом верхнем уровне пакета. Но Scala не ограничивает вас только этим перечнем — любые виды определений, которые можно помещать внутри класса, могут присутствовать и на верхнем уровне пакета. На самый верхний уровень пакета можно смело помещать любой вспомогательный метод, который хотелось бы иметь в области видимости всего пакета.

Для этого поместите определение в пакет, как вы бы сделали в случае с классом, чертой или объектом. Пример показан в листинге 12.14. Файл `ShowFruit.scala` объявляет вспомогательный метод `showFruit` из листинга 12.8 как участник пакета `bobsdelights`.

Листинг 12.14. Объект пакета

```
// в файле ShowFruit.scala
package bobsdelights

def showFruit(fruit: Fruit) =
  import fruit.*
  s"${name}s are $color"

// в файле PrintMenu.scala
package printmenu
```

```
import bobsdelights.Fruits
import bobsdelights.showFruit

object PrintMenu:
  def main(args: Array[String]) =
    println(
      for fruit <- Fruits.menu yield
        showFruit(fruit)
    )
```

При наличии этого определения любой другой код в любом пакете может импортировать метод точно так же, как класс. Например, в данном листинге показан самостоятельный объект `PrintMenu`, который находится в другом пакете. `PrintMenu` может импортировать вспомогательный метод `showFruit` так же, как и класс `Fruit`.

Забегая вперед, следует отметить, что есть и другие способы использования определений верхнего уровня, которые еще не были вам показаны. Эти определения часто применяются для содержания псевдонимов типов, предназначенных для всего пакета (см. главу 20) и методов расширения (см. главу 22). Пакет `scala` включает определения верхнего уровня, которые доступны всему коду Scala.

12.7. Экспорты

В разделе 10.11 мы рекомендовали предпочитать композицию, а не наследование, особенно если вашей основной целью является повторное использование кода. Это применение принципа наименьшей мощности: композиция рассматривает компоненты как «черные ящики», а наследование путем переопределения влияет на их внутреннюю работу. Иногда тесная связь, подразумеваемая наследованием, является лучшим решением проблемы, но там, где в этом нет необходимости, лучше использовать более слабую связь композиции.

В большинстве популярных объектно-ориентированных языков программирования проще использовать наследование. Например, в Scala 2 для него требовалось только предложение `extends`, в то время как композиция требовала подробного описания последовательности серверов пересылки. Таким образом, подавляющая часть объектно-ориентированных языков подталкивала программистов к решению, которое зачастую оказывается слишком мощным. Экспорты в Scala 3 направлены на устранение этого дисбаланса. Эта новая функция помогает выразить отношения композиции так же крат-

ко и просто, как и отношения наследования. Она обеспечивает большую гибкость, чем директива `extends`, поскольку в ней можно переименовывать или исключать элементы.

В качестве примера представьте, что вы хотите создать тип для представления целых положительных чисел. Вы могли бы определить его следующим образом¹:

```
case class PosInt(value: Int):
  require(value > 0)
```

Этот класс позволяет указать в типе, что целое число является положительным. Однако для выполнения любых арифметических операций с `Int` в этом коде вам потребуется получить доступ к значению `value`:

```
val x = PosInt(88)
x.value + 1 // 89
```

Вы можете сделать это удобнее, реализовав метод `+` на `PosInt`, который делегируется методу `+` базового значения `value`, например, так:

```
case class PosInt(value: Int):
  require(value > 0)
  def +(x: Int): Int = value + x
```

С добавлением этого метода пересылки теперь можно выполнять сложение целых чисел в `PosInts` без необходимости доступа к значению `value`:

```
val x = PosInt(77)
x + 1 // 78
```

Вы могли бы сделать `PosInt` еще более удобным, реализовав все методы `Int`, но их более ста. Если бы вы могли определить `PosInt` как подкласс `Int`, вы бы унаследовали все эти методы и вам бы не понадобилась их повторная реализация. Но поскольку `Int` является окончательным, вы не можете этого сделать. Вот почему класс `PosInt` должен использовать композицию и делегирование вместо наследования.

В Scala 3 вы можете использовать ключевое слово `export` для указания нужных вам методов пересылки, и компилятор сгенерирует их для вас. Вот как можно создать класс `PosInt`, который объявляет методы пересылки к соответствующим именованным методам базового значения `value`:

¹ Два альтернативных способа создания этого типа, которые позволяют избежать упаковки, — это `AnyVals` и непрозрачные типы. `AnyVals` будут рассмотрены в разделе 17.4.

```
case class PosInt(value: Int):  
  require(value > 0)  
  export value.*
```

С помощью этой конструкции вы можете вызывать любые методы в `PosInt`, которые объявлены непосредственно на `Int`:

```
val x = PosInt(99)  
x + 1 // 100  
x - 1 // 98  
x / 3 // 33
```

Директива экспорта создает окончательные методы, называемые *экспортными псевдонимами*, для каждой перегруженной формы каждого имени экспортируемого метода. Например, метод `+`, который примет значение `Int`, будет иметь такую подпись в `PosInt`:

```
final def +(x: Int): Int = value + x
```

Вам доступны все различные формы синтаксиса для импорта с экспортом. Например, вы можете не использовать операторы символического сдвига `<<`, `>>` и `>>>` в `PosInt`:

```
val x = PosInt(24)  
x << 1 // 48 (shift left)  
x >> 1 // 12 (shift right)  
x >>> 1 // 12 (unsigned shift right)
```

У вас есть возможность переименовывать эти операторы при экспорте так же, как и идентификаторы при импорте, — с помощью `as`. Давайте рассмотрим пример:

```
case class PosInt(value: Int):  
  require(value > 0)  
  export value.{<< as shl, >> as shr, >>> as ushr, *}
```

С учетом этой директивы экспорта операторы сдвига в `PosInt` больше не будут иметь символических имен:

```
val x = PosInt(24)  
x shl 1 // 48  
x shr 1 // 12  
x ushr 1 // 12
```

Вы также можете исключить методы из экспорта подстановочных знаков с помощью `as _`. Например, сдвиг вправо (`>>`) и беззнаковый сдвиг вправо (`>>>`) всегда дают одинаковый результат для целого положительного числа,

поэтому возможно использование только одного оператора сдвига вправо — `shr`. Этого можно добиться, опустив оператор `>>>` с помощью `>>> as _`, например, так:

```
case class PosInt(value: Int):
  require(value > 0)
  export value.{<< as shl, >> as shr, >>> as _, *}
```

Теперь для метода `>>>` не создается никакого псевдонима:

```
scala> val x = PosInt(39)
val x: PosInt = PosInt(39)
```

```
scala> x shr 1
val res0: Int = 19
```

```
scala> x >>> 1
1 |x >>> 1
  |^^^^^
  |value >>> is not a member of PosInt
```

Резюме

В данной главе мы показали основные конструкции, предназначенные для разбиения программы на пакеты. Благодаря этому вы имеете простую и полезную разновидность модульности и можете работать с весьма большими объемами кода, не допуская взаимного влияния различных его частей. Существующая в Scala система по духу аналогична пакетированию, используемому в Java, но с некоторыми отличиями: в Scala проявляется более последовательный или же более универсальный подход. Вы также видели новую функцию, `exports`, которая призвана сделать композицию такой же удобной, как и наследование, для повторного использования кода.

В следующей главе мы переключимся на сопоставление шаблонов.

13 Сопоставление с образцом

Данная глава описывает понятия *case-классов* и *сопоставления с образцом* (pattern matching) — конструкций, способствующих созданию обычных, неинкапсулированных структур данных. Особенно полезны эти две конструкции при работе с древовидными рекурсивными данными.

Если вам уже приходилось программировать на функциональном языке, то, возможно, сопоставление с образцом вам уже знакомо. А вот понятие *case-классов* будет для вас новым. *case-классы* в Scala позволяют применять сопоставление с образцом к объектам, добавляя к ним лишь ключевое слово *case*, не требуя при этом большого объема шаблонного кода.

Эту главу мы начнем с примера *case-классов* и сопоставления с образцом. Затем разберем все виды поддерживаемых шаблонов, рассмотрим роль *запечатанных классов* (sealed classes), обсудим перечисления, *Options* и покажем некоторые неочевидные места в языке, где используется сопоставление с образцом, а также более объемный и приближенный к реальному пример его использования.

13.1. Простой пример

Прежде чем вникать во все правила и нюансы сопоставления с образцом, есть смысл рассмотреть простой пример, дающий общее представление. Допустим, нужно написать библиотеку, которая работает с арифметическими выражениями и, возможно, является частью разрабатываемого предметно-ориентированного языка.

Первым шагом к решению этой задачи будет определение входных данных. Чтобы ничего не усложнять, сконцентрируемся на арифметических выра-

жениях, состоящих из переменных, чисел и унарных и бинарных операций. Все это выражается иерархией классов Scala, показанной в листинге 13.1.

Листинг 13.1. Определение case-классов

```
trait Expr
case class Var(name: String) extends Expr
case class Num(number: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
  left: Expr, right: Expr) extends Expr
```

Эта иерархия включает трейт `Expr` с четырьмя подклассами, по одному для каждого вида рассматриваемых выражений. Тела всех пяти классов пусты.

case-классы

Еще одна особенность объявлений в листинге 13.1 (см. выше), которая заслуживает внимания, — наличие у каждого подкласса модификатора `case`. Классы с таким модификатором называются *case-классами*. Как упоминалось в разделе 4.4, использование модификатора `case` заставляет компилятор Scala добавлять к вашему классу некоторые синтаксические удобства. Первое заключается в том, что к классу добавляется фабричный метод с именем данного класса. Это означает, к примеру, что для создания `var`-объекта можно применить код `Var("x")`:

```
val v = Var("x")
```

Особенно полезны фабричные методы благодаря их вложенности. Теперь код не загроможден ключевыми словами `new`, и структуру выражения можно воспринять с одного взгляда:

```
val op = BinOp("+", Num(1), v)
```

Второе синтаксическое удобство заключается в том, что все аргументы в списке параметров `case`-класса автоматически получают префикс `val`, то есть сохраняются в качестве полей:

```
v.name // x
op.left // Num(1.0)
```

Третье удобство состоит в том, что компилятор добавляет к вашему классу «естественную» реализацию методов `toString`, `hashCode` и `equals`. Они будут заниматься подготовкой данных к выводу, их хешированием и сравнением всего дерева, состоящего из класса, и (рекурсивно) всех его аргументов.

Поскольку метод `==` в Scala всегда передает полномочия методу `equals`, это означает, что элементы `case`-классов всегда сравниваются структурно:

```
op.toString          // BinOp(+,Num(1.0),Var(x))
op.right == Var("x") // true
```

И наконец, чтобы создать измененные копии, компилятор добавляет к вашему классу метод `copy`. Он пригодится для создания нового экземпляра класса, аналогичного другому экземпляру, за исключением того, что будет отличаться одним или двумя атрибутами. Метод работает за счет использования именованных параметров и параметров по умолчанию (см. раздел 8.8). Применение именованных параметров позволяет указать требуемые изменения. А для любого неуказанного параметра используется значение из старого объекта. Посмотрим в качестве примера, как можно создать операцию, похожую на `op` во всем, кроме того, что будет изменен параметр `operator`:

```
op.copy(operator = "-") // BinOp(-,Num(1.0),Var(x))
```

Все эти соглашения в качестве небольшого бонуса придают вашей работе массу удобств. Нужно просто указать модификатор `case`, и ваши классы и объекты приобретут гораздо более оснащенный вид. Они станут больше за счет создания дополнительных методов и неявного добавления поля для каждого параметра конструктора. Но самым большим преимуществом `case`-классов является то, что они поддерживают сопоставления с образцом¹.

Сопоставление с образцом

Предположим, нужно упростить арифметические выражения только что представленных видов. Существует множество возможных правил упрощения. В качестве иллюстрации подойдут три следующих правила:

```
UnOp("-", UnOp("-", e)) => e // двойное отрицание
BinOp("+", e, Num(0)) => e // прибавление нуля
BinOp("*", e, Num(1)) => e // умножение на единицу
```

Как показано в листинге 13.2, чтобы в Scala сформировать ядро функции упрощения с помощью сопоставления с образцом, эти правила можно взять практически в неизменном виде. Показанную в листинге функцию `simplifyTop` можно использовать следующим образом:

```
simplifyTop(UnOp("-", UnOp("-", Var("x")))) // Var(x)
```

¹ `case`-классы поддерживают сопоставление шаблонов путем создания метода извлечения `unapply` в объекте-компаньоне.

Листинг 13.2. Функция `simplifyTop`, выполняющая сопоставление с образцом

```
def simplifyTop(expr: Expr): Expr =
  expr match
    case UnOp("-", UnOp("-", e)) => e // двойное отрицание
    case BinOp("+", e, Num(0)) => e // прибавление нуля
    case BinOp("*", e, Num(1)) => e // умножение на единицу
    case _ => expr
```

Правая часть `simplifyTop` состоит из выражения `match`, которое соответствует `switch` в Java, но записывается после выражения выбора. Иными словами, оно выглядит как

```
выбор match { альтернативы }
```

вместо

```
switch (выбор) { альтернативы }
```

Сопоставление с образцом включает последовательность *альтернатив*, каждая из которых начинается с ключевого слова `case`. Каждая альтернатива состоит из *паттерна* и одного или нескольких выражений, которые будут вычислены при соответствии паттерну. Обозначение стрелки `=>` отделяет паттерн от выражений.

Выражение `match` вычисляется проверкой соответствия каждого из паттернов в порядке их написания. Выбирается первый же соответствующий паттерн, а также выбирается и выполняется та часть, которая следует за обозначением стрелки.

Паттерн-константа вида `+` или `1` соответствует значениям, равным константе в случае применения метода `==`. *Паттерн-переменная* вида `e` соответствует любому значению. Затем переменная ссылается на это же значение в правой части условия `case`. Обратите внимание: в данном примере первые три альтернативы вычисляются в `e`, то есть в переменную, связанную внутри соответствующего паттерна. *Подстановочный паттерн* (`_`) также соответствует любому значению, но без представления имени переменной для ссылки на это значение. Стоит отметить, что в листинге 13.2 выражение `match` заканчивается условием `case`, которое применяется при отсутствии соответствующих паттернов и не предполагает никаких действий с выражением. Вместо этого получается просто выражение `expr`, в отношении которого и выполняется сопоставление с образцом.

Паттерн-конструктор выглядит как `UnOp("-", e)`. Он соответствует всем значениям типа `UnOp`, первый аргумент которых соответствует `" - "`, а второй — `e`. Обратите внимание: аргументы конструктора сами являются

паттернами. Это позволяет составлять многоуровневые паттерны, используя краткую форму записи.

Примером может послужить следующий паттерн:

```
UnOp("-", UnOp("-", e))
```

Представьте попытку реализовать такую же функциональную возможность с помощью шаблона проектирования `visitor`¹. Практически так же трудно представить реализацию такой же функциональной возможности в виде длинной последовательности инструкций, проверок соответствия типам и явного приведения типов.

Сравнение `match` со `switch`

Выражения `match` могут быть представлены в качестве общих случаев `switch`-выражений в стиле Java. В Java `switch`-выражение может быть вполне естественно представлено в виде `match`-выражения, где каждый паттерн является константой, а последний паттерн может быть подстановочным (который представлен в `switch`-выражении вариантом, используемым при отсутствии других соответствий).

И тем не менее следует учитывать три различия. Во-первых, `match` является *выражением* языка Scala, то есть всегда вычисляется в значение. Во-вторых, применяемые в Scala выражения альтернатив никогда не «выпадают» в следующий вариант. В-третьих, если не найдено соответствие ни одному из паттернов, то выдается исключение `MatchError`. Следовательно, вам придется всегда обеспечивать охват всех возможных вариантов, даже если это будет означать вариант по умолчанию, в котором не делается ничего.

Листинг 13.3. Сопоставление с образцом с пустым вариантом по умолчанию

```
expr match
  case BinOp(op, left, right) =>
    println(s"$expr является бинарной операцией")
  case _ =>
```

Пример показан в листинге 13.3. Второй необходим, поскольку без него выражение `match` выдаст исключение `MatchError` для любого `expr`-аргумента,

¹ Гамма Э., Хелм Р., Джонсон Р., Влссидес Дж. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020.

не являющегося `BinOp`. В данном примере для этого второго варианта не указан никакой код, поэтому при его срабатывании ничего не произойдет. Результатом в любом случае будет `unit`-значение `()`, которое также будет результатом вычисления всего выражения `match`.

13.2. Разновидности паттернов

В предыдущем примере мы кратко описали некоторые разновидности паттернов. А теперь потратим немного времени на более подробное изучение каждого из них.

Синтаксис паттернов довольно прост, поэтому не стоит особо переживать из-за него. Все паттерны выглядят точно так же, как и соответствующие им выражения. Например, если взять иерархию из листинга 13.1, то паттерн `Var(x)` соответствует любому выражению, содержащему переменную, с привязкой `x` к имени переменной. Будучи использованным в качестве выражения, `Var(x)` с точно таким же синтаксисом воссоздает эквивалентный объект, предполагая, что идентификатор `x` уже привязан к имени переменной. В синтаксисе паттернов все прозрачно, поэтому главное, на что следует обратить внимание, — это какого вида паттерны можно применять.

Подстановочные паттерны

Подстановочный паттерн `(_)` соответствует абсолютно любому объекту. Вы уже видели, как он используется в качестве общего паттерна, выявляющего все оставшиеся альтернативы:

```
expr match
  case BinOp(op, left, right) =>
    s"$expr является бинарной операцией"
  case _ => // handle the default case
    s"Это что-то другое"
```

Кроме того, подстановочные паттерны могут использоваться для игнорирования тех частей объекта, которые не представляют для вас интереса. Например, в предыдущем примере нас не интересует, что представляют собой элементы бинарной операции, в нем лишь проверяется, является ли она бинарной. Поэтому, как показано в листинге 13.4, для элементов `BinOp` в коде также могут использоваться подстановочные паттерны.

Листинг 13.4. Сопоставление с образцом с использованием подстановочных паттернов

```
expr match
  case BinOp(_, _, _) => s"$expr является бинарной операцией"
  case _ => "Это что-то другое"
```

Паттерны-константы

Паттерн-константа соответствует только самому себе. В качестве константы может использоваться любой литерал. Например, паттернами-константами являются `5`, `true` и `"hello"`. В качестве константы может использоваться и любой `val`- или объект-одиночка. Так, объект-одиночка `Nil` является паттерном, соответствующим только пустому списку. Некоторые примеры паттернов-констант показаны в листинге 13.5. Вот как сопоставление с образцом выглядит в действии.

Листинг 13.5. Сопоставление с образцом с использованием паттернов-констант

```
def describe(x: Any) =
  x match
    case 5 => "пять"
    case true => "правда"
    case "hello" => "привет!"
    case Nil => "пустой список"
    case _ => "что-то другое"

describe(5)           // пять
describe(true)       // правда
describe("hello")    // привет!
describe(Nil)        // пустой список
describe(List(1,2,3)) // что-то другое
```

Паттерны-переменные

Паттерн-переменная соответствует любому объекту точно так же, как подстановочный паттерн, но в отличие от него Scala привязывает переменную к объекту. Затем с помощью этой переменной можно в дальнейшем воздействовать на объект. Например, в листинге 13.6 показано сопоставление с образцом, имеющее специальный вариант для нуля и общий вариант для всех остальных значений. В общем варианте используется паттерн-переменная, и поэтому у него есть имя для переменной независимо от того, что это на самом деле.

Листинг 13.6. Сопоставление с образцом с использованием паттерна-переменной

```
expr match
case 0 => "нуль"
case somethingElse => s"не нуль $somethingElse"
```

Переменная или константа?

У паттернов-констант могут быть символические имена. Вы уже это видели, когда в качестве образца использовался `Nil`. А вот похожий пример, где в сопоставлении с образцом задействуются константы `E` (2,718 28...) и `Pi` (3,141 59...):

```
scala> import math.{E, Pi}
import math.{E, Pi}

scala> E match
  case Pi => s"математический казус? Pi = $Pi"
  case _ => "OK"

val res0: String = OK
```

Как и ожидалось, значение `E` не равно значению `Pi`, поэтому вариант «математический казус» не выбирается.

А как компилятор Scala распознает, что `Pi` — это константа, импортированная из `scala.math`, а не переменная, обозначающая само значение селектора? Во избежание путаницы в Scala действует простое лексическое правило: обычное имя, начинающееся с буквы в нижнем регистре, считается переменной паттерна, а все другие ссылки считаются константами. Чтобы заметить разницу, создайте для `pi` псевдоним с первой буквой, указанной в нижнем регистре, и попробуйте в работе следующий код:

```
scala> val pi = math.Pi
pi: Double = 3.141592653589793

scala> E match
  case pi => s"математический казус? Pi = $pi"

val res1: String = математический казус? Pi = 2.718281828459045
```

Здесь компилятор даже не позволит вам добавить вариант по умолчанию. Поскольку `pi` — паттерн-переменная, то будет соответствовать всем вводимым данным, поэтому до следующих вариантов дело просто не дойдет:

```
scala> E match
  case pi => s"математический казус? Pi = $pi"
```

```

    case _ => "OK"

val res2: String = математический казус? Pi = 2.718281828459045
3 | _____ case _ => "OK"
  | _____ ^
  | _____ Unreachable case

```

Но при необходимости для паттерна-константы можно задействовать имя, начинающееся с буквы в нижнем регистре; для этого придется воспользоваться одним из двух приемов. Если константа является полем какого-нибудь объекта, то перед ней можно поставить префикс-классификатор. Например, `pi` — паттерн-переменная, а `this.pi` или `obj.pi` — константы, несмотря на то что их имена начинаются с букв в нижнем регистре. Если это не сработает (поскольку, скажем, `pi` — локальная переменная), то как вариант можно будет заключить имя переменной в обратные кавычки. Например, ``pi`` будет опять восприниматься как константа, а не как переменная:

```

scala> E match
  case `pi` => s"математический казус? Pi = $pi"
  case _ => "OK"

res4: String = OK

```

Как вы, наверное, заметили, использование для идентификаторов в Scala синтаксиса с обратными кавычками во избежание в коде необычных обстоятельств преследует две цели. Здесь показано, что этот синтаксис может применяться для рассмотрения идентификатора с именем, начинающимся с буквы в нижнем регистре, в качестве константы при сопоставлении с образцом. Ранее, в разделе 6.10, было показано, что этот синтаксис может использоваться также для трактовки ключевого слова в качестве обычного идентификатора. Например, в выражении `writingThread.`yield`()` слово `yield` трактуется как идентификатор, а не ключевое слово.

Паттерны-конструкторы

Реальная эффективность сопоставления с образцом проявляется именно в конструкторах. Паттерн-конструктор выглядит как `BinOp("+", e, Num(0))`. Он состоит из имени (`BinOp`), после которого в круглых скобках стоят несколько образцов: `"+"`, `e` и `Num(0)`. При условии, что имя обозначает `case`-класс, такой паттерн показывает следующее: сначала проверяется принадлежность элемента к названному `case`-классу, а затем соответствие

параметров конструктора объекта предоставленным дополнительным паттернам.

Эти дополнительные паттерны означают, что в паттернах Scala поддерживаются *глубокие сопоставления* (deep matches). Такой паттерн проверяет не только предоставленный объект верхнего уровня, но и его содержимое на соответствие следующим паттернам. Дополнительные паттерны сами по себе могут быть паттернами-конструкторами, поэтому их можно использовать для проверки объекта произвольной глубины. Например, паттерн, показанный в листинге 13.7, проверяет, что объект верхнего уровня относится к типу `BinOp`, третьим параметром его конструктора является число `Num` и значение поля этого числа — `0`. Весь паттерн умещается в одну строку кода, хотя выполняет проверку на глубину в три уровня.

Листинг 13.7. Сопоставление с образцом с использованием паттерна-конструктора

```
expr match
  case BinOp("+", e, Num(0)) => "глубокое соответствие"
  case _ => ""
```

Паттерны-последовательности

`case`-классы можно сопоставлять с такими типами последовательностей, как `List` или `Array`. Однако теперь в паттерне вы можете указать любое количество элементов, пользуясь тем же синтаксисом. В листинге 13.8 показан шаблон для проверки трехэлементного списка, начинающегося с нуля.

Листинг 13.8. Паттерн-последовательность фиксированной длины

```
xs match
  case List(0, _, _) => "соответствие найдено"
  case _ => ""
```

Если нужно сопоставить с последовательностью, не указывая ее длину, то в качестве последнего элемента паттерна-последовательности можно указать образец `_*`. Он имеет весьма забавный вид и соответствует любому количеству элементов внутри последовательности, включая ноль элементов. В листинге 13.9 показан пример, соответствующий любому списку, который начинается с нуля, независимо от длины этого списка.

Листинг 13.9. Паттерн-последовательность произвольной длины

```
xs match
  case List(0, _, _) => " соответствие найдено "
  case _ => ""
```

Паттерны-кортежи

Можно выполнять и сопоставление с кортежами. Паттерн вида `(a, b, c)` соответствует произвольному трехэлементному кортежу. Пример показан в листинге 13.10.

Если загрузить показанный в листинге 13.10 метод `tupleDemo` в интерпретатор и передать ему кортеж из трех элементов, то получится следующая картина.

Листинг 13.10. Сопоставление с образцом с использованием паттерна-кортежа

```
def tupleDemo(obj: Any) =
  obj match
    case (a, b, c) => s"matched $a$b$c"
    case _ => ""

tupleDemo(("a ", 3, "-tuple")) // соответствует а 3-tuple
```

Типизированные паттерны

Типизированный паттерн (typed pattern) можно использовать в качестве удобного заменителя для проверок типов и приведения типов. Пример показан в листинге 13.11.

Листинг 13.11. Сопоставление с образцом с использованием типизированных паттернов

```
def generalSize(x: Any) =
  x match
    case s: String => s.length
    case m: Map[_ , _] => m.size
    case _ => -1
```

А вот несколько примеров использования `generalSize` в интерпретаторе Scala:

```
generalSize("abc") // 3
generalSize(Map(1 -> 'a', 2 -> 'b')) // 2
generalSize(math.Pi) // -1
```

Метод `generalSize` возвращает размер или длину объектов различных типов. Типом его аргумента является `Any`, поэтому им может быть любое значение. Если в качестве типа аргумента выступает `String`, то метод возвращает длину строки. Образец `s: String` является типизированным паттерном и соответ-

ствует каждому (ненулевому) экземпляру класса `String`. Затем на эту строку ссылается паттерн-переменная `s`.

Заметьте: даже притом что `s` и `x` ссылаются на одно и то же значение, типом `x` является `Any`, а типом `s` является `String`. Поэтому в альтернативном выражении, соответствующем паттерну, можно воспользоваться кодом `s.length`, но нельзя — кодом `x.length`, поскольку в типе `Any` отсутствует член `length`.

Эквивалентный, но более многословный способ достичь такого же результата сопоставления с типизированным образцом — использовать проверку типа с его последующим приведением. В Scala для этого применяется не такой синтаксис, как в Java. К примеру, чтобы проверить, относится ли выражение `expr` к типу `String`, используется такой код:

```
expr.isInstanceOf[String]
```

Для приведения того же выражения к типу `String` используется код

```
expr.asInstanceOf[String]
```

Применяя проверку и приведение типа, можно переписать первый вариант предыдущего `match`-выражения, получив код, показанный в листинге 13.12.

Листинг 13.12. Использование `isInstanceOf` и `asInstanceOf` (плохой стиль)

```
if x.isInstanceOf[String] then
  val s = x.asInstanceOf[String]
  s.length
else ...
```

Операторы `isInstanceOf` и `asInstanceOf` считаются предопределенными методами класса `Any`, получающими параметр типа в квадратных скобках. Фактически `x.asInstanceOf[String]` — частный случай вызова метода с явно заданным параметром типа `String`.

Как вы уже заметили, написание проверок и приведений типов в Scala страдает излишним многословием. Сделано это намеренно, поскольку подобная практика не приветствуется. Как правило, лучше воспользоваться сопоставлением с типизированным образцом. В частности, подобный подход будет оправдан, если нужно выполнить две операции: проверку типа и его приведение, так как обе они будут сведены к единственному сопоставлению.

Второй вариант `match`-выражения в листинге 13.11 содержит типизированный паттерн `m: Map[_ , _]`. Он соответствует любому значению, являющемуся отображением каких-либо произвольных типов ключа

и значения, и позволяет `m` ссылаться на это значение. Поэтому `m.size` имеет правильный тип и возвращает размер отображения. Знаки подчеркивания в типизированном паттерне¹ подобны таким же знакам в подстановочных паттернах. Вместо них можно указывать переменные типа с символами в нижнем регистре.

Приписывание типов

Приведения по своей сути небезопасны. Например, даже если у компилятора достаточно информации, чтобы определить, что приведение из `Int` в `String` не сработает во время выполнения, оно все равно компилируется (и завершается сбоем во время выполнения):

```
3.asInstanceOf[String]
// java.lang.ClassCastException: java.lang.Integer
//     не может быть приведен к java.lang.String
```

Безопасной альтернативой приведения является приписывание типов: размещение двоеточия и типа после переменной или выражения. Приписывание типов безопасно, потому что любое неправильное приписывание, например приписывание `Int` к типу `String`, приведет к ошибке компилятора, а не к исключению во время выполнения:

```
scala> 3: String // ': String' – приписывание типов
1 |3: String
  | ^
  | Found:    (3 : Int)
  | Required: String
```

Приписывание типа будет компилироваться только в двух случаях. Во-первых, вы можете использовать его для расширения типа до одного из его супертипов. Например:

```
scala> Var("x"): Expr // Expr – супертип Var
val res0: Expr = Var(x)
```

Во-вторых, вы можете использовать его для неявного преобразования одного типа в другой, например для неявного преобразования `Int` в `Long`:

```
scala> 3: Long
val res1: Long = 3
```

¹ В типизированном паттерне `m: Map[_, _]`, часть "`Map[_, _]`" называется паттерном типа.

Затирание типов

А можно ли также проверять на отображение с конкретными типами элементов? Это пригодилось бы, скажем, для проверки того, является ли заданное значение отображением типа `Int` на тип `Int`. Попробуем:

```
scala> def isIntIntMap(x: Any) =
  x match
  case m: Map[Int, Int] => true
  case _ => false

def isIntIntMap(x: Any): Boolean
3 | case m: Map[Int, Int] => true
  | ~~~~~
  |
  | the type test for Map[Int, Int] cannot be
  | checked at runtime
```

В Scala точно так же, как и в Java, используется модель *затирания* обобщенных типов. Это значит, в ходе выполнения программы никакая информация об аргументах типов не сохраняется. Следовательно, способов определить в ходе выполнения программы, создавался ли заданный `Map`-объект с двумя `Int`-аргументами, а не с аргументами других типов, не существует. Система может лишь определить, что значение является отображением (`Map`) неких произвольных параметров типа. Убедиться в таком поведении можно, применив `isIntIntMap` к различным экземплярам класса `Map`:

```
isIntIntMap(Map(1 -> 1)) // true
isIntIntMap(Map("abc" -> "abc")) // true
```

Первое применение возвращает `true`, что выглядит вполне корректно, но второе тоже возвращает `true`, и это может оказаться сюрпризом. Чтобы оповестить вас о возможном непонятном поведении программы в ходе ее выполнения, компилятор выдает предупреждение о том, что не контролирует это поведение, похожее на показанные ранее.

Единственное исключение из правила затирания — массивы, поскольку в Java, а также в Scala они обрабатываются особым образом. Тип элемента массива сохраняется вместе со значением массива, поэтому к нему можно применить сопоставление с образцом. Пример выглядит так:

```
def isStringArray(x: Any) =
  x match
  case a: Array[String] => "yes"
  case _ => "no"

isStringArray(Array("abc")) // да
isStringArray(Array(1, 2, 3)) // нет
```

Привязка переменной

Кроме использования отдельно взятого паттерна-переменной, можно также добавить переменную к любому другому паттерну. Нужно указать имя переменной, знак «собачки» (@), а затем паттерн. Это даст вам паттерн с привязанной переменной, то есть паттерн для выполнения обычного сопоставления с образцом с возможностью в случае совпадения присвоить переменной соответствующий объект, как и при использовании обычного паттерна-переменной.

В качестве примера в листинге 13.13 показано сопоставление с образцом — поиск операции получения абсолютного значения, применяемой в строке дважды. Такое выражение можно упростить, однократно получив абсолютное значение.

Листинг 13.13. Паттерн с привязкой переменной (посредством использования знака @)

```
expr match
  case UnOp("abs", e @ UnOp("abs", _)) => e
  case _ =>
```

Пример, показанный в данном листинге, включает паттерн с привязкой переменной, где в качестве переменной выступает `e`, а в качестве паттерна — `UnOp("abs", _)`. Если будет найдено соответствие всему паттерну, то часть, которая соответствует `UnOp("abs", _)`, станет доступна как значение переменной `e`. Результатом варианта будет просто `e`, поскольку `e` имеет значение, равное `expr`, но с меньшим на единицу количеством операций получения абсолютного значения.

13.3. Ограждение образца

Иногда синтаксическое сопоставление с образцом является недостаточно точным. Предположим, перед вами стоит задача сформулировать правило упрощения, заменяющее выражение сложения с двумя одинаковыми операндами, такое как `e + e`, умножением на два, например `e * 2`. На языке деревьев `Expr` выражение вида

```
BinOp("+", Var("x"), Var("x"))
```

этим правилом будет превращено в

```
BinOp("*", Var("x"), Num(2))
```

Правило можно попробовать выразить следующим образом:

```
scala> def simplifyAdd(e: Expr) =
  e match
    case BinOp("+", x, x) => BinOp("*", x, Num(2))
    case _ => e
```

```
3 | case BinOp("+", x, x) => BinOp("*", x, Num(2))
  | ^
  | duplicate pattern variable: x
```

Попытка будет неудачной, поскольку в Scala паттерны должны быть *линейными*: паттерн-переменная может появляться в образце только один раз. Но, как показано в листинге 13.14, соответствие можно переформулировать с помощью *ограничителя паттернов* (pattern guard).

Листинг 13.14. Сопоставление с образцом с применением ограждения паттернов

```
def simplifyAdd(e: Expr) =
  e match
    case BinOp("+", x, y) if x == y =>
      BinOp("*", x, Num(2))
    case _ => e
```

Ограждение паттерна указывается после образца и начинается с ключевого слова `if`. В качестве ограждения может использоваться произвольное булево выражение, которое обычно ссылается на переменные в образце. При наличии ограждения паттернов соответствие считается найденным, только если ограждение вычисляется в `true`. Таким образом, первый вариант показанного ранее кода соответствует только бинарным операциям, имеющим два одинаковых операнда.

А вот как выглядят некоторые другие огражденные паттерны:

```
// соответствует только положительным целым числам
case n: Int if 0 < n => ...
```

```
// соответствует только строкам, начинающимся с буквы 'a'
case s: String if s(0) == 'a' => ...
```

13.4. Наложение паттернов

Паттерны применяются в порядке их указания. Версия метода `simplify`, показанная в листинге 13.15, представляет собой пример, в котором порядок следования вариантов имеет значение.

Листинг 13.15. Выражение сопоставления, в котором порядок следования вариантов имеет значение

```
def simplifyAll(expr: Expr): Expr =
  expr match
  case UnOp("-", UnOp("-", e)) =>
    simplifyAll(e) // '-' является своей собственной обратной величиной
  case BinOp("+", e, Num(0)) =>
    simplifyAll(e) // '0' нейтральный элемент для '+'
  case BinOp("*", e, Num(1)) =>
    simplifyAll(e) // '1' нейтральный элемент для '*'
  case UnOp(op, e) =>
    UnOp(op, simplifyAll(e))
  case BinOp(op, l, r) =>
    BinOp(op, simplifyAll(l), simplifyAll(r))
  case _ => expr
```

Версия метода `simplify`, показанная в данном листинге, станет применять правила упрощения в любом месте выражения, а не только в его верхней части, как это сделала бы версия `simplifyTop`. Данную версию можно вывести из версии `simplifyTop`, добавив два дополнительных варианта для обычных унарных и бинарных выражений (четвертый и пятый варианты `case` в листинге 13.15).

В четвертом варианте используется паттерн `UnOp(op, e)`, который соответствует любой унарной операции. Оператор и операнд унарной операции могут быть какими угодно. Они привязаны к паттернам-переменным `op` и `e` соответственно. Альтернативой в данном варианте будет рекурсивное применение `simplifyAll` к операнду `e` с последующим перестроением той же самой унарной операции с (возможно) упрощенным операндом. Пятый вариант для `BinOp` аналогичен четвертому: он является вариантом «поймать все» для произвольных бинарных операций, который рекурсивно применяет метод упрощения к своим двум операндам.

Важным обстоятельством в этом примере является то, что варианты «поймать все» следуют *после* более конкретизированных правил упрощения. Если расположить их в другом порядке, то вариант «поймать все» будет запущен вместо более конкретизированных правил. Во многих случаях компилятор будет жаловаться на такие попытки. Например, вот как выглядит выражение `match`, которое не пройдет компиляцию, поскольку первый вариант будет соответствовать всему тому, чему будет соответствовать второй вариант:

```
scala> def simplifyBad(expr: Expr): Expr =
  expr match
  case UnOp(op, e) => UnOp(op, simplifyBad(e))
  case UnOp("-", UnOp("-", e)) => e
  case _ => expr
```

```
def simplifyBad(expr: Expr): Expr
4 | case UnOp("-", UnOp("-", e)) => e
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  | Unreachable case
```

13.5. Запечатанные классы

При написании сопоставления с образцом нужно удостовериться в том, что охвачены все возможные варианты. Иногда это можно сделать, добавив в конец `match` вариант по умолчанию, но данный способ применим, только когда есть вполне определенное поведение по умолчанию. А что делать, если его нет? Как узнать, что охвачены все варианты и нет опасности упустить что-либо?

Чтобы определить пропущенные в выражении `match` комбинации паттернов, можно обратиться за помощью к компилятору Scala. Для этого компилятор должен иметь возможность сообщить обо всех потенциальных вариантах. По сути, в Scala это сделать нереально, поскольку классы могут быть определены в любое время и в произвольных блоках компиляции. Например, ничто не мешает вам добавить к иерархии класса `Expr` пятый `case`-класс не в том блоке компиляции, в котором определены четыре других `case`-класса, а в другом.

Альтернативой этому может стать превращение суперкласса ваших `case`-классов в *запечатанный* класс. У такого запечатанного класса не может быть никаких дополнительных подклассов, кроме тех, которые определены в том же самом файле. Особую пользу из этого можно извлечь при сопоставлении с образцом, поскольку запечатанность класса будет означать, что беспокоиться придется только по поводу тех подклассов, о которых вам уже известно. Более того, будет улучшена поддержка со стороны компилятора. При сопоставлении с образцом `case`-классам, являющимся наследниками запечатанного класса, компилятор в предупреждении отметит пропущенные комбинации паттернов.

Если создается иерархия классов, предназначенная для сопоставления с образцом, то нужно предусмотреть ее запечатанность. Чтобы это сделать, просто поставьте перед классом на вершине иерархии ключевое слово `sealed`. Программисты, использующие вашу иерархию классов, при сопоставлении с образцом будут чувствовать себя уверенно. Таким образом, ключевое слово `sealed` зачастую выступает лицензией на сопоставление с образцом. Пример, в котором `Expr` превращается в запечатанный класс, показан в листинге 13.16.

Листинг 13.16. Запечатанная иерархия case-классов

```
sealed trait Expr
case class Var(name: String) extends Expr
case class Num(number: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
  left: Expr, right: Expr) extends Expr
```

А теперь определим сопоставление с образцом, в котором пропущены некоторые возможные варианты:

```
def describe(e: Expr): String =
  e match
    case Num(_) => "число"
    case Var(_) => "переменная"
```

В результате будет получена следующая ошибка компилятора:

```
def describe(e: Expr): String
2 | e match
  | ^
  | match may not be exhaustive.
  |
  | It would fail on pattern case: UnOp(_, _),
  | BinOp(_, _, _)
```

Такая ошибка компилятора сообщает о существовании риска генерации вашим кодом исключения `MatchError`, поскольку некоторые возможные паттерны (`UnOp`, `BinOp`) не обрабатываются. Ошибка указывает на потенциальный источник сбоя в ходе выполнения программы и помогает при корректировке кода.

Но порой можно столкнуться с ситуацией, в которой компилятор при выдаче ошибки проявляет излишнюю дотошность. Например, из контекста может быть известно, что показанный ранее метод `describe` будет применяться только к выражениям типа `Num` или `Var`, следовательно, исключение `MatchError` не станет генерироваться. Чтобы избавиться от ошибки, к методу можно добавить третий вариант по умолчанию:

```
def describe(e: Expr): String =
  e match
    case Num(_) => "число"
    case Var(_) => "переменная"
    case _ => throw new RuntimeException // Не должно произойти
```

Решение вполне работоспособное, однако не идеальное. Вряд ли вас обрадует принуждение добавить код, который никогда не будет выполнен (по вашему мнению), лишь для того, чтобы успокоить компилятор.

Более экономной альтернативой станет добавление к селектору выражения сопоставления с образцом аннотации `@unchecked`. Делается это следующим образом:

```
def describe(e: Expr): String =
  (e: @unchecked) match
    case Num(_) => "число"
    case Var(_) => "переменная"
```

В общем, аннотации можно добавлять к выражению точно так же, как это делается при добавлении типа: нужно после выражения поставить двоеточие, знак «собачка» и указать название аннотации. Например, в данном случае к переменной `e` добавляется аннотация `@unchecked`, для чего используется код `e: @unchecked`. Аннотация `@unchecked` имеет особое значение для сопоставления с образцом. Если выражение селектора поиска содержит данную аннотацию, то исчерпывающая проверка последующих паттернов будет подавлена.

13.6. Сопоставление паттерна Options

Вы можете использовать сопоставление шаблонов для обработки стандартного типа `Option` в Scala. Как упоминалось в шаге 12 главы 3, `Option` может быть двух видов: это либо `Some(x)`, где `x` — реальное значение, либо `None`, у которого отсутствует значение.

Необязательные значения производятся некоторыми стандартными операциями над коллекциями Scala. Например, метод `get` из Scala-класса `Map` производит `Some(значение)`, если найдено *значение*, соответствующее заданному ключу, или `None`, если заданный ключ не определен в `Map`-объекте. Пример выглядит так:

```
val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")
capitals.get("France") // Some(Paris)
capitals.get("North Pole") // None
```

Самый распространенный способ разобрать необязательные значения — использовать сопоставление с образцом, например:

```
def show(x: Option[String]) =
  x match
    case Some(s) => s
    case None => "?"

show(capitals.get("Japan")) // Tokyo
show(capitals.get("France")) // Paris
show(capitals.get("North Pole")) // ?
```

Тип `Option` применяется в программах на языке Scala довольно часто. Его использование можно сравнить с доминирующей в Java идиомой `null`, показывающей отсутствие значения. Например, метод `get` из `java.util.HashMap` возвращает либо значение, сохраненное в `HashMap`, либо `null`, если значение не было найдено. В Java такой подход работает, но, применяя его, легко допустить ошибку, поскольку на практике довольно трудно отследить, каким переменным в программе разрешено иметь значение `null`.

В случае, когда переменной разрешено иметь значение `null`, вы должны вспомнить о ее проверке на наличие этого значения при каждом использовании. Если забыть выполнить эту проверку, то появится вероятность генерации в ходе выполнения программы исключений `NullPointerException`. Подобные исключения могут генерироваться довольно редко, поэтому с выявлением ошибки при тестировании могут возникнуть затруднения. В Scala такой подход вообще не работает, поскольку этот язык позволяет сохранять типы значений в хеш-отображениях, а `null` не является допустимым элементом для типов значений. Например, `HashMap[Int, Int]` не может вернуть `null`, чтобы обозначить отсутствие элемента.

Вместо этого в Scala для указания необязательного значения применяется тип `Option`. Такой способ имеет ряд преимуществ по сравнению с используемым в подходе `null`. Во-первых, тем, кто читает код, намного понятнее, что переменная, типом которой является `Option[String]`, — необязательная переменная `String`, а не переменная типа `String`, которая иногда может иметь значение `null`. Во-вторых, что более важно, рассмотренные ранее ошибки программирования, связанные с использованием переменной со значением `null` без предварительной проверки ее на `null`, превращаются в Scala в ошибку типа. Если переменная имеет тип `Option[String]`, то при попытке ее использования в качестве строки ваша программа на Scala не пройдет компиляцию.

13.7. Паттерны повсюду

Паттерны можно использовать не только в отдельно взятых `match`-выражениях, но и во многих других местах программы на языке Scala. Рассмотрим несколько подобных мест применения паттернов.

Паттерны в определениях переменных

При определении `val`- или `var`-переменной вместо простых идентификаторов можно использовать паттерны. Например, можно, как показано в ли-

стинге 13.17, разобрать кортеж и присвоить каждую его часть собственной переменной.

Листинг 13.17. Определение нескольких переменных с помощью одного присваивания

```
scala> val myTuple = (123, "abc")
val myTuple: (Int, String) = (123,abc)
```

```
scala> val (number, string) = myTuple
val number: Int = 123
val string: String = abc
```

Особенно полезной эта конструкция может быть при работе с `case`-классами. Если точно известен `case`-класс, с которым ведется работа, то вы можете разобрать его с помощью паттерна. Пример выглядит следующим образом:

```
scala> val exp = new BinOp("*", Num(5), Num(1))
val exp: BinOp = BinOp(*,Num(5.0),Num(1.0))
```

```
scala> val BinOp(op, left, right) = exp
val op: String = *
val left: Expr = Num(5.0)
val right: Expr = Num(1.0)
```

Последовательности вариантов в качестве частично примененных функций

Последовательность вариантов (то есть альтернатив), заключенную в фигурные скобки, можно задействовать везде, где может использоваться функциональный литерал. По сути, последовательность вариантов и есть функциональный литерал, только более универсальный. Вместо единственной точки входа и списка параметров последовательность вариантов имеет несколько точек входа, каждой из которых присущ собственный список параметров. Каждый вариант является точкой входа в функцию, а параметры указываются с помощью паттерна. Тело каждой точки входа — правосторонняя часть варианта.

Простой пример выглядит следующим образом:

```
val withDefault: Option[Int] => Int =
  case Some(x) => x
  case None => 0
```

В теле этой функции имеется два варианта. Первый соответствует `Some` и возвращает число, находящееся внутри `Some`. Второй соответствует

`None` и возвращает стандартное значение `0`. А вот как используется данная функция:

```
withDefault(Some(10)) // 10
withDefault(None)    // 0
```

Такая возможность особенно полезна для библиотеки акторов `Akka`, поскольку позволяет определить ее метод `receive` в виде серии вариантов:

```
var sum = 0
def receive =
  case Data(byte) =>
    sum += byte
  case GetChecksum(requester) =>
    val checksum = ~(sum & 0xFF) + 1
    requester ! checksum
```

Кроме того, стоит упомянуть еще одно общее правило: последовательность вариантов дает вам *частично* примененную функцию. Если применить такую функцию в отношении не поддерживаемого ею значения, то она сгенерирует исключение времени выполнения. Например, ниже показана частично примененная функция, которая возвращает второй элемент списка, состоящего из целых чисел:

```
val second: List[Int] => Int =
  case x :: y :: _ => y
```

При компиляции этого кода компилятор вполне резонно выведет предупреждение о том, что сопоставление с образцом не охватывает все возможные варианты:

```
2 | case x :: y :: _ => y
  | ^
  | match may not be exhaustive.
  |
  | It would fail on pattern case: List(_, Nil)
```

Функция справится со своей задачей, если ей передать список, состоящий из трех элементов, но не станет работать при передаче пустого списка:

```
scala> second(List(5, 6, 7))
val res24: Int = 6

scala> second(List())
scala.MatchError: List() (of class Nil$)
  at rs$line$10$.<init>$$anonfun$1(rs$line$10:2)
  at rs$line$12$.<init>(rs$line$12:1)
```

Если нужно проверить, определена ли частично примененная функция, то сначала следует сообщить компилятору: вы знаете, что работаете с частично примененными функциями. Тип `List[Int] => Int` включает все функции, получающие из целочисленных списков целочисленные значения независимо от того, частично они применяются или нет. Тип, который включает только *частично* примененные функции, которые получают из целочисленных списков целочисленные значения, записывается в виде `PartialFunction[List[Int], Int]`. Ниже представлен еще один вариант функции `second`, определенной с типом частично примененной функции:

```
val second: PartialFunction[List[Int], Int] =
  case x :: y :: _ => y
```

У частично примененных функций есть метод `isDefinedAt`, который может использоваться для тестирования того, определена ли функция в отношении конкретного значения. В данном случае функция определена для любого списка, состоящего по крайней мере из двух элементов:

```
second.isDefinedAt(List(5,6,7)) // true
second.isDefinedAt(List())      // false
```

Типичным образчиком частично примененной функции может послужить функциональный литерал сопоставления с образцом, подобный представленному в предыдущем примере. Фактически такое выражение преобразуется компилятором Scala в частично примененную функцию с помощью двойного преобразования паттернов: один раз для реализации реальной функции, а второй — для проверки того, определена ли функция.

Например, функциональный литерал `{ case x :: y :: _ => y }` преобразуется в следующее значение частично примененной функции:

```
new PartialFunction[List[Int], Int]:

  def apply(xs: List[Int]) =
    xs match
      case x :: y :: _ => y

  def isDefinedAt(xs: List[Int]) =
    xs match
      case x :: y :: _ => true
      case _ => false
```

Это преобразование осуществляется в том случае, когда в качестве объявляемого типа функционального литерала выступает `PartialFunction`. Если объявляемый тип — просто `Function1` или не указан, функциональный литерал вместо этого преобразуется в *полноценную функцию*.

Вообще-то, полноценными функциями нужно пробовать пользоваться везде, где только можно, поскольку использование частично примененных функций допускает возникновение ошибок времени выполнения, устранить которые компилятор вам не может помочь. Но иногда частично примененные функции приносят реальную пользу. Вам следует позаботиться о том, чтобы этим функциям не было предоставлено необрабатываемое значение. Как вариант, вы можете задействовать фреймворк, который допускает использование частично примененных функций и поэтому всегда перед вызовом функции выполняет проверку функцией `isDefinedAt`. Последнее проиллюстрировано приведенным ранее примером метода `receive`, где результатом выступает частично примененная функция с определением, данным в точности для тех сообщений, которые нужно обработать вызывающему коду.

Паттерны в выражениях `for`

Паттерны, как показано ниже, в листинге 13.18, можно использовать также в выражениях `for`. Это выражение извлекает все пары «ключ — значение» из отображения `capitals` (столицы). Каждая пара соответствует паттерну (`country, city`) (страна, город), который определяет две переменные: `country` и `city`.

Листинг 13.18. Выражение `for` с паттерном-кортежем

```
for (country, city) <- capitals yield
  s"Столицей $country является $city"
//
// List(Столицей France является Paris,
// Столицей Japan является Tokyo)
```

Паттерн пар, показанный в данном листинге, интересен, поскольку сопоставление с ним никогда не даст сбой. Конечно, `capitals` выдает последовательность пар, следовательно, можно быть уверенными, что каждая сгенерированная пара может соответствовать паттерну пар.

Но с равной долей вероятности возможно, что паттерн не будет соответствовать сгенерированному значению. Именно такой случай показан в листинге 13.19.

Листинг 13.19. Отбор элементов списка, соответствующих паттерну

```
val results = List(Some("apple"), None, Some("orange"))
for Some(fruit) <- results yield fruit
// List(apple, orange)
```

В этом примере показано, что сгенерированные значения, не соответствующие паттерну, отбрасываются. Так, второй элемент `None` в получившемся списке не соответствует паттерну `Some(fruit)`, поэтому отсутствует в результате.

13.8. Большой пример

После изучения различных форм паттернов может быть интересно посмотреть на их применение в более существенном примере. Предлагаемая задача заключается в написании класса, форматирующего выражения, который выводит арифметическое выражение в двумерной разметке. Такое выражение деления, как $x / (x + 1)$, должно быть выведено вертикально — с числителем, показанным над знаменателем:

$$\begin{array}{r} x \\ \text{-----} \\ x + 1 \end{array}$$

В качестве еще одного примера ниже в двумерной разметке показано выражение $((a / (b * c) + 1 / n) / 3)$:

$$\begin{array}{r} a \quad 1 \\ \text{-----} + - \\ b * c \quad n \\ \text{-----} \\ 3 \end{array}$$

Исходя из этих примеров, можно прийти к выводу, что манипулированием разметкой должен заняться класс — назовем его `ExprFormatter`, поэтому имеет смысл задействовать библиотеку разметки, разработанную в главе 10. Мы также используем семейство `case`-классов `Expr`, ранее уже встречавшееся в данной главе, и поместим в именованные пакеты как библиотеку разметки из главы 10, так и средство форматирования выражений. Полный код этого примера будет показан ниже, в листингах 13.20 и 13.21.

Сначала полезно будет сосредоточиться на горизонтальной разметке. Структурированное выражение

```
BinOp("+",
  BinOp("*",
    BinOp("+", Var("x"), Var("y")),
    Var("z")),
  Num(1))
```

должно привести к выводу $(x + y) * z + 1$. Обратите внимание на обязательность круглых скобок вокруг выражения $x + y$ и их необязательность вокруг выражения $(x + y) * z$. Чтобы разметка получилась максимально разборчивой, следует стремиться к отказу от избыточных круглых скобок и обеспечить наличие всех обязательных.

Чтобы узнать, куда ставить круглые скобки, код должен быть в курсе относительной приоритетности каждого оператора; как следствие, неплохо было бы сначала отрегулировать именно этот вопрос. Относительную приоритетность можно выразить непосредственно в виде литерала отображения следующей формы:

```
Map(
  "|" -> 0, "||" -> 0,
  "&" -> 1, "&&" -> 1, ...
)
```

Конечно, вам потребуется предварительно выполнить определенный объем вычислительной работы для назначения приоритетов. Удобнее будет просто определить группы операторов с нарастающим уровнем приоритета, а затем, исходя из этого, вычислить приоритет каждого оператора. Соответствующий код показан в листинге 13.20.

Листинг 13.20. Верхняя половина средства форматирования выражений

```
package org.stairwaybook.expr
import org.stairwaybook.layout.Element.elem

sealed abstract class Expr
case class Var(name: String) extends Expr
case class Num(number: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
  left: Expr, right: Expr) extends Expr

class ExprFormatter:

  // Содержит операторы в группах с нарастающей степенью приоритетности
  private val opGroups =
    Vector(
      Set("|", "||"),
      Set("&", "&&"),
      Set("^"),
      Set("==", "!="),
      Set("<", "<=", ">", ">="),
      Set("+", "-"),
      Set("*", "%")
    )
  )
```

```
// Отображение операторов на их степень приоритетности
private val precedence = {
  val assocs =
    for
      i <- 0 until opGroups.length
      op <- opGroups(i)
    yield op -> i
  assocs.toMap
}

private val unaryPrecedence = opGroups.length
private val fractionPrecedence = -1

// продолжение в листинге 13.21...
```

Переменная `precedence` — отображение операторов на уровень их приоритета, представленный целыми числами, начинающимися с нуля. Приоритет вычисляется с использованием выражения `for` с двумя генераторами. Первый выдает каждый индекс `i` вектора `opGroups`, второй — каждый оператор `op`, находящийся в `opGroups(i)`. Для каждого такого оператора выражение `for` выдает привязку оператора `op` к его индексу `i`. В результате приоритетность оператора берется из его относительной позиции в векторе.

Листинг 13.21. Нижняя половина средства форматирования выражений

```
// ...продолжение, начало в листинге 13.20
import org.stairwaybook.layout.Element

private def format(e: Expr, enclPrec: Int): Element =

  e match

    case Var(name) =>
      elem(name)

    case Num(number) =>
      def stripDot(s: String) =
        if s endsWith ".0" then s.substring(0, s.length - 2)
        else s
      elem(stripDot(number.toString))

    case UnOp(op, arg) =>
      elem(op) beside format(arg, unaryPrecedence)

    case BinOp("/", left, right) =>
      val top = format(left, fractionPrecedence)
      val bot = format(right, fractionPrecedence)
      val line = elem('-', top.width.max(bot.width), 1)
      val frac = top above line above bot
      if enclPrec != fractionPrecedence then frac
```

```

    else elem(" ") beside frac beside elem(" ")

case BinOp(op, left, right) =>
  val opPrec = precedence(op)
  val l = format(left, opPrec)
  val r = format(right, opPrec + 1)
  val oper = l beside elem(" " + op + " ") beside r
  if enclPrec <= opPrec then oper
  else elem("(") beside oper beside elem(")")

end match

def format(e: Expr): Element = format(e, 0)

end ExprFormatter

```

Привязки записываются с помощью инфиксной стрелки, например `op -> i`. До сих пор они были показаны только как часть конструкций отображений, но и сами по себе они имеют значение. Фактически привязка `op -> i` есть не что иное, как пара `(op, i)`.

Теперь, зафиксировав уровень приоритета всех бинарных операторов, за исключением `/`, имеет смысл обобщить данную концепцию, охватив также унарные операторы. Уровень приоритета унарного оператора выше, чем у любого бинарного оператора. Поэтому для переменной `unaryPrecedence`, показанной в листинге 13.20, устанавливается значение длины вектора `opGroups` на единицу большее, чем уровень приоритета операторов `*` и `%`. Приоритет оператора деления рассматривается не так, как у других операторов, поскольку для дробей используется вертикальная разметка. Но, конечно же, было бы удобно присвоить оператору деления специальное значение уровня приоритета `-1`, поэтому переменная `fractionPrecedence` будет инициализирована значением `-1`, как было показано в листинге 13.20.

После такой подготовительной работы можно приступить к написанию основного метода `format`. Этот метод получает два аргумента: выражение `e`, имеющее тип `Expr`, и уровень приоритета `enclPrec` того оператора, который непосредственно заключен в данное выражение. (Если в выражении нет никакого оператора, то значение `enclPrec` должно быть нулевым.) Метод выдаст элемент разметки, представленный в виде двумерного массива символов.

В листинге 13.21 показана остальная часть класса `ExprFormatter`, включающая два метода. Первый — приватный метод `format` — выполняет основную работу по форматированию выражений. Второй, который также называется `format`, представляет собой единственный публичный метод в библиотеке, получающий выражение для форматирования. Приватный метод `format`

проделяет свою работу, выполняя сопоставление с образцом по разновидностям выражения. У выражения `match` есть пять вариантов, каждый из которых будет рассмотрен отдельно.

Первый вариант имеет следующий вид:

```
case Var(name) =>
  elem(name)
```

Если выражение является переменной, то результатом станет элемент, сформированный из имени переменной.

Второй вариант выглядит так:

```
case Num(number) =>
  def stripDot(s: String) =
    if s endsWith ".0" then s.substring(0, s.length - 2)
    else s
  elem(stripDot(number.toString))
```

Если выражение является числом, то результатом станет элемент, сформированный из значения числа. Функция `stripDot` очистит изображение числа с плавающей точкой, удалив из строки любой суффикс вида `".0"`.

А вот как выглядит третий вариант:

```
case UnOp(op, arg) =>
  elem(op) beside format(arg, unaryPrecedence)
```

Если выражение представляет собой унарную операцию `UnOp(op, arg)`, то результат будет сформирован из операции `op` и результата форматирования аргумента `arg` с самым высоким из возможных уровнем приоритета, имеющимся в данном окружении¹. Это означает, что, если аргумент `arg` является бинарной операцией (но не делением), то всегда будет отображаться в круглых скобках.

Четвертый вариант представлен следующим кодом:

```
case BinOp("/", left, right) =>
  val top = format(left, fractionPrecedence)
  val bot = format(right, fractionPrecedence)
  val line = elem('-', top.width.max(bot.width), 1)
  val frac = top above line above bot
```

¹ Значение `unaryPrecedence` является самым высоким из возможных приоритетом, поскольку ему было присвоено значение, на единицу превышающее значения приоритета операторов `*` и `%`.

```
if enclPrec != fractionPrecedence then frac
else elem(" ") beside frac beside elem(" ")
```

Если выражение имеет вид дроби, то промежуточный результат `frac` формируется путем помещения отформатированных операндов `left` и `right` друг над другом с разделительным элементом в виде горизонтальной линии. Ширина горизонтальной линии равна максимальной ширине отформатированных операндов. Промежуточный результат становится окончательным, если только дробь сама по себе не появляется в виде аргумента еще одной функции. В последнем случае по обе стороны `frac` добавляется по пробелу. Чтобы понять, зачем это делается, рассмотрим выражение $(a / b) / c$.

Без коррекции по ширине при форматировании этого выражения получится следующая картинка:

```
  a
  -
  b
  -
  c
```

Вполне очевидна проблема с разметкой: непонятно, где именно находится дробная черта верхнего уровня. Показанное ранее выражение может означать либо $(a / b) / c$, либо $a / (b / c)$. Чтобы устранить неоднозначность, с обеих сторон разметки вложенной дроби a / b нужно добавить пробелы.

Тогда разметка станет однозначной:

```
  a
  -
  b
  ---
  c
```

Пятый и последний вариант выглядит следующим образом:

```
case BinOp(op, left, right) =>
  val opPrec = precedence(op)
  val l = format(left, opPrec)
  val r = format(right, opPrec + 1)
  val oper = l beside elem(" " + op + " ") beside r
  if enclPrec <= opPrec then oper
  else elem("(") beside oper beside elem(")")
```

Этот вариант применяется ко всем остальным бинарным операциям. Он указан после варианта, который начинается со следующего кода:

```
case BinOp("/", left, right) => ...
```

поэтому понятно, что оператор `op` в паттерне `BinOp(op, left, right)` не может быть оператором деления. Чтобы форматировать такую бинарную операцию, вам нужно сначала отформатировать его операнды `left` и `right`. В качестве параметра уровня приоритета для форматирования левого операнда используется `opPreс` оператора `op`, а для правого операнда берется уровень на единицу больше. Вдобавок такая схема обеспечивает правильную ассоциативность, выраженную круглыми скобками.

Например, операция

```
BinOp("-", Var("a"), BinOp("-", Var("b"), Var("c")))
```

будет вполне корректно выражена с применением круглых скобок в виде `a - (b - c)`. Затем с помощью выстраивания в линию операндов `left` и `right`, разделенных оператором, формируется промежуточный результат `oper`. Если уровень приоритета текущего оператора ниже уровня приоритета оператора, заключенного в скобки, то `oper` помещается между круглыми скобками, в противном случае возвращается в исходном виде.

На этом разработка приватной функции `format` завершена. Остается только публичный метод `format`, который позволяет программистам, создающим клиентский код, форматировать высокоуровневые выражения, не прибегая к передаче аргумента, содержащего уровень приоритета. Демонстрационная программа, с помощью которой проверяется `ExprFormatter`, показана в листинге 13.22.

Листинг 13.22. Приложение, выполняющее вывод отформатированных выражений

```
import org.stairwaybook.expr.*

object Express:

  def main(args: Array[String]): Unit =

    val f = new ExprFormatter

    val e1 = BinOp("*", BinOp("/", Num(1), Num(2)),
                      BinOp("+", Var("x"), Num(1)))

    val e2 = BinOp("+", BinOp("/", Var("x"), Num(2)),
                      BinOp("/", Num(1.5), Var("x")))

    val e3 = BinOp("/", e1, e2)

    def show(e: Expr) = println(s"${f.format(e)}\n\n")

    for e <- Vector(e1, e2, e3) do show(e)
```

Поскольку этот объект определяет метод `main`, он является работоспособным приложением. Запустить программу `Express` можно командой

```
scala Express
```

При этом будет получен следующий вывод:

```
1
- * (x + 1)
2
```

```
x  1.5
- + ---
2   x
```

```
1
- * (x + 1)
2
-----
x  1.5
- + ---
2   x
```

Резюме

В этой главе мы представили подробную информацию о `case`-классах и сопоставлении с образцом. Они позволяют получить преимущества в процессе использования ряда лаконичных идиом, которые обычно недоступны в объектно-ориентированных языках. Но реализованное в `Scala` сопоставление с образцом не ограничивается тем, что было показано в данной главе. Если нужно задействовать сопоставление с образцом, но при этом нежелательно открывать доступ к вашим классам, как делается в `case`-классах, то можно обратиться к *экстракторам*. В следующей главе мы рассмотрим списки.

14

Работа со списками

Вероятно, наиболее востребованные структуры данных в программах на Scala — списки. В этой главе мы подробно разьясим, что это такое. В ней мы представим много общих операций, которые могут выполняться над списками. Кроме того, раскроем некоторые наиболее важные принципы проектирования программ, работающих со списками.

14.1. Литералы списков

Списки уже попадались в предыдущих главах, следовательно, вам известно, что список, содержащий элементы 'a', 'b' и 'c', записывается как `List('a', 'b', 'c')`. А вот другие примеры:

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty = List()
```

Списки очень похожи на массивы, но имеют два важных отличия. Во-первых, списки являются неизменяемой структурой данных, то есть их элементы нельзя изменить путем присваивания. Во-вторых, списки имеют рекурсивную структуру (имеется в виду *связанный список*), а у массивов она линейная.

14.2. Тип List

Как и массивы, списки *однородны*: у всех элементов списка один и тот же тип. Тип списка, имеющего элементы типа `T`, записывается как `List[T]`. Например, далее показаны те же четыре списка, что и выше, с явным указанием типов:

```
val fruit: List[String] = List("apples", "oranges", "pears")
val nums: List[Int] = List(1, 2, 3, 4)
val diag3: List[List[Int]] =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty: List[Nothing] = List()
```

В Scala тип списка обладает *ковариантностью*. Это значит, что для каждой пары типов `S` и `T`, если `S` — подтип `T`, то `List[S]` — подтип `List[T]`. Например, `List[String]` — подтип `List[Object]`. И это вполне естественно, поскольку каждый список строк также может рассматриваться как список объектов¹.

Обратите внимание: типом пустого списка является `List[Nothing]`. `Nothing` считается «низшим типом» в иерархии классов Scala, так как он является подтипом любого другого имеющегося в Scala типа. Списки ковариантны, отсюда следует, что `List[Nothing]` — подтип `List[T]` для любого типа `T`. Следовательно, пустой списочный объект, имеющий тип `List[Nothing]`, также может рассматриваться в качестве объекта любого другого списочного типа, имеющего вид `List[T]`. Именно поэтому вполне допустимо написать такой код:

```
// List() также относится к типу List[String]!
val xs: List[String] = List()
```

14.3. Создание списков

Все списки создаются из двух основных строительных блоков: `Nil` и `::` (произносится как «конс», от слова «конструировать»). `Nil` представляет собой пустой список. Инфиксный оператор конструирования `::` обозначает расширение списка с начала. То есть запись `x :: xs` представляет собой список, первым элементом которого является `x`, за которым следует список `xs` (его

¹ Более подробно ковариантность и другие разновидности вариантности рассмотрены в главе 18.

элементы). Следовательно, предыдущие списочные значения можно определить так:

```
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
            (0 :: (1 :: (0 :: Nil))) ::
            (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

Фактически предыдущие определения `fruit`, `nums`, `diag3` и `empty`, выраженные в виде `List(...)`, всего лишь оболочки, которые разворачиваются в эти определения. Например, применение `List(1, 2, 3)` приводит к созданию списка `1 :: (2 :: (3 :: Nil))`.

То, что операция `::` заканчивается двоеточием, означает ее правую ассоциативность: `A :: B :: C` интерпретируется как `A :: (B :: C)`. Поэтому круглые скобки в предыдущих определениях можно отбросить. Например

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

будет эквивалентом предыдущего определения `nums`.

14.4. Основные операции над списками

Все действия со списками можно свести к трем основным операциям:

- `head` возвращает первый элемент списка;
- `tail` возвращает список, состоящий из всех элементов, за исключением первого;
- `isEmpty` возвращает `true`, если список пуст.

Эти операции определены как методы класса `List`. Некоторые примеры их использования показаны в табл. 14.1. Методы `head` и `tail` определены только для непустых списков. Будучи примененными к пустому списку, они генерируют исключение:

```
scala> Nil.head
java.util.NoSuchElementException: head of empty list
```

В качестве примера того, как можно обрабатывать список, рассмотрим сортировку элементов списка чисел в возрастающем порядке. Один из простых способов выполнить эту задачу заключается в *сортировке вставками*,

которая работает так: для сортировки непустого списка `x :: xs` сортируется остаток `xs` и первый элемент `x` вставляется в нужную позицию результата.

Таблица 14.1. Основные операции над списками

Что используется	Что этот метод делает
<code>empty.isEmpty</code>	Возвращает <code>true</code>
<code>fruit.isEmpty</code>	Возвращает <code>false</code>
<code>fruit.head</code>	Возвращает <code>"apples"</code>
<code>fruit.tail.head</code>	Возвращает <code>"oranges"</code>
<code>diag3.head</code>	Возвращает <code>List(1, 0, 0)</code>

Сортировка пустого списка выдает пустой список. Выраженный в виде кода Scala, алгоритм сортировки вставками выглядит так, как показано в листинге 14.1.

Листинг 14.1. Сортировка списка `List[Int]` с помощью алгоритма сортировки вставками

```
def isort(xs: List[Int]): List[Int] =
  if xs.isEmpty then Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]): List[Int] =
  if xs.isEmpty || x <= xs.head then x :: xs
  else xs.head :: insert(x, xs.tail)
```

14.5. Паттерны-списки

Разбирать списки можно и с помощью сопоставления с образцом. Паттерны-списки по порядку следования соответствуют выражениям списков. Используя паттерн вида `List(...)`, можно либо сопоставить все элементы списка, либо разобрать список поэлементно, применив паттерны, составленные из оператора `::` и константы `Nil`.

Пример использования первой разновидности паттерна выглядит следующим образом:

```
scala> val List(a, b, c) = fruit
val a: String = apples
val b: String = oranges
val c: String = pears
```

Паттерн `List(a, b, c)` соответствует спискам длиной три элемента и привязывает эти три элемента к паттернам-переменным `a`, `b` и `c`. Если количество элементов заранее не известно, то лучше вместо этого сопоставлять с помощью оператора `::`. Например, паттерн `a :: b :: rest` соответствует спискам длиной два и более элемента:

```
scala> val a :: b :: rest = fruit
val a: String = apples
val b: String = oranges
val rest: List[String] = List(pears)
```

О сопоставлении с образцом объектов класса `List`

Если провести беглый обзор возможных форм паттернов, рассмотренных в главе 15, то выяснится, что ничего похожего ни на `List(...)`, ни на `::` в определенных там разновидностях нет. Фактически `List(...)` — экземпляр определенного в библиотеке паттерна-экстрактора. Конс-паттерн `x :: xs` — особый случай паттерна инфиксной операции. В качестве выражения инфиксная операция выступает эквивалентом вызова метода. Для паттернов действуют иные правила: в качестве паттерна такая инфиксная операция, как `p op q`, является эквивалентом `op(p, q)`. То есть инфиксный оператор `op` рассматривается в качестве паттерна-конструктора. В частности, такой конс-паттерн, как `x :: xs`, рассматривается как `::(x, xs)`.

Это обстоятельство подсказывает, что должен быть класс по имени `::`, соответствующий паттерну-конструктору. Разумеется, это существующий одноименный класс, который создает непустые списки. Следовательно, `::` в Scala фигурирует дважды: как имя класса и как метод класса `List`. Результатом применения метода `::` является создание экземпляра класса `scala:::`.

Извлечение части списков с помощью паттернов — альтернатива использованию основных методов `head`, `tail` и `isEmpty`. Например, в коде ниже снова применена сортировка вставками, на этот раз записанная с сопоставлением с образцом:

```
def isort(xs: List[Int]): List[Int] =
  xs match
    case List() => List()
    case x :: xs1 => insert(x, isort(xs1))
def insert(x: Int, xs: List[Int]): List[Int] =
```

```
xs match
  case List() => List(x)
  case y :: ys => if x <= y then x :: xs
                 else y :: insert(x, ys)
```

Зачастую применение к спискам сопоставления с образцом оказывается более понятным, чем их декомпозиция с помощью методов, поэтому данное сопоставление должно стать частью вашего инструментария для обработки списков.

Вот и все, что нужно знать о списках в Scala, чтобы их правильно применять. Но существует также множество методов, которые вбирают в себя наиболее распространенные схемы проведения операций со списками. Эти методы делают программы обработки списков более лаконичными и зачастую более понятными. В следующих двух разделах мы представим наиболее важные методы, определенные в классе `List`.

14.6. Методы первого порядка класса `List`

В этом разделе мы рассмотрим большинство определенных в классе `List` методов *первого порядка*. Метод первого порядка не получает в качестве аргументов никаких функций. Мы также представим несколько рекомендованных приемов структурирования программ, работающих со списками, на двух примерах.

Конкатенация двух списков

Операцией, похожей на `::`, является конкатенация списков, записываемая в виде `:::`. В отличие от операции `::` операция `:::` получает в качестве операндов два списка. Результатом выполнения кода `xs ::: ys` выступает новый список, содержащий все элементы списка `xs`, за которыми следуют все элементы списка `ys`.

Рассмотрим несколько примеров:

```
List(1, 2) ::: List(3, 4, 5) // List(1, 2, 3, 4, 5)
List() ::: List(1, 2, 3)    // List(1, 2, 3)
List(1, 2, 3) ::: List(4)  // List(1, 2, 3, 4)
```

Как и конс-оператор, конкатенация списков правоассоциативна. Такое выражение, как

```
xs ::: ys ::: zs
```

интерпретируется следующим образом:

```
xs ::: (ys ::: zs)
```

Принцип «разделяй и властвуй»

Конкатенация (`:::`) реализована в виде метода класса `List`. Можно было бы также реализовать конкатенацию «вручную», используя сопоставление с образцом для списков. Будет поучительно попробовать сделать это самостоятельно, поскольку таким образом можно проследить общий путь реализации алгоритмов с помощью списков. Сначала мы остановимся на сигнатуре метода конкатенации, который назовем `append`. Чтобы не создавать большой путаницы, предположим, что `append` определен за пределами класса `List`, поэтому будет получать в качестве параметров два конкатенируемых списка. Оба они должны быть согласованы по типу их элементов, но сам тип может быть произвольным. Все это можно обеспечить, задав `append` параметр типа¹, представляющего тип элементов двух исходных списков:

```
def append[T](xs: List[T], ys: List[T]): List[T]
```

Чтобы спроектировать реализацию `append`, имеет смысл вспомнить принцип «разделяй и властвуй» для программ, работающих с рекурсивными структурами данных, такими как списки. Многие алгоритмы для работы со списками сначала разбивают исходный список на простые блоки, используя сопоставление с образцом. Это часть принципа, которая называется «разделяй». Затем конструируется результат для каждого варианта. Если результатом является непустой список, то некоторые из его частей можно сконструировать с помощью рекурсивных вызовов того же самого алгоритма. В этом будет заключаться часть принципа, называемая «властвуй».

Чтобы применить этот принцип к реализации метода `append`, сначала стоит ответить на вопрос: какой именно список нужно сопоставлять? Применительно к `append` данный вопрос не настолько прост, как для многих других методов, поскольку здесь имеются два варианта. Но следующая далее фаза «властвования» подсказывает: нужно сконструировать список, состоящий из всех элементов обоих исходных списков. Списки конструируются из конца в начало, поэтому `ys` можно оставить нетронутым, а вот `xs` нужно разобрать и пристроить впереди `ys`. Таким образом, имеет смысл

¹ Более подробно параметры типов будут рассмотрены в главе 18.

сконцентрироваться на `xs` как на источнике сопоставления с образцом. Самое частое сопоставление в отношении списков просто отличает пустой список от непустого. Итак, для метода `append` вырисовывается следующая схема:

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match
    case List() => ???
    case x :: xs1 => ???
```

Остается лишь заполнить два места, обозначенных ???¹. Первое такое место — альтернатива для случая, когда входной список `xs` пуст. В таком случае результатом конкатенации будет второй список:

```
case List() => ys
```

Второе место, оставленное незаполненным, — альтернатива для случая, когда входной список `xs` состоит из некоего `head`-элемента `x`, за которым следует остальная часть `xs1`. В таком случае результатом тоже будет непустой список. Чтобы сконструировать непустой список, нужно знать, какой должна быть его «голова» (`head`), а каким — «хвост» (`tail`). Вам известно, что первый элемент получающегося в результате списка — `x`. Что касается остальных элементов, то их можно вычислить, добавив второй список, `ys`, к оставшейся части первого списка, `xs1`.

Это завершает проектирование и дает:

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
```

Вычисление второй альтернативы — иллюстрация той части принципа, которая называется «властью»: сначала нужно продумать форму желаемого результата, затем вычислить отдельные части этой формы, используя, где возможно, рекурсивные вызовы алгоритма. И наконец, сконструировать из этих частей вывод.

¹ ??? — это метод, который генерирует ошибку `scala.NotImplementedError` и имеет результирующий тип `Nothing`. Он может применяться в качестве временной реализации в процессе разработки приложения.

Получение длины списка: length

Метод `length` вычисляет длину списка:

```
List(1, 2, 3).length // 3
```

Определение длины списков, в отличие от массивов, — довольно затратная операция. Чтобы ее выполнить, нужно пройти по всему списку в поисках его конца, и на это затрачивается время, пропорциональное количеству элементов списка. Поэтому вряд ли имеет смысл заменять `xs.isEmpty` выражением `xs.length == 0`. Результаты будут получены одинаковые, но второе выражение будет выполняться медленнее, в частности, если список `xs` имеет большую длину.

Обращение к концу списка: init и last

Вам уже известны основные операции `head` и `tail`, в результате выполнения которых извлекаются соответственно первый элемент списка и весь остальной список, за исключением первого элемента. У каждой из них есть обратная по смыслу операция: `last` возвращает последний элемент непустого списка, а `init` — список, состоящий из всех элементов, за исключением последнего:

```
val abcde = List('a', 'b', 'c', 'd', 'e')
abcde.last // e
abcde.init // List(a, b, c, d)
```

Подобно методам `head` и `tail`, эти методы, примененные к пустому списку, генерируют исключение:

```
scala> List().init
java.lang.UnsupportedOperationException: init of empty list
    at ...

scala> List().last
java.util.NoSuchElementException: last of empty list
    at ...
```

В отличие от `head` и `tail`, на выполнение которых неизменно затрачивается одно и то же время, для вычисления результата методы `init` и `last` должны обойти весь список. То есть их выполнение требует времени, пропорционального длине списка.

Неплохо было бы организовать ваши данные таким образом, чтобы основная часть обращений приходилось на головной, а не на последний элемент списка.

Реверсирование списков: `reverse`

Если в какой-то момент вычисления алгоритма требуется часто обращаться к концу списка, то порой более разумным будет сначала перестроить список в обратном порядке и работать уже с результатом. Получить такой список можно следующим образом:

```
abcde.reverse // List(e, d, c, b, a)
```

Как и все остальные операции со списками, `reverse` создает новый список, а не изменяет тот, с которым работает. Поскольку списки неизменяемы, сделать это все равно бы было невозможно. Чтобы убедиться в этом, убедитесь, что исходный список `abcde` после операции `reverse` не изменился:

```
abcde // List(a, b, c, d, e)
```

Операции `reverse`, `init` и `last` подчиняются ряду законов, с помощью которых можно анализировать вычисления и упрощать программы.

1. Операция `reverse` является собственной инверсией:

```
xs.reverse.reverse равно xs
```

2. Операция `reverse` превращает `init` в `tail`, а `last` в `head`, за исключением того, что все элементы стоят в обратном порядке:

```
xs.reverse.init равно xs.tail.reverse
xs.reverse.tail равно xs.init.reverse
xs.reverse.head равно xs.last
xs.reverse.last равно xs.head
```

Реверсирование можно реализовать, воспользовавшись конкатенацией (`:::`), как в следующем методе по имени `rev`:

```
def rev[T](xs: List[T]): List[T] =
  xs match
    case List() => xs
    case x :: xs1 => rev(xs1) :: List(x)
```

Но этот метод, вопреки предположениям, менее эффективен. Чтобы убедиться в высокой вычислительной сложности `rev`, представьте, будто список `xs` имеет длину n . Обратите внимание: придется делать n рекурсивных вызовов `rev`. Каждый вызов, за исключением последнего, влечет за собой конкатенацию списков. На конкатенацию `xs :: ys` затрачивается время, пропорциональное длине ее первого аргумента `xs`. Следовательно, общая вычислительная сложность `rev` выражается так:

$$n + (n - 1) + \dots + 1 = (1 + n) \times n / 2.$$

Иными словами, `rev` имеет квадратичную вычислительную сложность по длине его входного аргумента. Если сравнить это обстоятельство со стандартным реверсированием изменяемого связного списка, имеющего линейную вычислительную сложность, то оно вызывает разочарование. Но данная реализация `rev` — не самая лучшая из возможных. Пример, который начинается на с. 321, позволит вам увидеть, как можно ускорить все это.

Префиксы и суффиксы: `drop`, `take` и `splitAt`

Операции `drop` и `take` обобщают `tail` и `init` в том смысле, что возвращают произвольные префиксы или суффиксы списка. Выражение `xs.take(n)` возвращает первые n элементов списка `xs`. Если n больше `xs.length`, то возвращается весь список `xs`. Операция `xs.drop(n)` возвращает все элементы списка `xs`, за исключением первых n элементов. Если n больше `xs.length`, то возвращается пустой список.

Операция `splitAt` разбивает список по заданному индексу, возвращая пару из двух списков¹. Она определяется следующим равенством:

`xs.splitAt(n)` равно `(xs.take(n), xs.drop(n))`

Но операция `splitAt` избегает двойного прохода по элементам списка. Примеры применения этих трех методов выглядят следующим образом:

```
abcde.take(2)    // List(a, b)
abcde.drop(2)   // List(c, d, e)
abcde.splitAt(2) // (List(a, b),List(c, d, e))
```

¹ Как уже упоминалось в разделе 10.12, понятие пары — неформальное название для `Tuple2`.

Выбор элемента: `apply` и `indices`

Произвольный выбор элемента поддерживается методом `apply`, но эта операция менее востребована, чем аналогичная операция для массивов:

```
abcde.apply(2) // в Scala используется довольно редко
```

Что же касается всех остальных типов, то подразумевается, что `apply` вставляется в вызове метода, когда объект появляется в позиции функции. Поэтому показанную ранее строку кода можно сократить до следующей:

```
abcde(2) // в Scala используется довольно редко
```

Одной из причин того, что выбор произвольного элемента менее популярен для списков, чем для массивов, является то, что на выполнение кода `xs(n)` затрачивается время, пропорциональное величине значения индекса n . Фактически метод `apply` определен сочетанием методов `drop` и `head`:

```
xs.apply(n) равно (xs.drop(n)).head
```

Из этого определения также становится понятно, что индексы списков, как и индексы массивов, задаются в диапазоне от 0 до длины списка минус один. Метод `indices` возвращает список, состоящий из всех допустимых индексов заданного списка:

```
abcde.indices // Диапазон от 0 до 5
```

Линеаризация списка списков: `flatten`

Метод `flatten` принимает список списков и линеаризует его в единый список:

```
List(List(1, 2), List(3), List(), List(4, 5)).flatten
// List(1, 2, 3, 4, 5)
```

```
fruit.map(_.toList).flatten
// List(a, p, p, l, e, s, o, r, a, n, g, e,
//      s, p, e, a, r, s)
```

Он применим только к тем спискам, все элементы которых являются списками. Попытка линеаризации других списков выдаст ошибку компиляции:

```
scala> List(1, 2, 3).flatten
1 |List(1, 2, 3).flatten
  |_____^
```

```

| | No implicit view available from
| | Int => IterableOnce[B]
| | where, B is a type variable.

```

Объединение списков: zip и unzip

Операция `zip` получает два списка и формирует список из пар их значений:

```

abcde.indices.zip(abcde)
// Vector((0,a), (1,b), (2,c), (3,d), (4,e))

```

Если списки разной длины, то все элементы без пары отбрасываются:

```

val zipped = abcde.zip(List(1, 2, 3))
// List((a,1), (b,2), (c,3))

```

Особо полезен вариант объединения в пары списка с его индексами. Наиболее эффективно оно выполняется с помощью метода `zipWithIndex`, составляющего пары из каждого элемента списка и той позиции, в которой он появляется в этом списке.

```

abcde.zipWithIndex
// List((a,0), (b,1), (c,2), (d,3), (e,4))

```

Любой список кортежей можно превратить обратно в кортеж списков с помощью метода `unzip`:

```

zipped.unzip // (List(a, b, c),List(1, 2, 3))

```

Методы `zip` и `unzip` реализуют один из способов одновременной работы с несколькими списками. Более эффективный способ сделать то же самое показан в разделе 14.9.

Отображение списков: toString и mkString

Операция `toString` возвращает каноническое строковое представление списка:

```

abcde.toString // List(a, b, c, d, e)

```

Если требуется иное представление, то можно воспользоваться методом `mkString`. Операция `xs mkString (pre, sep, post)` задействует четыре операнда: отображаемый список `xs`, префиксную строку `pre`, отображаемую перед всеми элементами, строковый разделитель `sep`, отображаемый между

последовательно выводимыми элементами, и постфиксную строку, отображаемую в конце.

Результатом операции будет следующая строка:

```
pre + xs(0) + sep + . . . + sep + xs(xs.length - 1) + post
```

У метода `mkString` имеется два перегруженных варианта, которые позволяют отбрасывать некоторые или все его аргументы. Первый вариант получает только строковый разделитель:

```
xs.mkString(sep) равно xs.mkString("", sep, "")
```

Второй вариант позволяет опустить все аргументы:

```
xs.mkString равно xs.mkString("")
```

Рассмотрим несколько примеров:

```
abcde.mkString("[", ", ", "]") // [a,b,c,d,e]
abcde.mkString("") // abcde
abcde.mkString // abcde
abcde.mkString("List(", ", ", ", ")") // List(a, b, c, d, e)
```

Есть также вариант `mkString`, называющийся `addString`; он не возвращает созданную строку в качестве результата, а добавляет ее к объекту `StringBuilder`¹:

```
val buf = new StringBuilder
abcde.addString(buf, "(", ";", ")") // (a;b;c;d;e)
```

Методы `mkString` и `addString` наследуются из супертрейта `Iterable` класса `List`, поэтому их можно применять ко всем другим коллекциям.

Преобразование списков: `iterator`, `toArray`, `copyToArray`

Чтобы выполнить преобразование данных между линейным миром массивов и рекурсивным миром списков, можно воспользоваться методом `toArray` в классе `List` и методом `toList` в классе `Array`:

```
val arr = abcde.toArray // Array(a, b, c, d, e)
arr.toList // List(a, b, c, d, e)
```

¹ Имеется в виду класс `scala.StringBuilder`, а не `java.lang.StringBuilder`.

Есть также метод `copyToArray`, который копирует элементы списка в последовательные позиции массива внутри некоего результирующего массива. Операция

```
xs.copyToArray(arr, start)
```

копирует все элементы списка `xs` в массив `arr`, начиная с позиции `start`. Нужно обеспечить достаточную длину результирующего массива `arr`, чтобы в нем мог поместиться весь список. Рассмотрим пример:

```
val arr2 = new Array[Int](10)
    // Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
List(1, 2, 3).copyToArray(arr2, 3)
arr2 // Array(0, 0, 0, 1, 2, 3, 0, 0, 0, 0)
```

И наконец, если нужно получить доступ к элементам списка через итератор, то можно воспользоваться методом `iterator`:

```
val it = abcde.iterator
it.next() // a
it.next() // b
```

Пример: сортировка слиянием

Ранее представленная сортировка вставками записывается кратко, но эффективность ее невысока. Ее усредненная вычислительная сложность пропорциональна квадрату длины входного списка. Более эффективен алгоритм сортировки *слиянием*.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ

Этот пример — еще одна иллюстрация карринга и принципа «разделяй и властвуй». Кроме того, в нем говорится о вычислительной сложности алгоритма, что может оказаться полезным. Если же вы хотите поскорее перейти к практике, то раздел 14.7 можно спокойно пропустить.

Сортировка слиянием работает следующим образом: если список имеет один элемент или не имеет никаких элементов, то он уже отсортирован, поэтому может быть возвращен в исходном виде. Более длинные списки разбиваются на два подсписка, в каждом из которых содержится около половины элементов исходного списка. Каждый подсписок сортируется с помощью рекурсивного вызова функции `sort`, затем два отсортированных списка объединяются в ходе операции слияния.

Чтобы выполнить обобщенную реализацию сортировки слиянием, вам придется оставить публичными тип элементов сортируемого списка и функцию, которая будет использоваться для сравнения элементов. Максимально обобщенную функцию можно получить, передав ей эти два элемента в качестве параметров. При этом получится реализация, показанная в листинге 14.2.

Листинг 14.2. Функция сортировки слиянием объектов List

```
def msort[T](less: (T, T) => Boolean)
  (xs: List[T]): List[T] =

  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if less(x, y) then x :: merge(xs1, ys)
        else y :: merge(xs, ys1)

  val n = xs.length / 2
  if n == 0 then xs
  else
    val (ys, zs) = xs.splitAt(n)
    merge(msort(less)(ys), msort(less)(zs))
```

Вычислительная сложность `msort` — $n \log(n)$, где n — длина входного списка. Чтобы понять причину происходящего, следует отметить: и разбиение списка на два подсписка, и слияние двух отсортированных списков требуют времени, которое пропорционально длине аргумента `list(s)`. Каждый рекурсивный вызов `msort` вполнину уменьшает количество элементов, используемых им в качестве входных данных. Поэтому производится примерно $\log(n)$ последовательных вызовов, выполняемых до тех пор, пока не будет достигнут базовый вариант для списков длиной в единицу. Но для более длинных списков каждый вызов порождает два последующих вызова. Если все это сложить вместе, получится, что на каждом уровне вызова $\log(n)$ каждый элемент исходных списков примет участие в одной операции разбиения и одной операции слияния.

Следовательно, каждый уровень вызова имеет общий уровень затрат, пропорциональный n . Поскольку число уровней вызова равно $\log(n)$, мы получаем общий уровень затрат, пропорциональный $n \log(n)$. Он не зависит от исходного распределения элементов в списке, следовательно, в наихудшем варианте будет таким же, как уровень затрат в усредненном. Это свойство делает сортировку слиянием весьма привлекательным алгоритмом для сортировки списков.

Пример использования `msort` выглядит следующим образом:

```
msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))  
// List(1, 3, 5, 7)
```

Функция `msort` представляет собой классический образец карринга, рассмотренный в разделе 9.3. Карринг упрощает специализацию функции для конкретных функций сравнения. Рассмотрим пример:

```
val intSort = msort((x: Int, y: Int) => x < y)  
// intSort имеет тип List[Int] => List[Int]
```

Переменная `intSort` ссылается на функцию, которая получает список целочисленных значений и сортирует их в порядке следования чисел. Как объяснялось в разделе 8.6, в этом примере `msort` применяется частично, поскольку в нем отсутствует список аргументов. В данном случае в качестве недостающего аргумента фигурирует список, который должен быть отсортирован. А вот другой пример, который демонстрирует способ возможного определения функции, выполняющей сортировку списка целочисленных значений в обратном порядке следования чисел:

```
val reverseIntSort = msort((x: Int, y: Int) => x > y)
```

Поскольку функция сравнения уже представлена с помощью карринга, при вызове функций `intSort` или `reverseIntSort` нужно будет только предоставить сортируемый список. Рассмотрим несколько примеров:

```
val mixedInts = List(4, 1, 9, 0, 5, 8, 3, 6, 2, 7)  
intSort(mixedInts)  
// List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)  
  
reverseIntSort(mixedInts)  
// List(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

14.7. Методы высшего порядка класса List

У многих операций над списками схожая структура. Раз за разом используется несколько схем. К подобным примерам можно отнести какое-либо преобразование каждого элемента списка; проверку того, что свойство соблюдается для всех элементов списка; извлечение из списка элементов, удовлетворяющих неким критериям; или объединение элементов списка с помощью того или иного оператора. В императивных языках подобные схемы традиционно будут создаваться идиоматическими комбинациями циклов `for` или `while`. В Scala они могут быть выражены более коротко

и непосредственно за счет использования операторов высшего порядка¹, которые реализуются в виде методов, определенных в классе `List`. Этим операторам высшего порядка и посвящен данный раздел.

Отображения списков: `map`, `flatMap` и `foreach`

Операция `xs map f` получает в качестве операндов список `xs` типа `List[T]` и функцию `f` типа `T => U`. Она возвращает список, получающийся в результате применения `f` к каждому элементу списка `xs`, например:

```
List(1, 2, 3).map(_ + 1) // List(2, 3, 4)
val words = List("the", "quick", "brown", "fox")
words.map(_.length)    // List(3, 5, 5, 3)
words.map(_.toList.reverse.mkString)
  // List(eht, kciuq, nworb, xof)
```

Оператор `flatMap` похож на `map`, но в качестве правого операнда получает функцию, возвращающую список элементов. Он применяет функцию к каждому элементу списка и возвращает конкатенацию всех результатов выполнения функции. Разница между `map` и `flatMap` показана в следующем примере:

```
words.map(_.toList)
  // List(List(t, h, e), List(q, u, i, c, k),
  //       List(b, r, o, w, n), List(f, o, x))
words.flatMap(_.toList)
  // List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x)
```

Как видите, там, где `map` возвращает список списков, `flatMap` возвращает единый список, в котором все элементы списков сконкатенированы.

Различия и взаимодействие методов `map` и `flatMap` показаны также следующим выражением, с помощью которого создается список всех пар (i, j) , отвечающих условию $1 \leq j < i < 5$:

```
List.range(1, 5).flatMap(
  i => List.range(1, i).map(j => (i, j))
)
  // List((2,1), (3,1), (3,2), (4,1),
  //       (4,2), (4,3))
```

¹ Под операторами высшего порядка понимаются функции высшего порядка, используемые в системе записи операторов. Как упоминалось в разделе 9.1, функция является функцией высшего порядка, если получает в качестве параметров одну функцию и более.

Метод `List.range` является вспомогательным, создающим список из всех целых чисел в некотором диапазоне. В этом примере он используется дважды в целях создания списков целых чисел: в первый раз — списка целых чисел от 1 (включительно) до 5 (не включительно), во второй — списка целых чисел от 1 до i для каждого значения i , взятого из первого списка. Метод `map` в данном выражении создает список кортежей (i, j) , где $j < i$. Внешний метод `flatMap` в этом примере создает данный список для каждого i между 1 и 5, а затем конкатенирует все результаты. По-другому этот же список может быть создан с помощью выражения `for`:

```
for i <- List.range(1, 5); j <- List.range(1, i) yield (i, j)
```

Третья `map`-подобная операция — `foreach`. Но в отличие от `map` и `flatMap` она получает в качестве правого операнда процедуру (функцию, результирующим типом которой является `Unit`). Она просто применяет процедуру к каждому элементу списка. А сам результат операции также имеет тип `Unit`, то есть никакого результирующего списка не будет. В качестве примера рассмотрим краткий способ суммирования всех чисел списка:

```
scala> var sum = 0
var sum: Int = 0

scala> List(1, 2, 3, 4, 5).foreach(sum += _)

scala> sum
val res39: Int = 15
```

Фильтрация списков: `filter`, `partition`, `find`, `takeWhile`, `dropWhile` и `span`

Операция `xs filter p` получает в качестве операндов список `xs` типа `List[T]` и функцию-предикат `p` типа `T => Boolean`. Эта операция выдает список всех элементов `x` из списка `xs`, для которых `p(x)` вычисляется в `true`, например:

```
List(1, 2, 3, 4, 5).filter(_ % 2 == 0) // List(2, 4)
words.filter(_.length == 3)         // List(the, fox)
```

Метод `partition` похож на метод `filter`, но возвращает пару списков. Один список содержит все элементы, для которых предикат вычисляется в `true`, а другой — все элементы, для которых предикат вычисляется в `false`. Он определяется равенством

```
xs.partition(p) равно (xs.filter(p), xs.filter(!p(_)))
```

Пример его работы выглядит следующим образом:

```
List(1, 2, 3, 4, 5).partition(_ % 2 == 0)
// (List(2, 4),List(1, 3, 5))
```

Метод `find` тоже похож на метод `filter`, но возвращает только первый элемент, который удовлетворяет условию заданного предиката, а не все такие элементы. Операция `xs find p` получает в качестве операндов список `xs` и предикат `p`. Она возвращает `Option`. Если в списке `xs` есть элемент `x`, для которого `p(x)` вычисляется в `true`, то возвращается `Some(x)`. В противном случае `p` вычисляется в `false` для всех элементов и возвращается `None`. Вот несколько примеров работы этого метода:

```
List(1, 2, 3, 4, 5).find(_ % 2 == 0) // Some(2)
List(1, 2, 3, 4, 5).find(_ <= 0)    // None
```

Операторы `takeWhile` и `dropWhile` также получают в качестве правого операнда предикат. Операция `xs.takeWhile(p)` получает самый длинный префикс списка `xs`, в котором каждый элемент удовлетворяет условию предиката `p`. Аналогично этому операция `xs.dropWhile(p)` удаляет самый длинный префикс из списка `xs`, в котором каждый элемент удовлетворяет условию предиката `p`. Ряд примеров использования этих методов выглядит следующим образом:

```
List(1, 2, 3, -4, 5).takeWhile(_ > 0) // List(1, 2, 3)
words.dropWhile(_.startsWith("t"))   // List(quick, brown, fox)
```

Метод `span` объединяет `takeWhile` и `dropWhile` в одну операцию точно так же, как метод `splitAt` объединяет `take` и `drop`. Он возвращает пару из двух списков, определяемых следующим равенством:

```
xs span p равно (xs takeWhile p, xs dropWhile p)
```

Как и `splitAt`, метод `span` избегает двойного прохода элементов списка:

```
List(1, 2, 3, -4, 5).span(_ > 0)
// (List(1, 2, 3),List(-4, 5))
```

Применение предикатов к спискам: `forall` и `exists`

Операция `xs.forall(p)` получает в качестве аргументов список `xs` и предикат `p`. Она возвращает результат `true`, если все элементы списка удовлетворяют условию предиката `p`. Напротив, операция `xs.exists p` возвращает `true`, если в `xs` есть хотя бы один элемент, удовлетворяющий условию предиката `p`.

Например, чтобы определить наличие в матрице, представленной списком списков, строки, состоящей только из нулевых элементов, можно применить следующий код:

```
def hasZeroRow(m: List[List[Int]]) =  
  m.exists(row => row.forall(_ == 0))  
hasZeroRow(diag3) // false
```

Свертка списков: foldLeft и foldRight

Еще один распространенный вид операции объединяет элементы списка с помощью оператора, например:

```
sum(List(a, b, c)) равно 0 + a + b + c
```

Это особый случай операции свертки:

```
def sum(xs: List[Int]): Int = xs.foldLeft(0)(_ + _)
```

Аналогично этому

```
product(List(a, b, c)) равно 1 * a * b * c
```

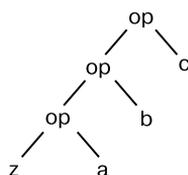
представляет собой особый случай этой операции свертки:

```
def product(xs: List[Int]): Int = xs.foldLeft(1)(_ * _)
```

Операция левой свертки `xs.foldLeft(z)(op)` задействует три объекта: начальное значение `z`, список `xs` и бинарную операцию `op`. Результат свертки — применение `op` между последовательно извлекаемыми элементами списка, где в качестве префикса выступает значение `z`, например:

```
List(a, b, c).foldLeft(z)(op) равно op(op(op(z, a), b), c)
```

Или в графическом представлении:



Вот еще один пример, иллюстрирующий использование операции `foldLeft`. Чтобы объединить все слова в списке из строковых значений с пробелами

между ними и пробелом в самом начале списка, можно задействовать следующий код:

```
words.foldLeft("")( _ + " " + _ ) // " the quick brown fox"
```

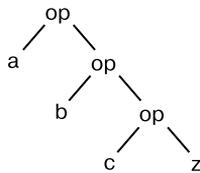
Этот код выдаст лишний пробел в самом начале. Избавиться от него можно с помощью слегка видоизмененного варианта кода:

```
words.tail.foldLeft(words.head)( _ + " " + _ )
// "the quick brown fox"
```

Операция `foldLeft` создает деревья операций с уклоном влево. По аналогии с этим оператор `foldRight` создает деревья операций с уклоном вправо, например:

```
List(a, b, c).foldRight(z)(op) равно op(a, op(b, op(c, z)))
```

Или в графическом представлении:



Для ассоциативных операций левая и правая свертки абсолютно эквивалентны, но эффективность их применения может существенно различаться. Рассмотрим, к примеру, операцию, соответствующую методу `flatten`, которая конкатенирует все элементы в список списков. Она может быть реализована с помощью либо левой, либо правой свертки:

```
def flattenLeft[T](xss: List[List[T]]) =
  xss.foldLeft(List[T]())(_ ::: _)
```

```
def flattenRight[T](xss: List[List[T]]) =
  xss.foldRight(List[T]())(_ ::: _)
```

Поскольку конкатенация списков `xs ::: ys` занимает время, пропорциональное длине его первого аргумента `xs`, то реализация в понятиях правой свертки в `flattenRight` более эффективна, чем реализация с применением левой свертки в `flattenLeft`. Дело в том, что `flattenLeft(xss)` копирует первый элемент списка `xss.head` $n - 1$ раз, где n — длина списка `xss`.

Учтите, что обе версии `flatten` требуют аннотации типа в отношении пустого списка, который является начальным значением свертки. Это связано

с ограничениями в имеющемся в Scala механизме вывода типов, который не в состоянии автоматически вывести правильный тип списка. При попытке игнорировать аннотацию будет выдано такое сообщение об ошибке:

```
scala> def flattenRight[T](xss: List[List[T]]) =
      xss.foldRight(List())(_ ::: _)

2 | xss.foldRight(List())(_ ::: _)
  |                               ^
  |                               Found:    (_$1 : List[T])
  |                               Required: List[Nothing]
```

Чтобы понять, почему не был выполнен надлежащий вывод типа, нужно узнать о типах методов свертки и способах их реализации. Более подробно этот вопрос рассматривается в разделе 14.10.

Пример: реверсирование списков с помощью свертки

Ранее в этой главе была показана реализация метода реверсирования по имени `rev`, время работы которой имело квадратичную вычислительную сложность по длине реверсируемого списка. Теперь будет показана другая реализация метода реверсирования с линейной вычислительной сложностью. Идея состоит в том, чтобы воспользоваться операцией левой свертки, основанной на следующей схеме:

```
def reverseLeft[T](xs: List[T]) =
  xs.foldLeft(стартовое_значение)(операция)
```

Остается заполнить части *стартовое_значение* и *операция*. Собственно, вы можете попытаться вывести эти части из нескольких простых примеров. Чтобы правильно вывести *стартовое_значение*, можно начать с наименьшего потенциального списка, `List()` и рассуждать следующим образом:

```
List()
  равен (в понятиях свойств reverseLeft)

reverseLeft(List())
  равен (по схеме для reverseLeft)

List().foldLeft(стартовое_значение)(операция)
  равен (по определению foldLeft)

стартовое_значение
```

Следовательно, *стартовое_значение* должно быть `List()`. Чтобы вывести второй операнд, в качестве примера можно взять следующий наименьший список. Поскольку уже известно, что *стартовое_значение* — это `List()`, можно рассуждать так:

```
List(x)
  равен (в понятиях свойств reverseLeft)

reverseLeft(List(x))

  равен (по схеме для reverseLeft со стартовым_значением = List())

List(x).foldLeft(List())(операция)
  равен (по определению foldLeft)

операция (List(), x)
```

Следовательно, *операция*(`List()`, `x`) — эквивалент `List(x)`, что можно записать также в виде `x :: List()`. Это наводит на такую мысль: нужно взять в качестве операции оператор `::` с его операндами, которые поменяли местами. (Иногда такую операцию называют «снок», ссылаясь на операцию `:::`, которую называют «конс».) И тогда мы приходим к следующей реализации метода `reverseLeft`:

```
def reverseLeft[T](xs: List[T]) =
  xs.foldLeft(List[T]()) { (ys, y) => y :: ys }
```

Чтобы заставить работать механизм вывода типов, здесь также в качестве аннотации типа требуется использовать код `List[T]()`. Если проанализировать вычислительную сложность `reverseLeft`, то можно прийти к выводу, что в нем n раз применяется постоянная по времени выполнения операция («снок»), где n — длина списка-аргумента. Таким образом, вычислительная сложность `reverseLeft` линейна.

Сортировка списков: `sortWith`

Операция `xs sortWith before`, где `xs` — это список, а `before` — функция, которая может использоваться для сравнения двух элементов, выполняет сортировку элементов списка `xs`. Выражение `x before y` должно возвращать `true`, если в желаемом порядке следования `x` должен стоять перед `y`, например:

```
List(1, -3, 4, 2, 6).sortWith(_ < _) // List(-3, 1, 2, 4, 6)
words.sortWith(_.length > _.length)
// List(quick, brown, the, fox)
```

Обратите внимание: `sortWith` выполняет сортировку слиянием подобно тому, как это делает алгоритм `msort`, показанный в последнем разделе. Но `sortWith` является методом класса `List`, а `msort` определен вне списков.

14.8. Методы объекта List

До сих пор все показанные в этой главе операции реализовывались в качестве методов класса `List`, поэтому вызывались в отношении отдельно взятых списочных объектов. Существует также ряд методов в глобально доступном объекте `scala.List`, который является объектом-компаньоном класса `List`. Одни такие операции — это фабричные методы, создающие списки. Другие же — это операции, работающие со списками некоторых конкретных видов. В этом разделе будут представлены обе разновидности методов.

Создание списков из их элементов: `List.apply`

В книге уже несколько раз попадались литералы списков вида `List(1, 2, 3)`. В их синтаксисе нет ничего особенного. Литерал вида `List(1, 2, 3)` — простое применение объекта `List` к элементам `1, 2, 3`. То есть это эквивалент кода `List.apply(1, 2, 3)`:

```
List.apply(1, 2, 3) // List(1, 2, 3)
```

Создание диапазона чисел: `List.range`

Метод `range`, который ранее подробно рассматривался при изучении методов `map` и `flatMap`, создает список, состоящий из диапазона чисел. Его самая простая форма, при которой создаются все числа, начиная с `from` и заканчивая `until` минус один, — `List.range(from, until)`. Следовательно, последнее значение, `until`, в диапазон не входит.

Существует также версия `range`, получающая в качестве третьего параметра значение `step`. В результате выполнения этой операции получится список элементов, которые следуют друг за другом с указанным шагом, начиная

с `from`. Указываемый шаг `step` может иметь положительное или отрицательное значение:

```
List.range(1, 5)    // List(1, 2, 3, 4)
List.range(1, 9, 2) // List(1, 3, 5, 7)
List.range(9, 1, -3) // List(9, 6, 3)
```

Создание единообразных списков: `List.fill`

Метод `fill` создает список, состоящий из нуля или более копий одного и того же элемента. Он получает два параметра: длину создаваемого списка и повторяемый элемент. Каждый параметр задается в отдельном списке:

```
List.fill(5>('a')) // List(a, a, a, a, a)
List.fill(3)("hello") // List(hello, hello, hello)
```

Если методу `fill` дать более двух аргументов, то он будет создавать многомерные списки, то есть списки списков, списки списков из списков и т. д. Дополнительный аргумент помещается в первый список аргументов.

```
List.fill(2, 3>('b')) // List(List(b, b, b), List(b, b, b))
```

Табулирование функции: `List.tabulate`

Метод `tabulate` создает список, элементы которого вычисляются согласно предоставляемой функции. Аргументы у него такие же, как и у метода `List.fill`: в первом списке аргументов задается размерность создаваемого списка, а во втором дается описание элементов списка. Единственное отличие — элементы не фиксируются, а вычисляются из функции:

```
val squares = List.tabulate(5)(n => n * n)
// List(0, 1, 4, 9, 16)
val multiplication = List.tabulate(5,5)(_ * _)
// List(List(0, 0, 0, 0, 0),
// List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8),
// List(0, 3, 6, 9, 12), List(0, 4, 8, 12, 16))
```

Конкатенация нескольких списков: `List.concat`

Метод `concat` объединяет несколько списков элементов. Конкатенируемые списки предоставляются `concat` в виде непосредственных аргументов:

```
List.concat(List('a', 'b'), List('c')) // List(a, b, c)
List.concat(List(), List('b'), List('c')) // List(b, c)
List.concat() // List()
```

14.9. Совместная обработка нескольких списков

Вы уже знакомы с методом `zip`, который создает список пар из двух списков, позволяя работать с ними одновременно:

```
List(10, 20).zip(List(3, 4, 5)).map { (x, y) => x * y }
// List(30, 80)
```

ПРИМЕЧАНИЕ

Последний `map` использует преимущества такой особенности Scala 3, как разгруппировка параметров, в которой литерал функции с двумя или более параметрами будет автоматически разгруппирован, если ожидаемый тип является функцией, которая принимает один параметр типа кортеж. Например, вызов `map` в предыдущем выражении означает то же самое, что и

```
map { case (x, y) => x * y }
```

Метод `map`, примененный к спискам, прошедшим через `zip`, перебирает не отдельные элементы, а их пары. Первая пара содержит элементы, идущие первыми в каждом списке, вторая — элементы, идущие вторыми, и т. д. Количество пар определяется длиной списков. Обратите внимание: третий элемент второго списка отбрасывается. Метод `zip` объединяет только то количество элементов, которое совместно появляется во всех списках. Любые лишние элементы в конце отбрасываются.

Один из недостатков работы с несколькими списками с помощью метода `zip` состоит в том, что мы получаем промежуточный список (после вызова `zip`), который в конечном счете отбрасывается (при вызове метода `map`). Создание этого промежуточного списка может потребовать существенных расходов, если у него много элементов. Эти две проблемы решает метод `lazyZip`. По своему синтаксису он похож на метод `zip`:

```
(List(10, 20).lazyZip(List(3, 4, 5))).map(_ * _)
// List(30, 80)
```

Разница между `lazyZip` и `zip` в том, что первый не возвращает коллекцию сразу (отсюда и префикс `lazy` — «ленивый»). Вместо этого вы получаете значение, предоставляющее методы (включая `map`) для работы с двумя списками, для которых метод `zip` выполнен отложено. В приведенном выше примере вы можете видеть, как метод `map` принимает функцию с двумя параметрами (вместо одной пары), позволяя нам использовать синтаксис заместителей.

Существуют также обобщающие аналоги для методов `exists` и `forall`. Они похожи на версии этих методов, предназначенные для работы с одним списком, но оперируют элементами не одного, а нескольких списков:

```
(List("abc", "de").lazyZip(List(3, 2))).forall(_.length == _)
// true
(List("abc", "de").lazyZip(List(3, 2))).exists(_.length != _)
// false
```

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ

В последнем разделе этой главы дается информация об имеющемся в Scala алгоритме вывода типов. Если такие подробности сейчас вас не интересуют, то можете пропустить весь раздел и сразу перейти к резюме на с. 330.

14.10. Понимание имеющегося в Scala алгоритма вывода типов

Одно из отличий предыдущего использования `sortWith` и `msort` касается допустимых синтаксических форм функции сравнения.

Сравните этот диалог с интерпретатором:

```
msort((x: Char, y: Char) => x > y)(abcde)
// List(e, d, c, b, a)
```

со следующим:

```
abcde.sortWith(_ > _) // List(e, d, c, b, a)
```

Эти два выражения эквивалентны, но в первом используется более длинная форма функции сравнения с именованными параметрами и явно заданными типами. Во втором задействована более краткая форма, `(_ > _)`, в которой вместо именованных параметров стоят знаки подчеркивания. Разумеется, с методом `sortWith` вы можете применить также первую, более длинную форму сравнения.

А вот с `msort` более краткая форма использоваться не может:

```
scala> msort(_ > _)(abcde)
1 |msort(_ > _)(abcde)
  |~~~~~
  |value > is not a member of Any, but could be made
  | available as an extension method.
```

Чтобы понять, почему именно так происходит, следует знать некоторые подробности имеющегося в Scala алгоритма вывода типов. Это поточный механизм. При использовании метода `m(args)` механизм вывода типов сначала проверяет, имеется ли известный тип у метода `m`. Если да, то именно он и применяется для вывода ожидаемого типа аргументов. Например, в выражении `abcde.sortWith(_ > _)` типом `abcde` является `List[Char]`. Таким образом, `sortWith` известен как метод, получающий аргумент типа `(Char, Char) => Boolean` и выдающий результат типа `List[Char]`. Поскольку типы параметров аргументов функции известны, то их не нужно записывать явным образом. По совокупности всего известного о методе `sortWith` механизм вывода типов может установить, что код `(_ > _)` нужно раскрыть в `((x: Char, y: Char) => x > y)`, где `x` и `y` — некие произвольные только что полученные имена.

Теперь рассмотрим второй вариант, `msort(_ > _)(abcde)`. Типом `msort` является каррированный полиморфный тип метода, который принимает аргумент типа `(T, T) => Boolean` в функцию из `List[T]` в `List[T]`, где `T` — некий пока еще неизвестный тип. Прежде чем он будет применен к своим аргументам, у метода `msort` должен быть создан экземпляр с параметром типа.

Точный тип экземпляра `msort` в приложении еще неизвестен, поэтому он не может быть использован для вывода типа своего первого аргумента. В этом случае механизм вывода типов меняет свою стратегию: сначала он проверяет тип аргументов метода для определения экземпляра метода с подходящим типом. Но когда перед ним стоит задача проверки типа функционального литерала в краткой форме записи, `(_ > _)`, он дает сбой из-за отсутствия информации о неявных типах заданных параметров функции, показанных знаками подчеркивания.

Один из способов решить проблему — передать `msort` явно заданный тип параметра:

```
msort[Char](_ > _)(abcde) // List(e, d, c, b, a)
```

Экземпляр `msort` подходящего типа теперь известен, поэтому его можно использовать для вывода типов аргументов. Еще одним потенциальным решением может стать перезапись метода `msort` таким образом, чтобы его параметры поменялись местами:

```
def msortSwapped[T](xs: List[T])(less:
  (T, T) => Boolean): List[T] = ...

// та же реализация, что и у msort,
// но с аргументами, которые поменялись местами
```

Теперь вывод типов будет выполнен успешно:

```
msortBySwapped(abcde)(_ > _) // List(e, d, c, b, a)
```

Получилось так, что механизм вывода типов воспользовался известным типом первого параметра `abcde`, чтобы определить параметр типа метода `msortBySwapped`. Точный тип `msortBySwapped` был известен, поэтому с его помощью может быть выведен тип второго параметра, `(_ > _)`.

В общем, когда ставится задача вывести параметры типа полиморфного метода, механизм вывода типов принимает во внимание типы всех значений аргументов в первом списке параметров, игнорируя все аргументы, кроме этих. Поскольку `msortBySwapped` — каррированный метод с двумя списками параметров, то не нужно обращать внимание на второй аргумент (то есть на функциональное значение), чтобы определить параметр типа метода.

Эта схема вывода типов предлагает следующий принцип разработки библиотек: при разработке полиморфного метода, который получает некие нефункциональные аргументы и функциональный аргумент, этот функциональный аргумент в самом каррированном списке параметров нужно поставить на последнее место. Тогда экземпляр метода подходящего типа можно вывести из нефункциональных аргументов, и этот тип, в свою очередь, можно использовать для проверки типа функционального аргумента. Совокупный эффект будет состоять в том, что пользователи метода получают возможность предоставить меньший объем информации и написания функциональных литералов более компактными способами.

Теперь рассмотрим более сложный случай, касающийся операции *свертки*. Почему необходимо явно указывать параметр типа в выражении, подобном телу метода `flattenRight`, показанного на с. 320–321?

```
xss.foldRight(List[T]())(_ ::: _)
```

Тип метода `flattenRight` полиморфен в двух переменных типа. Если взять выражение

```
xs.foldRight(z)(op)
```

то типом `xs` должен быть список какого-то произвольного типа `A`, скажем `xs: List[A]`. Начальное значение `z` может быть какого-нибудь другого типа `B`. Тогда операция `op` должна получать два аргумента типа, `A` и `B`, и возвращать результат типа `B`, то есть `op: (A, B) => B`. Тип значения `z` не связан с типом списка `xs`, поэтому у механизма вывода типов нет контекстной информации для `z`.

Теперь рассмотрим выражение в ошибочной версии метода `flattenRight`:

```
xss.foldRight(List())(_ ::: _) // этот код не пройдет компиляцию
```

Начальное значение `z` в данной свертке — пустой список, `List()`, следовательно, при отсутствии дополнительной информации о типе его тип будет выведен как `List[Nothing]`. Исходя из этого, механизм вывода типов установит, что типом `B` в свертке будет являться `List[Nothing]`. Таким образом, для операции `(_ ::: _)` в свертке будет ожидаться следующий тип:

```
(List[T], List[Nothing]) => List[Nothing]
```

Конечно же, такой тип возможен для операции в данной свертке, но пользы от него никакой! Он сообщает, что операция всегда получает в качестве второго аргумента пустой список и всегда в качестве результата выдает также пустой список.

Иными словами, вопрос вывода типов на основе `List()` был решен слишком рано — он должен был выждать, пока не станет виден тип операции `op`. Следовательно, весьма полезное в иных случаях правило о том, что для определения типа метода нужно принимать во внимание только первый список аргументов, будучи применен к каррированному методу, становится камнем преткновения. В то же время, даже если бы это правило было смягчено, механизм вывода типов все равно не смог бы определиться с типом для операции `op`, поскольку ее типы параметров не приведены. Таким образом, создается ситуация, как в уловке-22¹, которую можно разрешить с помощью явной аннотации типа, получаемой от программиста.

Данный пример выявляет ряд ограничений локальной, поточной схемы вывода типов, имеющейся в Scala. В более глобальном механизме вывода типов в стиле Хиндли — Милнера (Hindley — Milner), используемом в таких функциональных языках, как ML или Haskell, подобных ограничений нет. Но по сравнению со стилем Хиндли — Милнера имеющийся в Scala механизм вывода типов обходится с объектно-ориентированной системой подтипов намного изящнее. К счастью, ограничения проявляются только в некоторых крайних случаях, и обычно их без особого труда можно обойти, добавив явную аннотацию типа.

¹ Уловка-22 (Catch-22) — ситуация, возникающая в результате логического парадокса между взаимоисключающими правилами и процедурами. В этой ситуации индивид, подпадающий под действие таких норм, не может их никак контролировать, так как попытка нарушить эти установки автоматически подразумевает их соблюдение.

Добавление аннотаций типа пригодится также при отладке, когда вас поставят в тупик сообщения об ошибках типа, связанных с полиморфными методами. Если вы не уверены в причине возникновения конкретной ошибки типа, то нужно просто добавить некоторые аргументы типа или другие аннотации типа, в правильности которых вы не сомневаетесь. Тогда можно будет быстро понять, где реальный источник проблемы.

Резюме

В этой главе мы показали множество способов работы со списками. Рассмотрели основные операции, такие как `head` и `tail`; операции первого порядка, такие как `reverse`; операции высшего порядка, такие как `map`; и вдобавок полезные методы, определенные в объекте `List`. Попутно мы изучили принципы работы имеющегося в `Scala` механизма вывода типов.

Списки в `Scala` — настоящая рабочая лошадка, поэтому, узнав, как с ними работать, вы сможете извлечь для себя немалую выгоду. Именно с этой целью мы в данной главе погрузились в способы применения списков. Но списки — всего лишь одна из разновидностей коллекций, поддерживаемых в `Scala`. Тематика следующей главы будет скорее охватывающей, чем углубленной. В ней мы покажем вам порядок использования различных типов коллекций `Scala`.

15

Работа с другими коллекциями

В Scala содержится весьма богатая библиотека коллекций. В этой главе мы расскажем о наиболее часто используемых типах коллекций и операциях над ними.

15.1. Последовательности

Типы последовательностей позволяют работать с группами данных, выстроенных по порядку. Поскольку элементы упорядочены, то можно запрашивать первый элемент, второй, 103-й и т. д. В этом разделе мы бегло пройдемся по наиболее важным последовательностям.

Списки

Возможно, самым важным типом последовательности, о котором следует знать, является класс `List` — неизменяемый связный список, подробно рассмотренный в предыдущей главе. Списки поддерживают быстрое добавление и удаление элементов в начало списка, но не позволяют получить быстрый доступ к произвольным индексам, поскольку, чтобы это реализовать, требуется выполнить последовательный обход всех элементов списка.

Такое сочетание свойств может показаться странным, но оно попало в золотую середину и неплохо функционирует во многих алгоритмах. Как следует из описаний, представленных в данной главе, быстрое добавление и удаление начальных элементов означает хорошую работу сопоставления с образцом. Неизменяемость списков помогает разрабатывать корректные эффективные алгоритмы, поскольку избавляет от необходимости создавать копии списков.

Краткий пример, показывающий способ инициализации списка и получения доступа к его голове и хвосту, выглядит так:

```
val colors = List("red", "blue", "green")
colors.head // red
colors.tail // List(blue, green)
```

Чтобы освежить в памяти сведения о списках, обратитесь к шагу 8 в главе 3. А подробности использования списков можно найти в главе 14.

Массивы

Массивы позволяют хранить последовательность элементов и оперативно обращаться к элементу, находящемуся в произвольной позиции, чтобы либо получить его, либо обновить; для этого используется индекс, отсчитываемый от нуля. Массив известной длины, для которого пока неизвестны значения элементов, создается следующим образом:

```
val fiveInts = new Array[Int](5) // Array(0, 0, 0, 0, 0)
```

А вот как инициализируется массив, когда значения элементов известны:

```
val fiveToOne = Array(5, 4, 3, 2, 1) // Array(5, 4, 3, 2, 1)
```

Как уже упоминалось, получить доступ к элементам массивов в Scala можно, указав индекс в круглых, а не в квадратных, как в Java, скобках. Рассмотрим пример доступа к элементу массива и обновления элемента:

```
fiveInts(0) = fiveToOne(4)
fiveInts // Array(1, 0, 0, 0, 0)
```

Массивы в Scala представлены точно так же, как массивы в Java. Поэтому можно абсолютно свободно использовать имеющиеся в Java методы, возвращающие массивы¹.

В предыдущих главах действия с массивами встречались уже много раз. Основы этих действий были рассмотрены в шаге 7 главы 3. Ряд примеров поэлементного обхода массивов с помощью выражения `for` был показан в разделе 7.3.

¹ Разница вариантности массивов в Scala и в Java — то есть является ли `Array[String]` подтипом `Array[AnyRef]` — будет рассмотрена в разделе 18.3.

Буферы списков

Класс `List` предоставляет быстрый доступ к голове и хвосту списка, но не к его концу. Таким образом, при необходимости построить список с добавлением элементов в конец следует рассматривать возможность построить список в обратном порядке путем добавления элементов спереди. Затем, когда это будет сделано, нужно вызвать метод реверсирования `reverse`, чтобы получить элементы в требуемом порядке.

Другой вариант, который позволяет избежать реверсирования, — использовать объект `ListBuffer`. Это содержащийся в пакете `scala.collection.mutable` изменяемый объект, который может помочь более эффективно строить списки, когда нужно добавлять элементы в их конец. Объект обеспечивает постоянное время выполнения операций добавления элементов как в конец, так и в начало списка. В конец списка элемент добавляется с помощью оператора `+=1`, а в начало — с помощью оператора `+=:`. Когда построение будет завершено, можно получить список типа `List`, вызвав в отношении `ListBuffer` метод `toList`. Соответствующий пример выглядит так:

```
import scala.collection.mutable.ListBuffer

val buf = new ListBuffer[Int]
buf += 1 // ListBuffer(1)
buf += 2 // ListBuffer(1, 2)
3 +=: buf // ListBuffer(3, 1, 2)
buf.toList // List(3, 1, 2)
```

Еще один повод использовать `ListBuffer` вместо `List` — возможность предотвратить потенциальное переполнение стека. Если можно создать список в нужном порядке, добавив элементы в его начало, но рекурсивный алгоритм, который потребуется, не является алгоритмом с хвостовой рекурсией, то вместо этого можно задействовать выражение `for` или цикл `while` и `ListBuffer`.

Буферы массивов

Объект `ArrayBuffer` похож на массив, за исключением того, что в дополнение ко всему здесь предоставляет возможность добавлять и удалять элементы в начало и в конец последовательности. Доступны все те же операции, что и в классе `Array`, хотя выполняются они несколько медленнее, поскольку в реализации есть уровень-оболочка. На новые операции добавления

¹ Операторы `+=` и `+=:` являются псевдонимами для `append` и `prepend` соответственно.

и удаления затрачивается в среднем одно и то же время, но иногда требуется время, пропорциональное размеру, из-за реализации, требующей выделить новый массив для хранения содержимого буфера.

Чтобы воспользоваться `ArrayBuffer`, нужно сначала импортировать его из пакета изменяемых коллекций:

```
import scala.collection.mutable.ArrayBuffer
```

При создании `ArrayBuffer` нужно указать параметр типа, а длину указывать не обязательно. По мере надобности `ArrayBuffer` автоматически установит выделяемое пространство памяти:

```
val buf = new ArrayBuffer[Int]()
```

Добавить элемент в `ArrayBuffer` можно с помощью метода `+=`:

```
buf += 12 // ArrayBuffer(12)
buf += 15 // ArrayBuffer(12, 15)
```

Доступны все обычные методы работы с массивами. Например, можно запросить у `ArrayBuffer` его длину или извлечь элемент по его индексу:

```
buf.length // 2
buf(0)     // 12
```

Строки (реализуемые через `StringOps`)

Еще одной последовательностью, заслуживающей упоминания, является `StringOps`. В ней реализованы многие методы работы с последовательностями. Поскольку в `Predef` есть неявное преобразование из `String` в `StringOps`, то с любой строкой можно работать как с последовательностью. Вот пример:

```
def hasUpperCase(s: String) = s.exists(_.isUpper)
hasUpperCase("Robert Frost") // true
hasUpperCase("e e cummings") // false
```

В этом примере метод `exists` вызывается в отношении строки, которая в теле метода `hasUpperCase` называется `s`. В самом классе `String` не объявлено никакого метода по имени `exists`, поэтому компилятор Scala выполнит неявное преобразование `s` в `StringOps`, где такой метод есть. Метод `exists` считает строку последовательностью символов и вернет значение `true`, если какой-либо из них относится к верхнему регистру¹.

¹ Подобный пример представлен на с. 49.

15.2. Множества и отображения

В предыдущих главах, начиная с шага 10 в главе 3, уже были показаны основы множеств и отображений. Прочитав этот раздел, вы получите более глубокое представление о способах их использования и увидите несколько дополнительных примеров.

Ранее мы уже говорили, что библиотека коллекций Scala предлагает как изменяемые, так и неизменяемые версии множеств и отображений. Иерархия множеств показана на рис. 3.2 (см. с. 80), а иерархия отображений — на рис. 3.3 (см. с. 82). Из этих схем следует, что простые имена `Set` и `Map` используются тремя трейтами и все они находятся в разных пакетах.

По умолчанию, когда в коде используется `Set` или `Map`, вы получаете неизменяемый объект. Если нужен изменяемый вариант, то следует применить явно указанное импортное имя. К неизменяемым вариантам Scala предоставляет самый простой доступ — в качестве небольшого поощрения за то, что предпочтение отдано им, а не их изменяемым аналогам. Доступ предоставляется через объект `Predef`, неявно импортируемый в каждый файл исходного кода на языке Scala. Соответствующие определения показаны в листинге 15.1.

Листинг 15.1. Исходные определения отображений `map` и множеств в `set` в `Predef`

```
object Predef:
  type Map[A, +B] = collection.immutable.Map[A, B]
  type Set[A] = collection.immutable.Set[A]
  val Map = collection.immutable.Map
  val Set = collection.immutable.Set
  // ...
end Predef
```

Имена `Set` и `Map` в качестве псевдонимов для более длинных полных имен трейтов неизменяемых множеств и отображений в `Predef` определяются с помощью ключевого слова `type`¹. Чтобы сослаться на объекты-одиночки для неизменяемых `Set` и `Map`, выполняется инициализация `val`-переменных с именами `Set` и `Map`. Следовательно, `Map` является тем же, что и объект `Predef.Map`, который определен быть тем же самым, что и `scala.collection.immutable.Map`. Это справедливо как для типа `Map`, так и для объекта `Map`.

Если нужно воспользоваться как изменяемыми, так и неизменяемыми множествами или отображениями в одном и том же исходном файле, то

¹ Более подробно ключевое слово `type` мы рассмотрим в разделе 20.6.

рекомендуемым подходом является импортирование имен пакетов, содержащих изменяемые варианты:

```
import scala.collection.mutable
```

Можно продолжать ссылаться на неизменяемое множество, как и прежде `Set`, но теперь можно будет сослаться и на изменяемое множество, указав `mutable.Set`. Вот как выглядит соответствующий пример:

```
val mutaSet = mutable.Set(1, 2, 3)
```

Использование множеств

Ключевой характеристикой множества является то, что оно гарантирует наличие каждого объекта не более чем в одном экземпляре, по определению оператора `==`. В качестве примера воспользуемся множеством, чтобы вычислить количество уникальных слов в строке.

Если указать в качестве разделителей слов пробелы и знаки пунктуации, то метод `split` класса `String` может разбить строку на слова. Для этого вполне достаточно применить регулярное выражение `[!, .]+`: оно показывает, что строка должна быть разбита во всех местах, где есть один или несколько пробелов и/или знак пунктуации:

```
val text = "See Spot run. Run, Spot. Run!"
val wordsArray = text.split("[ !, . ]+")
// Array(See, Spot, run, Run, Spot, Run)
```

Чтобы посчитать уникальные слова, их можно преобразовать, приведя их символы к единому регистру, а затем добавить в множество. Поскольку множества исключают дубликаты, то каждое уникальное слово будет появляться в множестве только раз.

Сначала можно создать пустое множество, используя метод `empty`, предоставляемый объектом-компаньоном `Set`:

```
val words = mutable.Set.empty[String]
```

Далее, просто перебирая слова с помощью выражения `for`, можно преобразовать каждое слово, приведя его символы к нижнему регистру, а затем добавить его в изменяемое множество, воспользовавшись оператором `+=`:

```
for word <- wordsArray do
  words += word.toLowerCase
words // Set(see, run, spot)
```

Таким образом, в тексте содержится три уникальных слова: `spot`, `run` и `see`. Наиболее часто используемые методы, применяемые равно к изменяемым и неизменяемым множествам, показаны в табл. 15.1.

Таблица 15.1. Наиболее распространенные операторы для работы с множествами

Что используется	Что этот метод делает
<code>val nums = Set(1, 2, 3)</code>	Создает неизменяемое множество (<code>nums.toString</code> возвращает <code>Set(1, 2, 3)</code>)
<code>nums + 5</code>	Добавляет элемент в неизменяемое множество (возвращает <code>Set(1, 2, 3, 5)</code>)
<code>nums - 3</code>	Удаляет элемент из неизменяемого множества (возвращает <code>Set(1, 2)</code>)
<code>nums ++ List(5, 6)</code>	Добавляет несколько элементов (возвращает <code>Set(1, 2, 3, 5, 6)</code>)
<code>nums -- List(1, 2)</code>	Удаляет несколько элементов из неизменяемого множества (возвращает <code>Set(3)</code>)
<code>nums & Set(1, 3, 5, 7)</code>	Выполняет пересечение двух множеств (возвращает <code>Set(1, 3)</code>)
<code>nums.size</code>	Возвращает размер множества (возвращает 3)
<code>nums.contains(3)</code>	Проверка включения (возвращает <code>true</code>)
<code>import scala.collection.mutable</code>	Упрощает доступ к изменяемым коллекциям
<code>val words = mutable.Set.empty[String]</code>	Создает пустое изменяемое множество (<code>words.toString</code> возвращает <code>Set()</code>)
<code>words += "the"</code>	Добавляет элемент (<code>words.toString</code> возвращает <code>Set(the)</code>)
<code>words -= "the"</code>	Удаляет элемент, если он существует (<code>words.toString</code> возвращает <code>Set()</code>)
<code>words ++= List("do", "re", "mi")</code>	Добавляет несколько элементов (<code>words.toString</code> возвращает <code>Set(do, re, mi)</code>)
<code>words ---= List("do", "re")</code>	Удаляет несколько элементов (<code>words.toString</code> возвращает <code>Set(mi)</code>)
<code>words.clear</code>	Удаляет все элементы (<code>words.toString</code> возвращает <code>Set()</code>)

Применение отображений

Отображения позволяют связать значение с каждым элементом множества. Отображение и массив используются похожим образом, за исключением того, что вместо индексирования с помощью целых чисел, начинающихся с нуля, можно применить ключи любого вида. Если импортировать пакет с именем `mutable`, то можно создать пустое изменяемое отображение:

```
val map = mutable.Map.empty[String, Int]
```

Учтите, что при создании отображения следует указать два типа. Первый тип предназначен для *ключей* отображения, а второй — для их *значений*. В данном случае ключами являются строки, а значениями — целые числа. Задание записей в отображении похоже на задание записей в массиве:

```
map("hello") = 1
map("there") = 2
map // Map(hello > 1, there > 2)
```

По аналогии с этим чтение отображения похоже на чтение массива:

```
map("hello") // 1
```

Чтобы связать все воедино, рассмотрим метод, подсчитывающий количество появлений каждого из слов в строке:

```
def countWords(text: String) =
  val counts = mutable.Map.empty[String, Int]
  for rawWord <- text.split("[ ,!.]+") do
    val word = rawWord.toLowerCase
    val oldCount =
      if counts.contains(word) then counts(word)
      else 0
    counts += (word -> (oldCount + 1))
  counts

countWords("See Spot run! Run, Spot. Run!")
// Map(spot -> 2, see -> 1, run -> 3)
```

Этот код работает благодаря тому, что используется изменяемое отображение по имени `counts` и каждое слово отображается на количество его появлений в тексте. Для каждого слова в тексте выполняется поиск предыдущего количества появлений слова и его увеличение на единицу, а затем в `counts` сохраняется новое значение количества. Обратите внимание: проверка того, встречалось ли это слово раньше, выполняется с помощью метода `contains`.

Если `counts.contains(word)` не возвращает `true`, значит, слово еще не встречалось и за количество принимается ноль.

Многие из наиболее часто используемых методов работы как с изменяемыми, так и с неизменяемыми отображениями показаны в табл. 15.2.

Таблица 15.2. Наиболее часто используемые операции для работы с отображениями

Что используется	Что этот метод делает
<code>val nums = Map("i" -> 1, "ii" -> 2)</code>	Создает неизменяемое отображение (<code>nums.toString</code> возвращает <code>Map(i -> 1, ii -> 2)</code>)
<code>nums + ("vi" -> 6)</code>	Добавляет запись в неизменяемое отображение (возвращает <code>Map(i -> 1, ii -> 2, vi -> 6)</code>)
<code>nums - "ii"</code>	Удаляет запись из неизменяемого отображения (возвращает <code>Map(i -> 1)</code>)
<code>nums ++ List("iii" -> 3, "v" -> 5)</code>	Добавляет несколько записей (возвращает <code>Map(i -> 1, ii -> 2, iii -> 3, v -> 5)</code>)
<code>nums -- List("i", "ii")</code>	Удаляет несколько записей из неизменяемого отображения (возвращает <code>Map()</code>)
<code>nums.size</code>	Возвращает размер отображения (возвращает 2)
<code>nums.contains("ii")</code>	Проверяет на включение (возвращает <code>true</code>)
<code>nums("ii")</code>	Извлекает значение по указанному ключу (возвращает 2)
<code>nums.keys</code>	Возвращает ключи (возвращает результат итерации, выполненной над строками "i" и "ii")
<code>nums.keySet</code>	Возвращает ключи в виде множества (возвращает <code>Set(i, ii)</code>)
<code>nums.values</code>	Возвращает значения (возвращает <code>Iterable</code> над целыми числами 1 и 2)
<code>nums.isEmpty</code>	Показывает, является ли отображение пустым (возвращает <code>false</code>)
<code>import scala.collection.mutable</code>	Упрощает доступ к изменяемым коллекциям
<code>val words = mutable.Map.empty[String, Int]</code>	Создает пустое изменяемое отображение
<code>words += ("one" -> 1)</code>	Добавляет запись в отображение из ключа "one" и значения 1 (<code>words.toString</code> возвращает <code>Map(one -> 1)</code>)
<code>words -= "one"</code>	Удаляет запись из отображения, если она существует (<code>words.toString</code> возвращает <code>Map()</code>)

Таблица 15.2 (окончание)

Что используется	Что этот метод делает
<code>words += List("one" -> 1, "two" -> 2, "three" -> 3)</code>	Добавляет записи в изменяемое отображение (<code>words.toString</code> возвращает <code>Map(one -> 1, two -> 2, three -> 3)</code>)
<code>words -= List("one", "two")</code>	Удаляет несколько объектов (<code>words.toString</code> возвращает <code>Map(three -> 3)</code>)

Множества и отображения, используемые по умолчанию

Для большинства случаев реализаций изменяемых и неизменяемых множеств и отображений, предоставляемых `Set()`, `scala.collection.mutable.Map()` и тому подобными фабриками, наверное, вполне достаточно. Реализации, предоставляемые этими фабриками, используют алгоритм ускоренного поиска, в котором обычно задействуется хеш-таблица, поэтому они могут быстро обнаружить наличие или отсутствие объекта в коллекции.

Так, фабричный метод `scala.collection.mutable.Set()` возвращает `scala.collection.mutable.HashSet`, внутри которого используется хеш-таблица. Аналогично этому фабричный метод `scala.collection.mutable.Map()` возвращает `scala.collection.mutable.HashMap`.

История с неизменяемыми множествами и отображениями несколько сложнее. Как показано в табл. 15.3, класс, возвращаемый фабричным методом `scala.collection.immutable.Set()`, зависит, к примеру, от того, сколько элементов ему было передано. В целях достижения максимальной производительности для множеств, состоящих не более чем из пяти элементов, применяется специальный класс. Но при запросе множества из пяти и более элементов фабричный метод вернет реализацию, использующую хеш.

По аналогии с этим, как следует из данной таблицы, в результате выполнения фабричного метода `scala.collection.immutable.Map()` будет возвращен нужный класс в зависимости от того, сколько пар «ключ — значение» ему передано. Как и в случае с множествами, для того чтобы неизменяемые отображения с количеством элементов меньше пяти достигли максимальной производительности для отображения каждого конкретного размера, используется специальный класс. Но если отображение содержит пять и более пар «ключ — значение», то используется неизменяемый класс `HashMap`.

Таблица 15.3. Реализации используемых по умолчанию неизменяемых множеств

Количество элементов	Реализация
0	<code>scala.collection.immutable.EmptySet</code>
1	<code>scala.collection.immutable.Set1</code>
2	<code>scala.collection.immutable.Set2</code>
3	<code>scala.collection.immutable.Set3</code>
4	<code>scala.collection.immutable.Set4</code>
5 или более	<code>scala.collection.immutable.HashSet</code>

В целях обеспечения максимальной производительности используемые по умолчанию реализации неизменяемых классов, показанные в табл. 15.3 и 15.4, работают совместно. Например, если добавляется элемент к `EmptySet`, то возвращается `Set1`. Если добавляется элемент к этому `Set1`, то возвращается `Set2`. Если затем удалить элемент из `Set2`, то будет опять получен `Set1`.

Таблица 15.4. Реализации используемых по умолчанию неизменяемых отображений

Количество элементов	Реализация
0	<code>scala.collection.immutable.EmptyMap</code>
1	<code>scala.collection.immutable.Map1</code>
2	<code>scala.collection.immutable.Map2</code>
3	<code>scala.collection.immutable.Map3</code>
4	<code>scala.collection.immutable.Map4</code>
5 или более	<code>scala.collection.immutable.HashMap</code>

Отсортированные множества и отображения

Иногда может понадобиться множество или отображение, итератор которого возвращает элементы в определенном порядке. Для этого в библиотеке коллекций Scala имеются трейты `SortedSet` и `SortedMap`. Они реализованы с помощью классов `TreeSet` и `TreeMap`, которые в целях хранения элементов в определенном порядке применяют красно-черное дерево (в случае `TreeSet`) или ключи (в случае с `TreeMap`). Порядок определяется трейтом `Ordered`, неявный экземпляр которого должен быть определен для типа элементов и множества или типа ключей отображения. Эти классы поставляются в изменяемых и неизменяемых вариантах. Рассмотрим ряд примеров использования `TreeSet`:

```
import scala.collection.immutable.TreeSet
val ts = TreeSet(9, 3, 1, 8, 0, 2, 7, 4, 6, 5)
           // TreeSet(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
val cs = TreeSet('f', 'u', 'n') // TreeSet(f, n, u)
```

А это ряд примеров использования `TreeMap`:

```
import scala.collection.immutable.TreeMap
var tm = TreeMap(3 -> 'x', 1 -> 'x', 4 -> 'x')
           // TreeMap(1 -> x, 3 -> x, 4 -> x)
tm += (2 -> 'x')
tm // TreeMap(1 -> x, 2 -> x, 3 -> x, 4 -> x)
```

15.3. Выбор между изменяемыми или неизменяемыми коллекциями

При решении одних задач лучше работают изменяемые коллекции, а при решении других — неизменяемые. В случае сомнений лучше начать с неизменяемой коллекции, а позже при необходимости перейти к изменяемой, поскольку разобраться в работе неизменяемых коллекций гораздо проще.

Иногда может оказаться полезно двигаться в обратном направлении. Если код, использующий изменяемые коллекции, становится сложным и в нем трудно разобраться, то следует подумать, стоит ли заменить некоторые коллекции их неизменяемыми альтернативами. В частности, если вас волнует вопрос создания копий изменяемых коллекций только в нужных местах либо вы слишком много думаете над тем, кто владеет изменяемой коллекцией или что именно она содержит, то имеет смысл заменить некоторые коллекции их неизменяемыми аналогами.

Помимо того что в неизменяемых коллекциях потенциально легче разобраться, они, как правило, могут храниться более компактно, чем изменяемые, если количество хранящихся в них элементов невелико. Например, пустое изменяемое отображение в его представлении по умолчанию в виде `HashMap` занимает примерно 80 байт, и около 16 дополнительных байт требуется для добавления к нему каждой записи. А пустое неизменяемое отображение `Map` — это один объект, который совместно используется всеми ссылками, и потому ссылаться на него можно, по сути, одним полем указателя.

Более того, в настоящее время библиотека коллекций Scala хранит до четырех записей неизменяемых отображений и множеств в одном объекте, который в зависимости от количества хранящихся в коллекции записей

обычно занимает от 16 до 40 байт¹. Следовательно, для небольших множеств и отображений неизменяемые версии занимают намного меньше места, чем изменяемые. С учетом того, что многие коллекции весьма невелики, переход на их неизменяемый вариант может существенно сэкономить пространство памяти и обеспечить преимущество в производительности.

Чтобы облегчить переход с неизменяемых на изменяемые коллекции и наоборот, Scala предоставляет немного синтаксического сахара. Неизменяемые множества и отображения не поддерживают настоящий метод `+=`, однако в Scala дается полезная альтернативная интерпретация `+=`. Когда используется запись `a += b` и `a` не поддерживает метод по имени `+=`, Scala пытается интерпретировать эту запись как `a = a + b`.

Например, неизменяемые множества не поддерживают оператор `+=`:

```
scala> val people = Set("Nancy", "Jane")
val people: Set[String] = Set(Nancy, Jane)

scala> people += "Bob"
1 | people += "Bob"
  | ^^^^^^^^^^^
  | value += is not a member of Set[String]
```

Но если объявить `people` в качестве `var`-, а не `val`-переменной, то коллекцию можно обновить с помощью операции `+=` даже притом, что она неизменяемая. Сначала создается новая коллекция, а затем переменной `people` присваивается новое значение для ссылки на новую коллекцию:

```
var people = Set("Nancy", "Jane")
people += "Bob"
people // Set(Nancy, Jane, Bob)
```

После этой серии инструкций переменная `people` ссылается на новое неизменяемое множество, содержащее добавленную строку `"Bob"`. Та же идея применима не только к методу `+=`, но и к любому другому методу, заканчивающемуся знаком `=`. Вот как тот же самый синтаксис используется с оператором `-=`, который удаляет элемент из множества, и с оператором `++=`, добавляющим в множество коллекцию элементов:

```
people -= "Jane"
people ++= List("Tom", "Harry")
people // Set(Nancy, Bob, Tom, Harry)
```

¹ Под одним объектом, как следует из табл. 15.3 и 15.4, понимается экземпляр одного из классов: от `Set1` до `Set4` или от `Map1` до `Map4`.

Чтобы понять, насколько это полезно, рассмотрим еще раз пример отображения Map из раздела 1.1:

```
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

В этом коде используются неизменяемые коллекции. Если захочется попробовать задействовать вместо них изменяемые коллекции, то нужно будет всего лишь импортировать изменяемую версию Map, переопределив таким образом выполненный по умолчанию импорт неизменяемой версии Map:

```
import scala.collection.mutable.Map // единственное требуемое изменение!
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Так легко преобразовать удастся не все примеры, но особая трактовка методов, заканчивающихся знаком равенства, зачастую сокращает объем кода, который нуждается в изменениях.

Кстати, эта трактовка синтаксиса работает не только с коллекциями, но и с любыми разновидностями значений. Например, здесь она была использована в отношении чисел с плавающей точкой:

```
var roughlyPi = 3.0
roughlyPi += 0.1
roughlyPi += 0.04
roughlyPi // 3.14
```

Эффект от такого расширяющего преобразования похож на эффект, получаемый от операторов присваивания, использующихся в Java (`+=`, `-=`, `*=` и т. п.), однако носит более общий характер, поскольку преобразован может быть каждый оператор, заканчивающийся на `=`.

15.4. Инициализация коллекций

Как уже было показано, наиболее широко востребованный способ создания и инициализации коллекции — передача исходных элементов фабричному методу, определенному в объекте-компаньоне класса выбранной вами коллекции. Элементы просто помещаются в круглые скобки после имени объекта-компаньона, и компилятор Scala преобразует это в вызов метода `apply` в отношении объекта-компаньона:

```
List(1, 2, 3)
Set('a', 'b', 'c')
import scala.collection.mutable
mutable.Map("hi" -> 2, "there" -> 5)
Array(1.0, 2.0, 3.0)
```

Чаще всего можно позволить компилятору Scala вывести тип элемента коллекции из элементов, переданных ее фабричному методу. Но иногда может понадобиться создать коллекцию, указав притом тип, отличающийся от того, который выберет компилятор. Это особенно касается изменяемых коллекций. Рассмотрим пример:

```
scala> import scala.collection.mutable
scala> val stuff = mutable.Set(42)
val stuff: scala.collection.mutable.Set[Int] = HashSet(42)

scala> stuff += "abracadabra"
1 |stuff += "abracadabra"
  | ^^^^^^^^^^^^^^^^^
  | Found: ("abracadabra" : String)
  | Required: Int
```

Проблема здесь заключается в том, что переменной `stuff` был задан тип элемента `Int`. Если вы хотите, чтобы типом элемента был `Any`, то это нужно указать явно, поместив тип элемента в квадратные скобки:

```
scala> val stuff = mutable.Set[Any](42)
val stuff: scala.collection.mutable.Set[Any] = HashSet(42)
```

Еще одна особая ситуация возникает при желании инициализировать коллекцию с помощью другой коллекции. Допустим, есть список, но нужно получить коллекцию `TreeSet`, содержащую элементы, которые находятся в нем. Список выглядит так:

```
val colors = List("blue", "yellow", "red", "green")
```

Передать список названий цветов фабричному методу для `TreeSet` невозможно:

```
scala> import scala.collection.immutable.TreeSet

scala> val treeSet = TreeSet(colors)
1 |val treeSet = TreeSet(colors)
  | ^^^^^^^^^^^^^^^^^
  | No implicit Ordering defined for List[String]..
```

Вместо этого вам нужно будет преобразовать список в `TreeSet` с помощью метода `to`:

```
val treeSet = colors to TreeSet
    // TreeSet(blue, green, red, yellow)
```

Метод `to` принимает в качестве параметра объект-компаньон коллекции. С его помощью вы можете преобразовать любую коллекцию в другую.

Преобразование в массив или список

Помимо универсального метода `to` для преобразования коллекции в другую произвольную коллекцию, вы также можете использовать более конкретные методы для преобразования в наиболее распространенные типы коллекций Scala. Как было показано ранее, чтобы инициализировать новый список с помощью другой коллекции, следует просто вызвать в отношении этой коллекции метод `toList`:

```
treeSet.toList // List(blue, green, red, yellow)
```

Или же, если нужен массив, вызвать метод `toArray`:

```
treeSet.toArray // Array(blue, green, red, yellow)
```

Обратите внимание: несмотря на неотсортированность исходного списка `colors`, элементы в списке, создаваемом вызовом `toList` в отношении `TreeSet`, стоят в алфавитном порядке. Когда в отношении коллекции вызывается `toList` или `toArray`, порядок следования элементов в списке, получающемся в результате, будет таким же, как и порядок следования элементов, создаваемый итератором на этой коллекции. Поскольку итератор, принадлежащий типу `TreeSet[String]`, будет выдавать строки в алфавитном порядке, то они в том же порядке появятся и в списке, который создается в результате вызова `toList` в отношении объекта `TreeSet`.

Разница между `xs to List` и `xs.toList` в том, что реализация `toList` может быть переопределена конкретным типом коллекции `xs`. Это делает преобразование ее элементов в список более эффективным по сравнению с реализацией по умолчанию, копирующей все элементы коллекции. Например, коллекция `ListBuffer` переопределяет метод `toList` с помощью реализации, которая имеет постоянное время выполнения и объем памяти.

Но следует иметь в виду, что преобразование в списки или массивы, как правило, требует копирования всех элементов коллекции и потому для больших коллекций может выполняться довольно медленно. Но иногда это все же приходится делать из-за уже существующего API. Кроме того, многие коллекции содержат всего несколько элементов, а при этом потери в скорости незначительны.

Преобразования между изменяемыми и неизменяемыми множествами и отображениями

Иногда возникает еще одна ситуация, требующая преобразования изменяемого множества либо отображения в неизменяемый аналог или *наоборот*. Выполнить эти задачи следует с помощью метода, показанного чуть ранее. Преобразование неизменяемого множества `TreeSet` из предыдущего примера в изменяемый и обратно в неизменяемый выполняется так:

```
import scala.collection.mutable
treeSet // TreeSet(blue, green, red, yellow)
val mutableSet = treeSet to mutable.Set
      // mutable.HashSet(red, blue, green, yellow)
val immutableSet = mutableSet to Set //
      // Set(red, blue, green, yellow)
```

Ту же технику можно применить для преобразований между изменяемыми и неизменяемыми отображениями:

```
val mutable = mutable.Map("i" -> 1, "ii" -> 2)
mutable // mutable.HashMap(i -> 1, ii -> 2)
val immutable = mutable to Map // Map(ii -> 2, i -> 1)
```

15.5. Кортежи

Согласно описанию, которое дано в шаге 9 в главе 3, кортеж объединяет фиксированное количество элементов, позволяя выполнять их передачу в виде единого целого. В отличие от массива или списка кортеж может содержать объекты различных типов. Вот как, к примеру, выглядит кортеж, содержащий целое число, строку и консоль:

```
(1, "hello", Console)
```

Кортежи избавляют вас от скуки, возникающей при определении упрощенных, насыщенных данными классов. Даже притом что определить класс не составляет особого труда, все же требуется приложить определенное количество порой напрасных усилий. Кортежи избавляют вас от необходимости выбирать имя класса, область видимости, в которой определяется класс, и имена для членов класса. Если класс просто хранит целое число и строку, то добавление класса по имени `AnIntegerAndAString` особой ясности не внесет.

Поскольку кортежи могут сочетать объекты различных типов, они не являются наследниками класса `Iterable`. Если потребуется сгруппировать

ровно одно целое число и ровно одну строку, то понадобится кортеж, а не `List` или `Array`.

Довольно часто кортежи применяются для возвращения из метода нескольких значений. Рассмотрим, к примеру, метод, который выполняет поиск самого длинного слова в коллекции и возвращает наряду с ним его индекс:

```
def longestWord(words: Array[String]): (String, Int) =
  var word = words(0)
  var idx = 0
  for i <- 1 until words.length do
    if words(i).length > word.length then
      word = words(i)
      idx = i
  (word, idx)
```

А вот пример использования этого метода:

```
val longest = longestWord("The quick brown fox".split(" "))
// (quick,1)
```

Функция `longestWord` выполняет здесь два вычисления, получая при этом слово `word`, являющееся в массиве самым длинным, и его индекс `idx`. Во избежание усложнений в функции предполагается, что список имеет хотя бы одно слово, и она отдает предпочтение тому из одинаковых по длине слов, которое стоит в списке первым. Как только функция выберет, какое слово и какой индекс возвращать, она возвращает их вместе, используя синтаксис кортежа `(word, idx)`.

Доступ к элементам кортежа можно получить с помощью круглых скобок и индекса, основанного на нуле. Результат будет иметь соответствующий тип. Например:

```
scala> longest(0)
val res0: String = quick
```

```
scala> longest(1)
val res1: Int = 1
```

Кроме того, значение каждого элемента кортежа можно присвоить собственной переменной¹:

```
scala> val (word, idx) = longest
val word: String = quick
```

¹ Этот синтаксис является, по сути, особым случаем сопоставления с образцом, подробно рассмотренным в разделе 13.7.

```
val idx: Int = 1
```

```
scala> word  
val res55: String = quick
```

Кстати, если не поставить круглые скобки, то будет получен совершенно иной результат:

```
scala> val word, idx = longest  
val word: (String, Int) = (quick,1)  
val idx: (String, Int) = (quick,1)
```

Представленный синтаксис дает *множественное определение* одного и того же выражения. Каждая переменная инициализируется собственным вычислением выражения в правой части. В данном случае неважно, что это выражение вычисляется в кортеж. Обе переменные инициализируются всем кортежем целиком. Ряд примеров, в которых удобно применять множественные определения, можно увидеть в главе 16.

Следует заметить, что кортежи довольно просты в использовании. Они очень хорошо подходят для объединения данных, не имеющих никакого другого смысла, кроме «А и Б». Но когда объединение имеет какое-либо значение или нужно добавить к объединению некие методы, то лучше пойти дальше и создать класс. Например, не стоит использовать кортеж из трех значений, чтобы объединить месяц, день и год, — нужно создать класс **Date**. Так вы явно обозначите свои намерения, благодаря чему код станет более понятным для читателей, и это позволит компилятору и средствам самого языка помочь вам отловить возможные ошибки.

Резюме

В данной главе мы дали обзор библиотеки коллекций Scala и рассмотрели наиболее важные ее классы и трейты. Опираясь на эти знания, вы сможете эффективно работать с коллекциями Scala и будете знать, что именно нужно искать в Scaladoc, когда возникнет необходимость в дополнительных сведениях. Более подробную информацию о коллекциях Scala можно найти в главах 3 и 24. А в следующей главе мы переключим внимание с библиотеки Scala на сам язык и рассмотрим имеющуюся в Scala поддержку изменяемых объектов.

16

Изменяемые объекты

В предыдущих главах в центре внимания были функциональные (неизменяемые) объекты. Дело в том, что идея использования объектов без какого-либо изменяемого состояния заслуживала более пристального рассмотрения. Но в Scala также вполне возможно определять объекты с изменяемым состоянием. Подобные изменяемые объекты зачастую появляются естественным образом, когда нужно смоделировать объекты из реального мира, которые со временем подвергаются изменениям.

В этой главе мы раскроем суть изменяемых объектов и рассмотрим синтаксические средства для их выражения, предлагаемые Scala. Кроме того, рассмотрим большой пример моделирования дискретных событий, в котором используются изменяемые объекты, а также описан внутренний предметно-ориентированный язык (*domain-specific language*, DSL), предназначенный для определения моделируемых цифровых электронных схем.

16.1. Что делает объект изменяемым

Принципиальную разницу между чисто функциональным и изменяемым объектами можно проследить, даже не изучая реализацию объектов. При вызове метода или получении значения поля по указателю в отношении функционального объекта вы всегда будете получать один и тот же результат.

Например, если есть следующий список символов:

```
val cs = List('a', 'b', 'c')
```

то применение `cs.head` всегда будет возвращать `'a'`. То же самое произойдет, даже если между местом определения `cs` и местом, где будет применено об-

ращение `cs.head`, над списком `cs` будет проделано произвольное количество других операций.

Что же касается изменяемого объекта, то результат вызова метода или обращения к полю может зависеть от того, какие операции были ранее выполнены в отношении объекта. Хороший пример изменяемого объекта — банковский счет. Его упрощенная реализация показана в листинге 16.1.

Листинг 16.1. Изменяемый класс банковского счета

```
class BankAccount:

    private var bal: Int = 0

    def balance: Int = bal

    def deposit(amount: Int): Unit =
        require(amount > 0)
        bal += amount

    def withdraw(amount: Int): Boolean =
        if amount > bal then false
        else
            bal -= amount
            true
```

В классе `BankAccount` определяются приватная переменная `bal` и три публичных метода: `balance` возвращает текущий баланс, `deposit` добавляет к `bal` заданную сумму, `withdraw` предпринимает попытку вывести из `bal` заданную сумму, гарантируя при этом, что баланс не станет отрицательным. Возвращаемое `withdraw` значение, имеющее тип `Boolean`, показывает, были ли запрошенные средства успешно выведены.

Даже если ничего не знать о внутренней работе класса `BankAccount`, все же можно сказать, что экземпляры `BankAccounts` являются изменяемыми объектами:

```
val account = new BankAccount
account.deposit(100)
account.withdraw(80) // true
account.withdraw(80) // false
```

Обратите внимание: в двух последних операциях вывода средств в ходе работы с программой были возвращены разные результаты. По выполнении первой операции было возвращено значение `true`, поскольку на банковском счете сохранился достаточный объем, позволяющий вывести средства. Вторая операция вывода средств была такой же, как и первая, однако по ее выполнении было

возвращено значение `false`, поскольку баланс счета уменьшился настолько, что уже не мог покрыть запрошенные средства. Исходя из этого, мы понимаем, что банковским счетам присуще изменяемое состояние, так как при выполнении одной и той же операции в разное время получаются разные результаты.

Можно подумать, будто изменяемость `BankAccount` априори не вызывает сомнений, поскольку в нем содержится определение `var`-переменной. Изменяемость и `var`-переменные обычно идут рука об руку, но ситуация не всегда бывает столь очевидной. Например, класс может быть изменяемым и без определения или наследования каких-либо `var`-переменных, поскольку перенаправляет вызовы методов другим объектам, которые находятся в изменяемом состоянии. Может сложиться и обратная ситуация: класс содержит `var`-переменные и все же является чисто функциональным. Как образец, можно привести класс, кэширующий результаты дорогой операции в поле в целях оптимизации. Чтобы подобрать пример, предположим наличие неоптимизированного класса `Keyed` с дорогой операцией `computeKey`:

```
class Keyed:
  def computeKey: Int = ... // займет некоторое время
  ...
```

При условии, что `computeKey` не читает и не записывает никаких `var`-переменных, эффективность `Keyed` можно увеличить, добавив кэш:

```
class MemoKeyed extends Keyed:
  private var keyCache: Option[Int] = None
  override def computeKey: Int =
    if !keyCache.isDefined then
      keyCache = Some(super.computeKey)
    keyCache.get
```

Использование `MemoKeyed` вместо `Keyed` может ускорить работу: когда результат выполнения операции `computeKey` будет запрошен повторно, вместо еще одного запуска `computeKey` может быть возвращено значение, сохраненное в поле `keyCache`. Но за исключением такого ускорения поведение классов `Keyed` и `MemoKeyed` абсолютно одинаково. Следовательно, если `Keyed` является чисто функциональным классом, то таковым будет и класс `MemoKeyed`, даже притом что содержит переназначаемую переменную.

16.2. Переназначаемые переменные и свойства

В отношении переназначаемой переменной допускается выполнение двух основных операций: получения ее значения или присваивания ей нового.

В таких библиотеках, как JavaBeans, эти операции часто инкапсулированы в отдельные методы считывания (getter) и записи значения (setter), которые необходимо объявлять явно.

В Scala каждая `var`-переменная представляет собой неprivатный член какого-либо объекта, в отношении которого в нем неявно определены методы геттер и сеттер. Но названия таких методов отличаются от предписанных соглашениями Java. Метод получения значения (геттер) `var`-переменной `x` называется просто `x`, а метод присваивания значения (сеттер) — `x_ =`.

Например, появляясь в классе, определение `var`-переменной

```
var hour = 12
```

создает геттер `hour` и сеттер `hour_ =` вдобавок к переназначаемому полю, у которого всегда имеется внутренняя пометка `"object private"`. Она означает, что доступ к полю устанавливается только из объекта, который его содержит. В то же время геттер и сеттер обеспечивают исходной `var`-переменной некоторую видимость. Если `var`-переменная объявлена публичной (`public`), то таковыми же являются и геттер, и сеттер. Если она является защищенной (`protected`), то и они тоже, и т. д.

Рассмотрим, к примеру, класс `Time`, показанный в листинге 16.2, в котором определены две публичные `var`-переменные с именами `hour` и `minute`.

Листинг 16.2. Класс с публичными `var`-переменными

```
class Time:
  var hour = 12
  var minute = 0
```

Эта реализация в точности соответствует определению класса, показанного в листинге 16.3. В данном определении имена локальных полей `h` и `m` были выбраны произвольно, чтобы не конфликтовали с уже используемыми именами.

Листинг 16.3. Как публичные `var`-переменные расширяются в геттер и сеттер

```
class Time:

  private var h = 12
  private var m = 0

  def hour: Int = h
  def hour_(x: Int) =
    h = x

  def minute: Int = m
  def minute_(x: Int) =
    m = x
```

Интересным аспектом такого расширения `var`-переменных в геттер и сеттер является то, что вместо определения `var`-переменной можно также выбрать вариант непосредственного определения этих методов доступа. Он позволяет как угодно интерпретировать операции доступа к переменной и присваивания ей значения. Например, вариант класса `Time`, показанный в листинге 16.4, содержит необходимые условия, благодаря которым перехватываются все присваивания недопустимых значений часам и минутам, хранящимся в переменных `hour` и `minute`.

Листинг 16.4. Непосредственное определение геттера и сеттера

```
class Time:

  private var h = 12
  private var m = 0

  def hour: Int = h
  def hour_(x: Int) =
    require(0 <= x && x < 24)
    h = x

  def minute = m
  def minute_(x: Int) =
    require(0 <= x && x < 60)
    m = x
```

В некоторых языках для этих похожих на переменные величин, не являющихся простыми переменными из-за того, что их геттер и сеттер могут быть переопределены, имеются специальные синтаксические конструкции. Например, в `C#` эту роль играют свойства. По сути, принятое в `Scala` соглашение о постоянной интерпретации переменной как имеющей пару геттер и сеттер предоставляет вам такие же возможности, что и свойства `C#`, но при этом не требует какого-то специального синтаксиса.

Свойства могут иметь множество назначений. В примере, показанном в листинге 16.4, методы присваивания значений навязывают соблюдение конкретных условий, защищая таким образом переменную от присваивания ей недопустимых значений. Кроме того, свойства позволяют регистрировать все обращения к переменной со стороны геттера и сеттера. Или же можно объединять переменные с событиями, например уведомляя с помощью методов-подписчиков о каждом изменении переменной.

Вдобавок возможно, а иногда и полезно определять геттер и сеттер без связанных с ними полей. Например, в листинге 16.5 показан класс `Thermometer`,

в котором инкапсулирована переменная `temperature`, позволяющая читать и обновлять ее значение. Температурные значения могут выражаться в градусах Цельсия или Фаренгейта. Этот класс позволяет получать и устанавливать значение температуры в любых единицах измерения.

Листинг 16.5. Определение геттера и сеттера без связанного с ними поля

```
import scala.compiletime.uninitialized

class Thermometer:

  var celsius: Float = uninitialized

  def fahrenheit = celsius * 9 / 5 + 32

  def fahrenheit_(f: Float) =
    celsius = (f - 32) * 5 / 9

  override def toString = s"${fahrenheit}F/${celsius}C"
```

В первой строке тела этого класса определяется `var`-переменная `celsius`, в которой будет храниться значение температуры в градусах Цельсия. Для переменной `celsius` изначально устанавливается значение по умолчанию: в качестве инициализирующего значения для нее устанавливается знак `= uninitialized`. Точнее, инициализатором поля данному полю присваивается нулевое значение. Суть нулевого значения зависит от типа поля. Для числовых типов это `0`, для булевых — `false`, а для ссылочных — `null`. Получается то же самое, что и при определении в Java некоей переменной без инициализатора.

Учтите, что в Scala просто отбросить инициализатор `= uninitialized` нельзя. Если использовать код

```
var celsius: Float
```

то получится объявление абстрактной, а не инициализированной переменной¹.

За определением переменной `celsius` следуют геттер по имени `fahrenheit` и сеттер `fahrenheit_`, которые обращаются к той же температуре, но в градусах Фаренгейта. В листинге нет отдельного поля, содержащего значение текущей температуры в таких градусах. Вместо этого геттер и сеттер для

¹ Абстрактные переменные будут рассматриваться в главе 20.

значений в градусах Фаренгейта выполняют автоматическое преобразование из градусов Цельсия и в них же соответственно. Пример взаимодействия с объектом `Thermometer` выглядит следующим образом:

```
val t = new Thermometer
t // 32.0F/0.0C

t.celsius = 100
t // 212.0F/100.0C

t.fahrenheit = -40
t // -40.0F/-40.0C
```

16.3. Практический пример: моделирование дискретных событий

Далее в главе на расширенном примере будут показаны интересные способы возможного сочетания изменяемых объектов с функциями, являющимися значениями первого класса. Речь идет о проектировании и реализации симулятора цифровых схем. Эта задача разбита на несколько подзадач, каждая из которых интересна сама по себе.

Сначала мы покажем весьма лаконичный язык для цифровых схем. Его определение подчеркнет общий метод встраивания предметно-ориентированных языков (*domain-specific languages, DSL*) в язык их реализации, подобный `Scala`. Затем представим простую, но всеобъемлющую среду для моделирования дискретных событий. Ее основной задачей будет являться отслеживание действий, выполняемых в ходе моделирования. И наконец, мы покажем, как структурировать и создавать программы дискретного моделирования. Цели создания таких программ — моделирование физических объектов объектами-симуляторами и использование среды для моделирования физического времени.

Этот пример взят из классического учебного пособия Абельсона и Суссмана [Abe96]. Наша ситуация отличается тем, что языком реализации является `Scala`, а не `Scheme`, и тем, что различные аспекты примера структурно выделены в четыре программных уровня. Первый относится к среде моделирования, второй — к основному пакету моделирования схем, третий касается библиотеки определяемых пользователем электронных схем, а четвертый, последний уровень предназначен для каждой моделируемой схемы как таковой. Каждый уровень выражен в виде класса, и более конкретные уровни являются наследниками более общих.

РЕЖИМ УСКОРЕННОГО ЧТЕНИЯ

На разбор примера моделирования дискретных событий, представленного в данной главе, потребуется некоторое время. Если вы считаете, что его лучше было бы потратить на дальнейшее изучение самого языка Scala, то можете перейти к чтению следующей главы.

16.4. Язык для цифровых схем

Начнем с краткого языка для описания цифровых схем, состоящих из *проводников* и *функциональных блоков*. По проводникам проходят *сигналы*, преобразованием которых занимаются функциональные блоки. Сигналы представлены булевыми значениями, где `true` используется для сигнала высокого уровня, а `false` — для сигнала низкого уровня.

Основные функциональные блоки (или *логические элементы*) показаны на рис. 16.1.

- Блок «НЕ» выполняет инверсию входного сигнала.
- Блок «И» устанавливает на своем выходе конъюнкцию сигналов на входе.
- Блок «ИЛИ» устанавливает на своем выходе дизъюнкцию сигналов на входе.

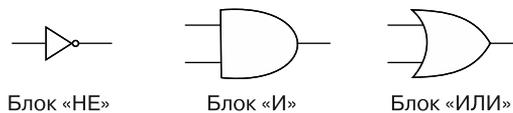


Рис. 16.1. Основные логические элементы

Этих логических элементов вполне достаточно для построения всех остальных функциональных блоков. У логических элементов существуют *задержки*, следовательно, сигнал на выходе элемента будет изменяться через некоторое время после изменения сигнала на его входе.

Элементы цифровой схемы будут описаны с применением набора классов и функций Scala. Сначала создадим класс `Wire` для проводников. Их можно сконструировать следующим образом:

```
val a = new Wire
val b = new Wire
val c = new Wire
```

или то же самое, но покороче:

```
val a, b, c = new Wire
```

Затем понадобятся три процедуры, создающие логические элементы:

```
def inverter(input: Wire, output: Wire): Unit
def andGate(a1: Wire, a2: Wire, output: Wire): Unit
def orGate(o1: Wire, o2: Wire, output: Wire): Unit
```

Необычно то, что в силу имеющегося в Scala функционального уклона логические элементы в этих процедурах вместо возвращения в качестве результата сконструированных элементов конструируются в виде побочных эффектов. Например, вызов `inverter(a, b)` помещает элемент «НЕ» между проводниками `a` и `b`. Получается, что данная конструкция, основанная на побочном эффекте, позволяет упростить постепенное создание все более сложных схем. Вдобавок, притом что имена большинства методов происходят от глаголов, имена этих методов происходят от существительных, показывающих, какой именно элемент создается. Тем самым отображается декларативная природа DSL-языка: он должен давать описание электронной схемы, а не выполняемых в ней действий.

Из логических элементов могут создаваться более сложные функциональные блоки. Например, метод, показанный в листинге 16.6, создает полусумматор. Метод `halfAdder` получает два входных параметра, `a` и `b`, и выдает сумму `s`, определяемую как $s = (a + b) \% 2$, и перенос в следующий разряд `c`, определяемый как $c = (a + b) / 2$. Схема полусумматора показана на рис. 16.2.

Листинг 16.6. Метод `halfAdder`

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) =
  val d, e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)
  andGate(d, e, s)
```

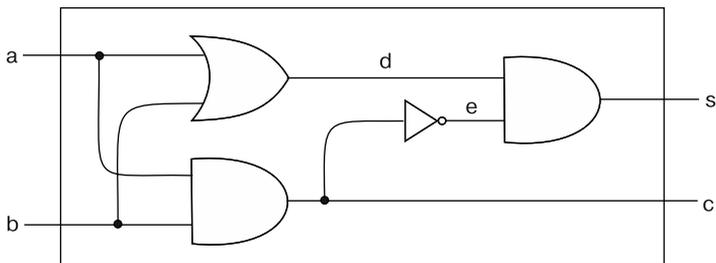


Рис. 16.2. Схема полусумматора

Обратите внимание: `halfAdder` является параметризованным функциональным блоком, как и три метода, составляющие логические элементы. Его можно использовать для составления более сложных схем. Например, в листинге 16.7 определяется полный одноразрядный сумматор (рис. 16.3), который получает два входных параметра, `a` и `b`, а также перенос из младшего разряда (`carry-in`) `cin` и выдает на выходе значение `sum`, определяемое как $sum = (a + b + cin) \% 2$, и перенос в старший разряд (`carry-out`), определяемый как $cout = (a + b + cin) / 2$.

Листинг 16.7. Метод `fullAdder`

```
def fullAdder(a: Wire, b: Wire, cin: Wire,
             sum: Wire, cout: Wire) =

    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
```

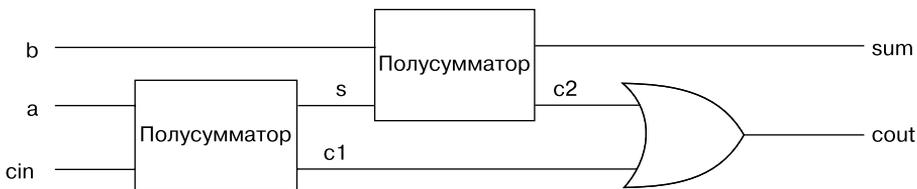


Рис. 16.3. Схема сумматора

Класс `Wire` и функции `inverter`, `andGate` и `orGate` представляют собой краткий язык, с помощью которого пользователи могут определять цифровые схемы. Это неплохой пример *внутреннего* DSL — предметно-ориентированного языка, определенного в виде не какой-то самостоятельной реализации, а библиотеки в языке его реализации.

Реализацию DSL-языка электронных логических схем еще предстоит разработать. Цель определения схемы средствами DSL — моделирование электронной схемы, поэтому вполне разумно будет основой реализации DSL сделать общий API для моделирования дискретных событий. В следующих двух разделах мы представим первый API моделирования, а затем в качестве надстройки над ним покажем реализацию DSL-языка электронных логических схем.

16.5. API моделирования

API моделирования показан в листинге 16.8. Он состоит из класса `Simulation` в пакете `org.stairwaybook.simulation`. Наследниками этого класса являются конкретные библиотеки моделирования, дополняющие его предметно-ориентированную функциональность. В данном разделе представлены элементы класса `Simulation`.

Листинг 16.8. Класс `Simulation`

```
abstract class Simulation:

  type Action = () => Unit

  case class WorkItem(time: Int, action: Action)

  private var curtime = 0
  def currentTime: Int = curtime

  private var agenda: List[WorkItem] = List()

  private def insert(ag: List[WorkItem],
    item: WorkItem): List[WorkItem] =

    if ag.isEmpty || item.time < ag.head.time then item :: ag
    else ag.head :: insert(ag.tail, item)

  def afterDelay(delay: Int)(block: => Unit) =

    val item = WorkItem(currentTime + delay, () => block)
    agenda = insert(agenda, item)

  private def next() =

    (agenda: @unchecked) match
      case item :: rest =>
        agenda = rest
        curtime = item.time
        item.action()

  def run() =

    afterDelay(0) {
      println("*** simulation started, time = " +
        currentTime + " ***")
    }
    while !agenda.isEmpty do next()
```

При моделировании дискретных событий *действия*, определенные пользователем, выполняются в указанные *моменты времени*. Действия, определенные конкретными подклассами моделирования, имеют один и тот же тип:

```
type Action = () => Unit
```

Эта инструкция определяет `Action` в качестве псевдонима типа процедуры, принимающей пустой список параметров и возвращающей тип `Unit`. Тип `Action` является *членом типа* класса `Simulation`. Его можно рассматривать как гораздо более легко читаемое имя для типа `() => Unit`. Члены типов будут подробно рассмотрены в разделе 20.6.

Момент времени, в который выполняется действие, является моментом моделирования — он не имеет ничего общего с временем «настенных часов». Моменты времени моделирования представлены просто как целые числа. Текущий момент хранится в приватной переменной:

```
private var curtime: Int = 0
```

У переменной есть публичный метод доступа, извлекающий текущее время:

```
def currentTime: Int = curtime
```

Это сочетание приватной переменной с публичным методом доступа служит гарантией невозможности изменить текущее время за пределами класса `Simulation`. Обычно не нужно, чтобы моделируемые вами объекты манипулировали текущим временем, за исключением, возможно, случая, когда моделируется путешествие во времени. Действие, которое должно быть выполнено в указанное время, называется *рабочим элементом*. Рабочие элементы реализуются следующим классом:

```
case class WorkItem(time: Int, action: Action)
```

Класс `WorkItem` сделан `case`-классом, чтобы иметь возможность получить следующие синтаксические удобства: для создания экземпляров класса можно использовать фабричный метод `WorkItem` и при этом без каких-либо усилий получить средства доступа к параметрам конструктора `time` и `action`. Следует также заметить, что класс `WorkItem` вложен в класс `Simulation`. Вложенные классы в Scala обрабатываются аналогично Java. Более подробно этот вопрос рассматривается в разделе 20.7.

В классе `Simulation` хранится *план действий* (`agenda`) всех остальных, еще не выполненных рабочих элементов. Они отсортированы по моделируемому времени, в которое должны быть запущены:

```
private var agenda: List[WorkItem] = List()
```

Список `agenda` будет храниться в надлежащем отсортированном порядке благодаря использованию метода `insert`, который обновляет этот список. Вызов метода `insert` в качестве единственного способа добавить рабочий элемент к плану действий можно увидеть в методе `afterDelay`:

```
def afterDelay(delay: Int)(block: => Unit) =
  val item = WorkItem(currentTime + delay, () => block)
  agenda = insert(agenda, item)
```

Как следует из названия, этот метод вставляет действие, задаваемое блоком, в план действий, планируя время задержки его выполнения `delay` после текущего момента моделируемого времени. Например, следующий вызов создаст новый рабочий элемент к выполнению в моделируемое время `currentTime + delay`:

```
afterDelay(delay) { count += 1 }
```

Код, предназначенный для выполнения, содержится во втором аргументе метода. Формальный параметр имеет тип `=> Unit`, то есть это вычисление типа `Unit`, передаваемое по имени. Следует напомнить, что параметры, передаваемые по имени (by-name parameters), при передаче методу не вычисляются. Следовательно, в показанном ранее вызове значение `count` будет увеличено на единицу, только когда среда моделирования вызовет действие, сохраненное в рабочем элементе. Обратите внимание: `afterDelay` — каррированная функция. Это хороший пример разъясненного в разделе 9.5 принципа, согласно которому карринг может использоваться для выполнения вызовов методов, больше похожих на встроены синтаксис языка.

Созданный рабочий элемент еще нужно вставить в план действий. Это делается с помощью метода `insert`, в котором поддерживается предварительное условие отсортированности плана по времени:

```
private def insert(ag: List[WorkItem],
  item: WorkItem): List[WorkItem] =

  if ag.isEmpty || item.time < ag.head.time then item :: ag
  else ag.head :: insert(ag.tail, item)
```

Ядро класса `Simulation` определяется методом `run`:

```
def run() =
  afterDelay(0) {
    println("*** simulation started, time = " +
      currentTime + " ***")
  }
  while !agenda.isEmpty do next()
```

Этот метод периодически берет первый элемент из плана действий, удаляет его из данного плана и выполняет. Он продолжает свою работу до тех пор, пока в плане не останется элементов для выполнения. Каждый шаг выполняется с помощью вызова метода `next`, имеющего такое определение:

```
private def next() =
  (agenda: @unchecked) match
    case item :: rest =>
      agenda = rest
      curtime = item.time
      item.action()
```

В методе `next` текущий план действий разбивается с помощью сопоставления с образцом на первый элемент `item` и весь остальной список рабочих элементов `rest`. Первый элемент из текущего плана удаляется, моделируемое время `curtime` устанавливается на время рабочего элемента, и выполняется действие рабочего элемента.

Обратите внимание: `next` может быть вызван, только если план действий еще не пуст. Варианта для пустого плана нет, поэтому при попытке запуска `next` в отношении пустого списка `agenda` будет выдано исключение `MatchError`.

В действительности же компилятор Scala обычно выдает предупреждение о том, что для списка не указан один из возможных паттернов:

```
27 |   agenda match
   |   ^^^^^^
   |   match may not be exhaustive.
   |
   |   It would fail on pattern case: Nil
```

В данном случае неуказанный вариант никакой проблемы не создает, поскольку известно, что `next` вызывается только в отношении непустого плана действий. Поэтому может возникнуть желание отключить предупреждение. Как было показано в разделе 13.5, это можно сделать, добавив к выражению селектора сопоставления с образцом аннотацию `@unchecked`. Именно поэтому в коде `Simulation` используется `(agenda: @unchecked) match`, а не `agenda match`.

И это правильно. Объем кода для среды моделирования может показаться весьма скромным. Может возникнуть вопрос: а как эта среда вообще может поддерживать содержательное моделирование, если всего лишь выполняет список рабочих элементов? В действительности же эффективность среды моделирования определяется тем фактом, что действия, сохраненные в рабочих элементах, в ходе своего выполнения могут самостоятельно добавлять следующие рабочие элементы в план действий. Тем самым открывается

возможность получить из вычисления простых начальных действий довольно продолжительную симуляцию.

16.6. Моделирование электронной логической схемы

Следующим шагом станет использование среды моделирования в целях реализации предметно-ориентированного языка для логических схем, показанного в разделе 16.4. Следует напомнить, что DSL логических схем состоит из класса для проводников и методов, создающих логические элементы «И», «ИЛИ» и «НЕ». Все это содержится в классе `BasicCircuitSimulation`, который расширяет среду моделирования. Он показан в листинге 16.9.

Листинг 16.9. Класс `BasicCircuitSimulation`

```
package org.stairwaybook.simulation

abstract class BasicCircuitSimulation extends Simulation:

  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire:

    private var sigVal = false
    private var actions: List[Action] = List.empty

    def getSignal = sigVal

    def setSignal(s: Boolean) =
      if s != sigVal then
        sigVal = s
        actions.foreach(_())

    def addAction(a: Action) =
      actions = a :: actions
      a()

  def inverter(input: Wire, output: Wire) =
    def invertAction() =
      val inputSig = input.getSignal
      afterDelay(InverterDelay) {
        output setSignal !inputSig
      }
    input addAction invertAction
```

```
def andGate(a1: Wire, a2: Wire, output: Wire) =
  def andAction() =
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  a1 addAction andAction
  a2 addAction andAction

def orGate(o1: Wire, o2: Wire, output: Wire) =
  def orAction() =
    val o1Sig = o1.getSignal
    val o2Sig = o2.getSignal
    afterDelay(OrGateDelay) {
      output setSignal (o1Sig | o2Sig)
    }
  o1 addAction orAction
  o2 addAction orAction

def probe(name: String, wire: Wire) =
  def probeAction() =
    println(name + " " + currentTime +
      " new-value = " + wire.getSignal)
  wire addAction probeAction
```

В классе `BasicCircuitSimulation` объявляются три абстрактных метода, представляющих задержки основных логических элементов: `InverterDelay`, `AndGateDelay` и `OrGateDelay`. Настоящие задержки на уровне этого класса неизвестны, поскольку зависят от технологии моделируемых логических микросхем. Поэтому задержки в классе `BasicCircuitSimulation` остаются абстрактными, и их конкретное определение делегируется подклассам¹. Далее мы рассмотрим реализацию остальных членов класса `BasicCircuitSimulation`.

Класс `Wire`

Проводникам нужно поддерживать три основных действия:

- `getSignal: Boolean` возвращает текущий сигнал в проводнике;
- `setSignal(sig: Boolean)` выставляет сигнал проводника в `sig`;

¹ Имена этих методов задержки начинаются с прописных букв, поскольку представляют собой константы. Но это методы, и они могут быть переопределены в подклассах. Как те же вопросы решаются с помощью `val`-переменных, мы покажем в разделе 20.3.

- `addAction(p: Action)` прикрепляет указанную процедуру `p` к *действиям* проводника. Замысел заключается в том, чтобы все процедуры действий, прикрепленные к какому-либо проводнику, выполнялись всякий раз, когда сигнал на проводнике изменяется. Как правило, действия добавляются к проводнику подключенными к нему компонентами. Прикрепленное действие выполняется в момент его добавления к проводнику, а после этого всякий раз при изменении сигнала в проводнике.

Реализация класса `Wire` имеет следующий вид:

```
class Wire:

  private var sigVal = false
  private var actions: List[Action] = List.empty

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if s != sigVal then
      sigVal = s
      actions.foreach(_())

  def addAction(a: Action) =
    actions = a :: actions
    a()
```

Состояние проводника формируется двумя приватными переменными. Переменная `sigVal` представляет текущий сигнал, а переменная `actions` — процедуры действий, прикрепленные в данный момент к проводнику. В реализациях методов представляет интерес только та часть, которая относится к методу `setSignal`: когда сигнал проводника изменяется, в переменной `sigVal` сохраняется новое значение. Кроме того, выполняются все действия, прикрепленные к проводнику. Обратите внимание на используемую для этого сокращенную форму синтаксиса: выражение `actions foreach (_())` вызывает применение функции `_()` к каждому элементу в списке действий. В соответствии с описанием, приведенным в разделе 8.5, функция `_()` является сокращенной формой записи для `f => f ()`, то есть получает функцию (назовем ее `f`) и применяет ее к пустому списку параметров.

Метод `inverter`

Единственный результат создания инвертора — то, что действие устанавливается на его входном проводнике. Данное действие вызывается при его установке, а затем всякий раз при изменении сигнала на входе. Эффект

от действия заключается в установке выходного значения (с помощью `setSignal`) на отрицание его входного значения. Поскольку у логического элемента «НЕ» имеется задержка, это изменение должно наступить только по прошествии определенного количества единиц моделируемого времени, хранящегося в переменной `InverterDelay`, после изменения входного значения и выполнения действия. Эти обстоятельства подсказывают следующий вариант реализации:

```
def inverter(input: Wire, output: Wire) =
  def invertAction() =
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  input addAction invertAction
```

Эффект метода `inverter` заключается в добавлении действия `invertAction` к `input`. При вызове данного действия берется входной сигнал и устанавливается еще одно действие, инвертирующее выходной сигнал в плане действий моделирования. Это другое действие должно быть выполнено по прошествии того количества единиц времени моделирования, которое хранится в переменной `InverterDelay`. Обратите внимание на то, как для создания нового рабочего элемента, предназначенного для выполнения в будущем, в методе используется метод `afterDelay`.

Методы `andGate` и `orGate`

Реализация моделирования логического элемента «И» аналогична реализации моделирования элемента «НЕ». Цель — выставить на выходе конъюнкцию его входных сигналов. Это должно произойти по прошествии того количества единиц времени моделирования, которое хранится в переменной `AndGateDelay`, после изменения любого из его двух входных сигналов. Стало быть, подойдет следующая реализация:

```
def andGate(a1: Wire, a2: Wire, output: Wire) =
  def andAction() =
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  a1 addAction andAction
  a2 addAction andAction
```

Эффект от вызова метода `andGate` заключается в добавлении действия `andAction` к обоим входным проводникам, `a1` и `a2`. При вызове данного действия берутся оба входных сигнала и устанавливается еще одно действие, которое выдает выходной сигнал в виде конъюнкции обоих входных сигналов. Это другое действие должно быть выполнено по прошествии того количества единиц времени моделирования, которое хранится в переменной `AndGateDelay`. Учтите, что при смене любого сигнала на входных проводниках выход должен вычисляться заново. Именно поэтому одно и то же действие `andAction` устанавливается на каждом из двух входных проводников, `a1` и `a2`. Метод `orGate` реализуется аналогичным образом, за исключением того, что моделирует логическую операцию «ИЛИ», а не «И».

Вывод симуляции

Для запуска симулятора нужен способ проверки изменения сигналов на проводниках. Чтобы выполнить эту задачу, можно смоделировать действие проверки (пробы) проводника:

```
def probe(name: String, wire: Wire) =
  def probeAction() =
    println(name + " " + currentTime +
      " new-value = " + wire.getSignal)

  wire addAction probeAction
```

Эффект от процедуры `probe` заключается в установке на заданный проводник действия `probeAction`. Как обычно, установленное действие выполняется всякий раз при изменении сигнала на проводнике. В данном случае он просто выводит на стандартное устройство название проводника (которое передается `probe` в качестве первого параметра), а также текущее моделируемое время и новое значение проводника.

Запуск симулятора

После всех этих приготовлений настало время посмотреть на симулятор в действии. Для определения конкретной симуляции нужно выполнить наследование из класса среды моделирования. Чтобы увидеть кое-что интересное, будет создан абстрактный класс моделирования, расширяющий `BasicCircuitSimulation` и содержащий определения методов для полусумматора и сумматора в том виде, в котором они были представлены

в листингах 16.6 и 16.7 соответственно. Этот класс, который будет назван `CircuitSimulation`, показан в листинге 16.10.

Листинг 16.10. Класс `CircuitSimulation`

```
package org.stairwaybook.simulation

abstract class CircuitSimulation
  extends BasicCircuitSimulation:

  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) =
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)

  def fullAdder(a: Wire, b: Wire, cin: Wire,
    sum: Wire, cout: Wire) =

    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
```

Конкретная модель логической схемы будет объектом-наследником класса `CircuitSimulation`. Этому объекту по-прежнему необходимо зафиксировать задержки на логических элементах в соответствии с технологией реализации моделируемой логической микросхемы. И наконец, понадобится также определить конкретную моделируемую схему.

Эти шаги можно проделать в интерактивном режиме в интерпретаторе Scala:

```
scala> import org.stairwaybook.simulation.*
```

Сначала займемся задержками логических элементов. Определим объект (назовем его `MySimulation`), предоставляющий несколько чисел:

```
scala> object MySimulation extends CircuitSimulation:
  def InverterDelay = 1
  def AndGateDelay = 3
  def OrGateDelay = 5
// Определяем объект MySimulation
```

Поскольку предполагается периодически получать доступ к элементам объекта `MySimulation`, импортирование этого объекта укоротит последующий код:

```
scala> import MySimulation.*
```

Далее займемся схемой. Определим четыре проводника и поместим пробы на два из них:

```
scala> val input1, input2, sum, carry = new Wire
val input1: MySimulation.Wire = ...
val input2: MySimulation.Wire = ...
val sum: MySimulation.Wire = ...
val carry: MySimulation.Wire = ...
```

```
scala> probe("sum", sum)
sum 0 new-value = false
```

```
scala> probe("carry", carry)
carry 0 new-value = false
```

Обратите внимание: пробы немедленно выводят выходные данные. Дело в том, что каждое действие, установленное на проводнике, первый раз выполняется при его установке.

Теперь определим подключение к проводникам полусумматора:

```
scala> halfAdder(input1, input2, sum, carry)
```

И наконец, установим один за другим сигналы на двух входящих проводниках на `true` и запустим моделирование:

```
scala> input1 setSignal true
```

```
scala> run()
*** simulation started, time = 0 ***
sum 8 new-value = true
```

```
scala> input2 setSignal true
```

```
scala> run()
*** simulation started, time = 8 ***
carry 11 new-value = true
sum 15 new-value = false
```

Резюме

В данной главе мы собрали воедино две на первый взгляд несопоставимые технологии: изменяемое состояние и функции высшего порядка. Изменяемое состояние было использовано для моделирования физических объектов, состояние которых со временем изменяется. Функции высшего порядка были применены в среде моделирования в целях выполнения действий в указан-

ные моменты моделируемого времени. Они также были использованы в моделировании логических схем в качестве *триггеров*, связывающих действия с изменениями состояния. Попутно мы показали простой способ определить предметно-ориентированный язык в виде библиотеки. Вероятно, для одной главы этого вполне достаточно.

Если вас привлекла эта тема, то можете попробовать создать дополнительные примеры моделирования. Можно объединить полусумматоры и сумматоры для формирования более крупных схем или на основе ранее определенных логических элементов разработать новые схемы и смоделировать их. В главе 19 мы рассмотрим имеющуюся в Scala параметризацию типов и покажем еще один пример, который сочетает в себе функциональный и императивный подходы, дающие весьма неплохое решение.

17

Иерархия Scala

В этой главе мы рассмотрим иерархию классов Scala в целом. В Scala каждый класс наследуется от общего суперкласса по имени `Any`. Поскольку каждый класс является подклассом `Any`, то методы, определенные в классе `Any`, универсальны: их можно вызвать в отношении любого объекта. В самом низу иерархии в Scala также определяются довольно интересные классы `Null` и `Nothing`, которые, по сути, выступают в роли общих *подклассов*. Например, в то время, как `Any` — суперкласс для всех классов, `Nothing` — подкласс для любого класса. В данной главе мы проведем экскурсию по имеющейся в Scala иерархии классов.

17.1. Иерархия классов Scala

На рис. 17.1 в общих чертах показана иерархия классов Scala. На вершине иерархии находится класс `Any`; в нем определяются методы, в число которых входят:

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```

Все классы — наследники класса `Any`, поэтому каждый объект в программе на Scala можно подвергнуть сравнению с помощью `==`, `!=` или `equals`, хешированию с использованием `##` или `hashCode` и форматированию, прибегнув к `toString`. Методы определения равенства `==` и неравенства `!=` объявлены в классе `Any` как `final`, следовательно, переопределить их в подклассах невозможно.

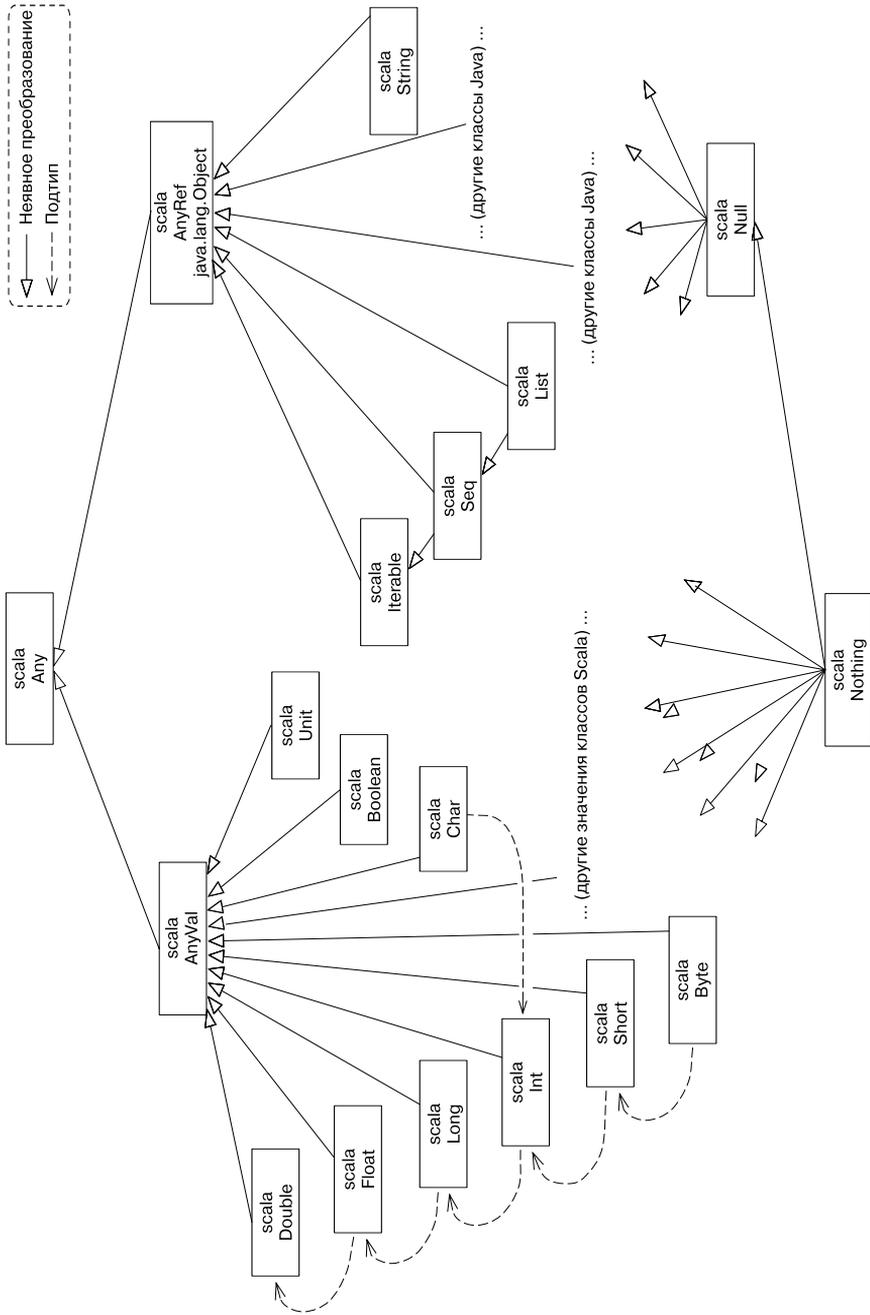


Рис. 17.1. Иерархия классов Scala

Метод `==` — по сути то же самое, что и `equals`, а `!=` всегда является отрицанием метода `equals`¹. Таким образом, отдельные классы могут переопределить смысл значения метода `==` или `!=`, переопределив `equals`.

Множественное равенство

В Scala 3 вводится понятие «множественное равенство», вызывающее ошибку компилятора при использовании методов `==` и `=`, которые отражают вероятные ошибки, например, при сравнении `String` и `Int` на равенство. Этот механизм будет описан в главе 23.

У корневого класса `Any` имеется два подкласса: `AnyVal` и `AnyRef`. Класс `AnyVal` является родительским для *классов значений* в Scala. Наряду с возможностью определять собственные классы значений (см. раздел 17.4) Scala имеет девять встроенных: `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean` и `Unit`. Первые восемь соответствуют примитивным типам Java, и их значения во время выполнения программы представляются в виде примитивных значений Java. Все экземпляры этих классов написаны в Scala в виде литералов. Например, `42`, `'x'`, `false` — экземпляры классов `Int`, `Char` и `Boolean` соответственно. Создать их, используя ключевое слово `new`, невозможно. Этому препятствует особый прием, в котором все классы значений определены одновременно и как абстрактные, и как финальные.

Поэтому, если воспользоваться следующим кодом:

```
scala> new Int
```

то будет получен такой результат:

```
1 | new Int
  |     ^^^
  | Int is abstract; it cannot be instantiated
```

¹ Единственный случай, когда использование `==` не приводит к непосредственному вызову `equals`, относится к упакованным числовым классам Java, таким как `Integer` или `Long`. В Java `new Integer(1)` не эквивалентен `new Long(1)` даже в случае применения примитивных значений `1 == 1L`. Поскольку Scala — более регулярный язык, чем Java, появилась необходимость скорректировать это несоответствие, задействовав для этих классов особую версию метода `==`. Точно так же метод `##` обеспечивает Scala-версию хеширования и похож на Java-метод `hashCode`, за исключением того, что для упакованных числовых типов он всегда работает с методом `==`. Например, для `new Integer(1)` и `new Long(1)` метод `##` вычисляет один и тот же хеш, тогда как Java-методы `hashCode` вычисляют разный хеш-код.

Класс значений `Unit` примерно соответствует имеющемуся в Java типу `void` — он используется в качестве результирующего типа выполнения метода, который не возвращает содержательного результата. Как упоминалось в разделе 7.2, у `Unit` имеется единственное значение экземпляра, оно записывается как `()`.

Как объяснялось в главе 5, в классах значений в качестве методов поддерживаются обычные арифметические и логические (булевы) операторы. Например, у класса `Int` имеются методы `+` и `*`, а у класса `Boolean` — методы `||` и `&&`. Классы значений также наследуют все методы из класса `Any`. Например:

```
42.toString // 42
42.hashCode // 42
42.equals(42) // true
```

Следует отметить, что пространство классов значений плоское: все классы значений являются подтипами `scala.AnyVal`, но не являются подклассами друг друга. Вместо этого между различными типами классов значений существует неявное преобразование типов. Например, экземпляр класса `scala.Int`, когда это требуется, автоматически расширяется (путем неявного преобразования) в экземпляр класса `scala.Long`.

Как упоминалось в разделе 5.10, неявное преобразование используется также для добавления большей функциональности к типам значений. Например, тип `Int` поддерживает все показанные далее операции:

```
42.max(43) // 43
42.min(43) // 42
1 until 5 // Range 1 until 5
1 to 5 // Range 1 to 5
3.abs // 3
-3.abs // 3
```

Работает это следующим образом: все методы `min`, `max`, `until`, `to` и `abs` определены в классе `scala.runtime.RichInt`, а между классами `Int` и `RichInt` существует неявное преобразование. Оно применяется при вызове в отношении `Int`-объекта метода, который определен не в классе `Int`, а в `RichInt`. По аналогии с этим «классы-усилители» и неявные преобразования существуют и для других классов значений¹.

¹ Этот вариант использования неявных преобразований со временем будет заменен методами расширения, описанными в главе 22.

Другим подклассом корневого класса `Any` является `AnyRef` — база всех *ссылочных классов* в Scala. Как упоминалось ранее, на платформе Java `AnyRef` фактически является псевдонимом класса `java.lang.Object`, а значит, все классы, написанные на Java и Scala, — наследники `AnyRef`¹. Поэтому `java.lang.Object` считается способом реализации `AnyRef` на платформе Java. Таким образом, хоть `Object` и `AnyRef` и можно взаимозаменяемо использовать в программах Scala на платформе Java, рекомендуемым стилем будет повсеместное применение `AnyRef`.

17.2. Как реализованы примитивы

Как все это реализовано? Фактически в Scala целочисленные значения хранятся так же, как и в Java, — в виде 32-разрядных слов. Это необходимо для эффективной работы виртуальной машины Java (JVM) и обеспечения возможности совместной работы с библиотеками Java. Такие стандартные операции, как сложение или умножение, реализуются в качестве примитивных операций. Однако Scala использует «резервный» класс `java.lang.Integer` везде, где целое число должно выглядеть как (Java) объект. Так происходит, например, при вызове метода `toString` для целого числа или присваивании этого числа переменной типа `Any`. При необходимости целочисленные значения типа `Int` явно преобразуются в упакованные целые числа типа `java.lang.Integer`.

Это во многом походит на автоупаковку (auto-boxing) в Java, два процесса действительно очень похожи. Но все-таки есть одно коренное различие: упаковка в Scala гораздо менее заметна, чем в Java. Попробуйте выполнить в Java следующий код:

```
// Это код на языке Java
boolean isEqual(int x, int y) {
    return x == y;
}
System.out.println(isEqual(421, 421));
```

В результате, конечно же, будет получено значение `true`. А теперь измените типы аргументов `isEqual` на `java.lang.Integer` (или с аналогичным результатом на `Object`):

¹ Одна из причин существования псевдонима `AnyRef`, заменяющего использование имени `java.lang.Object`, заключается в том, что Scala изначально разрабатывался для работы как на платформе Java, так и на платформе .NET. На платформе .NET `AnyRef` был псевдонимом для `System.Object`.

```
// Это код на языке Java
boolean isEqual(Integer x, Integer y) {
    return x == y;
}
System.out.println(isEqual(421, 421));
```

В итоге получите результат `false!` Оказывается, число `421` было упаковано дважды, поэтому аргументами для `x` и `y` стали два разных объекта. Применение `==` в отношении ссылочных типов означает равенство ссылок, а `Integer` — ссылочный тип, вследствие чего в результате получается `false`. Это один из аспектов, свидетельствующих о том, что Java не является чистым объектно-ориентированным языком. Существует четко видимая разница между примитивными и ссылочными типами.

Теперь попробуйте провести тот же самый эксперимент на Scala:

```
def isEqual(x: Int, y: Int) = x == y
isEqual(421, 421) // true
def isEqual(x: Any, y: Any) = x == y
isEqual(421, 421) // true
```

Операция `==` в Scala разработана так, чтобы быть понятной относительно представления типа. Для типов значений (числовых или логических) это вполне естественное равенство. Для ссылочных типов, отличающихся от упакованных числовых типов Java, `==` рассматривается в качестве псевдонима метода `equals`, унаследованного от класса `Object`. Данный метод изначально определен в целях выявления равенства ссылок, но во многих подклассах переопределяется для реализации их естественных представлений о равенстве. Это также означает, что в Scala вы никогда не попадете в хорошо известную в Java ловушку, касающуюся сравнения строк. В Scala оно работает вполне корректно:

```
val x = "abcd".substring(2) // cd
val y = "abcd".substring(2) // cd
x == y // true
```

В Java результатом сравнения `x` с `y` будет `false`. В этом случае программист должен был воспользоваться методом `equals`, но данный нюанс нетрудно упустить из виду.

Может сложиться и такая ситуация, при которой вместо равенства, определяемого пользователем, нужно проверить равенство ссылок. Так, в некоторых ситуациях, когда эффективность важнее всего, вы можете использовать хеш конс (*hash cons*) некоторых классов и сопоставить их экземпляры

с помощью равенства ссылок¹. Для таких случаев в классе `AnyRef` определен дополнительный метод `eq`, который не может быть переопределен и реализован как проверка равенства ссылок (то есть для ссылочных типов ведет себя подобно `==` в Java). Существует также отрицание `eq`, которое называется `ne`, например:

```
val x = new String("abc") // abc
val y = new String("abc") // abc
x == y // true
x eq y // false
x ne y // true
```

Более подробно равенство в Scala рассматривается в главе 8.

17.3. Низшие типы

Внизу иерархии на рис. 17.1 показаны два класса: `scala.Null` и `scala.Nothing`. Это особые типы, единообразно сглаживающие острые углы объектно-ориентированной системы типов в Scala.

Класс `Null` — тип нулевой ссылки `null`: он представляет собой подкласс каждого ссылочного класса (то есть каждого класса, который сам является наследником класса `AnyRef`)². `Null` несовместим с типами значений. Нельзя, к примеру, присвоить значение `null` целочисленной переменной:

```
scala> val i: Int = null
1 |val i: Int = null
  |           ^^^^^
  |           Found:    Null
  |           Required: Int
```

Тип `Nothing` находится в самом низу иерархии классов Scala: он представляет собой подтип любого другого типа, значений которого вообще не существу-

¹ Вы можете выполнить хеш конс экземпляров класса путем кэширования всех созданных экземпляров в слабую коллекцию. Затем, как только потребуются новый экземпляр класса, сначала проверяется кэш. Если в нем уже есть элемент, равный тому, который вы намереваетесь создать, то можно повторно воспользоваться существующим экземпляром. В результате такой систематизации любые два экземпляра, равенство которых определяется с помощью метода `equals()`, также равны на основе равенства ссылок.

² В Scala 3 есть опция `-Yexplicit-nulls`, которая позволяет использовать экспериментальную альтернативную обработку значения `null`, направленную на отслеживание переменных, которые могут и не могут быть нулевыми.

ет. А зачем нужен тип без значений? Как говорилось в разделе 7.4, `Nothing` используется, в частности, для того, чтобы сигнализировать об аварийном завершении операции.

Например, в объекте `sys` стандартной библиотеки Scala есть метод `error`, имеющий такое определение:

```
def error(message: String): Nothing =  
  throw new RuntimeException(message)
```

Возвращаемым типом метода `error` является `Nothing`, что говорит пользователю о ненормальном возвращении из метода (вместо этого метод сгенерировал исключение). Поскольку `Nothing` — подтип любого другого типа, то методы, подобные `error`, допускают весьма гибкое использование, например:

```
def divide(x: Int, y: Int): Int =  
  if y != 0 then x / y  
  else sys.error("деление на ноль невозможно")
```

Ветка `then` данного условия, представленная выражением `x / y`, имеет тип `Int`, а ветка `else`, то есть вызов `error`, имеет тип `Nothing`. Поскольку `Nothing` — подтип `Int`, то типом всего условного выражения, как и требовалось, является `Int`.

17.4. Определение собственных классов значений

В разделе 17.1 говорилось, что в дополнение к встроенным классам значений можно определять собственные. Как и экземпляры встроенных, экземпляры ваших классов значений будут, как правило, компилироваться в байт-код Java, который не задействует класс-оболочку. В том контексте, где нужна оболочка, например, при использовании обобщенного кода, значения будут упаковываться и распаковываться автоматически¹.

Классами значений можно сделать только вполне определенные классы. Чтобы класс стал классом значений, он должен иметь только один параметр и не должен иметь внутри ничего, кроме `def`-определений. Более того, класс значений не может расширяться никакими другими классами и в нем не могут переопределяться методы `equals` или `hashCode`.

¹ Scala 3 также предлагает непрозрачные типы, что является некоторым ограничением, но гарантирует, что значение никогда не будет упаковано.

Чтобы определить класс значений, его нужно сделать подклассом класса `AnyVal` и поставить перед его единственным параметром префикс `val`. Пример класса значений выглядит так:

```
class Dollars(val amount: Int) extends AnyVal:
  override def toString = "$" + amount
```

В соответствии с описанием, приведенным в разделе 10.6, префикс `val` позволяет иметь доступ к параметру `amount` как к полю. Например, следующий код создает экземпляр класса значений, а затем извлекает из него `amount`:

```
val money = new Dollars(1_000_000)
money.amount // 1000000
```

В данном примере `money` ссылается на экземпляр класса значений. Эта переменная в исходном коде Scala имеет тип `Dollars`, но скомпилированный байт-код Java будет напрямую использовать тип `Int`.

В этом примере определяется метод `toString`, и компилятор понимает, когда его использовать. Именно поэтому вывод значения `money` дает результат `$1000000` со знаком доллара, а вывод `money.amount` дает результат `1000000`. Можно даже определить несколько типов значений, и все они будут опираться на одно и то же `Int`-значение, например:

```
class SwissFrancs(val amount: Int) extends AnyVal:
  override def toString = s"$amount CHF"
```

Несмотря на то что `Dollars` и `SwissFrancs` во время выполнения представлены в виде целых чисел, в процессе компиляции они становятся разными типами:

```
scala> val dollars: Dollars = new SwissFrancs(1000)
1 | val dollars: Dollars = new SwissFrancs(1000)
  | _____
  |                               Found:    SwissFrancs
  |                               Required: Dollars
```

Уход от монокультурности типов

Чтобы получить наибольшие преимущества от использования иерархии классов Scala, старайтесь для каждого понятия предметной области определять новый класс, несмотря на то что будет возможность неоднократно применять его для различных целей. Даже если он относится к так называемому *крошечному (tiny) типу*, не имеющему методов или полей, определение дополнительного класса поможет компилятору принести вам больше пользы.

Предположим, вы написали некий код для генерации HTML. В HTML название стиля представлено в виде строки. То же самое касается и идентификаторов привязки. Сам код HTML также является строкой, поэтому при желании представить все здесь перечисленное можно с помощью определения вспомогательного кода, используя строки наподобие этих:

```
def title(text: String, anchor: String, style: String): String =  
  s"<a id='$anchor'><h1 class='$style'>${text}</h1></a>"
```

В данной сигнатуре четыре строки! Такой *строчно-типизированный* код с технической точки зрения является строго типизированным. Однако все, что находится здесь в поле зрения, относится к типу `String`, поэтому компилятор не может помочь вам отличить один элемент структуры от другого. Например, не сможет уберечь вас от следующего искажения структуры:

```
scala> title("chap:vcls", "bold", "Value Classes")  
val res17: String = <a id='bold'><h1 class='Value  
Classes'>chap:vcls</h1></a>
```

Код HTML нарушен. Предполагаемый для вывода на экран текст `Value Classes` используется в качестве класса стиля, в то время как для отображаемого текста `chap.vcls` предусматривалась роль гипертекстовой ссылки. В довершение ко всему в качестве идентификатора такой ссылки выступила строка `bold`, которая, в свою очередь, должна была выполнять роль класса стиля. Несмотря на всю череду ошибок, компилятор никак этому не воспротивился.

Если определить для каждого понятия предметной области *крошечный тип*, то компилятор сможет принести больше пользы. Например, можно определить собственный небольшой класс для стилей, идентификаторов гипертекстовых ссылок, отображаемого текста и кода HTML. Поскольку эти классы имеют один параметр и не имеют элементов, то могут быть определены как классы значений:

```
class Anchor(val value: String) extends AnyVal  
class Style(val value: String) extends AnyVal  
class Text(val value: String) extends AnyVal  
class Html(val value: String) extends AnyVal
```

Наличие этих классов позволяет создать версию `title`, обладающую менее тривиальной сигнатурой типов наподобие такой:

```
def title(text: Text, anchor: Anchor, style: Style): Html =  
  Html(  
    s"<a id='${anchor.value}'>" +
```

```
s"<h1 class='${style.value}'>" +
text.value +
"</h1></a>"
)
```

Теперь при попытке воспользоваться этой версией с аргументами, указанными в неверном порядке, компилятор сможет обнаружить ошибку, например:

```
scala> title(Anchor("chap:vcls"), Style("bold"),
             Text("Value Classes"))
1 | title(new Anchor("chap:vcls"), new Style("bold"),
   | ~~~~~
   | Found:    Anchor
   | Required: Text
1 | title(Anchor("chap:vcls"), Style("bold"),
   | ~~~~~
   | Found:    Style
   | Required: Anchor
2 |     Text("Value Classes"))
   | ~~~~~
   | Found:    Text
   | Required: Style
```

17.5. Типы пересечений

Вы можете объединить два и более типа с помощью амперсанда (&), чтобы сформировать тип пересечения, например `Incrementing & Filtering` (Приращение и фильтрация). Вот пример использования классов и признаков, показанных в листингах 11.5, 11.6 и 11.9.

```
scala> val q = new BasicIntQueue with
           Incrementing with Filtering
val q: BasicIntQueue & Incrementing & Filtering = anon$....
```

Здесь `q` инициализируется экземпляром анонимного класса, который расширяет `BasicIntQueue` и смешивает `Incrementing` (Приращение) с последующей `Filtering` (Фильтрация). Его выводимый тип, `BasicIntQueue & Incrementing & Filtering`, представляет собой тип пересечения, который указывает, что объект, на который ссылается `q`, является экземпляром всех трех упомянутых типов: `BasicIntQueue`, `Incrementing` и `Filtering`.

Тип пересечения является подтипом всех комбинаций составляющих его типов. Например, тип `B & I & F` является подтипом типов `B`, `I`, `F`, `B & I`, `B & F`, `I & F` и самого себя. Более того, поскольку типы пересечения являются коммутативными, порядок появления типов в типе пересечения не имеет значения:

например, тип `I & F` эквивалентен типу `F & I`. Следовательно, `B & I & F` также является подтипом `I & B`, `F & B`, `F & I`, `B & F & I`, `F & F & I`, `F & B & I` и т. д. Вот пример, иллюстрирующий эти взаимосвязи между типами пересечений:

```
// Компилируется, так как B & I & F <: I & F
val q2: Incrementing & Filtering = q

// Компилируется, так как I & F эквивалентно F & I
val q3: Filtering & Incrementing = q2
```

17.6. Типы объединения

Scala предлагает дубликат для типов пересечения, называемых типами *объединения*, которые состоят из двух или более типов, соединенных вертикальной чертой (`|`), например `Plum | Apricot`. Тип объединения указывает, что объект является экземпляром по крайней мере одного из упомянутых типов. Например, объект типа `Plum | Apricot` является либо экземпляром `Plum`, либо экземпляром `Apricot`, либо и тем и другим¹.

Как и типы пересечения, типы объединения являются коммутативными: `Plum | Apricot` эквивалентно `Apricot | Plum`. В отличие от типов пересечения тип объединения является *супертипом* всех комбинаций входящих в него типов. Например, `Plum | Apricot` является супертипом как `Plum`, так и `Apricot`. Важно отметить, что `Plum | Apricot` является не просто супертипом `Plum` и `Apricot`, а их ближайшим общим супертипом, или *наименьшей верхней границей*.

Добавление типов объединения и пересечения в Scala 3 гарантирует, что система типов Scala образует математическую *решетку*. Решетка — это частичный порядок, в котором любые два типа имеют как уникальную наименьшую верхнюю границу, или *LUB*, так и уникальную *наибольшую нижнюю границу*. В Scala 3 наименьшей верхней границей любых двух типов является их объединение, а наибольшей нижней границей — их пересечение. Например, наименьшей верхней границей `Plum` и `Apricot` является `Plum | Apricot`. Их наибольшая нижняя граница — `Plum & Apricot`.

Типы объединения имеют серьезные последствия для спецификации и реализации вывода типов и проверки типов в Scala. В то время как в Scala 2 алгоритм вывода типов должен был основываться на приближенном значении наименьшей верхней границы некоторых пар типов, фактическая наимень-

¹ Вы можете произносить `Plum | Apricot` как `Plum` или `Apricot`.

шая верхняя граница которых была пределом бесконечной серии, то в Scala 3 можно просто сформировать объединение этих типов.

Чтобы представить себе это, рассмотрим следующую иерархию:

```
trait Fruit
trait Plum extends Fruit
trait Apricot extends Fruit
trait Pluot extends Plum, Apricot
```

Эти четыре типа образуют иерархию, показанную на рис. 17.2. `Fruit` является супертипом как для `Plum`, так и для `Apricot`, но он не является *ближайшим* общим супертипом. Скорее, тип объединения `Plum | Apricot` является ближайшим общим супертипом, или наименьшей верхней границей, для `Plum` и `Apricot`. Как показано на рис. 17.2, это означает, что тип объединения `Plum | Apricot` является подтипом `Fruit`. И это действительно так, как показано на рисунке.

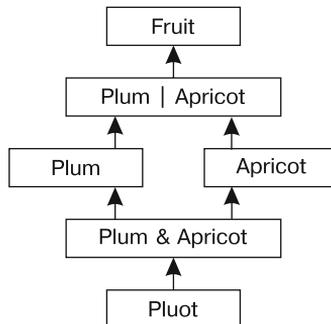


Рис. 17.2. Наименьшая верхняя и наибольшая нижняя границы

```
val plumOrApricot: Plum | Apricot = new Plum {}

// Компилируется без проблем, так как Plum | Apricot <: Fruit
val fruit: Fruit = plumOrApricot

// Нельзя использовать Fruit, так как нужен Plum | Apricot
scala> val doesNotCompile: Plum | Apricot = fruit
1 |val doesNotCompile: Plum | Apricot = fruit
  |                                     ^^^^^
  |                                     Found:    (fruit : Fruit)
  |                                     Required: Plum | Apricot
```

Двойной `Pluot` является подтипом и для `Plum`, и для `Apricot`, но он не является *ближайшим* общим подтипом. Скорее, тип пересечения `Plum & Apricot`

является ближайшим общим подтипом, или наибольшей нижней границей, для `Plum` и `Apricot`. Из представленной на рис. 17.2 схемы следует, что тип пересечения `Plum & Apricot` является *супертипом* `Pluot`. И это действительно так:

```
val pluot: Pluot = new Pluot {}

// Компилируется без проблем, так как Pluot <: Plum & Apricot
val plumAndApricot: Plum & Apricot = pluot

// Нельзя использовать Plum & Apricot, так как нужен Pluot
scala> val doesNotCompile: Pluot = plumAndApricot
1 |val doesNotCompile: Pluot = plumAndApricot
  |                               ~~~~~
  |                               Found:    (plumAndApricot : Plum & Apricot)
  |                               Required: Pluot
```

Вы можете вызвать любой метод или получить доступ к любому полю, определенному в каждом из составляющих типов типа пересечения. Например, для экземпляра `Plum & Apricot` вы можете вызывать любые методы, определенные в `Plum` или `Apricot`. В отличие от этого в типе объединения вы можете получить доступ только к тем элементам *супертипов*, которые являются *общими* для составляющих типов. Таким образом, в экземпляре `Plum | Apricot` вы можете получить доступ к членам `Fruit` (включая элементы, которые он наследует от `AnyRef` и `Any`), но вы не можете получить доступ к каким-либо элементам, добавленным в `Plum` или `Apricot`. Чтобы получить к ним доступ, вы должны выполнить сопоставление с образцом, чтобы определить реальный класс значения во время выполнения, например:

```
def errorMessage(msg: Int | String): String =
  msg match
    case n: Int => s"Error number: ${n.abs}"
    case s: String => s + "!"
```

Параметр `msg` метода `errorMessage` имеет тип `Int | String`. Поэтому вы можете напрямую вызывать в `msg` только методы, объявленные в `Any`, единственном общем супертипе `Int` и `String`. Вы не можете напрямую вызывать никакие другие методы, определенные либо в `Int`, либо в `String`. Чтобы получить доступ, например, к методу `abs` в `Int` или оператору конкатенации строк (+) в `String`, необходимо выполнить сопоставление с образцом в `msg`, как показано в теле метода `errorMessage`. Вот несколько примеров использования метода `errorMessage`:

```
errorMessage("Oops") // "Oops!"
errorMessage(-42)   // "Error number: 42"
```

17.7. Прозрачные трейты

У трейтов есть два основных применения: они позволяют определять классы с помощью композиции примешивания и определяют типы. В основном трейт используется как примешивание, а не как тип. Например, трейты `Incrementing` и `Filtering` из раздела 11.3 полезны в качестве примешиваний, однако они также имеют ограниченную ценность в качестве типов. По умолчанию можно выявить типы, определяемые этими трейтами. Например, компилятор Scala определит тип `q` в следующей инструкции как тип пересечения, в котором упоминаются и `Incrementing`, и `Filtering`:

```
scala> val q = new BasicIntQueue with
      Incrementing with Filtering
val q: BasicIntQueue & Incrementing & Filtering = anon$...
```

Вы можете указать, что не хотите, чтобы имя трейта отображалось в выводимых типах, объявив его с помощью модификатора `transparent` (прозрачный). Например, объявив `Incrementing` и `Filtering` как прозрачные следующим образом:

```
transparent trait Incrementing extends IntQueue:
  abstract override def put(x: Int) = super.put(x + 1)

transparent trait Filtering extends IntQueue:
  abstract override def put(x: Int) =
    if x >= 0 then super.put(x)
```

Теперь, когда трейты `Incrementing` и `Filtering` определены как прозрачные, их имена больше не будут отображаться в выводимых типах. Например, тип, выведенный из того же выражения создания экземпляра, показанного ранее, больше не будет упоминать `Incrementing` или `Filtering`:

```
scala> val q = new BasicIntQueue with
      Incrementing with Filtering
val q: BasicIntQueue = anon$...
```

Модификатор `transparent` влияет только на *вывод* типов. Вы все еще можете использовать прозрачные трейты в качестве типов, если напишете их в явном виде. Вот пример, в котором прозрачные трейты `Incrementing` и `Filtering` отображаются в аннотации явного типа для переменной `q`:

```
scala> val q: BasicIntQueue & Incrementing & Filtering =
      new BasicIntQueue with Incrementing with Filtering

val q: BasicIntQueue & Incrementing & Filtering = anon$...
```

Помимо трейтов, явно помеченных как прозрачные, Scala 3 будет также считать прозрачными `scala.Product`, `java.lang.Serializable` и `java.lang.Comparable`. Поскольку эти типы никогда не будут выводиться в Scala 3, когда вы захотите использовать их, вам придется делать это с помощью явных аннотаций типов или приписываний.

Резюме

В этой главе мы показали классы, находящиеся в верхней и нижней части иерархии классов Scala. Вы также увидели, как создавать свои собственные классы значений, в том числе как использовать их для «крошечных типов». Вы узнали о типах пересечений и объединений и увидели, как они превращают иерархию типов Scala в решетку. Наконец, вы узнали, как использовать модификатор `transparent`, чтобы алгоритм вывода типов Scala не использовал трейты, созданные в основном как примешивания, в качестве типов. В следующей главе вы узнаете о параметризации типов.

18

Параметризация типов

В этой главе мы рассмотрим детали параметризации типов в Scala. Попутно продемонстрируем несколько техник сокрытия информации, представленных в главе 12, на конкретном примере: проектирования класса для чисто функциональных очередей.

Параметризация типов позволяет создавать обобщенные классы и трейты. Например, множества имеют обобщенный характер и получают параметр типа: они определяются как `Set[T]`. В результате любой отдельно взятый экземпляр множества может иметь тип `Set[String]`, `Set[Int]` и т. д., но должен быть множеством *чего-либо*. В отличие от языка Java, в котором разрешено использовать «сырые» типы (raw types), Scala требует указывать параметры типа. Вариантность определяет взаимоотношения наследования параметризованных типов, к примеру, таких, при которых `Set[String]` является подтипом `Set[AnyRef]`.

Глава состоит из трех частей. В первой разрабатывается структура данных для чисто функциональных очередей. Во второй разрабатываются технологические приемы сокрытия внутреннего представления деталей этой структуры. В третьей объясняется вариантность параметров типов и то, как она взаимодействует со сокрытием информации.

18.1. Функциональные очереди

Функциональная очередь представляет собой структуру данных с тремя операциями:

- `head` — возвращает первый элемент очереди;
- `tail` — возвращает очередь без первого элемента;

- `enqueue` — возвращает новую очередь с заданным элементом, добавленным в ее конец.

В отличие от изменяемой функциональная очередь не изменяет свое содержимое при добавлении элемента. Вместо этого возвращается новая очередь, содержащая элемент. В данной главе наша цель — создать класс по имени `Queue`, работающий так:

```
val q = Queue(1, 2, 3) // Queue(1, 2, 3)
val q1 = q.enqueue(4) // Queue(1, 2, 3, 4)
q                    // Queue(1, 2, 3)
```

Будь у `Queue` изменяемая реализация, операция `enqueue` в показанной ранее второй строке ввода повлияла бы на содержимое `q`: по сути, после этой операции оба результата, и `q1`, и исходная очередь `q`, будут содержать последовательность 1, 2, 3, 4. А для функциональной очереди добавленное значение обнаруживается только в результате `q1`, но не в очереди `q`, в отношении которой выполнялась операция.

Кроме того, чистые функциональные очереди слегка похожи на списки. Обе эти коллекции имеют так называемую *абсолютно постоянную* структуру данных, где старые версии остаются доступными даже после расширений или модификаций. Обе они поддерживают операции `head` и `tail`. Но список растет, как правило, с начала с помощью операции `::`, а очередь — с конца с помощью операции `enqueue`.

Как добиться эффективной реализации очереди? В идеале функциональная (неизменяемая) очередь не должна иметь высоких издержек, существенно больших по сравнению с императивной (изменяемой) очередью. То есть все три операции: `head`, `tail` и `enqueue` — должны выполняться за постоянное время.

Одним из простых подходов к реализации функциональной очереди станет использование списка в качестве типа представления. Тогда `head` и `tail` — просто аналогичные операции над списком, а `enqueue` — конкатенация.

В таком варианте получится следующая реализация:

```
class SlowAppendQueue[T](elems: List[T]): // Неэффективное решение
  def head = elems.head
  def tail = new SlowAppendQueue(elems.tail)
  def enqueue(x: T) = SlowAppendQueue(elems :: List(x))
```

Проблемной в данной реализации является операция `enqueue`. На нее уходит время, пропорциональное количеству элементов, хранящихся в очереди. Если требуется постоянное время добавления, то можно также попробовать

изменить в списке, представляющем очередь, порядок следования элементов на обратный, чтобы последний добавляемый элемент стал в списке первым. Тогда получится такая реализация:

```
class SlowHeadQueue[T](smele: List[T]): // Неэффективное решение
  // smele – это реверсированный elems
  def head = smele.last
  def tail = new SlowHeadQueue(smele.init)
  def enqueue(x: T) = SlowHeadQueue(x :: smele)
```

Теперь у операции `enqueue` постоянное время выполнения, а вот у `head` и `tail` — нет. На их выполнение теперь уходит время, пропорциональное количеству элементов, хранящихся в очереди.

При изучении этих двух примеров реализация, в которой на все три операции будет затрачиваться постоянное время, не представляется такой уж простой. И действительно, возникают серьезные сомнения в возможности подобной реализации! Но, воспользовавшись сочетанием двух операций, можно подойти к желаемому результату очень близко. Замысел состоит в представлении очереди в виде двух списков: `leading` и `trailing`. Список `leading` содержит элементы, которые располагаются от конца к началу, а элементы списка `trailing` следуют из начала в конец очереди, то есть в обратном порядке. Содержимое всей очереди в любой момент времени равно коду `leading :: trailing.reverse`.

Теперь, чтобы добавить элемент, следует просто провести конс-операцию в отношении списка `trailing`, воспользовавшись оператором `::`, и тогда операция `enqueue` будет выполняться за постоянное время. Это значит, если изначально пустая очередь выстраивается на основе последовательно проведенных операций `enqueue`, то список `trailing` будет расти, а список `leading` останется пустым. Затем перед выполнением первой операции `head` или `tail` в отношении пустого списка `leading` весь список `trailing` копируется в `leading` в обратном порядке следования элементов. Это делается с помощью операции по имени `mirror`. Реализация очередей с использованием данного подхода показана в листинге 18.1.

Листинг 18.1. Базовая функциональная очередь

```
class Queue[T](
  private val leading: List[T],
  private val trailing: List[T]
):
  private def mirror =
    if leading.isEmpty then
      new Queue(trailing.reverse, Nil)
```

```
    else
      this

def head = mirror.leading.head

def tail =
  val q = mirror
  new Queue(q.leading.tail, q.trailing)

def enqueue(x: T) =
  new Queue(leading, x :: trailing)
```

Какова вычислительная сложность этой реализации очереди? Операция `mirror` может занять время, пропорциональное количеству элементов очереди, но только при условии, что список `leading` пуст. Если же нет, то возврат из метода происходит немедленно. Поскольку `head` и `tail` вызывают `mirror`, то их вычислительная сложность также может иметь линейную зависимость от размера очереди. Но чем длиннее становится очередь, тем реже вызывается `mirror`.

И действительно, допустим, есть очередь длиной n с пустым списком `leading`. Тогда операции `mirror` придется скопировать в обратном порядке список длиной n . Однако следующий раз, когда операции `mirror` придется делать что-либо, наступит только по опустошению списка `leading`, что произойдет после n операций `tail`. То есть вы можете расплатиться за каждую из этих n операций `tail` одной n -ной от вычислительной сложности операции `mirror`, что означает постоянный объем работы. При условии, что операции `head`, `tail` и `enqueue` используются примерно с одинаковой частотой, *амортизированная* вычислительная сложность является, таким образом, константой для каждой операции. Следовательно, функциональные очереди асимптотически так же эффективны, как и изменяемые очереди.

Но этот аргумент следует сопроводить некоторыми оговорками. Во-первых, речь шла только об асимптотическом поведении, а постоянно действующие факторы могут несколько различаться. Во-вторых, аргумент основывается на том, что операции `head`, `tail` и `enqueue` вызываются с примерно одинаковой частотой. Если операция `head` вызывается намного чаще, чем остальные две, то данный аргумент утратит силу, поскольку каждый вызов `head` может повлечь за собой весьма затратную реорганизацию списка с помощью операции `mirror`. Негативных последствий, названных во второй оговорке, можно избежать, поскольку есть возможность сконструировать функциональные очереди таким образом, что в последовательности операций `head` реорганизация потребуется только для первой из них. Как это делается, мы покажем в конце главы.

18.2. Скрытие информации

Теперь по эффективности реализация класса `Queue`, показанная в листинге 18.1, нас вполне устраивает. Конечно, вы можете возразить, что за эту эффективность пришлось расплачиваться неоправданно подробной детализацией. Глобально доступный конструктор `Queue` получает в качестве параметров два списка, в одном из которых элементы следуют в обратном порядке, что вряд ли совпадает с интуитивным представлением об очереди. Нам нужен способ, позволяющий скрыть этот конструктор от кода клиента. Некоторые способы решения данной задачи на Scala мы покажем в текущем разделе.

Приватные конструкторы и фабричные методы

В Java конструктор можно скрыть, объявив его приватным. В Scala первичный конструктор не имеет явно указываемого определения — подразумевается, что он автоматически определяется с параметрами и телом класса. Тем не менее, как показано в листинге 18.2, скрыть первичный конструктор можно, добавив перед списком параметров класса модификатор `private`.

Листинг 18.2. Скрытие первичного конструктора путем превращения его в приватный

```
class Queue[T] private (
  private val leading: List[T],
  private val trailing: List[T]
)
```

Модификатор `private`, указанный между именем класса и его параметрами, служит признаком того, что конструктор класса `Queue` является приватным: доступ к нему можно получить только изнутри самого класса и из его объекта-компаньона. Имя класса `Queue` по-прежнему остается публичным, поэтому вы можете использовать его в качестве типа, но не можете вызвать его конструктор:

```
scala> Queue(List(1, 2), List(3))
1 | Queue(List(1, 2), List(3))
   | ^^^^^
   | constructor Queue cannot be accessed as a member of
   | Queue from module class rs$line$4$.
```

Теперь, когда первичный конструктор класса `Queue` уже не может быть вызван из кода клиента, нужен какой-то другой способ создания новых очередей. Одна из возможностей — добавить вспомогательный конструктор:

```
def this() = this( Nil, Nil)
```

Этот конструктор, показанный в предыдущем примере, создает пустую очередь. В качестве уточнения он может получать список исходных элементов очереди:

```
def this(elems: T*) = this(elems.toList, Nil)
```

Следует напомнить, что в соответствии с описанием из раздела 8.8 T^* — форма записи для повторяющихся параметров.

Еще одна возможность состоит в добавлении фабричного метода, создающего очередь из последовательности исходных элементов. Прямой путь сделать это — определить объект `Queue`, который имеет такое же имя, как и определяемый класс, и, как показано в листинге 18.3, содержит метод `apply`.

Листинг 18.3. Фабричный метод `apply` в объекте-компаньоне

```
object Queue:
  // создает очередь с исходными элементами xs
  def apply[T](xs: T*) = new Queue[T](xs.toList, Nil)
```

Помещая этот объект в тот же самый исходный файл, в котором находится определение класса `Queue`, вы превращаете объект в объект-компаньон этого класса. В разделе 12.5 было показано: объект-компаньон имеет такие же права доступа, что и его класс. Поэтому метод `apply` в объекте `Queue` может создать новый объект `Queue`, несмотря на то что конструктор класса `Queue` является приватным.

Обратите внимание: по причине вызова фабричным методом метода `apply` клиенты могут создавать очереди с помощью выражений вида `Queue(1, 2, 3)`. Данное выражение разворачивается в `Queue.apply(1, 2, 3)`, поскольку `Queue` — объект, заменяющий функцию. В результате этого клиенты видят `Queue` в качестве глобально определенного фабричного метода. На самом же деле в Scala нет методов с глобальной областью видимости, поскольку каждый метод должен быть заключен в объект, класс или пакет. Но, используя методы по имени `apply` внутри глобальных объектов, вы можете поддерживать схемы, похожие на вызов глобальных методов.

Альтернативный вариант: приватные классы

Приватные конструкторы и приватные члены класса — всего лишь один из способов скрыть инициализацию и представить класс. Еще один более радикальный способ — сокрытие самого класса и экспорт только трейта, показывающего публичный интерфейс класса. Код реализации такой конструкции

представлен в листинге 18.4. В нем показан трейт `Queue`, в котором объявляются методы `head`, `tail` и `enqueue`. Все три метода реализованы в подклассе `QueueImpl`, который является приватным подклассом внутри класса объекта `Queue`. Тем самым клиентам открывается доступ к той же информации, что и раньше, но с помощью другого приема. Вместо сокрытия отдельно взятых конструкторов и методов в этой версии скрывается весь реализующий очередь класс.

Листинг 18.4. Абстракция типа для функциональных очередей

```
trait Queue[T]:
  def head: T
  def tail: Queue[T]
  def enqueue(x: T): Queue[T]

object Queue:

  def apply[T](xs: T*): Queue[T] =
    QueueImpl[T](xs.toList, Nil)

  private class QueueImpl[T](
    private val leading: List[T],
    private val trailing: List[T]
  ) extends Queue[T]:

    def mirror =
      if leading.isEmpty then
        QueueImpl(trailing.reverse, Nil)
      else
        this

    def head: T = mirror.leading.head

    def tail: QueueImpl[T] =
      val q = mirror
      QueueImpl(q.leading.tail, q.trailing)

    def enqueue(x: T) =
      QueueImpl(leading, x :: trailing)
```

18.3. Аннотации вариантности

Элемент `Queue` согласно определению в листинге 18.4 — трейт, но не тип, поскольку получает параметр типа¹. В результате вы не можете создавать переменные типа `Queue`:

¹ `Queue` можно считать типом более высокого рода.

```
scala> def doesNotCompile(q: Queue) = {}
1 | def doesNotCompile(q: Queue) = {}
  |                               ^^^^^^
  |                               Missing type parameter for Queue
```

Вместо этого `Queue` позволяет указывать *параметризованные* типы, такие как `Queue[String]`, `Queue[Int]` или `Queue[AnyRef]`:

```
scala> def doesCompile(q: Queue[AnyRef]) = {}
def doesCompile: (q: Queue[AnyRef]): Unit
```

Таким образом, `Queue` — трейт, а `Queue[String]` — тип. `Queue` также называют *конструктором типа*, поскольку вы можете сконструировать тип с его участием, указав параметр типа. (Это аналогично конструированию экземпляра объекта с использованием самого обычного конструктора с указанием параметра значения.) Конструктор типа `Queue` генерирует семейство типов, включающее `Queue[Int]`, `Queue[String]` и `Queue[AnyRef]`.

Можно также сказать, что `Queue` — *обобщенный* трейт. (Классы и трейты, которые получают параметры типа, являются обобщенными, а вот типы, генерируемые ими, являются параметризованными, а не обобщенными.) Понятие «обобщенный» означает, что вы определяете множество конкретных типов, используя один обобщенно написанный класс или трейт. Например, трейт `Queue` в листинге 18.4 определяет обобщенную очередь. Конкретными очередями будут `Queue[Int]`, `Queue[String]` и т. д.

Сочетание параметров типа и системы подтипов вызывает ряд интересных вопросов. Например, существуют ли какие-то особые подтиповые отношения между членами семейства типов, генерируемого `Queue[T]`? Конкретнее говоря, следует ли рассматривать `Queue[String]` как подтип `Queue[AnyRef]`? Или в более широком смысле: если `S` — подтип `T`, то следует ли рассматривать `Queue[S]` как подтип `Queue[T]`? Если да, то можно сказать, что трейт `Queue` *ковариантный* (или гибкий) в своем параметре типа `T`. Или же, поскольку у него всего один параметр типа, можно просто сказать, что `Queue`-очереди ковариантны. Такая ковариантность `Queue` будет означать, к примеру, что вы можете передать `Queue[String]` ранее показанному методу `doesCompile`, который принимает параметр значения типа `Queue[AnyRef]`.

Интуитивно все это выглядит хорошо, поскольку очередь из `String`-элементов похожа на частный случай очереди из `AnyRef`-элементов. Но в Scala у обобщенных типов изначально имеется *нонвариантная* (или жесткая) подтипизация. То есть при использовании трейта `Queue`, определенного в показанном выше листинге 18.4, очереди с различными типами элементов никогда не будут состоять в подтиповых отношениях. `Queue[String]` не будет использоваться

как `Queue[AnyRef]`. Но получить ковариантность (гибкость) подтипизации очередей можно следующим изменением первой строчки трейта `Queue`:

```
trait Queue[+T] { ... }
```

Указанный знак плюс (+) в качестве префикса формального параметра типа показывает, что подтипизация в этом параметре ковариантна (гибка). Добавляя этот единственный знак, вы сообщаете Scala о необходимости, к примеру, рассматривать тип `Queue[String]` как подтип `Queue[AnyRef]`. Компилятор проверит факт определения `Queue` в соответствии со способом, предполагаемым подобной подтипизацией.

Помимо префикса +, существует префикс -, который показывает *контравариантность* подтипизации. Если определение `Queue` имеет вид

```
trait Queue[-T] { ... }
```

и если тип `T` — подтип типа `S`, то это будет означать, что `Queue[S]` — подтип `Queue[T]` (что в случае с очередями было бы довольно неожиданно!). Ковариантность, контравариантность или невариантность параметра типа называются *вариантностью* параметров. Знаки + и -, которые могут размещаться рядом с параметрами типа, называются *аннотациями вариантности*.

В чисто функциональном мире многие типы по своей природе ковариантны (гибки). Но ситуация становится иной, как только появляются изменяемые данные. Чтобы выяснить, почему так происходит, рассмотрим показанный в листинге 18.5 простой тип одноэлементных ячеек, допускающих чтение и запись.

Листинг 18.5. Нонвариантный (жесткий) класс `Cell`

```
class Cell[T](init: T):
  private var current = init
  def get = current
  def set(x: T) =
    current = x
```

Показанный здесь тип `Cell` объявлен невариантным (жестким). Ради аргументации представим на время, что `Cell` был объявлен ковариантным, то есть был объявлен класс `Cell[+T]`, и этот код передан компилятору Scala. (Этого делать не стоит, и скоро мы объясним почему.) Значит, можно сконструировать следующую проблематичную последовательность инструкций:

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get
```

Если рассмотреть строки по отдельности, то все они выглядят вполне нормально. В первой строке создается строковая ячейка, которая сохраняется в `val`-переменной по имени `c1`. Во второй строке определяется новая `val`-переменная `c2`, имеющая тип `Cell[Any]`, которая инициализируется значением переменной `c1`. Кажется, все в порядке, поскольку экземпляры класса `Cell` считаются ковариантными. В третьей строке для `c2` устанавливается значение `1`. С этим тоже все в порядке, так как присваиваемое значение `1` — экземпляр, относящийся к объявленному для `c2` типу элемента `Any`. И наконец, в последней строке значение элемента `c1` присваивается строковой переменной. Здесь нет ничего странного, поскольку с обеих сторон выражения находятся значения одного и того же типа. Но если взять все в совокупности, то эти четыре строки в конечном счете присваивают целочисленное значение `1` строковому значению `s`. Это явное нарушение целостности типа.

Какая из операций вызывает сбой в ходе выполнения кода? Видимо, вторая, в которой используется ковариантная подтипизация. Все другие операции слишком простые и базовые. Стало быть, ячейка `Cell`, хранящая значение типа `String`, не является также ячейкой `Cell`, хранящей значение типа `Any`, поскольку есть вещи, которые можно делать с `Cell` из `Any`, но нельзя делать с `Cell` из `String`. К примеру, в отношении `Cell` из `String` нельзя использовать `set` с `Int`-аргументом.

Получается, если передать ковариантную версию `Cell` компилятору Scala, то будет выдана ошибка компиляции:

```
4 | def set(x: T) =
  |           ^^^^
  | covariant type T occurs in contravariant position
  | in type T of value x
```

Вариантность и массивы

Интересно сравнить данное поведение с массивами в Java. В принципе, массивы подобны ячейкам, за исключением того, что могут иметь несколько элементов. При этом массивы в Java считаются ковариантными.

Пример, аналогичный работе с ячейкой, можно проверить в работе с массивами Java:

```
// Это код Java
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0];
```

Если запустить данный пример, то окажется, что он пройдет компиляцию. Но в ходе выполнения, когда элементу `a2[0]` будет присваиваться значение `Integer`, программа сгенерирует исключение `ArrayStore`:

```
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
    at JavaArrays.main(JavaArrays.java:8)
```

Дело в том, что в Java сохраняется тип элемента массива во время выполнения программы. При каждом обновлении элемента массива новое значение элемента проверяется на принадлежность к сохраненному типу. Если оно не является экземпляром этого типа, то выдается исключение `ArrayStore`.

Может возникнуть вопрос: а почему в Java принята такая на вид небезопасная и затратная конструкция? Отвечая на данный вопрос, Джеймс Гослинг (James Gosling), основной изобретатель языка Java, говорил, что создатели хотели получить простые средства обобщенной обработки массивов. Например, им хотелось получить возможность писать метод сортировки всех элементов массива с использованием следующей сигнатуры, которая получает массив из `Object`-элементов:

```
void sort(Object[] a, Comparator cmp) { ... }
```

Ковариантность массивов требовалась для того, чтобы этому методу сортировки можно было передавать массивы произвольных ссылочных типов. Разумеется, после появления в Java обобщенных типов (дженериков) подобный метод сортировки уже мог быть написан с параметрами типа, поэтому необходимость в ковариантности массивов отпала. Сохранение ее до наших дней обусловлено соображениями совместимости.

Scala старается быть чище, чем Java, и не считает массивы ковариантными. Вот что получится, если перевести первые две строки кода примера с массивом на язык Scala:

```
scala> val a1 = Array("abc")
val a1: Array[String] = Array(abc)

scala> val a2: Array[Any] = a1
1 |val a2: Array[Any] = a1
  |                   ^^
  |                   Found:   (a1 : Array[String])
  |                   Required: Array[Any]
```

В данном случае получилось, что Scala считает массивы нековариантными (жесткими), следовательно, `Array[String]` не считается соответствующим

`Array[Any]`. Но иногда необходимо организовать взаимодействие между имеющимися в Java устаревшими методами, которые используют в качестве средства эмуляции обобщенного массива `Object`-массив. Например, может возникнуть потребность вызвать метод сортировки наподобие того, который был рассмотрен ранее в отношении `String`-массива, передаваемого в качестве аргумента. Чтобы допустить такую возможность, Scala позволяет выполнять приведение массива из элементов типа `T` к массиву элементов любого из супертипов `T`:

```
val a2: Array[Object] = a1.asInstanceOf[Array[Object]]
```

Приведение типов во время компиляции не вызывает никаких возражений и всегда будет успешно работать во время выполнения программы, поскольку положенная в основу работы JVM-модель времени выполнения рассматривает массивы в качестве ковариантных точно так же, как это делается в языке Java. Но впоследствии, как и на Java, может быть получено исключение `ArrayStore`.

18.4. Проверка аннотаций вариантности

Рассмотрев несколько примеров ненадежности вариантности, можно задать вопрос: какие именно определения классов следует отвергать, а какие — принимать? До сих пор во всех нарушениях целостности типов фигурировали переназначаемые поля или элементы массивов. В то же время чисто функциональная реализация очередей представляется хорошим кандидатом на ковариантность. Но в следующем примере показано, что ситуацию ненадежности можно подстроить даже при отсутствии переназначаемого поля.

Чтобы выстроить пример, предположим: очереди из показанного выше листинга 18.4 определены как ковариантные. Затем создадим подкласс очередей с указанным типом `Int` и переопределим метод `enqueue`:

```
class StrangeIntQueue extends Queue[Int]:
  override def enqueue(x: Int) =
    println(math.sqrt(x))
    super.enqueue(x)
```

Перед выполнением добавления метод `enqueue` в подклассе `StrangeIntQueue` выводит на стандартное устройство квадратный корень из своего (целочисленного) аргумента.

Теперь можно создать контрпример из двух строк кода:

```
val x: Queue[Any] = new StrangeIntQueue
x.enqueue("abc")
```

Первая из этих двух строк вполне допустима, поскольку `StrangeIntQueue` — подкласс `Queue[Int]`, и, если предполагается ковариантность очередей, то `Queue[Int]` является подтипом `Queue[Any]`. Вполне допустима и вторая строка, так как `String`-значение можно добавлять в `Queue[Any]`. Но если взять их вместе, то у этих двух строк проявляется не имеющий никакого смысла эффект применения метода извлечения квадратного корня к строке.

Это явно не те простые изменяемые поля, которые делают ковариантные поля ненадежными. Проблема носит более общий характер. Получается, как только обобщенный параметр типа появляется в качестве типа параметра метода, содержащие данный метод класс или трейт в этом параметре типа могут не быть ковариантными.

Для очередей метод `enqueue` нарушает это условие:

```
class Queue[+T]:
  def enqueue(x: T) =
    ...
```

При запуске модифицированного класса очередей, подобного показанному ранее, в компиляторе Scala последний выдаст следующий результат:

```
17 | def enqueue(x: T) =
    |           ^^^^
    | covariant type T occurs in contravariant position
    | in type T of value x
```

Переназначаемые поля — частный случай правила, которое не позволяет параметрам типа, имеющим аннотацию `+`, использоваться в качестве типов параметра метода. Как упоминалось в разделе 16.2, переназначаемое поле `var x: T` рассматривается в Scala как геттер `def x: T` и как сеттер `def x_=(y: T)`. Как видите, сеттер имеет параметр поля типа `T`. Следовательно, этот тип не может быть ковариантным.

УСКОРЕННЫЙ РЕЖИМ ЧТЕНИЯ

Далее в этом разделе мы рассмотрим механизм, с помощью которого компилятор Scala проверяет аннотацию вариантности. Если данные подробности вас пока не интересуют, то можете смело переходить к разделу 18.5. Следует усвоить главное: компилятор Scala будет проверять любую аннотацию вариантности, которую вы укажете в отношении параметров типа. Например, при попытке объявить ковариантный параметр типа (путем добавления знака `+`), способного вызвать потенциальные ошибки в ходе выполнения программы, программа откомпилирована не будет.

Чтобы проверить правильность аннотаций вариантности, компилятор Scala классифицирует все позиции в теле класса или трейта как *положительные*, *отрицательные* или *нейтральные*. Под позицией понимается любое место в теле класса или трейта (далее в тексте будет фигурировать только термин «класс»), где может использоваться параметр типа. Например, позицией является каждый параметр значения в методе, поскольку у параметра значения метода есть тип. Следовательно, в этой позиции может применяться параметр типа.

Компилятор проверяет каждое использование каждого из имеющихся в классе параметров типа. Параметры типа, для аннотации которых применяется знак +, могут быть задействованы только в положительных позициях, а параметры, для аннотации которых используется знак -, могут применяться лишь в отрицательных. Параметр типа, не имеющий аннотации вариантности, может использоваться в любой позиции. Таким образом, он является единственной разновидностью параметров типа, которая применима в нейтральных позициях тела класса.

В целях классификации позиций компилятор начинает работу с объявления параметра типа, а затем идет через более глубокие уровни вложенности. Позиции на верхнем уровне объявляемого класса классифицируются как положительные. По умолчанию позиции на более глубоких уровнях вложенности классифицируются таким же образом, как и охватывающие их уровни, но есть небольшое количество исключений, в которых классификация меняется. Позиции параметров значений метода классифицируются по *перевернутой* схеме относительно позиций за пределами метода, там положительная классификация становится отрицательной, отрицательная — положительной, а нейтральная классификация так и остается нейтральной.

Текущая классификация действует по перевернутой схеме в отношении не только позиций параметров значений методов, но и параметров типа методов. В зависимости от вариантности соответствующего параметра типа классификация иногда бывает перевернута в имеющейся в типе позиции аргумента типа, например, это касается `Arg` в `C[Arg]`. Если параметр типа у `C` аннотирован с помощью знака +, то классификация остается такой же. Если с помощью знака -, то классификация определяется по перевернутой схеме. При отсутствии у параметра типа у `C` аннотации вариантности текущая классификация изменяется на нейтральную.

В качестве слегка надуманного примера рассмотрим следующее определение класса, в котором несколько позиций аннотированы согласно их классификации с помощью обозначений $\wedge+$ (для положительных) или $\wedge-$ (для отрицательных):

```
abstract class Cat[-T, +U]:  
  def meow[W-](volume: T-, listener: Cat[U+, T-]-)  
    : Cat[Cat[U+, T-]-, U+]+
```

Позиции параметра типа `W` и двух параметров значений, `volume` и `listener`, помечены как отрицательные. Если посмотреть на результирующий тип метода `meow`, то позиция первого аргумента, `Cat[U, T]`, помечена как отрицательная, поскольку первый параметр типа у `Cat`, `T`, аннотирован с помощью знака `-`. Тип `U` внутри этого аргумента опять имеет положительную позицию (после двух перевертываний), а тип `T` внутри этого аргумента остается в отрицательной.

Из всего вышесказанного можно сделать вывод, что отследить позиции вариантностей очень трудно. Поэтому помощь компилятора Scala, прорабатывающего эту работу за вас, несомненно, приветствуется.

После вычисления классификации компилятор проверяет, используется ли каждый параметр типа только в позициях, которые имеют соответствующую классификацию. В данном случае `T` используется лишь в отрицательных позициях, а `U` — лишь в положительных. Следовательно, класс `Cat` типизирован корректно.

18.5. Нижние ограничители

Вернемся к классу `Queue`. Вы видели, что прежнее определение `Queue[T]`, показанное выше в листинге 18.4, не может быть превращено в ковариантное в отношении `T`, поскольку `T` фигурирует в качестве типа параметра метода `enqueue` и находится в отрицательной позиции.

К счастью, существует способ выйти из этого положения: можно обобщить `enqueue`, превратив данный метод в полиморфный (то есть предоставить самому методу `enqueue` параметр типа), и воспользоваться *нижним ограничителем* его параметра типа. Новая формулировка `Queue` с реализацией этой идеи показана в листинге 18.6.

Листинг 18.6. Параметр типа с нижним ограничителем

```
class Queue[+T] (private val leading: List[T],  
  private val trailing: List[T]):  
  def enqueue[U >: T](x: U) =  
    new Queue[U](leading, x :: trailing) // ...
```

В новом определении `enqueue` дается параметр типа `U`, и с помощью синтаксиса `U >: T` тип `T` определяется как нижний ограничитель для `U`. В результате

от типа `U` требуется, чтобы он был супертипом для `T`¹. Теперь параметр для `enqueue` имеет тип `U`, а не `T`, а возвращаемое значение метода теперь не `Queue[T]`, а `Queue[U]`.

Предположим, есть класс `Fruit`, имеющий два подкласса: `Apple` и `Orange`. С новым определением класса `Queue` появилась возможность добавить `Orange` в `Queue[Apple]`. Результатом будет `Queue[Fruit]`.

В этом пересмотренном определении `enqueue` типы используются правильно. Интуитивно понятно, что если `T` — более конкретный тип, чем ожидалось (например, `Apple` вместо `Fruit`), то вызов `enqueue` все равно будет работать, поскольку `U (Fruit)` по-прежнему будет супертипом для `T (Apple)`².

Возможно, новое определение `enqueue` лучше старого, поскольку имеет более обобщенный характер. В отличие от старой версии новое определение позволяет добавлять в очередь с элементами типа `T` элементы произвольного супертипа `U`. Результат получается типа `Queue[U]`. Наряду с ковариантностью очереди это позволяет получить правильную разновидность гибкости для моделирования очередей из различных типов элементов вполне естественным образом.

Это показывает, что в совокупности аннотации вариантности и нижние ограничители — хорошо сыгранная команда. Они являются хорошим примером *разработки, управляемой типами*, где типы интерфейса управляют ее детальным проектированием и реализацией. В случае с очередями высока вероятность того, что вы не стали бы продумывать улучшенную реализацию `enqueue` с нижним ограничителем. Но у вас могло созреть решение сделать очереди ковариантными, в случае чего компилятор указал бы для `enqueue` на ошибку вариантности. Исправление ошибки вариантности путем добавления нижнего ограничителя придает `enqueue` большую обобщенность, а очереди в целом делает более полезными.

Вдобавок это наблюдение — главная причина того, почему в Scala предпочитаема вариантность по месту объявления (*declaration-site variance*), а не вариантность по месту использования (*use-site variance*), встречающаяся в Java в подстановочных символах (*wildcards*). В случае вариантности по месту

¹ Отношения супертипов и подтипов рефлексивны. Это значит, что тип является одновременно супертипом и подтипом по отношению к себе. Даже притом что `T` — нижняя граница для `U`, `T` все же можно передавать методу `enqueue`.

² С технической точки зрения произошедшее — переворот для нижних границ. Параметр типа `U` находится в отрицательной позиции (один переворот), а нижняя граница (`>`: `T`) — в положительной (два переворота).

использования вы разрабатываете класс самостоятельно. А вот клиентам данного класса придется вставлять подстановочные символы, и если они сделают это неправильно, то применить некоторые важные методы экземпляра станет невозможно. Вариантность — дело непростое, пользователи зачастую понимают ее неправильно и избегают ее, полагая, что подстановочные символы и дженерики для них слишком сложны. При использовании вариантности по месту объявления ваши намерения выражаются для компилятора, который выполнит двойную проверку, чтобы убедиться, что метод, который вам нужно сделать доступным, будет действительно доступен.

18.6. Контравариантность

До сих пор во всех представленных в данной главе примерах встречалась либо ковариантность, либо нековариантность. Но бывают такие обстоятельства, при которых вполне естественно выглядит и контравариантность. Рассмотрим, к примеру, трейт каналов вывода, показанный в листинге 18.7.

Листинг 18.7. Контравариантный канал вывода

```
trait OutputChannel[-T]:  
  def write(x: T): Unit
```

Здесь трейт `OutputChannel` определен с контравариантностью, указанной для `T`. Следовательно, получается, что канал вывода для `AnyRef` является подтипом канала вывода для `String`. Хотя на интуитивном уровне это может показаться непонятным, в действительности здесь есть определенный смысл. Понять, почему так, можно, рассмотрев возможные действия с `OutputChannel[String]`. Единственная поддерживаемая операция — запись в него значения типа `String`. Аналогичная операция может быть выполнена также в отношении `OutputChannel[AnyRef]`. Следовательно, вполне безопасно будет вместо `OutputChannel[String]` подставить `OutputChannel[AnyRef]`. В отличие от этого подставить `OutputChannel[String]` туда, где требуется `OutputChannel[AnyRef]`, будет небезопасно. В конце концов, на `OutputChannel[AnyRef]` можно отправить любой объект, а `OutputChannel[String]` требует, чтобы все записываемые значения были строками.

Эти рассуждения указывают на общий принцип разработки систем типов: вполне безопасно предположить, что тип `T` — подтип типа `U`, если значение типа `T` можно подставить там, где требуется значение типа `U`. Это называется *принципом подстановки Лисков*. Он соблюдается, если `T` поддерживает те же операции, что и `U`, и все принадлежащие `T` операции требуют

меньшего, а предоставляют большее, чем соответствующие операции в U. В случае с каналами вывода `OutputChannel[AnyRef]` может быть подтипом `OutputChannel[String]`, поскольку в обоих типах поддерживается одна и та же операция `write` и она требует меньшего в `OutputChannel[AnyRef]`, чем в `OutputChannel[String]`. Меньшее означает следующее: от аргумента в первом случае требуется только, чтобы он был типа `AnyRef`, а вот во втором случае от него требуется, чтобы он был типа `String`.

Иногда в одном и том же типе смешиваются ковариантность и контравариантность. Известный пример — функциональные трейты Scala. Например, при написании функционального типа `A => B` Scala разворачивает этот код, приводя его к виду `Function1[A, B]`. Определение `Function1` в стандартной библиотеке использует как ковариантность, так и контравариантность: в листинге 18.8 показано, что трейт `Function1` контравариантен в аргументе функции типа `S` и ковариантен в результирующем типе `T`. Принцип подстановки Лисков здесь не нарушается, поскольку аргументы — это то, что требуется, а вот результаты — то, что предоставляется.

Листинг 18.8. Ковариантность и контравариантность `Function1`

```
trait Function1[-S, +T]:
  def apply(x: S): T
```

Рассмотрим в качестве примера приложение, показанное в листинге 18.9. Здесь класс `Publication` содержит одно параметрическое поле `title` типа `String`. Класс `Book` расширяет `Publication` и пересылает свой строковый параметр `title` конструктору своего суперкласса. В объекте-одиночке `Library` определяются набор книг `books` и метод `printBookList`, получающий функцию `info`, у которой есть тип `Book => AnyRef`. Иными словами, типом единственного параметра `printBookList` является функция, которая получает один аргумент типа `Book` и возвращает значение типа `AnyRef`. В приложении `Customer` определяется метод `getTitle`, получающий в качестве единственного своего параметра значение типа `Publication` и возвращающий значение типа `String`, которое содержит название переданной публикации `Publication`.

Листинг 18.9. Демонстрация вариантности параметра типа функции

```
class Publication(val title: String)
class Book(title: String) extends Publication(title)

object Library:
  val books: Set[Book] =
    Set(
      Book("Programming in Scala"),
```

```
    Book("Walden")
  )
  def printBookList(info: Book => AnyRef) =
    for book <- books do println(info(book))

object Customer:
  def getTitle(p: Publication): String = p.title
  def main(args: Array[String]): Unit =
    Library.printBookList(getTitle)
```

Теперь посмотрим на последнюю строку в объекте `Customer`. В ней вызывается принадлежащий `Library` метод `printBookList`, которому в инкапсулированном в значение функции виде передается `getTitle`:

```
Library.printBookList(getTitle)
```

Эта строка кода проходит проверку на соответствие типу даже притом, что `String`, результирующий тип выполнения функции, является подтипом `AnyRef`, типом результата параметра `info` метода `printBookList`. Данный код проходит компиляцию, поскольку результирующие типы функций объявлены ковариантными (+T в листинге 18.8). Если заглянуть в тело `printBookList`, то можно получить представление о том, почему в этом есть определенный смысл.

Метод `printBookList` последовательно перебирает элементы своего списка книг и вызывает переданную ему функцию в отношении каждой книги. Он передает `AnyRef`-результат, возвращенный `info`, методу `println`, который вызывает в отношении этого результата метод `toString` и выводит на стандартное устройство возвращенную им строку. Данный процесс будет работать со `String`-значениями, а также с любыми другими подклассами `AnyRef`, в чем, собственно, и заключается смысл ковариантности результирующих типов функций.

Теперь рассмотрим параметр типа той функции, которая была передана методу `printBookList`. Хотя тип параметра, принадлежащего функции `info`, объявлен как `Book`, функция `getTitle` при ее передаче в этот метод получает значение типа `Publication`, а этот тип является для `Book` *супертипом*. Все это работает, поскольку, хотя типом параметра метода `printBookList` является `Book`, телу метода `printBookList` будет разрешено только передать значение типа `Book` в функцию. А ввиду того, что параметром типа функции `getTitle` является `Publication`, телу этой функции будет лишь разрешено обращаться к его параметру `p`, относящемуся к элементам, объявленным в классе `Publication`. Любой метод, объявленный в классе `Publication`, доступен также в его подклассе `Book`, поэтому все должно работать, в чем, собственно,

и заключается смысл контравариантности типов результатов функций. Графическое представление всего вышесказанного можно увидеть на рис. 18.1.

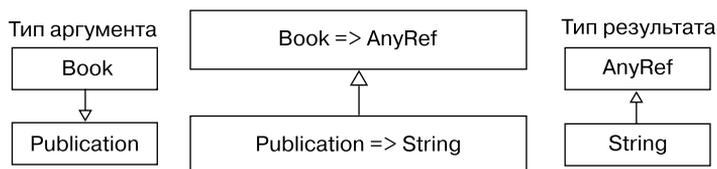


Рис. 18.1. Ковариантность и контравариантность в параметрах типа функции

Код в представленном выше листинге 18.9 проходит компиляцию, поскольку `Publication => String` является подтипом `Book => AnyRef`, что и показано в центре рис. 18.1. Результирующий тип `Function1` определен в качестве ковариантного, и потому показанное в правой части схемы отношение наследования двух результирующих типов имеет то же самое направление, что и две функции, показанные в центре. В отличие от этого, поскольку тип параметра функции `Function1` определен в качестве контравариантного, отношение наследования двух типов параметров, показанное в левой части схемы, имеет направление, обратное направлению отношения наследования двух функций.

18.7. Верхние ограничители

В листинге 14.2 была показана предназначенная для списков функция сортировки слиянием, получавшая в качестве своего первого аргумента функцию сравнения, а в качестве второго, каррированного, — сортируемый список. Еще один способ, который может вам пригодиться для организации подобной функции сортировки, заключается в требовании того, чтобы тип списка примешивал трейт `Ordered`. Как упоминалось в разделе 11.2, примешивание `Ordered` к классу и реализация в `Ordered` одного абстрактного метода, `compare`, позволит клиентам сравнивать экземпляры класса с помощью операторов `<`, `>`, `<=` и `>=`. В качестве примера в листинге 18.10 показан трейт `Ordered`, примешанный к классу `Person`.

Листинг 18.10. Класс `Person`, к которому примешан трейт `Ordered`

```
class Person(val firstName: String, val lastName: String)
  extends Ordered[Person]:

  def compare(that: Person) =
```

```

val lastNameComparison =
  lastName.compareToIgnoreCase(that.lastName)
if lastNameComparison != 0 then
  lastNameComparison
else
  firstName.compareToIgnoreCase(that.firstName)

override def toString = s"$firstName $lastName"

```

В результате двух людей можно сравнивать так:

```

val robert = new Person("Robert", "Jones")
val sally = new Person("Sally", "Smith")
robert < sally // true

```

Чтобы выставить требование о примешивании `Ordered` в тип списка, переданного вашей новой функции сортировки, следует задействовать *верхний ограничитель*. Он указывается так же, как нижний, за исключением того, что вместо обозначения `>:`, используемого для нижних ограничителей, применяется, как показано выше, в листинге 18.11, обозначение `<:`.

Используя синтаксис `T <: Ordered[T]`, вы показываете, что параметр типа `T` имеет верхний ограничитель `Ordered[T]`. Это значит, тип элемента, переданного `orderedMergeSort`, должен быть подтипом `Ordered`. Следовательно, `List[Person]` можно передать `orderedMergeSort`, поскольку `Person` примешивает `Ordered`.

Рассмотрим, к примеру, следующий список:

```

val people = List(
  Person("Larry", "Wall"),
  Person("Anders", "Hejlsberg"),
  Person("Guido", "van Rossum"),
  Person("Alan", "Kay"),
  Person("Yukihiro", "Matsumoto")
)

```

Поскольку тип элемента этого списка `Person` примешивает `Ordered[Person]` (и поэтому является его подтипом), список можно передать методу `orderedMergeSort`:

```

scala> val sortedPeople = orderedMergeSort(people)
val sortedPeople: List[Person] = List(Anders Hejlsberg,
  Alan Kay, Yukihiro Matsumoto, Guido van Rossum, Larry Wall)

```

А теперь следует заметить, что, хотя функция сортировки, показанная в том же листинге 18.11, и служит неплохой иллюстрацией верхних ограничителей,

в действительности это не самый универсальный способ в Scala для разработки функции сортировки, получающей преимущества от трейта `Ordered`.

Листинг 18.11. Функция сравнения с верхним ограничителем

```
def orderedMergeSort[T <: Ordered[T]](xs: List[T]): List[T] =
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if x < y then x :: merge(xs1, ys)
        else y :: merge(xs, ys1)

  val n = xs.length / 2
  if n == 0 then xs
  else
    val (ys, zs) = xs.splitAt(n)
    merge(orderedMergeSort(ys), orderedMergeSort(zs))
```

Так, функцию `orderedMergeSort` нельзя использовать для сортировки списка целых чисел, поскольку класс `Int` не является подтипом `Ordered[Int]`:

```
scala> val wontCompile = orderedMergeSort(List(3, 2, 1))
<console>:5: error: inferred type arguments [Int] do
  not conform to method orderedMergeSort's type
  parameter bounds [T <: Ordered[T]]
  val wontCompile = orderedMergeSort(List(3, 2, 1))
```

В целях получения более универсального решения в разделе 21.4 будет показан порядок использования *заданных параметров* и *типовых классов*.

Резюме

В этой главе мы показали ряд техник, применяемых для сокрытия информации: приватные конструкторы, фабричные методы, абстракцию типов и приватные члены объекта. Кроме этого, продемонстрировали способы указать вариантность типов данных и объяснили, что вариантность означает для реализации класса. И наконец, показали технику, помогающую получить гибкие аннотации вариантности: нижние ограничители для параметров типов методов. В следующей главе мы рассмотрим перечисления.

19

Перечисления

В Scala 3 появилась конструкция `enum`, которая позволяет сделать определение иерархий запечатанных `case`-классов более компактным. Перечисления можно использовать для определения перечисляемых типов данных, распространенных в популярных объектно-ориентированных языках, таких как Java, равно как и в функциональных языках наподобие Haskell, где эти типы относятся к алгебраическим. В Scala эти понятия находятся на противоположных концах спектра, и для их определения используется механизм `enum`. В этой главе будут описаны как перечисляемые, так и алгебраические типы данных.

19.1. Перечисляемые типы данных

Перечисляемый тип данных (enumerated data type, EDT)¹ полезен в ситуациях, когда вам нужен тип, ограниченный конечным множеством именованных значений. Эти именованные значения называются *образцами* EDT. Например, EDT для представления четырех направлений компаса (севера, востока, юга и запада) можно определить так:

```
enum Direction:  
  case North, East, South, West
```

Это простое перечисление сгенерирует запечатанный класс с именем `Direction`² и объект-компаньон с четырьмя значениями, объявленными как

¹ Несмотря на то что `enum` чаще встречается в качестве краткого названия перечисляемых типов данных, в этой книге мы будем использовать аббревиатуру EDT, поскольку конструкция `enum` в Scala применяется в том числе и для определения алгебраических типов, которые называют ADT (algebraic data types).

² Запечатанный класс называется типом перечисления.

val. Значения с именами `North`, `East`, `South` и `West` будут иметь тип `Direction`. С помощью этого определения можно, к примеру, создать метод, который будет инвертировать направление компаса, используя сопоставление с образцом, как показано ниже:

```
import Direction.{North, South, East, West}

def invert(dir: Direction): Direction =
  dir match
    case North => South
    case East  => West
    case South => North
    case West  => East
```

Вот несколько примеров использования метода `invert`:

```
invert(North) // Юг
invert(East)  // Запад
```

Перечисляемые типы данных называются так, потому что компилятор назначает каждому образцу порядковый номер типа `Int`. Порядковые номера начинаются с 0 и увеличиваются на единицу для каждого образца в том порядке, в котором он объявлен в перечислении. Для доступа к порядковым номерам можно использовать метод `ordinal`, который компилятор генерирует для каждого EDT. Например:

```
North.ordinal // 0
East.ordinal  // 1
South.ordinal // 2
West.ordinal  // 3
```

Компилятор также генерирует метод под названием `values` в объекте-компаньоне для каждого типа перечисления EDT. Этот метод возвращает `Array` со всеми образцами EDT в порядке объявления. Тип элементов массива совпадает с типом перечисления. Например, `Direction.values` возвращает `Array[Direction]` с элементами `North`, `East`, `South` и `West` (в этом порядке):

```
Direction.values // Array(North, East, South, West)
```

Наконец, компилятор добавляет в объект-компаньон метод `valueOf`, который преобразует строку в экземпляр типа перечисления — при условии, что эта строка в точности совпадает с названием одного из образцов. Если соответствий не обнаружено, вы получите сгенерированное исключение. Вот несколько примеров использования этого метода:

```
Direction.valueOf("North") // Север
Direction.valueOf("East")  // Восток
```

```
Direction.valueOf("Up")
// IllegalArgumentException: enum case not found: Up
```

Вы также можете назначать типу EDT параметры. Вот новая версия `Direction`, принимающая значение `Int`, которая представляет угол вывода направления в компасе:

```
enum Direction(val degrees: Int):
  case North extends Direction(0)
  case East extends Direction(90)
  case South extends Direction(180)
  case West extends Direction(270)
```

Поскольку значение `degrees` объявлено в виде параметрического поля, оно доступно в любом экземпляре `Direction`. Вот несколько примеров:

```
import Direction.*
North.degrees // 0
South.degrees // 180
```

Вы также можете определять свои собственные методы для типа перечисления, размещая их в теле `enum`. Например, вы могли бы переопределить метод `invert`, показанный ранее, чтобы он стал членом `Direction`:

```
enum Direction(val degrees: Int):

  def invert: Direction =
    this match
      case North => South
      case East  => West
      case South => North
      case West  => East

  case North extends Direction(0)
  case East  extends Direction(90)
  case South extends Direction(180)
  case West  extends Direction(270)
```

Теперь `Direction` сможет себя инвертировать:

```
North.invert // Юг
East.invert  // Запад
```

Если задать для EDT объект-компаньон, Scala все так же предоставит методы `values` и `valueOf`, если вы их не определите. Например, вот объект-компаньон для `Direction` с методом, который находит ближайшее направление компаса относительно переданного угла:

```
object Direction:
  def nearestTo(degrees: Int): Direction =
    val rem = degrees % 360
    val angle = if rem < 0 then rem + 360 else rem
    val (ne, se, sw, nw) = (45, 135, 225, 315)
    angle match
      case a if a > nw || a <= ne => North
      case a if a > ne && a <= se => East
      case a if a > se && a <= sw => South
      case a if a > sw && a <= nw => West
```

Интеграция с перечислениями Java

Чтобы выявить перечисление Java в Scala, достаточно сделать так, чтобы ваш EDT наследовал `java.lang.Enum`, и передать тип перечисления Scala в качестве параметра типа. Например:

```
enum Direction extends java.lang.Enum[Direction]:
  case North, East, South, West
```

Помимо стандартных возможностей, которыми обладают EDT в Scala, эта версия `Direction` также имеет тип `java.lang.Enum`. Например, вы можете воспользоваться методом `compareTo`, который определен в `java.lang.Enum`:

```
Direction.East.compareTo(Direction.South) // -1
```

Объект-компаньон предлагает как объявленные, так и сгенерированные методы. Вот пример одновременного использования двух методов объекта `Direction`: объявленного `nearestTo` и сгенерированного `values`:

```
def allButNearest(degrees: Int): List[Direction] =
  val nearest = Direction.nearestTo(degrees)
  Direction.values.toList.filter(_ != nearest)
```

Функция `allButNearest` возвращает список, содержащий все направления, кроме ближайшего относительно переданного угла компаса. Вот пример ее использования:

```
allButNearest(42) // List(East, South, West)
```

У перечислений есть одно ограничение: вы не можете определять методы для отдельных образцов. Вместо этого любые методы должны объявляться в качестве членов самого типа перечисления, что сделает их доступными

во всех его образцах¹. Образцы перечисления в первую очередь нужны для предоставления фиксированного множества способов создания экземпляров типа перечисления.

19.2. Алгебраические типы данных

Алгебраический тип данных (algebraic data type, ADT) состоит из конечного набора образцов. Это естественный способ выражения моделей предметной области, позволяющий моделировать данные для каждого отдельного образца, который представляет один «конструктор данных» — определенный механизм создания экземпляра типа. В Scala запечатанное семейство `case`-классов составляет ADT — при условии, что как минимум один образец принимает параметры². Например, вот тип ADT, описывающий три возможности: ожидаемое значение («хороший» тип), ошибочное значение («плохой» тип) и исключение («злой» тип):

```
enum Eastwood[+G, +B]:
  case Good(g: G)
  case Bad(b: B)
  case Ugly(ex: Throwable)
```

Как и в случае с EDT, вы не можете определять методы ни для каких конкретных образцов, будь то `Good`, `Bad` или `Ugly`, но это можно сделать из общего суперкласса `Eastwood`. Вот пример метода `map`, который преобразует значение `Good`, если `Eastwood` является `Good`:

```
enum Eastwood[+G, +B]:
  case Good(g: G)
  case Bad(b: B)
  case Ugly(ex: Throwable)

def map[G2](f: G => G2): Eastwood[G2, B] =
  this match
    case Good(g) => Good(f(g))
    case Bad(b) => Bad(b)
    case Ugly(ex) => Ugly(ex)
```

А вот пример его использования:

¹ Вы могли бы определять методы расширения для типов образцов, но в таких ситуациях, наверное, лучше вручную написать иерархию запечатанных классов-образцов.

² Для сравнения, EDT — это запечатанное семейство классов-образцов, ни один образец которого не принимает параметры.

```
val eastWood = Good(41)
eastWood.map(n => n + 1) // Good(42)
```

Реализация ADT и EDT немного отличается. Для каждого образца ADT, принимающего параметры, компилятор генерирует `case`-класс в объекте-компаньоне типа перечисления. Таким образом, для `Eastwood` компилятор сгенерирует код, похожий на следующий:

```
// Сгенерированный запечатанный трейт ("тип перечисления")
sealed trait Eastwood[+G, +B]

object Eastwood: // Generated companion object

  // Сгенерированные классы-образцы
  case class Good[+G, +B](g: G) extends Eastwood[G, B]
  case class Bad[+G, +B](b: B) extends Eastwood[G, B]
  case class Ugly[+G, +B](ex: Throwable) extends Eastwood[G, B]
```

Несмотря на то что итоговый тип фабричного метода, созданного `case`-классами, будет соответствовать типам отдельных классов-экземпляров, компилятор расширит последние до более общего типа перечисления. Вот несколько примеров:

```
scala> Good(42)
val res0: Eastwood[Int, Nothing] = Good(42)

scala> Bad("oops")
val res1: Eastwood[Nothing, String] = Bad(oops)

scala> Ugly(new Exception)
val res2: Eastwood[Nothing, Nothing] = Ugly(java.lang.Exception)
```

Если вам нужен более конкретный тип для своего образца, можете создать экземпляр с помощью `new` вместо фабричного метода. Например, `Good(1)` будет иметь тип `Eastwood[Int, Nothing]`, однако у `new Good(1)` будет более конкретный тип, `Good[Int, Nothing]`.

ADT могут быть рекурсивными. Например, образец может принимать тип перечисления в качестве параметра. Хорошим примером такого рекурсивного ADT является связный список. Его можно определить в виде запечатанного типа с двумя подтипами: объектом-одиночкой, представляющим пустой список, и классом `::`, который принимает два параметра — элемент (начало, или `head`) и остальную часть списка (конец, или `tail`). Ниже показан тип связного списка, в котором объект с пустым списком называется `Nada`, а класс `::` — `Yada`:

```
enum Seinfeld[+E]:
  def ::[E2 >: E](o: E2): Seinfeld[E2] = Yada(o, this)
  case Yada(head: E, tail: Seinfeld[E])
  case Nada
```

ADT `Seinfeld` является рекурсивным типом, поскольку образец `Yada` принимает другой тип `Seinfeld[E]` в качестве своего параметра `tail`. Учитывая, что `Seinfeld` объявляет метод `::`, вы можете создать экземпляр, который похож на `List` из состава `Scala`, но начинается с `Nada`, а не с `Nil`:

```
scala> val xs = 1 :: 2 :: 3 :: Nada
val xs: Seinfeld[Int] = Yada(1,Yada(2,Yada(3,Nada)))
```

19.3. Обобщенные ADT

Обобщенные алгебраические типы данных (generalized algebraic data types, GADT) — это ADT, в которых запечатанный трейт принимает параметр типа, который заполняется образцами. Например:

```
enum Literal[T]:
  case IntLit(value: Int) extends Literal[Int]
  case LongLit(value: Long) extends Literal[Long]
  case CharLit(value: Char) extends Literal[Char]
  case FloatLit(value: Float) extends Literal[Float]
  case DoubleLit(value: Double) extends Literal[Double]
  case BooleanLit(value: Boolean) extends Literal[Boolean]
  case StringLit(value: String) extends Literal[String]
```

Перечисление `Literal` представляет GADT, поскольку оно принимает параметр типа `T`, который указывается каждым его образцом в инструкции `extends`. Например, образец `IntLit` уточняет `T` до `Int`, расширяя `Literal[Int]`.

Такого рода иерархия запечатанных типов носит специальное название «обобщенные ADT», ввиду особых проблем, которые она создает для проверки и вывода типов. Вот наглядный пример:

```
import Literal.*

def valueOfLiteral[T](lit: Literal[T]): T =
  lit match
    case IntLit(n) => n
    case LongLit(m) => m
    case CharLit(c) => c
    case FloatLit(f) => f
    case DoubleLit(d) => d
    case BooleanLit(b) => b
    case StringLit(s) => s
```

Метод `valueOfLiteral` передает средство проверки типов, хотя ни один из его вариантов сопоставления не приводит к нужному итоговому типу `T`. Например, вариант `case IntLit(n)` выдает значение `n`, которое имеет тип `Int`.

Проблема в том, что `Int` не является ни типом `T`, ни его подтипом. Проверка этого типа происходит только лишь из-за того, что, как замечает компилятор, для образца `IntList` роль `T` может играть только `Int`. То же самое касается других вариантов. Кроме того, этот более конкретный тип передается обратно вызывающей стороне. Вот несколько примеров:

```
valueOfLiteral(BooleanLit(true)) // true: Boolean
valueOfLiteral(IntLit(42))       // 42: Int
```

19.4. Что делает типы ADT алгебраическими

ADT называют алгебраическими, потому что они представляют собой применение алгебраической теории к типам. Эту связь с математикой можно наблюдать при сопоставлении каждого типа с его *мощностью* — количеством элементов, из которых он состоит. Если представить тип в виде множества значений, то его мощность будет равна мощности (количеству элементов) этого множества.

КРАТЧАЙШИЙ ПУТЬ

Этот раздел содержит материал о математических свойствах типов данных, которые можно определить с помощью `enum`. Если вы хотите вместо этого познакомиться с коллекциями `Scala`, можете переходить к следующей главе.

Например, у `Boolean` есть два возможных значения, `true` и `false`. Это два элемента, из которых состоит тип `Boolean`. Таким образом, мощность `Boolean` составляет 2. У типа `Unit` есть всего одно возможное значение — пустое множество `()`, — поэтому его мощность — 1, а у типа `Nothing` — 0, так как он не содержит никаких элементов.

Вы можете найти или определить другие типы с мощностью 0, 1 или 2, однако `Nothing`, `Unit` и `Boolean` будет достаточно, чтобы проиллюстрировать алгебраические свойства. Что насчет типа мощностью 3? Если вам не приходится на ум очевидных вариантов из стандартной библиотеки, вы можете легко создать такой тип с помощью EDT:

```
enum TrafficLight:
  case Red, Yellow, Green
```

У типа `TrafficLight` есть три возможных значения: `Red`, `Yellow` и `Green`, что делает его мощность равной 3.

Некоторые типы имеют очень большую мощность. Например, у типа `Byte` есть 256 (2^8) возможных значений в диапазоне от `Byte.MinValue` до

`Byte.MaxValue` включительно. Эти восьмидесятибитные целочисленные значения являются элементами, составляющими `Byte`, поэтому мощность этого типа равна 256. Тип `Int` состоит из 2^{32} элементов, что делает его мощность равной 2^{32} , или 42 949 672 962. Многие типы, такие как `String`, имеют неограниченное множество возможных значений и, следовательно, бесконечную мощность. Алгебра применима и к бесконечным мощностям, но для иллюстрации этих понятий я буду использовать типы с относительно небольшими, конечными мощностями.

Вы можете объединить несколько типов в новый, составной тип, сделав так, чтобы их мощности следовали закону сложения. Такой составной тип называется *типом-суммой*. В Scala кратчайшим способом определения типа-суммы является перечисление. Например:

```
enum Hope[+T]:
  case Glad(o: T)
  case Sad
```

Тип `Hope` позволяет надеяться на лучшее, но готовиться к худшему. Он похож на тип `Option` в Scala, где `Glad` играет роль `Some`, а `Sad` — `None`. Из скольких элементов состоит `Hope[T]`? Поскольку это тип-сумма, мощность `Hope[T]` равна *сумме* мощностей составляющих его типов: `Glad[T]` и `Sad`.

Каждый экземпляр `Glad[T]` служит оберткой для типа `T`, поэтому мощность `Glad[T]` равна мощности `T`. Например, мощность `Glad[Boolean]` — 2, как и `Boolean`. Составными элементами этого типа являются `Glad(true)` и `Glad(false)`. `Sad` — объект-одиночка наподобие `Unit`, поэтому его мощность составляет 1. Таким образом, мощность `Hope[Boolean]` равна 3 (`Glad[Boolean]` — 2 и `Sad` — 1). У этого типа есть три возможных экземпляра: `Glad(true)`, `Glad(false)` и `Sad`. В табл. 19.1 показаны другие примеры.

```
case class Both[A, B](a: A, b: B)
```

Таблица 19.1. Мощность `Hope`

Тип	Мощность	Составные элементы
<code>Hope[Nothing]</code>	$0 + 1 = 1$	<code>Sad</code>
<code>Hope[Unit]</code>	$1 + 1 = 2$	<code>Glad(())</code> , <code>Sad</code>
<code>Hope[Boolean]</code>	$2 + 1 = 3$	<code>Glad(true)</code> , <code>Glad(false)</code> , <code>Sad</code>
<code>Hope[TrafficLight]</code>	$3 + 1 = 4$	<code>Glad(Red)</code> , <code>Glad(Yellow)</code> , <code>Glad(Green)</code> , <code>Sad</code>
<code>Hope[Byte]</code>	$256 + 1 = 257$	<code>Glad(Byte.MinValue)</code> , ... <code>Glad(Byte.MaxValue)</code> , <code>Sad</code>

Теперь вы знаете, как происходит сложение. Но что насчет умножения? Аналогичным образом несколько типов можно объединить в один составной так, чтобы их мощности следовали закону умножения. Такой составной тип называют *типом-произведением*. В Scala кратчайшим способом определения типа-произведения является `case`-класс. Например: тип `Both` позволяет сочетать два значения типов `A` и `B`, подобно тому как это делает тип `Tuple2` из состава Scala. Из скольких элементов состоит `Both[A, B]`? Поскольку это тип-произведение, мощность `Both[A, B]` равна *произведению* мощностей составляющих его типов, `A` и `B`.

Чтобы перечислить все элементы `Both[A, B]`, нужно взять сочетание каждого элемента типа `A` с каждым элементом типа `B`. Например, `TrafficLight` и `Boolean` имеют мощность 3 и 2 соответственно, поэтому мощность `Both[TrafficLight, Boolean]` составит 3×2 , или 6. В табл. 19.2 показано шесть возможных экземпляров этого типа вместе с некоторыми примерами.

Таблица 19.2. Мощность `Both`

Тип	Мощность	Составные элементы
<code>Both[Nothing, Nothing]</code>	$0 \times 0 = 0$	Нет элементов
<code>Both[Unit, Nothing]</code>	$1 \times 0 = 0$	Нет элементов
<code>Both[Unit, Unit]</code>	$1 \times 1 = 1$	<code>Both((), ())</code>
<code>Both[Boolean, Nothing]</code>	$2 \times 0 = 0$	Нет элементов
<code>Both[Boolean, Unit]</code>	$2 \times 1 = 2$	<code>Both(false, ())</code> , <code>Both(true, ())</code>
<code>Both[Boolean, Boolean]</code>	$2 \times 2 = 4$	<code>Both(false, false)</code> , <code>Both(false, true)</code> , <code>Both(true, false)</code> , <code>Both(true, true)</code>
<code>Both[TrafficLight, Nothing]</code>	$3 \times 0 = 0$	Нет элементов
<code>Both[TrafficLight, Unit]</code>	$3 \times 1 = 3$	<code>Both(Red, ())</code> , <code>Both(Yellow, ())</code> , <code>Both(Green, ())</code>
<code>Both[TrafficLight, Boolean]</code>	$3 \times 2 = 6$	<code>Both(Red, false)</code> , <code>Both(Red, true)</code> , <code>Both(Yellow, false)</code> , <code>Both(Yellow, true)</code> , <code>Both(Green, false)</code> , <code>Both(Green, true)</code>

Таблица 19.2 (окончание)

Тип	Мощность	Составные элементы
<code>Both[TrafficLight, TrafficLight]</code>	$3 \times 3 = 9$	<code>Both(Red, Red),</code> <code>Both(Red, Yellow),</code> <code>Both(Red, Green),</code> <code>Both(Yellow, Red),</code> <code>Both(Yellow, Yellow),</code> <code>Both(Yellow, Green),</code> <code>Both(Green, Red),</code> <code>Both(Green, Yellow),</code> <code>Both(Green, Green)</code>

В целом, алгебраические типы данных представляют *суммы произведений*, где образцы формируют варианты типа-суммы и каждый образец представляет тип-произведение, который может состоять из любого количества составных типов, начиная с нуля. EDT является частным случаем ADT, в котором каждый тип-произведение представлен объектом-одиночкой.

За счет понимания алгебраических свойств своих структур данных вы можете использовать соответствующие математические законы, чтобы доказать наличие определенных характеристик у своего кода. Например, что определенные процедуры рефакторинга сохранят логику вашей программы. Показатели мощности ADT подчиняются законам сложения и вычитания, таким как тождество, коммутативность, ассоциативность и дистрибутивность. Функциональное программирование нередко предлагает возможность лучше понять свой код в контексте разных разделов математики.

Резюме

В этой главе вы познакомились с перечислениями в Scala — компактным механизмом определения иерархий запечатанных `case`-классов, формирующих перечисляемые и алгебраические типы данных. Вы узнали, что в Scala EDT и ADT находятся на разных концах одного спектра, и рассмотрели суть алгебраических типов. Конструкция `enum` в Scala делает распространенный подход к функциональному моделированию данных лаконичным и указывает на то, что EDT и ADT являются важными шаблонами проектирования.

20

Абстрактные члены

Член класса или трейта называется *абстрактным*, если у него нет в классе полного определения. Реализовывать абстрактные элементы предполагается в подклассах того класса, в котором они объявлены. Воплощение этой идеи можно найти во многих объектно-ориентированных языках. Например, в Java можно объявить абстрактные методы. В Scala тоже, что было показано в разделе 10.2. Но Scala этим не ограничивается, и в нем данная идея реализуется самым универсальным образом: в качестве членов классов и трейтов можно объявлять не только методы, но и абстрактные поля и даже абстрактные типы.

В этой главе мы рассмотрим все четыре разновидности абстрактных членов: `val`- и `var`-переменные, методы и типы. Попутно изучим предварительно инициализированные поля, ленивые `val`-переменные и типы, зависящие от пути.

20.1. Краткий обзор абстрактных членов

В следующем трейте объявляется по одному абстрактному члену каждого вида: абстрактный тип (`T`), метод (`transform`), `val`-переменная (`initial`) и `var`-переменная (`current`):

```
trait Abstract:  
  type T  
  def transform(x: T): T  
  val initial: T  
  var current: T
```

Конкретная реализация трейта `Abstract` нуждается в заполнении определений для каждого из его абстрактных членов. Пример реализации, предоставляющий эти определения, выглядит так:

```
class Concrete extends Abstract:  
  type T = String  
  def transform(x: String) = x + x  
  val initial = "hi"  
  var current = initial
```

Реализация придает конкретное значение типу `T`, определяя его в качестве псевдонима типа `String`. Операция `transform` конкатенирует предоставленную ей строку с ней же самой, а для `initial`, как и для `current`, устанавливается значение `"hi"`.

Указанные примеры дают вам первое приблизительное представление о разновидностях абстрактных членов, существующих в `Scala`. Далее мы рассмотрим подробности, касающиеся этих членов, и объясним, для чего могут пригодиться эти новые формы абстрактных членов, а также члены-типы в целом.

20.2. Члены-типы

В примере, приведенном в предыдущем разделе, было показано, что понятие «*абстрактный тип*» в `Scala` означает объявление типа (с ключевым словом `type`) в качестве члена класса или трейта, без указания определения. Абстрактными могут быть и сами классы, а трейты по определению абстрактные, однако ни один из них не является в `Scala` тем, что называют *абстрактным типом*. Абстрактный тип в `Scala` всегда выступает членом какого-либо класса или трейта, как тип `T` в трейте `Abstract`.

Неабстрактный (или конкретный) член-тип, такой как тип `T` в классе `Concrete`, можно представить себе в качестве способа определения нового имени, или *псевдонима*, для типа. К примеру, в классе `Concrete` типу `String` дается псевдоним `T`. В результате везде, где в определении класса `Concrete` появляется `T`, подразумевается `String`. Сюда включаются преобразования типов параметров и результирующих типов, как исходных, так и текущих, в которых при их объявлении в супертрейте `Abstract` упоминается `T`. Следовательно, когда в классе `Concrete` реализуются эти методы, такие обозначения `T` интерпретируются как `String`.

Один из поводов использовать член-тип — определение краткого описательного псевдонима для типа, чье имя длиннее или значение менее понятно,

чем у псевдонима. Такие члены-типы могут сделать понятнее код класса или трейта. Другое основное применение членов-типов — объявление абстрактного типа, который должен быть определен в подклассе. Более подробно этот вариант использования, продемонстрированный в предыдущем разделе, мы рассмотрим чуть позже в данной главе.

20.3. Абстрактные val-переменные

Объявление абстрактной val-переменной выглядит следующим образом:

```
val initial: String
```

Val-переменной даются имя и тип, но не указывается значение. Оно должно быть предоставлено конкретным определением val-переменной в подклассе. Например, в классе `Concrete` для реализации val-переменной используется такой код:

```
val initial = "hi"
```

Объявление в классе абстрактной val-переменной применяется, когда в этом классе еще неведомо нужное ей значение, но известно, что переменная в каждом экземпляре класса получит неизменяемое значение.

Объявление абстрактной val-переменной напоминает объявление абстрактного метода без параметров:

```
def initial: String
```

Клиентский код будет ссылаться как на val-переменную, так и на метод абсолютно одинаково (то есть `obj.initial`). Но если `initial` является абстрактной val-переменной, то клиенту гарантируется, что `obj.initial` будет при каждом обращении выдавать одно и то же значение. Если `initial` — абстрактный метод, то данная гарантия соблюдаться не будет, поскольку в таком случае метод `initial` можно реализовать с помощью конкретного метода, возвращающего при каждом своем вызове разные значения.

Иными словами, val-переменная ограничивает свою допустимую реализацию: любая реализация должна быть определением val-переменной — она не может быть var- или def-определением. А вот объявления абстрактных методов можно реализовать как конкретными определениями методов, так и конкретными определениями var-переменных. Если взять абстрактный класс `Fruit`, показанный в листинге 20.1, то класс `Apple` будет допустимой реализацией подкласса, а класс `BadApple` — нет.

Листинг 20.1. Переопределение абстрактных `val`-переменных и методов без параметров

```
abstract class Fruit:
  val v: String // `v` – значение (value)
  def m: String // `m` – метод (method)

abstract class Apple extends Fruit:
  val v: String
  val m: String // нормально воспринимаемое переопределение 'def'
                // в 'val'

abstract class BadApple extends Fruit:
  def v: String // ОШИБКА: переопределять 'val' в 'def' нельзя
  def m: String
```

20.4. Абстрактные `var`-переменные

Как и для абстрактной `val`-переменной, для абстрактной `var`-переменной объявляются только имя и тип, но не начальное значение. Например, в листинге 20.2 показан трейт `AbstractTime`, в котором объявляются две абстрактные переменные с именами `hour` и `minute`.

Листинг 20.2. Объявление абстрактных `var`-переменных

```
trait AbstractTime:
  var hour: Int
  var minute: Int
```

Что означают такие абстрактные `var`-переменные, как `hour` и `minute`? В разделе 16.2 было показано, что `var`-переменные, объявленные в качестве членов класса, оснащаются геттером и сеттером. Это справедливо и для абстрактных `var`-переменных. Если, к примеру, объявляется абстрактная `var`-переменная по имени `hour`, то подразумевается, что для нее объявляется абстрактный геттер `hour` и абстрактный сеттер `hour_ =`. Тем самым не определяется никакое переназначаемое поле, а конкретная реализация абстрактной `var`-переменной будет выполнена в подклассах. Например, определение `AbstractTime`, показанное выше, в листинге 20.2, абсолютно эквивалентно определению, показанному в листинге 20.3.

Листинг 20.3. Расширение абстрактных `var`-переменных в геттеры и сеттеры

```
trait AbstractTime:
  def hour: Int           // get-метод для 'hour'
  def hour_(x: Int): Unit // set-метод для 'hour'
  def minute: Int        // get-метод для 'minute'
  def minute_(x: Int) : Unit // set-метод для 'minute'
```

20.5. Инициализация абстрактных val-переменных

Иногда абстрактные val-переменные играют роль, аналогичную роли параметров суперкласса: они позволяют предоставить в подклассе подробности, не указанные в суперклассе. Рассмотрим в качестве примера переформулировку класса `Rational` из главы 6, который был показан в листинге 6.5, в трейт:

```
trait RationalTrait:  
  val numerArg: Int  
  val denomArg: Int
```

У класса `Rational` из главы 6 были два параметра: `n` для числителя рационального числа и `d` для его знаменателя. Представленный здесь трейт `RationalTrait` определяет вместо них две абстрактные val-переменные: `numerArg` и `denomArg`. Чтобы создать конкретный экземпляр этого трейта, нужно реализовать определения абстрактных val-переменных, например:

```
new RationalTrait:  
  val numerArg = 1  
  val denomArg = 2
```

Здесь появляется ключевое слово `new` перед `RationalTrait`, после которого стоит двоеточие и отступ от тела класса. Это выражение выдает экземпляр *анонимного класса*, примешивающего трейт и определяемого телом. Создание экземпляра данного анонимного класса дает эффект, аналогичный созданию экземпляра с помощью кода `new Rational(1, 2)`.

Но аналогия здесь неполная. Есть небольшое различие, касающееся порядка, в котором инициализируются выражения. При записи следующего кода:

```
new Rational(expr1, expr2)
```

два выражения, `expr1` и `expr2`, вычисляются перед инициализацией класса `Rational`, следовательно, значения `expr1` и `expr2` доступны для инициализации класса `Rational`.

С трейтами складывается обратная ситуация. При записи кода

```
new RationalTrait:  
  val numerArg = expr1  
  val denomArg = expr2
```

выражения `expr1` и `expr2` вычисляются как часть инициализации анонимного класса, но анонимный класс инициализируется *после* `RationalTrait`.

Следовательно, значения `numerArg` и `denomArg` в ходе инициализации `RationalTrait` недоступны (точнее говоря, выбор любого значения даст значение по умолчанию для типа `Int`, то есть ноль). Для представленного ранее определения `RationalTrait` это не проблема, поскольку при инициализации трейта значения `numerArg` или `denomArg` не используются. Но проблема возникает в варианте `RationalTrait`, показанном в листинге 20.4, где определяются нормализованные числитель и знаменатель.

Листинг 20.4. Трейт, использующий абстрактные `val`-переменные

```
trait RationalTrait:
  val numerArg: Int
  val denomArg: Int
  require(denomArg != 0)
  private val g = gcd(numerArg, denomArg)
  val numer = numerArg / g
  val denom = denomArg / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  override def toString = s"$numer/$denom"
```

При попытке создать экземпляр этого трейта с какими-либо выражениями для числителя и знаменателя, не являющимися простыми литералами, выдается исключение:

```
scala> val x = 2
val x: Int = 2

scala> new RationalTrait:
  val numerArg = 1 * x
  val denomArg = 2 * x
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:280)
  at RationalTrait.$init$(<console>:4)
  ... 28 elided
```

Исключение в этом примере было сгенерировано потому, что при инициализации класса `RationalTrait` у `denomArg` сохранилось исходное нулевое значение, из-за чего вызов `require` завершился сбоем.

В данном примере показано, что порядок инициализации для параметров класса и абстрактных полей разный. Аргумент параметра класса вычисляется *до* его передачи конструктору класса (если только это не параметр, передаваемый по имени). А вот реализация определения `val`-переменной, которая находится в подклассе, вычисляется только *после* инициализации суперкласса.

Теперь вы понимаете, почему поведение абстрактных `val`-переменных отличается от поведения параметров, и было бы неплохо узнать, что

с этим делать. Получится ли определить `RationalTrait`, который можно надежно инициализировать, не опасаясь, что возникнут ошибки из-за неинициализированных полей? В действительности в Scala предлагаются два альтернативных решения этой проблемы: *параметрические поля трейтов* и *ленивые val-переменные*. Эти решения рассматриваются в остальной части раздела.

Параметрические поля трейтов

Первое решение — параметрические поля трейтов — позволяет вычислять значения для полей трейтов до того, как сам трейт будет инициализирован. Для этого определите поля как параметрические. Пример приведен в листингах 20.5 и 20.6.

Листинг 20.5. Трейт, принимающий параметрические поля

```
trait RationalTrait(val numerArg: Int, val denomArg: Int):
  require(denomArg != 0)
  private val g = gcd(numerArg, denomArg)
  val numer = numerArg / g
  val denom = denomArg / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  override def toString = s"$numer/$denom"
```

Листинг 20.6. Параметрические поля трейта в выражении анонимного класса

```
scala> new RationalTrait(1 * x, 2 * x) {}
val res1: RationalTrait = 1/2
```

Сфера применения параметрических полей не ограничивается анонимными классами, они могут использоваться также в объектах или именованных подклассах. Соответствующие примеры показаны в листингах 20.7 и 20.8. Класс `RationalClass`, показанный в листинге 20.8, иллюстрирует общую схему доступности параметров класса для инициализации супертрейта.

Листинг 20.7. Параметрические поля трейта в определении объекта

```
object TwoThirds extends RationalTrait(2, 3)
```

Листинг 20.8. Параметрические поля трейта в определении класса

```
class RationalClass(n: Int, d: Int) extends RationalTrait(n, d):
  def + (that: RationalClass) = new RationalClass(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )
```

Ленивые val-переменные

Параметры трейта могут применяться для точной имитации поведения инициализации аргументов конструктора класса. Но иногда лучше позволить самой системе разобраться, как и что должно быть проинициализировано. Добиться этого можно с помощью определения *ленивой* val-переменной. Если перед определением val-переменной поставить модификатор *lazy*, то выражение инициализации справа будет вычисляться только при первом использовании val-переменной.

Определим, к примеру, объект `Demo` с val-переменной:

```
object Demo:
  val x = { println("initializing x"); "done" }
```

Теперь сначала сошлемся на `Demo`, а затем на `Demo.x`:

```
scala> Demo
initializing x
val res0: Demo.type = Demo$@3002e397
```

```
scala> Demo.x
val res1: String = done
```

Как видите, на момент использования объекта `Demo` его поле `x` становится проинициализированным. Инициализация `x` составляет часть инициализации `Demo`. Но ситуация изменится, если определить поле `x` как *lazy*:

```
object Demo:
  lazy val x = { println("initializing x"); "done" }
```

```
scala> Demo
val res2: Demo.type = Demo$@24e5389c
```

```
scala> Demo.x
initializing x
val res3: String = done
```

Теперь инициализация `Demo` не включает инициализацию `x`. Она будет отложена до первого использования `x`. Это похоже на ситуацию определения `x` в качестве метода без параметров с помощью ключевого слова `def`. Но в отличие от `def` ленивая val-переменная никогда не вычисляется более одного раза. Фактически после первого вычисления ленивой val-переменной результат вычисления сохраняется, чтобы его можно было применить повторно при последующем использовании той же самой val-переменной.

При изучении данного примера создается впечатление, что объекты, подобные `Demo`, сами ведут себя как ленивые `val`-переменные, поскольку инициализируются по необходимости при их первом использовании. Так и есть. Действительно, определение объекта может рассматриваться как сокращенная запись для определения ленивой `val`-переменной с анонимным классом, в котором описывается содержимое объекта.

Используя ленивые `val`-переменные, можно переделать `RationalTrait`, как показано в листинге 20.9. В новом определении трейта все конкретные поля определены как `lazy`. Есть еще одно изменение, касающееся предыдущего определения `RationalTrait`, показанного выше, в листинге 20.4. Данное изменение заключается в том, что условие `require` было перемещено из тела трейта в инициализатор приватного поля `g`, вычисляющий наибольший общий делитель для `numerArg` и `denomArg`. После внесения этих изменений при инициализации `LazyRationalTrait` делать больше ничего не нужно, поскольку весь код инициализации теперь является правосторонней частью ленивой `val`-переменной. Таким образом, теперь вполне безопасно инициализировать абстрактные поля `LazyRationalTrait` после того, как класс уже определен.

Давайте рассмотрим пример:

```
scala> val x = 2
val x: Int = 2
```

Листинг 20.9. Инициализация трейта с ленивыми `val`-переменными

```
trait LazyRationalTrait:
  val numerArg: Int
  val denomArg: Int

  lazy val numer = numerArg / g
  lazy val denom = denomArg / g

  override def toString = s"$numer/$denom"

  private lazy val g =
    require(denomArg != 0)
    gcd(numerArg, denomArg)

  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
```

Рассмотрим пример:

```
scala> new LazyRationalTrait:
  val numerArg = 1 * x
  val denomArg = 2 * x

val res4: LazyRationalTrait = 1/2
```

Здесь не нужны какие-либо предварительные вычисления. Проследим последовательность инициализации, приводящей к тому, что в показанном ранее коде на стандартное устройство будет выведена строка 1/2.

1. Создается новый экземпляр `LazyRationalTrait`, и запускается код инициализации `LazyRationalTrait`. Этот код пуст — ни одно из полей `LazyRationalTrait` еще не проинициализировано.
2. С помощью вычисления выражения `new` определяется первичный конструктор анонимного подкласса. Данная процедура включает в себя инициализацию `numerArg` значением 2 и инициализацию `denomArg` значением 4.
3. Далее интерпретатор в отношении создаваемого объекта вызывает метод `toString`, чтобы получившееся значение можно было бы вывести на стандартное устройство.
4. Метод `toString`, определенный в трейте `LazyRationalTrait`, выполняет первое обращение к полю `numer`, что вызывает вычисление инициализатора.
5. Инициализатор поля `numer` обращается к приватному полю `g`; таким образом, следующим вычисляется `g`. При этом вычислении происходит обращение к `numerArg` и `denomArg`, которые были определены в шаге 2.
6. Метод `toString` обращается к значению `denom`, что вызывает вычисление `denom`. При этом происходит обращение к значениям `denomArg` и `g`. Инициализатор поля `g` заново уже не вычисляется, поскольку был вычислен в шаге 5.
7. Создается и выводится строка результата 1/2.

Обратите внимание: в классе `LazyRationalTrait` определение `g` появляется в тексте кода после определений в нем `numer` и `denom`. Несмотря на это, ввиду того что все три значения ленивые, `g` инициализируется до завершения инициализации `numer` и `denom`.

Тем самым демонстрируется важное свойство ленивых `val`-переменных: порядок следования их определений в тексте кода не играет никакой роли, поскольку инициализация значений выполняется по требованию. Стало быть, ленивые `val`-переменные могут освободить вас как программиста от необходимости обдумывать порядок расстановки определений `val`-переменных, чтобы гарантировать, что к моменту востребованности все будет определено.

Но данное преимущество сохраняется до тех пор, пока инициализация ленивых `val`-переменных не производит никаких побочных эффектов, а также

не зависит от них. Если есть побочные эффекты, то порядок инициализации становится значимым. И тогда могут возникнуть серьезные трудности в отслеживании порядка запуска инициализационного кода, как было показано в предыдущем примере. Следовательно, ленивые `val`-переменные — идеальное дополнение к функциональным объектам, в которых порядок инициализации не имеет значения до тех пор, пока все в конечном счете не будет проинициализировано. А вот для преимущественно императивного кода эти переменные подходят меньше.

Ленивые функциональные языки

Scala — далеко не первый язык, использующий идеальную пару ленивых определений и функционального кода. Существует целая категория ленивых языков функционального программирования, в которых каждое значение и каждый параметр инициализируются лениво. Яркий представитель этого класса языков — Haskell [SPJ02].

20.6. Абстрактные типы

В начале этой главы в качестве объявления абстрактного типа мы показали код `type T`. Далее мы рассмотрим, что означает такое объявление абстрактного типа и для чего оно может пригодиться. Как и все остальные объявления абстракций, объявление абстрактного типа — заместитель для чего-либо, что будет конкретно определено в подклассах. В данном случае это тип, который будет определен ниже по иерархии классов. Следовательно, обозначение `T` ссылается на тип, который на момент его объявления еще неизвестен. Разные подклассы могут обеспечивать различные реализации `T`.

Рассмотрим широко известный пример, в который абстрактные типы вписываются вполне естественно. Предположим, что получена задача смоделировать привычный рацион животных. Начать можно с определения класса питания `Food` и класса животных `Animal` с методом питания `eat`:

```
class Food
abstract class Animal:
  def eat(food: Food): Unit
```

Затем можно попробовать создать специализацию этих двух классов в виде класса коров `Cow`, питающихся травой `Grass`:

```
class Grass extends Food
class Cow extends Animal:
  override def eat(food: Grass) = {} // Этот код не пройдет компиляцию
```

Но при попытке компиляции этих новых классов будут получены следующие ошибки:

```
2 | class Cow extends Animal:
  |   ^
  |   class Cow needs to be abstract, since
  |   def eat(food: Food): Unit is not defined (Note that Food
  |   does not match Grass: class Grass is a subclass of class
  |   Food, but method parameter types must match exactly.)
3 |   override def eat(food: Grass) = {} // This won't...
  |   ^
  |   method eat has a different signature than the
  |   overridden declaration
```

Дело в том, что метод `eat` в классе `Cow` не переопределяет метод `eat` класса `Animal`, поскольку типы их параметров различаются: в классе `Cow` это `Grass`, а в классе `Animal` это `Food`.

Некоторые считают, что в отклонении этих двух классов виновата слишком строгая система типов. Они говорят, что допустимо специализировать параметр метода в подклассе. Но если бы классы были позволены в том виде, в котором написаны, вы быстро оказались бы в весьма небезопасной ситуации.

К примеру, механизму проверки типов будет передан следующий скрипт:

```
class Food
abstract class Animal:
  def eat(food: Food): Unit

class Grass extends Food
class Cow extends Animal
  override def eat(food: Grass) = {} // Этот код не пройдет компиляцию,
  // но если бы это случилось...

class Fish extends Food
val bessy: Animal = new Cow
bessy.eat(new Fish) // ...коров можно было бы накормить рыбой.
```

Если снять ограничения, то программа пройдет компиляцию, поскольку коровы из класса `Cow` — животные из класса `Animal`, а у класса `Animal` есть метод кормления `eat`, который принимает любую разновидность питания `Food`, включая рыбу, то есть `Fish`. Но коровы не едят рыбу!

Вместо этого вам нужно применить более точное моделирование. Животные из класса `Animal` потребляют (`eat`) питание `Food`, но какое именно питание потребляет каждое животное, зависит от самого животного. Это довольно четко можно выразить с помощью абстрактного типа, что и показано в листинге 20.10.

Листинг 20.10. Моделирование подходящего питания с помощью абстрактных типов

```
class Food
abstract class Animal:
  type SuitableFood <: Food
  def eat(food: SuitableFood): Unit
```

С новым определением класса животное `Animal` может потреблять только то питание, которое ему подходит. Какое именно питание будет подходящим, нельзя определить на уровне класса `Animal`. Поэтому подходящее питание `SuitableFood` моделируется в виде абстрактного типа. У него есть верхний ограничитель `Food`, что выражено условием `<: Food`. Это значит, что любая конкретная реализация `SuitableFood` (в подклассе класса `Animal`) должна быть подклассом `Food`. К примеру, реализовать `SuitableFood` классом `IOException` не получится.

После определения `Animal` можно, как показано в листинге 20.11, перейти к коровам. Класс `Cow` устанавливает в качестве подходящего для коров питания `SuitableFood` траву `Grass`, а также определяет конкретный метод `eat` для данной разновидности питания.

Листинг 20.11. Реализация абстрактного типа в подклассе

```
class Grass extends Food
class Cow extends Animal:
  type SuitableFood = Grass
  override def eat(food: Grass) = {}
```

Эти новые определения класса компилируются без ошибок. При попытке запустить с новыми определениями класс контрпримера про коров, которые едят рыбу (`cows-that-eat-fish`), будут получены следующие ошибки компиляции:

```
class Fish extends Food
val bessy: Animal = new Cow

scala> bessy.eat(new Fish)
1 |bessy.eat(new Fish)
  |                ~~~~~
  |                Found: Fish
  |                Required: bessy.SuitableFood
```

20.7. Типы, зависящие от пути

Еще раз посмотрим на последнее сообщение об ошибке. Нас интересует тип, требующийся для метода `eat: bessy.SuitableFood`. Указание типа состоит из ссылки на объект, `bessy`, за которой следует поле типа объекта, `SuitableFood`. Тем самым показывается, что объекты в Scala в качестве членов могут иметь типы. Смысл `bessy.SuitableFood` — «тип `SuitableFood`, являющийся членом объекта, на который ссылается `bessy`», или, иначе, тип питания, подходящего для `bessy`.

Тип вида `bessy.SuitableFood` называется *типом, зависящим от пути* (path-dependent type). Слово «путь» здесь означает ссылку на объект. Это может быть единственное имя, такое как `bessy`, или более длинный путь доступа, такой как `farm.barn.bessy`, где все составляющие, `farm`, `barn` и `bessy`, — переменные (или имена объектов-одиночек), которые ссылаются на объекты.

Термин «тип, зависящий от пути» подразумевает, что тип зависит от пути; и в целом различные пути дают начало разным типам. Например, предположим, что для определения классов собачьего питания `DogFood` и собак `Dog` используется следующий код:

```
class DogFood extends Food
class Dog extends Animal:
  type SuitableFood = DogFood
  override def eat(food: DogFood) = {}
```

При попытке накормить собаку едой для коров ваш код не пройдет компиляцию:

```
val bessy = new Cow
val lassie = new Dog

scala> lassie.eat(new bessy.SuitableFood)
1 | lassie.eat(new bessy.SuitableFood)
  |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |                                     Found:      Grass
  |                                     Required: DogFood
```

Проблема заключается в том, что типом объекта `SuitableFood`, переданного методу `eat`, выступает `bessy.SuitableFood`, а он несовместим с параметром типа `eat`, которым является `lassie.SuitableFood`.

В случае с двумя собаками из класса `Dog` ситуация другая. Поскольку в классе `Dog` тип `SuitableFood` определен в качестве псевдонима для класса `DogFood`, то типы `SuitableFood` двух представителей класса `Dog` по факту одинаковы.

В результате экземпляра класса `Dog`, называемый `lassie`, фактически может питаться тем, что подходит другому экземпляру класса `Dog`, который мы назовем `bootsie`:

```
val bootsie = new Dog
lassie.eat(new bootsie.SuitableFood)
```

Тип, зависящий от пути, напоминает синтаксис для типа внутреннего класса в Java, но есть существенное различие: в типе, зависящем от пути, называется внешний *объект*, а в типе внутреннего класса — внешний *класс*. Типы внутренних классов в стиле Java могут быть выражены и в Scala, но записываются по-другому. Рассмотрим два класса — наружный `Outer` и внутренний `Inner`:

```
class Outer:
  class Inner
```

В Scala вместо применяемого в Java выражения `Outer.Inner` к внутреннему классу обращаются с помощью выражения `Outer#Inner`. Синтаксис с использованием точки (.) зарезервирован для объектов. Представим, к примеру, что создаются экземпляры двух объектов типа `Outer`:

```
val o1 = new Outer
val o2 = new Outer
```

Здесь `o1.Inner` и `o2.Inner` — два типа, зависящих от пути, и это разные типы. Оба они соответствуют более общему типу `Outer#Inner` (являются его подтипами), который представляет класс `Inner` с произвольным внешним объектом типа `Outer`. В отличие от этого тип `o1.Inner` ссылается на класс `Inner` с конкретным внешним объектом, на который ссылается `o1`. Точно так же тип `o2.Inner` ссылается на класс `Inner` с другим конкретным внешним объектом, на который ссылается `o2`.

В Scala, как и в Java, экземпляры внутреннего класса содержат ссылку на экземпляр охватывающего их внешнего класса. Это, к примеру, позволяет внутреннему классу обращаться к членам его внешнего класса. Таким образом, невозможно создать экземпляр внутреннего класса, не имея какого-либо способа указать экземпляр внешнего класса. Один из способов заключается в создании экземпляра внутреннего класса внутри тела внешнего класса. В подобном случае будет использован текущий экземпляр внешнего класса (в ссылке на который можно задействовать `this`).

Еще один способ заключается в использовании типа, зависящего от пути. Например, в типе `o1.Inner` присутствует название конкретного внешнего объекта, поэтому можно создать его экземпляр:

```
new o1.Inner
```

Получившийся внутренний объект будет содержать ссылку на свой внешний объект, то есть на объект, на который ссылается `o1`. В отличие от этого, поскольку тип `Outer#Inner` не содержит названия какого-либо конкретного экземпляра класса `Outer`, создать экземпляр данного класса невозможно:

```
scala> new Outer#Inner
1 | new Outer#Inner
  | ~~~~~
  | Outer is not a valid class prefix, since it is
  | not an immutable path
```

20.8. Уточняющие типы

Когда класс является наследником другого класса, первый класс называют *номинальным* подтипом другого класса. Этот подтип *номинальный*, поскольку у каждого типа есть *имя* и имена явным образом объявлены имеющими отношение подтипирования. Кроме того, в Scala дополнительно поддерживается *структурное* подтипирование, где отношение подтипирования возникает просто потому, что у двух типов есть совместимые элементы. Для получения в Scala структурного подтипирования нужно задействовать имеющиеся в данном языке *уточняющие типы*.

Обычно удобнее применять номинальное подтипирование, поэтому в любой новой конструкции нужно сначала попробовать воспользоваться именно им. Имя — один краткий идентификатор, следовательно, короче явного перечисления типов членов. Кроме того, структурное подтипирование зачастую более гибко, чем требуется. Тем не менее оно имеет свои преимущества. Одно из них заключается в том, что иногда действительно не нужно ничего определять в виде типов, кроме членов самого класса. Предположим, к примеру, что необходимо определить класс пастбища `Pasture`, который может содержать животных, поедающих траву. Одним из вариантов может быть определение трейта для животных, питающихся травой, — `AnimalThatEatsGrass`, и его примешивание в каждый класс, где он применяется. Но это будет слишком многословным решением. В классе `Cow` уже есть объявление, что это животное и оно ест траву, а теперь придется объявлять, что это животное, поедающее траву.

Вместо определения `AnimalThatEatsGrass` можно воспользоваться уточняющим типом. Просто запишите основной тип, `Animal`, а за ним последовательность членов, перечисленных в фигурных скобках. Члены в фигурных скобках представляют дальнейшие указания, или, если хотите, уточнения, типов элементов из основного класса.

Вот как записывается тип «животное, поедающее траву»:

```
Animal { type SuitableFood = Grass }
```

Теперь, имея в своем распоряжении этот тип, класс «пастбища» можно записать следующим образом:

```
class Pasture:
  var animals: List[Animal { type SuitableFood = Grass }] = Nil
  // ...
```

20.9. Практический пример: работа с валютой

Далее в главе рассмотрен практический пример, объясняющий порядок использования в Scala абстрактных типов. При этом будет поставлена задача разработать класс `Currency`. Обычный его экземпляр будет представлять денежную сумму в долларах, евро, йенах и некоторых других валютах. Он позволит совершать с валютой ряд арифметических операций. Например, можно будет сложить две суммы в одной и той же валюте. Или умножить текущую сумму на коэффициент процентной ставки.

Эти соображения приводят к следующей первой конструкции класса валют:

```
// первая (нерабочая) конструкция класса Currency
abstract class Currency:
  val amount: Long
  def designation: String
  override def toString = s"$amount $designation"
  def + (that: Currency): Currency = ...
  def * (x: Double): Currency = ...
```

Поле `amount` (сумма) в классе валют — количество представляемых ею валютных единиц. Оно имеет тип `Long`, то есть представляемая сумма денежных средств может быть очень крупной, сравнимой с рыночной капитализацией Google или Apple. Здесь поле оставлено абстрактным в ожидании своего определения, когда в подклассе пойдет речь о конкретной сумме. Наименование валюты `designation` — строка, которая идентифицирует эту валюту. Метод `toString` класса `Currency` показывает сумму и наименование валюты. Он будет выдавать результат следующего вида:

```
79 USD
11000 Yen
99 Euro
```

И наконец, имеются методы `+` для сложения сумм в валюте и `*` для умножения суммы в валюте на число с плавающей точкой. Конкретное значение в валюте можно создать, предоставив конкретные значения суммы и наименования валюты:

```
new Currency:  
  val amount = 79L  
  def designation = "USD"
```

Эта конструкция не вызовет нареканий, если задумано моделирование с использованием лишь одной валюты, например только долларов или только евро. Но она не будет работать при необходимости иметь дело сразу с несколькими валютами. Предположим, выполняется моделирование долларов и евро в качестве двух подклассов класса валюты:

```
abstract class Dollar extends Currency:  
  def designation = "USD"  
  
abstract class Euro extends Currency:  
  def designation = "Euro"
```

На первый взгляд все выглядит вполне разумно. Но данный код позволит складывать доллары с евро. Результатом такого сложения будет тип `Currency`. Но это будет весьма забавная валюта — смесь евро и долларов. Вместо этого нужно получить более специализированную версию метода `+`. При его реализации в классе `Dollar` он должен получать аргументы типа `Dollar` и выдавать результат типа `Dollar`; при реализации в классе `Euro` — получать аргументы типа `Euro` и выдавать результат типа `Euro`. Следовательно, тип метода сложения будет изменяться в зависимости от того, в каком классе находится. И все же хотелось бы создать метод сложения единожды, а не делать это при каждом новом определении валюты.

Чтобы помочь справиться с подобными ситуациями, `Scala` предоставляет весьма простую технологию. Если к моменту определения класса что-то еще неизвестно, то нужно сделать это «что-то» абстрактным. Технология применима как к значениям, так и к типам. В случае с валютами точные типы аргументов и результирующие типы метода сложения неизвестны, следовательно, являются подходящими кандидатами для того, чтобы стать абстрактными.

Это привело бы к следующей предварительной версии кода класса `AbstractCurrency`:

```
// вторая (все еще несовершенная) конструкция класса Currency  
abstract class AbstractCurrency:
```

```

type Currency <: AbstractCurrency
val amount: Long
def designation: String
override def toString = s"$amount $designation"
def + (that: Currency): Currency = ...
def * (x: Double): Currency = ...

```

Единственное отличие от прежней ситуации заключается в том, что класс теперь называется `AbstractCurrency` и содержит абстрактный тип `Currency`, представляющий стоящую под вопросом реальную валюту. Каждому конкретному подклассу `AbstractCurrency` придется фиксировать тип `Currency`, чтобы обозначать конкретный подкласс как таковой, тем самым затягивая узел.

Вот как, к примеру, выглядит новая версия класса `Dollar`, которая теперь расширяет класс `AbstractCurrency`:

```

abstract class Dollar extends AbstractCurrency:
  type Currency = Dollar
  def designation = "USD"

```

Данная конструкция вполне работоспособна, но по-прежнему далека от совершенства. Есть проблема, скрывающаяся за многоточиями, которые показывают в классе `AbstractCurrency` пропущенные определения методов `+` и `*`. В частности, как в этом классе должен быть реализован метод сложения? Нетрудно вычислить правильную сумму в новой валюте как `this.amount + that.amount`, но как преобразовать сумму в валюту нужного типа?

Можно попробовать применить следующий код:

```

def + (that: Currency): Currency =
  new Currency:
    val amount = this.amount + that.amount

```

Но он не пройдет компиляцию:

```

7 |     new Currency:
  |         ^^^^^^^^^
  |         AbstractCurrency.this.Currency is not a class type
8 |     val amount = this.amount + that.amount
  |                   ^
  |                   Recursive value amount needs type

```

Одно из ограничений в трактовке в `Scala` абстрактных типов заключается в невозможности создать экземпляр абстрактного типа, а также в невозможности абстрактного типа играть роль супертипа для другого класса. Следовательно, компилятор будет отвергать код показанного здесь примера, в котором предпринимается попытка создать экземпляр `Currency`.

Но эти ограничения можно обойти, используя *фабричный метод*. Вместо того чтобы создавать экземпляр абстрактного класса, напрямую объявите абстрактный метод, который делает это. Затем там, где абстрактный тип устанавливается в какой-либо конкретный тип, нужно предоставить конкретную реализацию фабричного метода. Для класса `AbstractCurrency` все вышесказанное будет выглядеть так:

```
abstract class AbstractCurrency:
  type Currency <: AbstractCurrency // абстрактный тип
  def make(amount: Long): Currency // фабричный метод
  ... // вся остальная часть
  // определения класса
```

Подобную конструкцию, конечно, можно заставить работать, но выглядит она как-то подозрительно. Зачем помещать фабричный метод *внутри* класса `AbstractCurrency`? Это выглядит довольно сомнительно как минимум по двум причинам. Во-первых, если есть некая сумма в валюте (скажем, один доллар), то есть и возможность нарастить сумму в той же валюте, используя следующий код:

```
myDollar.make(100) // здесь еще сто!
```

В эпоху цветных ксероксов данный скрипт может стать заманчивым, но следует надеяться, что никто не сможет проделывать это слишком долго, не будучи пойманным за руку. Во-вторых, этот код, если у вас уже есть ссылка на объект `Currency`, позволяет создавать дополнительные объекты `Currency`. Но как получить первый объект данной валюты `Currency`? Вам понадобится другой метод создания, выполняющий практически ту же работу, что и `make`. То есть вы столкнулись со случаем дублирования кода, являющимся верным признаком «кода с дублированием».

Решение, конечно же, будет заключаться в перемещении абстрактного типа и фабричного класса за пределы класса `AbstractCurrency`. Нужно создать другой класс, содержащий класс `AbstractCurrency`, тип `Currency` и фабричный метод `make`.

Назовем этот класс `CurrencyZone`:

```
abstract class CurrencyZone:
  type Currency <: AbstractCurrency
  def make(x: Long): Currency
  abstract class AbstractCurrency:
    val amount: Long
    def designation: String
    override def toString = s"$amount $designation"
    def + (that: Currency): Currency =
```

```

    make(this.amount + that.amount)
  def * (x: Double): Currency =
    make((this.amount * x).toLong)

```

Пример конкретизации `CurrencyZone` — объект `US`, который можно определить следующим образом:

```

object US extends CurrencyZone:
  abstract class Dollar extends AbstractCurrency:
    def designation = "USD"

  type Currency = Dollar
  def make(x: Long) = new Dollar { val amount = x }

```

Здесь `US` — объект, расширяющий `CurrencyZone`. В нем определяется класс `Dollar`, являющийся подклассом `AbstractCurrency`. Следовательно, тип денежных единиц в этой зоне — доллар США, `US.Dollar`. Объект `US` также устанавливает, что тип `Currency` будет псевдонимом для `Dollar`, и предоставляет реализацию фабричного метода `make` для возвращения суммы в долларах.

Конструкция вполне работоспособна. Нужно лишь добавить несколько уточнений. Первое из них касается разменных монет. До сих пор каждая валюта измерялась в целых единицах: в долларах, евро или йенах. Но у большинства валют имеются разменные монеты, например, в США есть доллары и центы. Наиболее простой способ моделировать центы — использовать поле `amount` в `US.Currency`, представленное в центах, а не в долларах. Чтобы вернуться к доллару, будет полезно ввести в класс `CurrencyZone` поле `CurrencyUnit`, содержащее одну стандартную единицу в данной валюте:

```

abstract class CurrencyZone:
  ...
  val CurrencyUnit: Currency

```

Как показано в листинге 20.12, в объекте `US` могут быть определены величины `Cent`, `Dollar` и `CurrencyUnit`. Это определение объекта похоже на предыдущее, за исключением того, что в него добавлены три новых поля. Поле `Cent` представляет сумму в 1 `US.Currency`. Это объект, аналогичный одноцентовой монете. Поле `Dollar` представляет сумму в 100 `US.Currency`. Следовательно, объект `US` теперь определяет имя `Dollar` двумя способами. *Теперь* `Dollar`, определенный абстрактным внутренним классом по имени `Dollar`, представляет общее название валюты `Currency`, действительное в валютной зоне `US`. В отличие от этого *значение* `Dollar`, на которое ссылается `val`-поле по имени `Dollar`, представляет 1 доллар США, аналогичный однодолларовой купюре. Третье определение поля `CurrencyUnit` указывает на то, что стандартная

денежная единица в зоне США — доллар, `Dollar`, то есть значение `Dollar`, на которое ссылается поле, не является типом `Dollar`.

Метод `toString` в классе `AbstractCurrency` также нуждается в адаптации для восприятия разменных монет на счету. Например, сумма 10 долларов 23 цента должна выводиться как десятичное число: `10.23 USD`. Чтобы добиться этого результата, принадлежащий `AbstractCurrency` метод `toString` можно реализовать следующим образом:

```
override def toString =
  ((amount.toDouble / CurrencyUnit.amount.toDouble)
    .formatted(s"%.${decimals(CurrencyUnit.amount)}f")
    + " " + designation)
```

Здесь `formatted` является методом, доступным в Scala в нескольких классах, включая `Double`¹. Метод `formatted` возвращает строку, полученную в результате форматирования исходного `Double`, в отношении которой он был вызван, в соответствии со строкой форматирования, переданной ему в виде его правого операнда. Синтаксис строк форматирования, передаваемых методу `formatted`, аналогичен синтаксису, используемому для Java-метода `String.format`.

Листинг 20.12. Зона валюты США

```
object US extends CurrencyZone:
  abstract class Dollar extends AbstractCurrency:
    def designation = "USD"
    type Currency = Dollar
    def make(cents: Long) =
      new Dollar:
        val amount = cents
  val Cent = make(1)
  val Dollar = make(100)
  val CurrencyUnit = Dollar
```

Например, строка форматирования `%.2f` форматирует число с двумя знаками после точки. Строка форматирования, примененная в показанном ранее методе `toString`, собирается путем вызова метода `decimals` в отношении `CurrencyUnit.amount`. Данный метод возвращает число десятичных знаков десятичной степени за вычетом единицы. Например, `decimals(10)` — это 1, `decimals(100)` — это 2 и т. д. Метод `decimals` реализован в виде простой рекурсии:

¹ Чтобы обеспечить доступность метода `formatted`, в Scala используются обогащающие оболочки, рассмотренные в разделе 5.10.

```
private def decimals(n: Long): Int =
  if n == 1 then 0 else 1 + decimals(n / 10)
```

В листинге 20.13 показаны некоторые другие валютные зоны. В качестве еще одного уточнения к модели можно добавить свойство обмена валют. Сначала, как показано в листинге 20.14, можно создать объект `Converter`, содержащий применяемые обменные курсы валют. Затем к классу `AbstractCurrency` можно добавить метод обмена, `from`, который выполняет конвертацию из заданной исходной валюты в текущий объект `Currency`:

```
def from(other: CurrencyZone#AbstractCurrency): Currency =
  make(math.round(
    other.amount.toDouble * Converter.exchangeRate
      (other.designation)(this.designation)))
```

Листинг 20.13. Валютные зоны для Европы и Японии

```
object Europe extends CurrencyZone:
  abstract class Euro extends AbstractCurrency:
    def designation = "EUR"

    type Currency = Euro
    def make(cents: Long) =
      new Euro:
        val amount = cents

    val Cent = make(1)
    val Euro = make(100)
    val CurrencyUnit = Euro

object Japan extends CurrencyZone:
  abstract class Yen extends AbstractCurrency:
    def designation = "JPY"

    type Currency = Yen
    def make(yen: Long) =
      new Yen:
        val amount = yen

    val Yen = make(1)
    val CurrencyUnit = Yen
```

Листинг 20.14. Объект `converter` с отображением курсов обмена

```
object Converter:
  var exchangeRate =
    Map(
      "USD" -> Map("USD" -> 1.0, "EUR" -> 0.8498,
        "JPY" -> 1.047, "CHF" -> 0.9149),
      "EUR" -> Map("USD" -> 1.177, "EUR" -> 1.0,
        "JPY" -> 1.232, "CHF" -> 1.0765),
```

```

"JPY" -> Map("USD" -> 0.9554, "EUR" -> 0.8121,
             "JPY" -> 1.0, "CHF" -> 0.8742),
"CHF" -> Map("USD" -> 1.093, "EUR" -> 0.9289,
             "JPY" -> 1.144, "CHF" -> 1.0)
)

```

Метод `from` получает в качестве аргумента произвольную валюту. Это выражено его формальным типом параметра `CurrencyZone#AbstractCurrency`, который показывает, что переданный как `other` аргумент должен быть типа `AbstractCurrency` в некоторой произвольной и неизвестной валютной зоне `CurrencyZone`. Результат метода — перемножение суммы в другой валюте с курсом обмена между другой и текущей валютами¹.

Финальная версия класса `CurrencyZone` показана в листинге 20.15.

Листинг 20.15. Полный код класса `CurrencyZone`

```

abstract class CurrencyZone:

  type Currency <: AbstractCurrency
  def make(x: Long): Currency

  abstract class AbstractCurrency:

    val amount: Long
    def designation: String

    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
    def - (that: Currency): Currency =
      make(this.amount - that.amount)
    def / (that: Double) =
      make((this.amount / that).toLong)
    def / (that: Currency) =
      this.amount.toDouble / that.amount

    def from(other: CurrencyZone#AbstractCurrency): Currency =
      make(math.round(
        other.amount.toDouble * Converter.exchangeRate
          (other.designation)(this.designation)))

    private def decimals(n: Long): Int =
      if (n == 1) 0 else 1 + decimals(n / 10)

```

¹ Кстати, если вы полагаете, что сделка по японской йене будет неудачной, то курсы обмена валют основаны на числовых показателях в их `CurrencyZone`. Таким образом, 1.211 — курс обмена центов США на японскую йену.

```

    override def toString =
      ((amount.toDouble / CurrencyUnit.amount.toDouble)
        .formatted(s"%.${decimals(CurrencyUnit.amount)}f")
        + " " + designation)

  end AbstractCurrency

  val CurrencyUnit: Currency

end CurrencyZone

```

Класс можно опробовать, вводя команды в REPL Scala. Предполагается, что класс `CurrencyZone` и все конкретные объекты `CurrencyZone` определены в пакете `org.stairwaybook.currencies`. Сперва нужно импортировать `org.stairwaybook.currencies.*` в REPL. Затем можно будет выполнить ряд обменных операций с валютой:

```
scala> val yen = Japan.Yen.from(US.Dollar * 100)
val yen: Japan.Currency = 10470 JPY
```

```
scala> val euros = Europe.Euro.from(yen)
val euros: Europe.Currency = 85.03 EUR
```

```
scala> val dollars = US.Dollar.from(euros)
val dollars: US.Currency = 100.08 USD
```

Из факта получения почти такого же значения после трех конвертаций следует, что у нас весьма выгодные курсы обмена! Кроме того, можно нарастить значение в некоторой валюте:

```
scala> US.Dollar * 100 + dollars
res3: US.Currency = 200.08 USD
```

В то же время складывать суммы разных валют нельзя:

```
scala> US.Dollar + Europe.Euro
1 |US.Dollar + Europe.Euro
  |_____
  | Found:    (Europe.Euro : Europe.Currency)
  | Required: US.Currency(2)
  | where:    Currency is a type in object Europe which
  |           is an alias of Europe.Euro
  |           Currency(2) is a type in object US which is
  |           an alias of US.Dollar
```

Абстракция типов выполняет свою работу, не позволяя складывать два значения в разных единицах измерения (в данном случае валютах). Она мешает нам выполнять необоснованные вычисления. Неверные преобразования

между различными единицами могут показаться небольшими недочетами, но способны привести к весьма серьезным системным сбоям. Например, к аварии спутника Mars Climate Orbiter 23 сентября 1999 года, вызванной тем, что одна команда инженеров использовала метрическую систему мер, а другая — систему мер, принятую в Великобритании. Если бы единицы измерения были запрограммированы так же, как сделано с валютой в текущей главе, то данная ошибка была бы выявлена во время простого запуска кода на компиляцию. Вместо этого она стала причиной аварии космического аппарата после почти десятимесячного полета.

Резюме

В Scala предлагается рационально структурированная и самая общая поддержка объектно-ориентированной абстракции. При этом допускается применение не только абстрактных методов, но и значений, переменных и типов. В данной главе мы показали способы извлечь преимущества из использования абстрактных членов класса. С их помощью реализуется простой, но весьма эффективный принцип структурирования систем: все неизвестное при разработке класса нужно превращать в абстрактные члены. Тогда система типов задаст направление развитию вашей модели точно так же, как вы увидели в примере с валютой. И неважно, что именно будет неизвестно: тип, метод, переменная или значение. Все это в Scala можно объявить абстрактным.

21

Гивены

Поведение функции зачастую зависит от контекста, в котором она вызывается. Например, она может менять свое поведение в зависимости от контекстных данных, таких как системные свойства, разрешения безопасности, аутентифицированный пользователь, транзакция базы данных или заданное время ожидания. Функция также может зависеть от контекстного *поведения* — алгоритма, имеющего смысл в контексте, в котором эта функция вызывается. Например, функция сортировки может зависеть от алгоритма сравнения, который определяет, как упорядочивать сортируемые элементы. Разные контексты могут требовать разных алгоритмов сравнения.

Для предоставления функции такой контекстной информации и поведения существует множество приемов, однако в функциональном программировании решение традиционно сводится к одному: передавать все в качестве параметров. И хотя это вполне рабочий подход, у него есть недостаток: чем больше вы передаете функции данных и алгоритмов, тем более общей и полезной она становится, но при этом увеличивается количество аргументов, которые нужно указывать при каждом ее вызове. К сожалению, передача всего в виде параметров может быстро сделать ваш код повторяющимся и шаблонным.

В этой главе описываются *контекстные параметры*, которые часто называют гивенами (*given*). Они позволяют вам опускать некоторые аргументы при вызове функций, давая возможность компилятору подставить подходящие значения для каждого контекста в зависимости от типа.

21.1. Как это работает

Компилятор иногда меняет `someCall(a)` на `someCall(a)(b)` или `SomeClass(a)` на `new SomeClass(a)(b)`, добавляя тем самым один или несколько недостающих списков параметров, чтобы сделать вызов функции завершенным. Предоставляются не отдельные параметры, а целые их каррированные списки. Например, если недостающий список параметров `someCall` состоит из трех значений, компилятор может подставить `someCall(a)(b, c, d)` вместо `someCall(a)`. В этом случае подставленные идентификаторы, такие как `b`, `c` и `d` в `(b, c, d)`, должны быть помечены как заданные (*given*) в месте их определения, а сам список параметров в определении `someCall` или `someClass` должен начинаться с `using`.

Представьте, к примеру, что у вас есть множество методов, принимающих приглашение командной строки (например, "\$ " или "> "), которое предпочитает текущий пользователь. Вы можете сократить количество шаблонного кода, сделав запрос контекстным параметром. Для начала нужно создать специальный тип, инкапсулирующий строку с предпочитаемым приглашением:

```
class PreferredPrompt(val preference: String)
```

Далее нужно отредактировать каждый метод, который принимает приглашение, заменив параметр отдельным списком параметров с ключевым словом `using`. Например, у следующего объекта `Greeter` есть метод `greet`, который принимает `PreferredPrompt` в качестве контекстного параметра:

```
object Greeter:
  def greet(name: String)(using prompt: PreferredPrompt) =
    println(s"Welcome, $name. The system is ready.")
    println(prompt.preference)
```

Чтобы компилятор мог неявно подставлять контекстный параметр, вы должны определить `given`-экземпляр ожидаемого типа (в данном случае `PreferredPrompt`) с использованием ключевого слова `given`. Это можно сделать в объекте настроек, как показано далее:

```
object JillsPrefs:
  given prompt: PreferredPrompt =
    PreferredPrompt("Your wish> ")
```

Теперь компилятор может автоматически подставлять этот экземпляр `PreferredPrompt`, но только при условии, что тот находится в области видимости:

```
scala> Greeter.greet("Jill")
1 |Greeter.greet("Jill")
  |^
  |no implicit argument of type PreferredPrompt was found
  |for parameter prompt of method greet in object Greeter
```

Если сделать этот объект доступным, он будет использоваться для предоставления недостающего списка параметров:

```
scala> import JillsPrefs.prompt
scala> Greeter.greet("Jill")
Welcome, Jill. The system is ready.
Your wish>
```

Поскольку приглашение командной строки объявлено в качестве контекстного параметра, оно не скомпилируется, если вы попытаетесь передать аргумент как обычно, явным образом:

```
scala> Greeter.greet("Jill")(JillsPrefs.prompt)
1 | Greeter.greet("Jill")(JillsPrefs.prompt)
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  | method greet in object Greeter does not take more
  | parameters
```

Вместо этого вам следует указать, что вы хотите явно подставить контекстный параметр, используя в момент вызова ключевое слово `using`, как показано ниже:

```
scala> Greeter.greet("Jill")(using JillsPrefs.prompt)
Welcome, Jill. The system is ready.
Your wish>
```

Обратите внимание на то, что ключевое слово `using` относится не к отдельным параметрам, а ко всему списку. В листинге 21.1 показан пример, в котором второй список параметров метода `greet` из объекта `Greeter` (который опять же помечен как `using`) состоит из двух элементов: `prompt` (типа `PreferredPrompt`) и `drink` (типа `PreferredDrink`).

Листинг 21.1. Неявный список с несколькими параметрами

```
class PreferredPrompt(val preference: String)
class PreferredDrink(val preference: String)

object Greeter:
  def greet(name: String)(using prompt: PreferredPrompt,
    drink: PreferredDrink) =
    println(s"Welcome, $name. The system is ready.")
    print("But while you work, ")
    println(s"why not enjoy a cup of ${drink.preference}?" )
    println(prompt.preference)

object JoesPrefs:
  given prompt: PreferredPrompt =
    PreferredPrompt("relax> ")
  given drink: PreferredDrink =
    PreferredDrink("tea")
```

Объект-одиночка объявляет два `given`-экземпляра: `prompt` типа `PreferredPrompt` и `drink` типа `PreferredDrink`. Но, как и прежде, они не будут использоваться для подстановки недостающего списка параметров в `greet`, если они находятся вне области видимости:

```
scala> Greeter.greet("Joe")
1 | Greeter.greet("Joe")
  | ~
  | no implicit argument of type PreferredPrompt was found
  | for parameter prompt of method greet in object Greeter
```

Вы можете сделать оба `given`-экземпляра из листинга 21.1 доступными с помощью инструкции `import`:

```
scala> import JoesPrefs.{prompt, drink}
```

Поскольку и `prompt`, и `drink` теперь находятся в области видимости в качестве отдельных идентификаторов, вы можете использовать их для явного предоставления последнего списка параметров:

```
scala> Greeter.greet("Joe")(using prompt, drink)
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
relax>
```

И поскольку ваши контекстные параметры теперь удовлетворяют всем правилам, вы можете также позволить компилятору Scala подставить `prompt` и `drink` автоматически, целиком опустив весь список параметров:

```
scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
relax>
```

Одной из особенностей предыдущих примеров является то, что мы не использовали `String` в качестве типа для `prompt` или `drink`, хотя в итоге оба этих значения предоставили именно `String` через свои поля `preference`. Поскольку компилятор выбирает контекстные параметры путем сопоставления типов параметров и типов `given`-экземпляров, контекстные параметры должны иметь достаточно редкие, или особенные типы, которые делают случайное совпадение маловероятным. Например, типы `PreferredPrompt` и `PreferredDrink` в листинге 21.1 были определены исключительно для контекстных параметров. В результате `given`-экземпляры этих типов, скорее всего, не будут существовать, если только они не предназначены для использования в качестве контекстных параметров для таких методов, как `greet`.

21.2. Параметризованные given-типы

Контекстные параметры, наверное, чаще всего используются для предоставления информации о типе, *явно* указанном в предыдущем списке параметров, подобно классам типов (type class) в Haskell. Это важный механизм достижения *специального полиморфизма* (ad hoc polymorphism) при написании функций в Scala: ваши функции можно применять к значениям с подходящими типами, но при использовании для значений любых других типов код не скомпилируется. Представьте, к примеру, двухстрочную сортировку вставками, показанную в листинге 14.1. Это определение `isort` работает только для списка целых чисел. Чтобы сортировать списки других типов, вам нужно сделать тип аргумента `isort` более общим. Для этого первым делом можно ввести параметр типа, `T`, и подставить его вместо `Int` в параметре типа `List`:

```
// Не компилируется
def isort[T](xs: List[T]): List[T] =
  if xs.isEmpty then Nil
  else insert(xs.head, isort(xs.tail))

def insert[T](x: T, xs: List[T]): List[T] =
  if xs.isEmpty || x <= xs.head then x :: xs
  else xs.head :: insert(x, xs.tail)
```

Но, попытавшись скомпилировать `isort` после внесения этого изменения, вы получите от компилятора следующее сообщение:

```
6 | if xs.isEmpty || x <= xs.head then x :: xs
  |                   ^^^^^
  |                   value <= is not a member of T, ...
```

Если класс `Int` определяет метод `<=`, устанавливающий, является ли одно целое число меньше или равно другому, то для других типов могут потребоваться альтернативные стратегии сравнения или же их и вовсе нельзя сравнивать. Чтобы метод `isort` мог работать со списками, элементы которых имеют типы, отличные от `Int`, ему нужно предоставить чуть больше информации, позволяющей определить способ сравнения двух элементов.

Чтобы решить эту проблему, методу `isort` можно передать функцию «меньше или равно», подходящую для типа `List`. Эта функция должна принимать два экземпляра `T` и возвращать значение `Boolean`, указывающее на то, является ли первый экземпляр `T` меньше или равным второму:

```
def isort[T](xs: List[T])(lteq: (T, T) => Boolean): List[T] =
  if xs.isEmpty then Nil
```

```

    else insert(xs.head, isort(xs.tail)(lteq))(lteq)

def insert[T](x: T, xs: List[T])
  (lteq: (T, T) => Boolean): List[T] =
  if xs.isEmpty || lteq(x, xs.head) then x :: xs
  else xs.head :: insert(x, xs.tail)(lteq)

```

Теперь вместо `<=` вспомогательная функция `insert` использует параметр `lteq` для сравнения двух элементов во время сортировки. Это позволяет сортировать список любого типа `T`, главное — предоставить методу `isort` функцию сравнения, которая подходит для `T`. Например, с помощью этой версии `isort` можно сортировать списки `Int`, `String` и класса `Rational`, представленного в листинге 6.5:

```

isort(List(4, -10, 10))((x: Int, y: Int) => x <= y)
// List(-10, 4, 10)

isort(List("cherry", "blackberry", "apple", "pear"))
  ((x: String, y: String) => x.compareTo(y) <= 0)
// List(apple, blackberry, cherry, pear)

isort(List(Rational(7, 8), Rational(5, 6), Rational(1, 2)))
  ((x: Rational, y: Rational) =>
    x.numer * y.denom <= x.denom * y.numer)
// List(1/2, 5/6, 7/8)

```

Как уже описывалось в разделе 14.10, компилятор `Scala` последовательно определяет типы параметров в каждом списке, продвигаясь слева направо. Таким образом, он может определить типы `x` и `y`, указанные во втором списке параметров, исходя из типа элемента `T` экземпляра `List[T]`, переданного в первом списке параметров:

```

isort(List(4, -10, 10))((x, y) => x <= y)
// List(-10, 4, 10)

isort(List("cherry", "blackberry", "apple", "pear"))
  ((x, y) => x.compareTo(y) < 1)
// List(apple, blackberry, cherry, pear)

isort(List(Rational(7, 8), Rational(5, 6), Rational(1, 2)))
  ((x, y) => x.numer * y.denom <= x.denom * y.numer)
// List(1/2, 5/6, 7/8)

```

Теперь функция `isort` полезна в более общем смысле, однако за эту обобщенность приходится платить потерей лаконичности: при каждом вызове необходимо указывать функцию сравнения, которую определение `isort` теперь должно передавать каждому рекурсивному вызову `isort`, а также

каждому вызову вспомогательной функции `insert`. Эта версия `isort` больше не является простым выражением сортировки, как прежде.

Вы можете сделать более лаконичной как реализацию метода `isort`, так и его вызовы, если оформите функцию сравнения в виде контекстного параметра. Вы *могли бы* использовать контекстный параметр `(Int, Int) => Boolean`, но этот тип слишком общий, что делает его не самым оптимальным решением. У вашей программы, к примеру, может быть много функций, которые принимают целочисленные параметры и возвращают логическое значение, но при этом не имеют ничего общего с сортировкой. Поскольку поиск `given`-значений происходит по типу, вы должны позаботиться о том, чтобы тип вашего `given`-экземпляра выражал его назначение.

Определение типов с определенным назначением, таким как сортировка, обычно является хорошим решением, но, как упоминалось ранее, некоторые типы становятся особенно полезными при использовании контекстных параметров. Помимо гарантии использования подходящего `given`-экземпляра, тщательно определенные типы могут помочь вам более ясно выразить ваши намерения. Это позволяет вам развивать ваши программы постепенно, расширяя типы за счет дополнительного функционала, но не нарушая при этом существующие между ними контракты. Вы можете определить тип, чтобы выбрать, в каком порядке должны размещаться два элемента:

```
trait Ord[T]:
  def compare(x: T, y: T): Int
  def lteq(x: T, y: T): Boolean = compare(x, y) < 1
```

Этот трейт реализует функцию «меньше или равно» в виде более общего абстрактного метода `compare`. Контракт этого метода состоит в том, что он возвращает `0`, если два параметра равны, положительное целое число, если первый параметр больше второго, и отрицательное целое число, если второй параметр больше первого. Теперь, имея это определение, вы можете указать стратегию сравнения для `T`, используя `Ord[T]` в качестве контекстного параметра, как показано в листинге 21.2.

Листинг 21.2. Контекстные параметры, передаваемые с помощью `using`

```
def isort[T](xs: List[T])(using ord: Ord[T]): List[T] =
  if xs.isEmpty then Nil
  else insert(xs.head, isort(xs.tail))

def insert[T](x: T, xs: List[T])
  (using ord: Ord[T]): List[T] =
  if xs.isEmpty || ord.lteq(x, xs.head) then x :: xs
  else xs.head :: insert(x, xs.tail)
```

Как уже описывалось ранее, чтобы параметры можно было передавать неявно, перед ними нужно указать `using`. После этого вам больше не нужно предоставлять эти параметры вручную при вызове функции: если доступно значение подходящего типа, компилятор *возьмет* его и передаст вашей функции. Чтобы сделать значение `given`-экземпляром типа, его следует объявить с помощью ключевого слова `given`.

Хорошим местом для размещения `given`-экземпляров, представляющих «естественный» вариант использования типа, такой как сортировка целых чисел в порядке возрастания, является объект-компаньон «вовлеченного» типа. Например, естественный `given`-экземпляр `Ord[Int]` можно было бы разместить в объекте-компаньоне для `Ord` или `Int` — двух типов, «фигурирующих» в `Ord[Int]`. Если компилятор не найдет `given`-экземпляр `Ord[Int]` в лексической области видимости, он проведет дополнительный поиск в этих двух объектах-компаньонах. Поскольку компаньон `Int` не подлежит изменению, лучшим выбором является компаньон `Ord`:

```
object Ord:
  // (Пока что не является устоявшимся решением)
  given intOrd: Ord[Int] =
    new Ord[Int]:
      def compare(x: Int, y: Int) =
        if x == y then 0 else if x > y then 1 else 1
```

Все примеры `given`-объявлений, показанные до сих пор в этой главе, называются *псевдонимными* (*alias*). Имя по левую сторону от знака равенства является псевдонимом значения, указанного справа. Поскольку при объявлении псевдонимного `given`-экземпляра справа от знака равенства зачастую определяют анонимный экземпляр трейта или класса, Scala предлагает сокращенный синтаксис, который позволяет подставить вместо знака равенства и «имени нового класса» ключевое слово `with`¹. В листинге 21.3 показано более компактное определение `intOrd`.

Листинг 21.3. Объявление естественного `given`-экземпляра в компаньоне

```
object Ord:
  // Общепринятое решение
  given intOrd: Ord[Int] with
    def compare(x: Int, y: Int) =
      if x == y then 0 else if x > y then 1 else 1
```

¹ Этот способ использования `with` отличается от того, который был описан в главе 11 и предназначался для объединения трейтов.

Теперь, когда в объекте `Ord` имеется given-экземпляр `Ord[Int]`, сортировка с использованием `isort` снова становится лаконичной:

```
isort(List(10, 2, -10))
// List(-10, 2, 10)
```

Если опустить второй параметр `isort`, компилятор начнет искать для него заданное значение с учетом его типа. Если речь идет о сортировке значений `Int`, этим типом будет `Ord[Int]`. Вначале компилятор поищет given-экземпляр `Ord[Int]` в лексической области видимости, и, если его там не обнаружится, он пройдет по объектам-компаньонам вовлеченных типов `Ord` и `Int`. Поскольку в листинге 21.3 заданное значение `intOrd` имеет явно указанный тип, компилятор подставит `intOrd` вместо недостающего списка параметров.

Для сортировки строк достаточно предоставить given-экземпляр для параметра, предназначенного для сравнения строковых значений:

```
// Добавлено в объект Ord
given stringOrd: Ord[String] with
  def compare(s: String, t: String) = s.compareTo(t)
```

Теперь, когда в компаньоне `Ord` определен given-экземпляр `Ord[String]`, вы можете использовать `isort` для сортировки списков строк:

```
isort(List("mango", "jackfruit", "durian"))
// List(durian, jackfruit, mango)
```

Если заданное объявление не принимает параметризованные значения, given-экземпляр инициализируется при первом к нему обращении, что похоже на ленивые значения. Эта инициализация проводится потокобезопасным образом. Если же given-экземпляр принимает параметры, он создается заново при каждом обращении, подобно тому как ведет себя `def`. Действительно, компилятор Scala преобразует given-экземпляры в `val` или `def`, дополнительно делая их доступными для параметров `using`.

21.3. Анонимные given-экземпляры

Заданное объявление можно считать частным случаем ленивого `val` или `def`, однако оно обладает одной важной особенностью. При объявлении `val`, к примеру, нужно задать выражение, указывающее на значение `val`:

```
val age = 42
```

В этом выражении компилятор должен определить тип `age`. Поскольку для инициализации `age` используется значение `42`, которое, как известно компилятору, имеет тип `Int`, для `age` будет выбран тот же тип. Вы фактически предоставляете *выражение*, `age`, а компилятор определяет его тип, `Int`.

С контекстными параметрами все наоборот: вы предоставляете тип, а компилятор синтезирует выражение, которое его представляет, с учетом доступных `given`-экземпляров, и затем использует это выражение автоматически, когда этот тип необходим. Это называется *определением выражения* (чтобы не путать с определением типа).

Поскольку компилятор ищет `given`-экземпляр по типу и зачастую на него вообще не нужно ссылаться, вы можете объявить свое `given`-значение анонимно. Вместо кода

```
given revIntOrd: Ord[Int] with
  def compare(x: Int, y: Int) =
    if x == y then 0 else if x > y then -1 else 1

given revStringOrd: Ord[String] with
  def compare(s: String, t: String) = -s.compareTo(t)
```

можно написать

```
given Ord[Int] with
  def compare(x: Int, y: Int) =
    if x == y then 0 else if x > y then -1 else 1

given Ord[String] with
  def compare(s: String, t: String) = -s.compareTo(t)
```

Компилятор автоматически синтезирует имена для этих анонимных `given`-экземпляров. Вместо второго параметра функции `isort` будет подставлено это синтезированное значение, которое затем станет доступным внутри функции. Таким образом, если вам нужно, чтобы `given`-экземпляр был неявно предоставлен в качестве контекстных параметров, вам не нужно объявлять для него выражение.

21.4. Параметризованные `given`-экземпляры в виде классов типов

Вы можете предоставить гивен `Ord[T]` для любого типа `T`, который вам нужно сортировать. Например, вы могли бы сделать доступными для сортировки экземпляры класса `Rational`, показанного в листинге 6.5, определив

гивен `Ord[Rational]`. Поскольку объект-компаньон `Rational` представляет естественный способ сортировки рациональных чисел, он послужит подходящим местом размещения этого given-экземпляра:

```
object Rational:
  given rationalOrd: Ord[Rational] with
    def compare(x: Rational, y: Rational) =
      if x.numer * y.denom < x.denom * y.numer then -1
      else if x.numer * y.denom > x.denom * y.numer then 1
      else 0
```

Теперь вы можете сортировать списки элементов `Rational`:

```
isort(List(Rational(4, 5), Rational(1, 2), Rational(2, 3)))
// List(1/2, 2/3, 4/5)
```

Согласно принципу подстановки Лисков объект можно заменить его подтипом без изменения желаемых свойств программы. Этот принцип лежит в основе отношений между подтипом и супертипом, характерных для объектно-ориентированного программирования. В последней версии `isort`, показанной в листинге 21.2, все выглядит так, что вы можете подставить вместо списка строк список значений типа `Int` или `Rational`, и `isort` продолжит работать так, как ожидалось. Это является признаком того, что `Int`, `Rational` и `String` могут иметь общий «сортируемый» супертип¹. Однако это не так. Более того, определение *нового* супертипа для типов вроде `Int` или `String` было бы невозможным, так как они являются частью стандартных библиотек `Java` и `Scala`.

Предоставление given-экземпляров `Ord[T]` делает конкретные типы `T` частью множества «типов, доступных для сортировки», несмотря на отсутствие какого-либо общего сортируемого супертипа. Это множество называется *классом типов* (typeclass)². Например, на данном этапе класс типов `Ord` состоит из трех типов: `Int`, `String` и `Rational`; это множество типов `T`, для которых существуют гивены `Ord[T]`. В главе 23 даются дополнительные примеры класса типов. Поскольку реализация `isort`, представленная в листинге 21.2, принимает контекстный параметр типа `Ord[T]`, это пример специального полиморфизма: `isort` может сортировать списки

¹ Таким сортируемым типом является трейт `Ordered`, описанный в разделах 11.2 и 18.7.

² В термине «класс типов» слово «класс» употребляется не в объектно-ориентированном смысле. Оно означает множество (или класс в привычном смысле этого слова) типов, для которого существуют экземпляры определенного (объектно-ориентированного) класса или трейта.

определенных типов `T`, для которых существуют гивены `Ord[T]`, и не компилируется ни для каких других типов. Специальный полиморфизм с классами типов — это важная и распространенная методика в идиоматическом программировании на Scala.

В стандартной библиотеке Scala есть готовые классы типов для различных целей, таких как определение равенства или определение порядка размещения элементов при сортировке. Класс типов `Ord`, который используется в этой главе, является частично переписанной реализацией класса типов `math.Ordering` из состава Scala. В библиотеке Scala определены экземпляры `Ordering` для таких распространенных типов, как `Int` и `String`.

В листинге 21.4 показана версия `isort`, в которой используется класс типов `Ordering` из Scala. Обратите внимание на то, что у контекстного параметра в этой версии нет имени. Это просто ключевое слово `using`, за которым идет тип параметра, `Ordering[T]`. Такие параметры называются *анонимными*. Поскольку данный параметр используется неявно только внутри функции (он неявно передается функциям `insert` и `isort`), Scala не требует, чтобы ему назначили имя.

В качестве еще одного примера специального полиморфизма можно снова взять метод `orderedMergeSort` из листинга 18.11. Этот метод может сортировать списки любого типа `T` при условии, что `T` — подтип `Ordered[T]`. Это называется *полиморфизмом подтипов* (или *подтипизацией*), и, как проиллюстрировано в разделе 18.7, верхняя граница `Ordered[T]` означает, что `orderedMergeSort` нельзя использовать для списков с элементами `Int` или `String`. Для сравнения: вы можете сортировать списки `Int` и `String` с помощью функции `msort` из листинга 21.5, так как нужный ей тип `Ordering[T]` формирует *отдельную* от `T` иерархию. И хотя тип `Int` нельзя изменить так, чтобы он наследовал `Ordering[T]`, вы можете определить и предложить гивен `Ordering[Int]`.

Листинг 21.4. Функция сортировки вставками, использующая `Ordering`

```
def isort[T](xs: List[T])(using Ordering[T]): List[T] =
  if xs.isEmpty then Nil
  else insert(xs.head, isort(xs.tail))

def insert[T](x: T, xs: List[T])
  (using ord: Ordering[T]): List[T] =
  if xs.isEmpty || ord.lteq(x, xs.head) then x :: xs
  else xs.head :: insert(x, xs.tail)
```

Функции `isort` (см. листинг 21.4) и `msort` (см. листинг 21.5) являются примерами использования контекстного параметра с целью предоставле-

ния дополнительной информации о типе, явно упомянутом в предыдущем списке параметров. В частности, контекстный параметр типа `Ordering[T]` предоставляет больше информации о типе `T` — в данном случае как сортировать его экземпляры. Тип `T` упоминается в `List[T]`, типе параметра `xs`, который фигурирует в предыдущем списке параметров. Поскольку `xs` всегда нужно явно указывать при любом вызове `isort` или `msort`, тип `T` будет известен на этапе компиляции, что позволит определить, доступен ли гивен типа `Ordering[T]`. Если да, то компилятор автоматически передаст второй список параметров.

Листинг 21.5. Функция сортировки слиянием, использующая `Ordering`

```
def msort[T](xs: List[T])(using ord: Ordering[T]): List[T] =
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if ord.lt(x, y) then x :: merge(xs1, ys)
        else y :: merge(xs, ys1)

  val n = xs.length / 2
  if n == 0 then xs
  else
    val (ys, zs) = xs.splitAt(n)
    merge(msort(ys), msort(zs))
```

21.5. Импорт гивенов

Предоставляя `given`-значения в объекте-компаньоне класса, вы делаете их доступными для поиска. Это хороший подход для организации разумного поведения по умолчанию, которое пользователи, скорее всего, хотели бы получать в исполнении своего гивена — например, естественное упорядочение элементов типа. В противном случае гивены рекомендуется размещать в объектах-одиночках, которые *не* будут найдены автоматически; пользователи, которым нужны эти гивены, должны их предварительно импортировать. Чтобы вам было легче понять, откуда был взят гивен, в Scala предусмотрен специальный синтаксис импортирования.

Допустим, вы хотите определить объект, как показано в листинге 21.6. В главе 12 вы уже видели, как импортировать все `val` и `def` с помощью подстановочного знака. Однако обычный синтаксис импорта с подстановочным знаком не позволяет импортировать гивены:

```
// Импортирует только favoriteColor и favoriteFood
import TomsPrefs.*
```

Это выражение импортирует все члены `TomsPrefs`, кроме гивенов. Чтобы импортировать гивены, можно явно указать их имена.

Листинг 21.6. Объект настроек

```
object TomsPrefs:
  val favoriteColor = "blue"
  def favoriteFood = "steak"
  given prompt: PreferredPrompt =
    PreferredPrompt("enjoy> ")
  given drink: PreferredDrink =
    PreferredDrink("red wine")
  given prefPromptOrd: Ordering[PreferredPrompt] with
    def compare(x: PreferredPrompt, y: PreferredPrompt) =
      x.preference.compareTo(y.preference)
  given prefDrinkOrd: Ordering[PreferredDrink] with
    def compare(x: PreferredDrink, y: PreferredDrink) =
      x.preference.compareTo(y.preference)

import TomsPrefs.prompt // импортирует prompt
```

Если вы хотите импортировать все гивены, используйте *специальный подстановочный шаблон*:

```
// импортирует prompt, drink, prefPromptOrd и prefDrinkOrd
import TomsPrefs.given
```

Поскольку имя гивена зачастую не фигурирует явно в исходном коде, а вместо этого используется только его тип, данный механизм позволяет импортировать гивены по их типу¹:

```
// импортирует drink, так как это гивен PreferredDrink
import TomsPrefs.{given PreferredDrink}
```

Если хотите импортировать `prefPromptOrd` и `prefDrinkOrd` по типу, можете явно указать их тип после ключевого слова `given`:

```
// импортирует prefPromptOrd и prefDrinkOrd
import TomsPrefs.{given Ordering[PreferredPrompt],
  given Ordering[PreferredDrink]}
```

¹ Анонимные гивены, описанные в разделе 21.3, не имеют имен, поэтому их можно импортировать только по типу или с помощью подстановочного шаблона.

В качестве альтернативы вместо параметра типа можно подставить знак вопроса (?), чтобы импортировать сразу оба гивена:

```
// импортирует prefPromptOrd и prefDrinkOrd
import TomsPrefs.{given Ordering[?]}
```

21.6. Правила для контекстных параметров

Контекстными являются параметры, определенные в инструкции `using`. Компилятору позволено вставлять их для исправления любых ошибок, вызванных отсутствием списков параметров. Например, если вызов `someCall(a)` не проходит проверку типов, компилятор может поменять его на `someCall(a)(b)`, где недостающий список параметров помечен как `given`, а `b` является гивеном¹. Это изменение может исправить программу так, чтобы она успешно проверяла типы и работала корректно. Если явная передача `b` сводится к шаблонному коду, то отсутствие этого параметра в исходном коде можно считать уточнением.

В целом, контекстные параметры подчиняются следующим правилам.

Правило разметки: доступны только определения, помеченные как `given`. Ключевое слово `given` указывает, какие определения компилятор может использовать в качестве контекстных параметров. Вот пример определения гивена:

```
given amysPrompt: PreferredPrompt = PreferredPrompt("hi> ")
```

Компилятор заменит `greet("Amy")` на `greet("Amy")(amysPrompt)`, только если пометить `amysPrompt` как `given`. Таким образом удастся избежать путаницы, которая бы возникла, если бы компилятор выбрал случайные значения, оказавшиеся в области видимости, и вставил их автоматически. Компилятор выбирает только среди тех определений, которые вы явно пометили как `given`.

Правило видимости: вставленный `given`-экземпляр должен быть доступен в виде единого идентификатора или связан с типом, который фигурирует в типе параметра. Компилятор Scala рассматривает только видимые `given`-экземпляры. Чтобы `given`-экземпляр был доступным, его необходимо каким-то образом сделать видимым. Более того, за одним исключением, `given`-экземпляр должен быть видим в лексической области видимости *в качестве единого идентификатора*. Компилятор не вставит `given`-экземпляр вида

¹ Когда компилятор выполняет это переопределение внутри, перед явно переданным параметром не нужно указывать `using`.

`prefslib.AmysPrefs.amysPrompt`. Например, он не развернет `greet("Amy")` в `greet("Amy")(prefslib.AmysPrefs.amysPrompt)`. Если вы хотите сделать `prefslib.AmysPrefs.amysPrompt` `given`-экземпляром, вам нужно его импортировать, что делает его доступным в качестве единого идентификатора. После этого у компилятора будет возможность применять его с помощью этого идентификатора в виде `greet("Amy")(amysPrompt)`. На самом деле многие библиотеки включают объект `Preamble` с рядом полезных `given`-экземпляров, для доступа к которым достаточно одного выражения `import Preamble.given`¹.

У правила единого идентификатора есть одно исключение. Если компилятор не находит в лексической области видимости подходящий `given`-экземпляр, он в качестве запасного варианта ищет определения `given`-экземпляров в объекте-компаньоне всех типов, фигурирующих в типе контекстного параметра. Например, если вы попытаетесь вызвать метод, не указав явно аргумент для контекстного параметра типа `Ordering[Rational]`, компилятор начнет искать в объектах-компаньонах `Ordering`, `Rational` и их супертипов. В связи с этим такие `given`-экземпляры можно упаковывать в объекте-компаньоне любого из этих классов, `Ordering` или `Rational`. Но, поскольку `Ordering` является частью стандартной библиотеки, лучшим местом размещения `given`-экземпляра будет объект-компаньон `Rational`:

```
object Rational:
  given rationalOrdering: Ordering[Rational] with
    def compare(x: Rational, y: Rational) =
      if x.numer * y.denom < x.denom * y.numer then 1
      else if x.numer * y.denom > x.denom * y.numer then 1
      else 0
```

В данном случае `given`-экземпляр `rationalOrdering` считается *привязанным* к типу `Rational`. Компилятор будет находить его каждый раз, когда ему нужно синтезировать контекстный параметр типа `Ordering[Rational]`. Вам не нужно импортировать `given`-экземпляр отдельно в своей программе.

Правило видимости помогает анализировать код с точки зрения модулей. При чтении кода вас должны интересовать только те аспекты других файлов, которые либо импортируются, либо явно упоминаются с использованием полностью определенного имени. Это преимущество важно для `given`-экземпляров не меньше, чем для явно написанного кода. Если

¹ Импорт `Preamble.given` делает доступными в лексической области видимости синтезированные имена любых анонимных `given`-экземпляров, объявленных в `Preamble`, и они будут иметь вид единых экземпляров.

бы given-экземпляры действовали на уровне системы, то для того, чтобы разобраться в файле, вам нужно было бы знать о каждом их определении в программе!

Явные определения имеют повышенный приоритет: если в коде проводится проверка типов в том виде, в котором они написаны, компилятор не пытается использовать given-экземпляры. Компилятор не станет менять код, который уже работает. Из этого правила следует, что неявно предоставленные заданные идентификаторы всегда можно заменить явными, используя `using`; это сделает код более длинным, но вместе с тем его прояснит. Вы можете искать баланс между этими двумя свойствами в каждом отдельном случае. Если код кажется повторяющимся и многословным, контекстные параметры могут сделать его менее однообразным. Если код кажется настолько сжатым, что его сложно понять, вы можете явно передавать аргументы для контекстных параметров с помощью `using`. Так или иначе, количество контекстных параметров, которые позволено подставлять компилятору, — это вопрос стиля.

Назначение имени гивену

Гивены имеют произвольные имена. Имя гивена важно лишь в двух ситуациях: если вы хотите явно его записать при передаче, используя ключевое слово `using`, и когда вам нужно определить, какие гивены доступны в том или ином месте программы. Чтобы проиллюстрировать вторую ситуацию, представьте, что вы хотите воспользоваться гивеном `prefPromptOrd` из объекта-одиночки `TomsPrefs`, представленного в листинге 21.6, но при этом вам не нравится предпочитаемый Томом напиток, который используется в `prefDrinkOrd`. В этом случае вы можете импортировать лишь один из этих гивенов:

```
import TomsPrefs.prefPromptOrd
```

В этом примере наличие имен у гивенов оказалось полезным, позволив вам выборочно импортировать только один из них.

21.7. Когда подходит сразу несколько гивенов

Иногда бывает так, что в области видимости есть несколько гивенов, каждый из которых является подходящим. Обычно в таких случаях Scala отказывается заменять контекстный параметр. Контекстные параметры

работают хорошо, когда опущенный список параметров является совершенно очевидным и шаблонным. Если же нам подходит сразу несколько гивенов, выбор получается не настолько простым. Рассмотрим в качестве примера листинг 21.7.

Листинг 21.7. Несколько гивенов

```
class PreferredPrompt(val preference: String)

object Greeter:
  def greet(name: String)(using prompt: PreferredPrompt) =
    println(s"Welcome, $name. The system is ready.")
    println(prompt.preference)

  object JillsPrefs:
    given jillsPrompt: PreferredPrompt =
      PreferredPrompt("Your wish> ")

  object JoesPrefs:
    given joesPrompt: PreferredPrompt =
      PreferredPrompt("relax> ")
```

Гивен `PreferredPrompt` предоставляется сразу двумя объектами, показанными в листинге 21.7: `JillsPrefs` и `JoesPrefs`. Если импортировать и тот и другой, в лексической области видимости будет два разных идентификатора, `jillsPrompt` и `joesPrompt`:

```
scala> import JillsPrefs.jillsPrompt

scala> import JoesPrefs.joesPrompt
```

Теперь, если вы попытаетесь вызвать `Greeter.greet`, компилятор откажется выбирать между двумя подходящими гивенами.

```
scala> Greeter.greet("Who's there?")
1 | Greeter.greet("Who's there?")
  | ^
  | ambiguous implicit arguments: both given instance
  | joesPrompt in object JoesPrefs and given instance
  | jillsPrompt in object JillsPrefs match type
  | PreferredPrompt of parameter prompt of method
  | greet in object Greeter
```

Здесь мы имеем дело с настоящей двусмысленностью. Джилл и Джо предпочитают совершенно разные приглашения командной строки. В данном случае программист должен явно указать, какую из них следует использовать. В ситуациях, когда может подойти сразу несколько гивенов, компилятор отказывается выбирать — разве что один из них является более

конкретным, чем другие. Это полный аналог перегрузки методов. Если вы попытаетесь вызвать `foo(null)`, а в вашем коде есть две разные перегруженные версии `foo`, которые принимают `null`, компилятор откажется делать выбор и выведет сообщение о том, что вызов метода имеет неоднозначную цель.

Однако если один из гивенов безусловно является *более конкретным*, чем другие, компилятор выберет именно его. Идея в следующем: если есть причина полагать, что программист в любом случае предпочтет один из гивенов, его не нужно заставлять записывать это явно. В конце концов, перегрузка методов имеет такое же послабление. Возвращаясь к предыдущему примеру, если один из доступных методов `foo` принимает `String`, остальные — `Any`, компилятор выберет версию со `String`. Она явно более конкретная.

Строго говоря, один гивен *конкретнее* другого, если выполняется одно из следующих условий:

- тип первого является подтипом второго;
- внешний класс первого наследует внешний класс второго.

Если у вас есть два подходящих гивена, один из которых явно должен быть выбран в первую очередь, вы можете разместить другой экземпляр в трейте `LowPriority`, а свой первостепенный выбор — в подклассе или подобъекте этого трейта. Компилятор выберет первый экземпляр, если тот подходит, даже если второй был бы неоднозначным вариантом. Если гивен с более низким приоритетом подходит, а с более высоким — нет, компилятор сделает выбор в пользу первого.

21.8. Отладка гивенов

Гивены являются мощным инструментом Scala, с которым тем не менее может быть сложно управиться. Этот раздел содержит несколько советов об отладке гивенов.

Иногда не совсем понятно, почему компилятор не нашел гивен, который, казалось бы, должен был быть доступен. В таких случаях бывает полезно отредактировать код так, чтобы гивен передавался явно, с помощью `using`. Если сообщение об ошибке остается и после этого, причина, по которой компилятор не сумел применить ваш гивен, становится очевидной. Если же явная подстановка гивена позволяет избавиться от ошибки, вы можете быть уверены в том, что проблема была в одном из других правил (например, в правиле видимости).

При отладке программы иногда бывает полезно видеть, какие гивены подставляет компилятор. Для этого компилятору можно передать параметр `-Xprint:typer`. В этом случае вы узнаете, как выглядит ваш код после того, как средство проверки типов добавило все гивены. Пример этого показан в листингах 21.8 и 21.9. Если взглянуть на последнюю инструкцию каждого из этих листингов, можно увидеть, что компилятор подставил второй список параметров, `enjoy("reader")`, который был опущен в листинге 21.8:

```
Mocha.enjoy("reader")(Mocha.pref)
```

Если вы чувствуете жажду приключений, попробуйте выполнить команду `scala -Xprint:typer`, чтобы получить интерактивную оболочку с выводом исходного кода после типизации, предназначенного для внутреннего использования компилятором. Но будьте готовы к тому, что интересующие вас участки будут спрятаны в грудe шаблонного кода¹.

Листинг 21.8. Пример кода, в котором используется контекстный параметр object `Mocha`:

```
class PreferredDrink(val preference: String)

given pref: PreferredDrink = new PreferredDrink("mocha")

def enjoy(name: String)(using drink: PreferredDrink): Unit =
  print(s"Welcome, $name")
  print(". Enjoy a ")
  print(drink.preference)
  println("!")

def callEnjoy: Unit = enjoy("reader")
```

Листинг 21.9. Пример кода, полученного после проверки типов и подстановки гивенов

```
$ scalac -Xprint:typer Mocha.scala
package <empty> {
  final lazy module val Mocha: Mocha$ = new Mocha$()
    def callEnjoy: Unit = Mocha.enjoy("reader")(Mocha.pref)
  final module class Mocha$() extends Object() {
    this: Mocha.type =>
    // ...
  final lazy given val pref: Mocha.PreferredDrink =
    new Mocha.PreferredDrink("mocha")
  def enjoy(name: String)(using drink:
```

¹ У таких IDE, как IntelliJ и Metals, есть параметры для вывода подставленных гивенов.

```
    Mocha.PreferredDrink): Unit = {  
    print(  
      _root_.scala.StringContext.apply(["Welcome,  
      ", "" : String]:String*).s([name : Any]:Any*)  
    )  
    print(". Enjoy a ")  
    print(drink.preference)  
    println("!")  
  }  
  def callEnjoy: Unit = Mocha.enjoy("reader")(Mocha.pref)  
}  
}
```

Резюме

Контекстные параметры могут сделать сигнатуры ваших функций более легкими для восприятия: вместо того чтобы пытаться разобраться в шаблонных аргументах, читатель может сосредоточиться на том, для чего в действительности предназначена ваша функция, делегируя предоставление этих шаблонных аргументов ее контексту. Поиск этих контекстных параметров тоже происходит во время компиляции, что гарантирует доступность их значений на этапе выполнения.

В этой главе был описан механизм гивенов, способный неявно передавать аргументы функциям. Это уменьшает количество шаблонного кода и при этом позволяет функциям потреблять все данные, с которыми они работают, в виде аргументов. Как вы сами видели, поиск контекстных параметров основан на типе параметров функции: если для неявной передачи доступно значение подходящего типа, компилятор передаст его функции. Вы также видели несколько примеров использования этого механизма для реализации специального полиморфизма, как в случае с классом типов `Ordering`. В следующей главе вы узнаете, как классы типов можно использовать в сочетании с методами расширения, а в главе 23 будет представлено несколько примеров использования заданных экземпляров для классов типов.

22

Методы расширения

Если вы пишете функцию, которая в основном работает с конкретным классом объектов, у вас может возникнуть желание определить ее в качестве члена этого класса. В таком объектно-ориентированном языке, как Scala, программистам, которые будут вызывать эту функцию, данный подход может показаться наиболее естественным. Тем не менее в некоторых случаях класс нельзя изменять. Бывают также ситуации, когда функционал должен принадлежать given-экземпляру класса типа (type class), определенному для класса. В связи с этим в Scala предусмотрен механизм, создающий *видимость* того, что функция определена как метод класса, хотя в реальности она ему не принадлежит.

В Scala 3 на смену подходу с неявными классами пришел новый механизм, *методы расширения*. В этой главе вы узнаете, как создавать собственные и использовать чужие методы расширения.

22.1. Основы

Представьте, что вам нужно проверять строки на равенство, применяя два особых правила к пробельным символам. Во-первых, пробельные символы, находящиеся в начале или конце строки, нужно игнорировать. Во-вторых, участки пробельных символов внутри строк должны совпадать по размеру и позиции, но могут отличаться по содержанию. Для этого вы могли бы обрезать пробельные символы в начале и в конце обеих строк, заменить любые внутренние последовательности пробельных символов одним пробелом и затем проверить получившиеся строки на равенство. Вот функция, которая выполняет это преобразование:

```
def singleSpace(s: String): String =
  s.trim.split("\\s+").mkString(" ")
```

Функция `singleSpace` принимает строку и делает ее подходящей для сравнения с использованием `==`. Сначала она удаляет пробельные символы на обоих концах строки с помощью `trim`. Затем она вызывает `split`, чтобы разделить обрезанную строку по участкам последовательных пробельных символов. В результате получается массив. И в завершение она использует `mkString` для объединения непобельных строк в массиве, разделяя каждую из них единственным символом пробела. Вот несколько примеров:

```
singleSpace("A Tale\tof Two Cities")
// "A Tale of Two Cities"
singleSpace(" It was the\t\tbest\nof times. ")
// "It was the best of times."
```

`singleSpace` можно использовать для проверки строк на равенство, игнорируя различия в пробельных символах:

```
val s = "One Fish, Two\tFish "
val t = " One Fish, Two Fish"
singleSpace(s) == singleSpace(t) // true
```

Этот подход вполне логичен. `singleSpace` можно поместить в подходящий объект-одиночку и перейти к следующей задаче. Но, если взглянуть на это с человеческой точки зрения, вам может показаться, что ваши пользователи предпочли бы вызывать этот метод напрямую из `String`, как показано ниже:

```
s.singleSpace == t.singleSpace // к сожалению, это выражение ложное
```

Данный синтаксис придал бы ощущение объектной ориентированности при работе с этой функцией. Но, поскольку класс `String` входит в стандартную библиотеку, этот синтаксис было бы проще всего реализовать путем определения `singleSpace` в качестве метода расширения¹. Это продемонстрировано в листинге 22.1.

Листинг 22.1. Метод расширения для `String`

```
extension (s: String)
  def singleSpace: String =
    s.trim.split("\\s+").mkString(" ")
```

¹ К числу других, более сложных вариантов относится добавление `singleSpace` в `String` с использованием Java Community ProcessSM или Scala Improvement Process.

Ключевое слово `extension` позволяет создать *иллюзию* того, что вы добавили в класс функцию-член, не меняя при этом сам класс. В скобках после `extension` указывается одна переменная того типа, к которому вы хотите добавить метод. Объект, на который ссылается эта переменная, называется *получателем* метода расширения. В данном случае (`s: String`) означает, что вы хотите добавить метод в `String`. Вслед за этой вводной частью идет самый обычный метод, единственная особенность которого в том, что в его теле используется получатель, `s`.

Процесс использования метода расширения называется *применением*. Например, здесь `singleSpace` применяется дважды для сравнения двух строк:

```
s.singleSpace == t.singleSpace // Возвращает true!
```

Определение метода расширения чем-то напоминает определение анонимного класса, который принимает объект-получатель в качестве параметра своего конструктора и тем самым делает этот объект доступным для своих методов. Однако это впечатление обманчивое. Определение метода расширения *заменяется* методом, который принимает получатель напрямую в качестве параметра. Например, компилятор переписывает определение метода расширения из листинга 22.1, чтобы привести его к виду, показанному в листинге 22.2.

Листинг 22.2. Метод расширения после переписывания компилятором

```
// С внутренним обозначением расширения
def singleSpace(s: String): String =
  s.trim.split("\\s+").mkString(" ")
```

Единственной особенностью переписанного метода является то, что компилятор присваивает ему внутреннее обозначение, сигнализирующее о том, что это метод расширения. Чтобы сделать этот метод доступным, проще всего разместить имя его переписанной версии в лексической области видимости. Вот пример в REPL:

```
scala> extension (s: String)
      def singleSpace: String =
        s.trim.split("\\s+").mkString(" ")
def singleSpace(s: String): String
```

Поскольку в этом сеансе REPL `singleSpace` находится в лексической области видимости и обозначен как метод расширения, его можно применить:

```
scala> s.singleSpace == t.singleSpace
val res0: Boolean = true
```

Ввиду того что Scala заменяет методы расширения, при их применении не происходит ненужных преобразований. В Scala 2, где используется подход с неявными классами, это было не всегда так. Таким образом методы расширения предоставляют «синтаксический сахар без отрицательных эффектов». Метод расширения, вызываемый из получателя, как в случае с `s.singleSpace`, всегда имеет ту же производительность, что и передача получателя соответствующему нерасширяющему методу — например, `singleSpace(s)`.

22.2. Обобщенные расширения

Вы можете определять методы расширения, которые являются обобщенными. В качестве примера возьмем метод `head` из класса `List`, который возвращает первый элемент списка, но генерирует исключение, если список пустой¹:

```
List(1, 2, 3).head // 1
List.empty.head // генерирует NoSuchElementException
```

Если вы не уверены в том, что имеющийся у вас список что-то содержит, можете использовать вместо этого метод `headOption`, который возвращает первый элемент, завернутый в `Some`. Если же список пустой, `headOption` возвращает `None`:

```
List(1, 2, 3).headOption // Some(1)
List.empty.headOption // None
```

`List` также предлагает метод `tail`, который возвращает все, кроме первого элемента. Как и `head`, он генерирует исключение, если список пустой:

```
List(1, 2, 3).tail // List(2, 3)
List.empty.tail // генерирует NoSuchElementException
```

Однако класс `List` не предлагает безопасную альтернативу для получения оставшейся части списка, завернутой в `Option`. Если вам нужен такой метод, можете реализовать его в виде *обобщенного расширения*. Для этого нужно указать один или несколько параметров типа после ключевого слова `extension`, но перед скобками с получателем внутри. Пример показан в листинге 22.3.

Листинг 22.3. Обобщенный метод расширения

```
extension [T](xs: List[T])
  def tailOption: Option[List[T]] =
    if xs.nonEmpty then Some(xs.tail) else None
```

¹ Метод `head` из `List` описан в разделе 14.4.

Метод расширения `tailOption` является обобщенным только для одного типа, `T`. Вот несколько примеров использования `tailOption`, в которых из `T` создается экземпляр `Int` и `String`:

```
List(1, 2, 3).tailOption    // Some(List(2, 3))
List.empty[Int].tailOption // None
List("A", "B", "C").tailOption // Some(List(B, C))
List.empty[String].tailOption // None
```

Обычно имеет смысл разрешить автоматическое определение такого параметра типа, как это было сделано в предыдущих примерах, но вы можете указать его явно. Для этого метод необходимо вызвать напрямую, то есть не как метод расширения:

```
tailOption[Int](List(1, 2, 3)) // Some(List(2, 3))
```

22.3. Групповые расширения

Когда несколько методов нужно добавить в один и тот же тип, их можно определить вместе с помощью *группового расширения*. Например, поскольку многие операции с `Int` могут вызывать переполнение буфера, вам, возможно, захочется определить несколько методов расширения для `Int`, способных распознать переполнение.

Чтобы представить значение `Int` в дополнительном коде, нужно инвертировать все его биты и прибавить единицу. Это представление позволяет реализовать вычитание в виде операции в дополнительном коде, вслед за которой идет сложение. Оно также имеет всего одно нулевое значение вместо двух: положительного и отрицательного¹. К тому же, учитывая отсутствие отрицательного нуля, у нас остается один разряд для дополнительного значения. Это значение размещается в самом конце отрицательных целых чисел. Вот почему наименьшее отрицательное значение `Int`, взятое по модулю, на единицу меньше наибольшего положительного целого числа, которое можно выразить:

```
Int.MaxValue // 2147483647
Int.MinValue // -2147483648
```

Именно из-за этой асимметричности между максимальным и минимальным значениями некоторые методы `Int` могут переполняться. Например, метод

¹ В обратном коде целые нулевые значения могут быть как положительными, так и отрицательными, по аналогии с форматом чисел с плавающей запятой IEEE 754, который используется в `Float` и `Double`.

`abs` из `Int` вычисляет абсолютное значение целого числа. В `Int` абсолютное значение минимума составляет 2 147 483 648, однако это число невозможно выразить с помощью этого типа. Максимальное значение `Int` равно 2 147 483 647, что на единицу меньше, поэтому вызов `abs` для `Int.MinValue` приводит к переполнению и вы получаете исходное значение `Int.MinValue`:

```
Int.MinValue.abs // -2147483648 (переполнение)
```

Если вам нужен метод, который возвращает абсолютное значение `Int`, но при этом распознает переполнение, вы можете определить метод расширения, как показано далее:

```
extension (n: Int)
  def absOption: Option[Int] =
    if n != Int.MinValue then Some(n.abs) else None
```

Для значения `Int.MinValue`, из-за которого переполняется `abs`, `absOption` возвращает `None`. Для остальных значений `absOption` возвращает результат работы `abs`, завернутый в `Some`. Вот несколько примеров использования `absOption`:

```
42.absOption           // Some(42)
-42.absOption          // Some(42)
Int.MaxValue.absOption // Some(2147483647)
Int.MinValue.absOption // None
```

Еще одной операцией, способной вызвать переполнение для минимального значения `Int`, является изменение знака числа на отрицательный. Операция `unary_-` с `Int.MinValue` возвращает все то же `Int.MinValue`¹:

```
-Int.MinValue // -2147483648 (переполнение)
```

Если вам нужен безопасный вариант `unary_-`, вы можете определить его вместе с `absOption` в *групповом расширении*, как показано в листинге 22.4.

Листинг 22.4. Групповое расширение

```
extension (n: Int)
  def absOption: Option[Int] =
    if n != Int.MinValue then Some(n.abs) else None
  def negateOption: Option[Int] =
    if n != Int.MinValue then Some(-n) else None
```

¹ Как было описано в разделе 5.4, компилятор Scala подставляет вместо `-Int.MinValue` метод `unary_-` для `Int.MinValue` — то есть `Int.MinValue.unary_-`.

Это расширение добавляет в `Int` сразу два метода: `absOption` и `negateOption`. Вот несколько примеров использования последнего.

```
-42.negateOption      // Some(42)
42.negateOption       // Some(42)
Int.MaxValue.negateOption // Some(2147483647)
Int.MinValue.negateOption // None
```

Методы, определенные вместе в групповом расширении, называются *методами того же уровня*. Из одного метода в групповом расширении можно вызывать другие, как если бы они были членами одного класса. Например, как показано в листинге 22.5, если вы решите добавить в `Int` еще один метод расширения, `isMinValue`, у вас будет возможность вызывать его напрямую из двух других методов, `absOption` и `negateOption`.

Листинг 22.5. Вызов метода расширения того же уровня

```
extension (n: Int)
  def isMinValue: Boolean = n == Int.MinValue
  def absOption: Option[Int] =
    if !isMinValue then Some(n.abs) else None
  def negateOption: Option[Int] =
    if !isMinValue then Some(-n) else None
```

В групповом расширении, показанном в листинге 22.5, метод `isMinValue` вызывается как из `absOption`, так и из `negateOption`. В таких случаях компилятор переопределяет вызов так, чтобы он выполнялся из получателя. В данном расширении, к примеру, компилятор подставит `n.isMinValue` вместо `isMinValue`, как показано в листинге 22.6.

Листинг 22.6. Групповое расширение, после переопределения компилятором

```
// Все с внутренними обозначениями расширения
def isMinValue(n: Int): Boolean = n == Int.MinValue
def absOption(n: Int): Option[Int] =
  if !n.isMinValue then Some(n.abs) else None
def negateOption(n: Int): Option[Int] =
  if !n.isMinValue then Some(-n) else None
```

22.4. Использование класса типов

Распознавание переполнения в операциях с получением абсолютного значения и изменением знака на противоположный имеет смысл не только для типа `Int`. Любому целочисленному типу, основанному на арифметике в дополнительном коде, свойственна одна и та же проблема с переполнением:

```

Long.MinValue.abs // -9223372036854775808 (переполнение)
-Long.MinValue   // -9223372036854775808 (переполнение)
Short.MinValue.abs // -32768 (переполнение)
-Short.MinValue   // -32768 (переполнение)
Byte.MinValue.abs // -128 (переполнение)
-Byte.MinValue     // -128 (переполнение)

```

Если вам нужны безопасные альтернативы `abs` и `unary_` - для всех этих типов, вы можете определить отдельное групповое расширение для каждого из них, но в этом случае все реализации будут выглядеть одинаково. Чтобы не дублировать код, вы можете определить вместо этого расширение на основе класса типов. Такое *специализированное расширение* будет работать для любого типа с `given`-экземпляром класса типа.

Чтобы проверить, существует ли трейт с подходящим классом типов, стоит заглянуть в стандартную библиотеку. Трейт `Numeric` слишком общий, так как `given`-экземпляры предоставляются для типов вроде `Double` или `Float`, которые не основаны на арифметике в дополнительном коде. То же самое можно сказать о трейте `Integral`, только вместо `Double` или `Float` `given`-экземпляр предоставлен для типа `BigInt`, который не переполняется. Таким образом, самый оптимальный вариант состоит в определении нового трейта специально для целочисленных типов в дополнительном коде, как, например, трейт `TwosComplement`, показанный в листинге 22.7.

После этого следует определить `given`-экземпляры для типов в дополнительном коде, которые будут содержать методы расширения. Подходящим местом их размещения будет объект-компаньон, доступ к которому, как вы ожидаете, всегда будет нужен пользователям¹. В листинге 22.7 `given`-экземпляры `TwosComplement` определены для `Byte`, `Short`, `Int` и `Long`.

Листинг 22.7. Класс типов для чисел в дополнительном коде

```

trait TwosComplement[N]:
  def equalsMinValue(n: N): Boolean
  def absOf(n: N): N
  def negationOf(n: N): N

object TwosComplement:
  given tcOfByte: TwosComplement[Byte] with
    def equalsMinValue(n: Byte) = n == Byte.MinValue
    def absOf(n: Byte) = n.abs
    def negationOf(n: Byte) = (-n).toByte

```

¹ Совет о том, где лучше определять `given`-экземпляры, был дан в разделе 21.5.

```

given tcOfShort: TwosComplement[Short] with
  def equalsMinValue(n: Short) = n == Short.MinValue
  def absOf(n: Short) = n.abs
  def negationOf(n: Short) = (-n).toShort

given tcOfInt: TwosComplement[Int] with
  def equalsMinValue(n: Int) = n == Int.MinValue
  def absOf(n: Int) = n.abs
  def negationOf(n: Int) = -n

given tcOfLong: TwosComplement[Long] with
  def equalsMinValue(n: Long) = n == Long.MinValue
  def absOf(n: Long) = n.abs
  def negationOf(n: Long) = -n

```

Имея в своем распоряжении эти определения, вы можете написать обобщенный метод расширения, как показано в листинге 22.8. Это позволит использовать `absOption` и `negateOption` для подходящих типов.

Листинг 22.8. Использование класса типов в расширении

```

Byte.MaxValue.negateOption // Some(-127)
Byte.MinValue.negateOption // None
Long.MaxValue.negateOption // -9223372036854775807
Long.MinValue.negateOption // None

extension [N](n: N)(using tc: TwosComplement[N])
  def isMinValue: Boolean = tc.equalsMinValue(n)
  def absOption: Option[N] =
    if !isMinValue then Some(tc.absOf(n)) else None
  def negateOption: Option[N] =
    if !isMinValue then Some(tc.negationOf(n)) else None

```

С другой стороны, любая попытка использования этих методов расширения для неподходящих типов приведет к ошибке компиляции:

```

BigInt(42).negateOption
1 | BigInt(42).negateOption
  | ~~~~~
  | value negateOption is not a member of BigInt.
  | An extension method was tried, but could not be
  | fully constructed:
  |
  |   negateOption[BigInt](BigInt.apply(42))(
  |     /* missing */summon[TwosComplement[BigInt]]
  |   )

```

Как уже обсуждалось в разделе 21.4, классы типов обеспечивают специальный полиморфизм: функционал, доступный только для определенных типов (тех, для которых существуют `given`-экземпляры класса типов) и выдающий

ошибку компиляции для любого другого типа. Классы типов можно использовать для получения синтаксического сахара в виде методов расширения для определенных типов. Любые попытки применения методов расширения к другим типам не позволят скомпилировать код.

22.5. Методы расширения для гивенов

В предыдущем разделе задача класса типов `TwosComplement` состояла в предоставлении методов расширения для определенного множества типов. Поскольку основным потребителем методов расширения является пользователь, у него не должно вызывать затруднений решение о том, когда их следует делать доступными и стоит ли это делать в принципе. В таких ситуациях расширение лучше всего размещать в объекте-одиночке. Ваши пользователи могут импортировать методы расширения из этого объекта в лексическую область видимости, что сделает возможным их применение. Вы можете, к примеру, поместить групповое расширение для распознавания переполнений в объект с именем `TwosComplementOps`, как показано в листинге 22.9.

Листинг 22.9. Размещение методов расширения в объекте-одиночке

```
object TwosComplementOps:
  extension [N](n: N)(using tc: TwosComplement[N])
    def isMinValue: Boolean = tc.equalsMinValue(n)
    def absOption: Option[N] =
      if !isMinValue then Some(tc.absOf(n)) else None
    def negateOption: Option[N] =
      if !isMinValue then Some(tc.negationOf(n)) else None
```

Затем ваши пользователи могут добавить в свой код немного синтаксического сахара:

```
import TwosComplementOps.*
```

Благодаря этому импорту методы расширения будут доступны для применения:

```
-42.absOption // Some(42)
```

В случае с `TwosComplementOps` методы расширения представляют основную цель проектирования, а класс типов играет вспомогательную роль. Но зачастую все наоборот: класс типов служит главной целью, а методы расширения помогают упростить использование этого класса. В таких ситуациях методы расширения лучше всего размещать в трейте самого класса типов.

Например, в главе 21 класс типов `Ord` был определен, чтобы сделать метод сортировки вставками, `isort`, более общим. И хотя эта цель была достигнута с помощью решений, представленных в главе 21 (метод `isort` можно использовать с любым типом `T`, для которого доступен `given`-экземпляр `Ord[T]`), добавление нескольких методов расширения сделает класс типов `Ord` более приятным в использовании.

Каждый трейт с классом типов принимает параметр, поскольку экземпляр этого класса знает, как обращаться с объектами этого типа. Например, `Ord[T]` знает, как сравнивать два экземпляра типа `T` для определения того, какой из них больше или равен другому. Поскольку экземпляр класса типов для `T` — не то же самое, что экземпляр или экземпляры `T`, синтаксис использования классов типов может быть немного громоздким. Например, в листинге 21.1 метод `insert` принимает `given`-экземпляр `Ord[T]` и определяет с его помощью, является ли экземпляр `T` равным начальному элементу уже отсортированного списка или меньше его. Вот как выглядит метод `insert` из этого листинга:

```
def insert[T](x: T, xs: List[T])(using ord: Ord[T]): List[T] =
  if xs.isEmpty || ord.lteq(x, xs.head) then x :: xs
  else xs.head :: insert(x, xs.tail)
```

Вызов `ord.lteq(x, xs.head)` является вполне нормальным, но его можно было бы записать более естественным и, наверное, более понятным образом:

```
x <= xs.head // Теперь намного понятнее!
```

Синтаксический сахар `<=` (а также `<`, `>` и `>=`) можно сделать доступным с помощью группового расширения. На этом этапе методы расширения размещаются в объекте-одиночке `OrdOps`, как показано в листинге 22.10.

Листинг 22.10. Размещение расширений для `Ord` в объекте-одиночке

```
// (Это еще не лучшее решение)
object OrdOps:
  extension [T](lhs: T)(using ord: Ord[T])
    def < (rhs: T): Boolean = ord.lt(lhs, rhs)
    def <= (rhs: T): Boolean = ord.lteq(lhs, rhs)
    def > (rhs: T): Boolean = ord.gt(lhs, rhs)
    def >= (rhs: T): Boolean = ord.gteq(lhs, rhs)
```

Имея в своем распоряжении определение `OrdOps` из листинга 22.10, пользователи могут добавить в свой код синтаксический сахар с помощью операции импорта, как показано здесь:

```
def insert[T](x: T, xs: List[T])(using Ord[T]): List[T] =
  import OrdOps.*
  if xs.isEmpty || x <= xs.head then x :: xs
  else xs.head :: insert(x, xs.tail)
```

Вместо `ord.lteq(x, xs.head)` можно написать `x <= xs.head`. К тому же вам на самом деле не нужно имя экземпляра `Ord`, так как вы его больше не используете. В итоге (`using ord: Ord[T]`) можно упростить до (`using Ord[T]`).

Этот подход работает, но было бы неплохо иметь под рукой этот синтаксический сахар всякий раз, когда доступен экземпляр `Ord`. Поскольку такие ситуации не редкость, Scala ищет `given`-экземпляры для применимых расширений. Таким образом, эти расширения лучше всего размещать не в объекте-одиночке, таком как `OrdOps`, а в трейте самого класса типов `Ord`. Это позволит гарантировать, что методы расширения можно применять всегда, когда экземпляр класса типов уже находится в области видимости. Это выглядело бы так, как показано в листинге 22.11.

Листинг 22.11. Размещение расширения в трейте класса типов

```
trait Ord[T]:
  def compare(x: T, y: T): Int
  def lt(x: T, y: T): Boolean = compare(x, y) < 0
  def lteq(x: T, y: T): Boolean = compare(x, y) <= 0
  def gt(x: T, y: T): Boolean = compare(x, y) > 0
  def gteq(x: T, y: T): Boolean = compare(x, y) >= 0

  // (Это лучшее решение)
  extension (lhs: T)
    def < (rhs: T): Boolean = lt(lhs, rhs)
    def <= (rhs: T): Boolean = lteq(lhs, rhs)
    def > (rhs: T): Boolean = gt(lhs, rhs)
    def >= (rhs: T): Boolean = gteq(lhs, rhs)
```

Благодаря размещению в трейте самого класса типов методы расширения будут доступны всегда, когда используется `given`-экземпляр этого класса. Например, методы расширения будут просто доступны внутри `insert`, и для этого не нужно ничего импортировать. Это видно в листинге 22.12.

Листинг 22.12. Использование расширения, определенного в трейте класса типов

```
def insert[T](x: T, xs: List[T])(using Ord[T]): List[T] =
  if xs.isEmpty || x <= xs.head then x :: xs
  else xs.head :: insert(x, xs.tail)
```

Поскольку вам больше не нужно импортировать `OrdOps.*`, версия `insert`, показанная в листинге 22.12, получилась более компактной по сравнению с предыдущей. Более того, упростилось и само расширение. Сравните реализации группового расширения в листингах 22.10 и 22.11. Методы расширения являются частью трейта самого класса типов, поэтому у него уже есть ссылка на экземпляр этого класса, то есть `this`. Таким образом, в начале больше не

нужно указывать [T] и (using ord: Ord[T]); это выражение упростилось до `extension (lhs: T)`. К тому же, поскольку у вас больше нет переданного экземпляра `Ord[T]` с именем `ord`, вы не можете использовать его для вызова методов класса типов, таких как `lt` и `lteq`. Вместо этого их можно вызывать из ссылки `this`. Таким образом, `ord.lt(lhs, rhs)` превращается в `lt(lhs, rhs)`.

Scala переопределяет методы расширения, делая их членами трейта самого класса типов, как показано в листинге 22.13.

Листинг 22.13. Расширения класса типов после переопределения компилятором

```
trait Ord[T]:

  def compare(x: T, y: T): Int
  def lt(x: T, y: T): Boolean = compare(x, y) < 0
  def lteq(x: T, y: T): Boolean = compare(x, y) <= 0
  def gt(x: T, y: T): Boolean = compare(x, y) > 0
  def gteq(x: T, y: T): Boolean = compare(x, y) >= 0

  // С внутренними обозначениями расширения:
  def < (lhs: T)(rhs: T): Boolean = lt(lhs, rhs)
  def <= (lhs: T)(rhs: T): Boolean = lteq(lhs, rhs)
  def > (lhs: T)(rhs: T): Boolean = gt(lhs, rhs)
  def >= (lhs: T)(rhs: T): Boolean = gteq(lhs, rhs)
```

Чтобы исправить ошибку выбора типа, Scala заглядывает внутрь `given`-экземпляров `Ord[T]` при поиске методов расширений. Для этого компилятор Scala использует немного запутанный алгоритм, который подробно описан далее.

22.6. Где Scala ищет методы расширения

Когда компилятор встречает попытку вызвать метод из ссылки на объект, он проверяет, определен ли этот метод в классе самого объекта. Если да, то он выбирает этот метод и не переходит к поиску метода расширения¹. В противном случае во время компиляции возникает ошибка выбора кандидата. Но прежде, чем выводить эту ошибку, компилятор ищет метод расширения или неявное преобразование, которые могут ее исправить². Компилятор сообщит об ошибке, только если ему не удастся найти метод расширения или неявное преобразование, которые позволили бы от нее избавиться.

¹ Это общее правило: если участок кода компилируется как есть, компилятор Scala не преобразует его во что-то другое.

² Неявные преобразования будут описаны в главе 23.

Scala выполняет поиск метода расширения в два этапа. На первом этапе компилятор проверяет лексическую область видимости. На втором он анализирует члены `given`-экземпляров в лексической области видимости, члены объектов-компаньонов класса получателя, родительских классов и трейтов, а также члены `given`-экземпляров в этих самых объектах-компаньонах. В рамках второго этапа он также пытается выполнить неявное приведение типа получателя.

Если на каком-либо этапе компилятор находит сразу несколько подходящих методов расширения, он выбирает из них самый конкретный, подобно тому как происходит выбор перегруженного метода из нескольких вариантов. Если найдено два и больше метода расширения с одинаковой степенью конкретности, выводится ошибка компиляции со списком равнозначных расширений.

Определение может встречаться в лексической области видимости по одной из трех причин: его определили напрямую, импортировали или унаследовали. Например, следующий вызов `absOption` из 88 успешно компилируется, потому что перед использованием метод расширения `absOption` импортируется в виде единого идентификатора:

```
import TwosComplementOps.absOption
88.absOption // Some(88)
```

Таким образом, поиск методов расширения для `absOption` заканчивается уже на первом этапе. Для сравнения: поиск, спровоцированный использованием `<=` в листинге 22.12, доходит до второго этапа. Примененным методом расширения выступает `<=` из листинга 22.11. Он вызывается из гивена `Ord[T]`, переданного в виде параметра `using`.

Резюме

Методы расширения позволяют улучшить ваш код за счет синтаксического сахара: все выглядит так, будто функция вызывается из объекта и является методом, объявленным в его классе, хотя на самом деле вы передаете объект этой функции. Из этой главы вы узнали, как определять собственные методы расширения и использовать те, которые определил кто-то другой. Здесь было показано, как методы расширения и классы типов дополняют друг друга и как их лучше всего использовать вместе. В следующей главе мы углубимся в классы типов.

23

Классы типов

Если вам нужно написать функцию, которая реализует поведение, полезное только для каких-то определенных типов, в Scala у вас есть несколько вариантов. Первый вариант состоит в определении перегруженных методов. Второй — потребовать, чтобы класс любого экземпляра, переданного вашей функции, был примесью в определенном трейте. Третий (и более гибкий) заключается в том, чтобы определить класс типов и адаптировать функцию для работы с типами, для которых определен given-экземпляр трейта этого класса.

В данной главе мы проведем сравнение этих разных подходов и затем углубимся в классы типов. Мы разберемся с синтаксисом классов типов, который привязан к контексту, и рассмотрим несколько примеров таких классов из стандартной библиотеки: для численных литералов, многостороннего равенства, неявных преобразований и главных методов. В заключение будет дан пример, иллюстрирующий использование класса типов для сериализации JSON.

23.1. Зачем нужны классы типов

Термин «класс типов» (typeclass) может сбивать с толку в контексте Scala, так как под *типами* подразумеваются типы языка, а вот *класс* употребляется в широком смысле и означает группу или множество каких-то вещей. Таким образом, «класс типов» — это группа или множество типов.

Как упоминалось в разделе 21.4, классы типов поддерживают *специальный полиморфизм* (ad hoc polymorphism), позволяя применять функции с конкретным, перечисляемым множеством типов. Любая попытка использования такой функции с типом, который не входит в это перечисляемое множество,

приводит к ошибке компиляции. Например, термин «специальный полиморфизм» во многих языках изначально описывал то, как операторы вроде + или - можно сочетать только с определенными типами [Str00]. В Scala для этого используются перегруженные методы. Например, `scala.Int` содержит семь перегруженных абстрактных методов с именем «минус» (-):

```
def -(x: Double): Double
def -(x: Float): Float
def -(x: Long): Long
def -(x: Int): Int
def -(x: Char): Int
def -(x: Short): Int
def -(x: Byte): Int
```

Следовательно, методу «минус» из `Int` можно передавать экземпляры семи конкретных типов. Это своего рода группа или множество (или класс в общем смысле этого слова) типов, принимаемых методом «минус». Это проиллюстрировано на рис. 23.1.

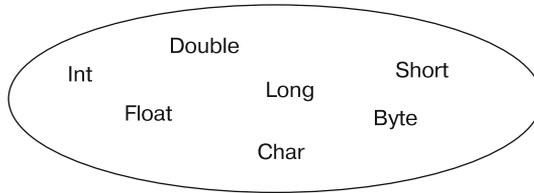


Рис. 23.1. Множество типов, принимаемых методами «минус» (-) из `Int`

Еще один способ организации полиморфизма в Scala состоит в использовании иерархии классов. Вот пример, в котором для определения семейства цветов используется запечатанный трейт:

```
sealed trait RainbowColor
class Red extends RainbowColor
class Orange extends RainbowColor
class Yellow extends RainbowColor
class Green extends RainbowColor
class Blue extends RainbowColor
class Indigo extends RainbowColor
class Violet extends RainbowColor
```

На основе этой иерархии можно определить метод, который принимает `RainbowColor` в качестве аргумента:

```
def paint(rc: RainbowColor): Unit
```

Поскольку трейт `RainbowColor` запечатан, методу `paint` можно передавать только аргументы одного из восьми типов, показанных на рис. 23.2. С любым другим типом он не скомпилируется. Этот подход можно считать специальным полиморфизмом, но его называют подтипизацией (или полиморфизмом подтипов), чтобы подчеркнуть его важную особенность: классы всех экземпляров, передаваемых методу `paint`, должны быть примесью в трейте `RainbowColor` и соблюдать любые ограничения, установленные его интерфейсом. Для сравнения: типы, принимаемые методом «минус» (-) из `Int` (см. рис. 23.1), не обязаны соблюдать правила никакого общего интерфейса, кроме самого верхнего типа в иерархии `Scala`, `Any`. Если подытожить, то подтипизация делает возможным полиморфизм *родственных* типов, тогда как подходы наподобие перегрузки и классов типов позволяют организовать полиморфизм *не связанных между собой* типов (специальный полиморфизм).

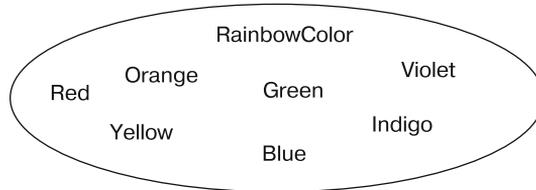


Рис. 23.2. Множество типов, принимаемых методом `paint`

Ввиду ограничений, накладываемых интерфейсом, подтипизация работает лучше всего, когда иерархии классов определяют небольшие семейства типов, ориентированных на какую-то единую концепцию. Отличными примерами являются запечатанные иерархии и перечисления. В таких самодостаточных семействах типов легко обеспечить совместимость интерфейсов. Подтипизация также может применяться для моделирования более крупных, незапечатанных семейств, ориентированных на единую концепцию. В качестве хорошего примера можно привести библиотеку коллекций `Scala`. Однако при моделировании поведения, широко применяемого к не связанным между собой типам (такого, как сериализация или упорядочение), подход на основе подтипизации становится более громоздким.

Рассмотрим в качестве примера трейт `Ordered` из состава `Scala`, который использует подтипизацию для моделирования операций упорядочения. Как было показано в разделах 11.2 и 18.7, если сделать трейт `Ordered` примесью класса и реализовать абстрактный метод `compare`, можно наследовать реализации `<`, `>`, `<=` и `>=`. Вы также можете использовать `Ordered` в качестве верхней границы для определения метода сортировки, такого как `orderedMergeSort` из листинга 18.11.

Недостаток этого подхода в том, что любой тип `T`, передаваемый методу `orderedMergeSort`, обязан быть примесью типа `Ordered[T]` и соблюдать правила его интерфейса. Из этого следует одна потенциальная проблема: в классе, в который вы примешали `Ordered`, уже могут быть определены методы, чьи имена или контракты конфликтуют с `Ordered`. Еще одна проблема может быть связана с конфликтами вариантности. Представьте, что вы хотите примешать `Ordered` в перечисление `Hope` из раздела 19.4. Возможно, вы надеетесь, что вам удастся реализовать метод `compare` путем использования объекта `Sad` в качестве наименьшего значения `Hope` и упорядочения экземпляров `Glad` с учетом порядка размещения содержащихся в них объектов. К сожалению, компилятору такой план не понравится, поскольку тип `Hope` ковариантный по своему параметру типа, тогда как `Ordered` инвариантный:

```
class Hope[+T <: Ordered[T]] extends Ordered[Hope[T]]
1 | class Hope[+T <: Ordered[T]] extends Ordered[Hope[T]]
  | .....
  | covariant type T occurs in invariant position in type
  | Object with Ordered[Hope[T]] {...} of class Hope
```

Таким образом одним из потенциальных недостатков полиморфизма подтипов является наличие несовместимых интерфейсов. Еще одна более распространенная проблема связана с существованием совместимых интерфейсов, *которые не поддаются изменению*. Например, вы не можете использовать метод `orderedMergeSort`, показанный в листинге 18.11, для сортировки `List[Int]`, поскольку `Int` не наследует `Ordered[Int]`, — и с этим ничего не поделаешь. На практике же основной трудностью при использовании подтипизации для общих концепций, применяемых ко многим не связанным между собой типам, является то, что многие из этих типов определены в библиотеках, которые нельзя изменить.

Классы типов решают эту проблему за счет определения *отдельной* иерархии, ориентированной на общую концепцию, с использованием параметра для задания типа, возможности которого расширяются. Поскольку эта иерархия ориентирована только на какую-то одну процедуру, такую как сериализация или упорядочение, обеспечение совместимости интерфейсов не составляет труда. Экземпляр класса типов использует параметр для определения типа, который расширяется, поэтому вам не нужно изменять сам тип, чтобы его расширить¹. Благодаря этому можно с легкостью определять `given`-экземпляры класса типов, размещенных в библиотеках, которые вы не можете изменять.

¹ Использование параметров типа таким образом называют универсальным полиморфизмом.

Хорошим примером этого является класс типов `Ordering` из состава `Scala`, который определяет иерархию, ориентированную на упорядочение. Иерархия типов `Ordering` отделена от типов, которые упорядочиваются. В результате, несмотря на то что `Ordered` нельзя применять в `Hope`, вы можете определить для `Hope` `given`-экземпляр `Ordered`. Это работает, даже несмотря на то, что `Hope` находится в библиотеке, недоступной для изменения, и вопреки различиям в вариантности между ковариантным типом `Hope` и инвариантным `Ordering`. Реализация показана в листинге 23.1.

Листинг 23.1. `Given`-экземпляр `Ordering` для `Hope[T]`

```
import org.stairwaybook.enums_and_adts.hope.Hope

object HopeUtils:

  given hopeOrdering[T](using
    ord: Ordering[T]): Ordering[Hope[T]] with

  def compare(lh: Hope[T], rh: Hope[T]): Int =
    import Hope.{Glad, Sad}
    (lh, rh) match
      case (Sad, Sad) => 0
      case (Sad, _) => 1
      case (_, Sad) => +1
      case (Glad(lhv), Glad(rhv)) =>
        ord.compare(lhv, rhv)
```

`Ordering` — это множество всех типов `T`, для которых существуют `given`-экземпляры `Ordering[T]`. Стандартная библиотека предоставляет `given`-экземпляры `Ordering` для многих типов, включая `Int` и `String`, что делает их стандартными членами класса типов `Ordering`. `Given` `hopeOrdering`, показанный в листинге 23.1, добавляет типы класса типов `Ordering` вида `Hope[T]` для всех типов `T`, которые также являются членами данного класса. Множество типов, составляющее класс типов `Ordering`, проиллюстрировано на рис. 23.3.

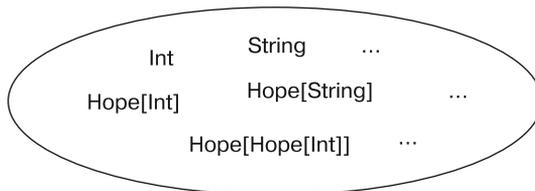


Рис. 23.3. Множество типов `T` с `given`-экземплярами `Ordering[T]`

Классы типов поддерживают специальный полиморфизм, так как вы можете создавать функции, доступные только для типов, для которых существуют given-экземпляры определенного класса типов. Любая попытка использования таких функций в сочетании с типом, у которого нет given-экземпляра необходимого класса типов, приведет к ошибке компиляции. Например, методу `msort`, показанному в листинге 21.5, можно передать `List[T]` с любым типом `T`, для которого определен given-экземпляр `Ordering[T]`. Поскольку в стандартной библиотеке имеются given-экземпляры `Ordering[Int]` и `Ordering[String]`, функции `msort` можно передавать `List[Int]` и `List[String]`. Более того, если импортировать given `hopeOrdering`, показанный в листинге 23.1, `msort` можно будет также передавать `List[Hope[Int]]`, `List[Hope[String]]`, `List[Hope[Hope[Int]]]` и т. д. С другой стороны, любая попытка передать функции `msort` список с элементами типа, для которого не был определен экземпляр `Ordering`, приведет к ошибке компиляции.

Если подытожить, то классы типов решают проблему, состоящую в том, что вместить все операции с типом в его иерархию классов слишком сложно, неудобно или попросту невозможно. На практике не всякий тип, с которым нужно выполнить операцию общего характера, может реализовать интерфейс, который сделает его частью общей иерархии. Подход с использованием классов типов позволяет применять вторую, отдельную иерархию, ориентированную на предоставление операции.

23.2. Границы контекста

Классы типов являются важным шаблоном проектирования в Scala, поэтому для них предусмотрен сокращенный синтаксис под названием «*границы контекста*». Возьмем в качестве примера функцию `maxList` из листинга 23.2, которая возвращает максимальный элемент в переданном списке. В качестве первого аргумента она принимает `List[T]`, но у нее есть еще один список аргументов, содержащий значение `using` типа `Ordering[T]`. В теле `maxList` переданный аргумент `Ordering[T]` используется в двух местах: в рекурсивном вызове `maxList` и в выражении `if`, которое проверяет, превышает ли начальный элемент значение максимального элемента остальной части списка.

Листинг 23.2. Функция с параметром `using`

```
def maxList[T](elements: List[T])
  (using ordering: Ordering[T]): T =
  elements match
```

```

case List() =>
  throw new IllegalArgumentException("empty list!")
case List(x) => x
case x :: rest =>
  val maxRest = maxList(rest)(using ordering)
  if ordering.gt(x, maxRest) then x
  else maxRest

```

Функция `maxList` демонстрирует использование параметра `using` для предоставления дополнительной информации о типе, явно указанном в предыдущем списке параметров. В частности, параметр `ordering` типа `Ordering[T]` дополнительно описывает тип `T`; в данном случае он уточняет, как упорядочивать экземпляры этого типа. Тип `T` фигурирует в `List[T]`, типе параметра `elements`, указанном в предыдущем списке параметров. Поскольку `elements` всегда нужно указывать явно при вызове `maxList`, тип `T` будет известен на этапе компиляции, что позволяет компилятору определить, доступно ли `given`-определение `Ordering[T]`. Если да, то компилятор может автоматически передать второй список параметров, `ordering`.

В реализации `maxList`, показанной в листинге 23.2, параметр `ordering` передается явно с помощью `using`, однако делать это не обязательно. Когда для параметра указано ключевое слово `using`, компилятор не только пытается *предоставить* этому параметру `given`-значение, но и *определяет* его в качестве доступного `given` в теле метода! Таким образом, первое использование `ordering` в теле метода можно опустить, как показано в листинге 23.3.

Анализируя код из листинга 23.3, компилятор обнаружит несоответствие типов. Выражение `maxList(rest)` предоставляет всего один список параметров, а `maxList` требует два. Но, поскольку второй список параметров помечен как `using`, компилятор не сразу прекращает проверку типов. Вместо этого он ищет `given`-параметр подходящего типа, которым является `Ordering[T]`. В этом случае он находит один такой параметр и преобразует вызов в `maxList(rest)(using ordering)`, после чего проверка типов проходит успешно.

Листинг 23.3. Функция, внутри которой используется параметр `using`

```

def maxList[T](elements: List[T])
  (using ordering: Ordering[T]): T =

  elements match
  case List() =>
    throw new IllegalArgumentException("empty list!")

```

```

case List(x) => x
case x :: rest =>
  val maxRest = maxList(rest)      // Использует given-
                                  // экземпляр.
  if ordering.gt(x, maxRest) then x // Этот параметр ordering
  else maxRest                    // по-прежнему задан явно

```

Существует также способ опустить второе использование `ordering`. Для этого в стандартной библиотеке предусмотрен следующий метод:

```
def summon[T](using t: T) = t
```

В результате вызова `summon[Foo]` компилятор будет искать `given`-определение типа `Foo`. Затем он вызовет метод `summon` с этим объектом и получит этот же объект в ответ. Таким образом, если вам нужно найти в текущей области видимости `given`-экземпляр `Foo`, можете просто написать `summon[Foo]`. Например, в листинге 23.4 демонстрируется использование `summon[Ordering[T]]` для извлечения параметра `ordering` по его типу.

Листинг 23.4. Функция, использующая `summon`

```

def maxList[T](elements: List[T])
  (using ordering: Ordering[T]): T =

  elements match
  case List() =>
    throw new IllegalArgumentException("empty list!")
  case List(x) => x
  case x :: rest =>
    val maxRest = maxList(rest)
    if summon[Ordering[T]].gt(x, maxRest) then x
    else maxRest

```

Внимательно взгляните на эту последнюю версию `maxList`. В ее теле нет ни единого упоминания параметра `ordering`. С тем же успехом второй параметр можно было бы назвать `comparator`:

```

def maxList[T](elements: List[T])
  (using comparator: Ordering[T]): T = // то же тело...

```

Если на то пошло, данная версия тоже работает:

```

def maxList[T](elements: List[T])
  (using iceCream: Ordering[T]): T = ??? // то же тело...

```

Ввиду распространенности этого приема Scala позволяет опустить имя данного параметра и сократить заголовок метода с помощью *границы контекста*. Граница контекста позволяет сделать так, чтобы сигнатура `maxList`

выглядела как в листинге 23.5. Границей в данном случае служит синтаксис `[T : Ordering]`, и она имеет двойное назначение. Во-первых, она вводит параметр типа `T`, как это обычно происходит. Во-вторых, она добавляет параметр `using` типа `Ordering[T]`. В предыдущей версии `maxList` этот параметр назывался `ordering`, но при использовании границы контекста его имя неизвестно. Как было показано ранее, вам зачастую не обязательно знать имя параметра.

Листинг 23.5. Функция с границей контекста

```
def maxList[T : Ordering](elements: List[T]): T =
  elements match
  case List() =>
    throw new IllegalArgumentException("empty list!")
  case List(x) => x
  case x :: rest =>
    val maxRest = maxList(rest)
    if summon[Ordering[T]].gt(x, maxRest) then x
    else maxRest
```

К границе контекста можно относиться как к информации о параметре типа. Когда вы пишете `[T <: Ordered[T]]`, это означает, что `T` является типом `Ordered[T]`. Для сравнения `[T : Ordering]` говорит не столько о том, что такое `T`, сколько о существовании какой-то процедуры упорядочения, относящейся к `T`.

Границы контекста — это, в сущности, синтаксический сахар для классов типов. Сам факт наличия этого сокращенного синтаксиса в `Scala` является свидетельством того, насколько полезной идиомой служат классы типов для программирования на этом языке.

23.3. Главные методы

Как упоминалось в шаге 2 в главе 2, в `Scala` главный метод можно объявить с помощью аннотации `@main`. Например:

```
// В файле echoargs.scala
@main def echo(args: String*) =
  println(args.mkString(" "))
```

Как было проиллюстрировано в главе 2, этот код можно выполнять как скрипт, если запустить `scala` с именем исходного файла, `echoargs.scala`:

```
$ scala echoargs.scala Running as a script
Running as a script
```

Этот код также можно выполнить в качестве приложения, скомпилировав исходный файл и затем снова вызвав `scala`. Но в этом случае нужно указать имя *главного метода*, `echo`:

```
$ scalac echoargs.scala

$ scala echo Running as an application
Running as an application
```

Все главные методы, показанные до сих пор, принимают один аргумент с повторяющимися параметрами `String*`, это не является обязательным требованием. В Scala главные методы могут принимать любое количество аргументов любых типов. Вот, например, главный метод, который ожидает получить строку и целое число:

```
// В файле repeat.scala
@main def repeat(word: String, count: Int) =

  val msg =
    if count > 0 then
      val words = List.fill(count)(word)
      words.mkString(" ")
    else
      "Please enter a word and a positive integer count."

  println(msg)
```

Учитывая объявление этого главного метода, при запуске `repeat` в командной строке необходимо указать одно строковое и одно целочисленное значение:

```
$ scalac repeat.scala

$ scala repeat hello 3
hello hello hello
```

Откуда Scala знает, как превратить строковой параметр "3" в число? Для этого используется класс типов `FromString`, который является членом `scala.util.CommandLineParser`. Его объявление показано в листинге 23.6.

Листинг 23.6. Трейт класса типов `FromString`

```
trait FromString[T]:
  def fromString(s: String): T
```

В стандартной библиотеке Scala определены гивены `FromString` для нескольких часто используемых типов, таких как `String` и `Int`. Эти экземпляры на-

ходятся в объекте-компаньоне `FromString`. Если вы хотите написать главный метод, принимающий нестандартный тип, можете объявить для этого типа `given`-экземпляр класса типов `FromString`.

Представьте, к примеру, что вам нужно дополнить главный метод `repeat` третьим параметром командной строки, описывающим одно из трех настроек: удивлен, зол и нейтрален. Чтобы описать настроение, можно определить перечисление `Mood`, как показано в листинге 23.7.

Листинг 23.7. Перечисление `Mood`

```
// В файле moody.scala
enum Mood:
  case Surprised, Angry, Neutral
```

Создав перечисление, вы можете улучшить главный метод `repeat` так, чтобы он принимал `Mood` в качестве третьего параметра. Это показано в листинге 23.8.

Листинг 23.8. Главный метод, принимающий нестандартный тип

```
// В файле moody.scala
val errmsg =
  "Please enter a word, a positive integer count, and\n" +
  "a mood (one of 'angry', 'surprised', or 'neutral')"
```

```
@main def repeat(word: String, count: Int, mood: Mood) =
  val msg =
    if count > 0 then
      val words = List.fill(count)(word.trim)
      val punc =
        mood match
          case Mood.Angry => "!"
          case Mood.Surprised => "?"
          case Mood.Neutral => ""
      val sep = punc + " "
      words.mkString(sep) + punc
    else errmsg

  println(msg)
```

Теперь остается только научить компилятор преобразовывать строковой параметр командной строки в `Mood`. В этой связи для типа `Mood` нужно определить экземпляр `FromString`. Хорошим местом размещения этого экземпляра будет объект-компаньон `Mood`, так как компилятор заглянет в него при поиске гивена `FromString[Mood]`. В листинге 23.9 показан один из вариантов реализации.

Листинг 23.9. Given-экземпляр FromString для Mood

// В файле moody.scala
object Mood:

```
import scala.util.CommandLineParser.FromString

given moodFromString: FromString[Mood] with
  def fromString(s: String): Mood =
    s.trim.toLowerCase match
      case "angry" => Mood.Angry
      case "surprised" => Mood.Surprised
      case "neutral" => Mood.Neutral
      case _ => throw new IllegalArgumentException(errmsg)
```

Имея в своем распоряжении определение given-экземпляра FromString[Mood], вы можете запустить свое приложение repeat:

```
$ scalac moody.scala

$ scala repeat hello 3 neutral
hello hello hello

$ scala repeat hello 3 surprised
hello? hello? hello?

$ scala repeat hello 3 angry
hello! hello! hello!
```

Решение, основанное на классе типов, хорошо подходит для анализаторов аргументов командной строки, которые принимают главные методы, так как эта возможность требуется лишь для определенных типов, которые в остальном не имеют друг к другу никакого отношения. Помимо String и Int, объект-компаньон FromString определяет given-экземпляры FromString для Byte, Short, Long, Boolean, Float и Double. Если прибавить к этому given-экземпляр FromString[Mood], размещенный в объекте-компаньоне Mood из листинга 23.9, множество типов, составляющих класс типов FromString, будет выглядеть как на рис. 23.4.

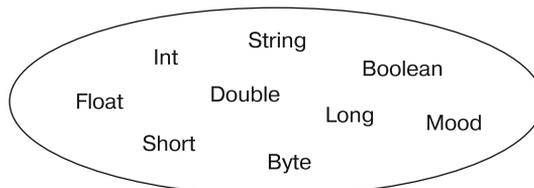


Рис. 23.4. Множество типов T с given-экземплярами FromString[T]

23.4. Многостороннее равенство

В Scala 2 было реализовано *универсальное равенство*, которое позволяет проверять на равенство два любых объекта с использованием `==` и `!=`. Пользователям было легко понять этот подход, и он хорошо сочетался с методом `equals` из Java, с помощью которого можно сравнить два любых экземпляра `Object`. Это также сделало возможной поддержку в Scala 2 *совместного равенства* — это когда на равенство проверяются разные взаимодействующие между собой типы. Совместное равенство, к примеру, позволило Scala 2 продолжить традицию Java, состоящую в том, что проверку равенства `Int` и `Long` можно осуществлять без явного приведения первого типа ко второму.

Тем не менее у универсального равенства был один серьезный недостаток: оно скрывало программные дефекты. Например, в Scala 2 вы могли проверить на равенство строки и объекты `Option`. Например:

```
scala> "hello" == Option("hello") // (в Scala 2)
val res0: Boolean = false
```

И хотя на этапе выполнения Scala 2 дает верный ответ (строка `"hello"` действительно не равна `Option("hello")`), не существует такой строки, которая была бы равна какому-либо объекту `Option`. Результатом этого сравнения всегда будет `false`. Таким образом, любая проверка строки и объекта `Option` на равенство, скорее всего, является ошибкой, которую не выявил компилятор Scala 2. Такого рода ошибки можно легко допустить во время рефакторинга — например, вы можете поменять тип переменной с `String` на `Option[String]` и не заметить, что теперь в другом месте на равенство проверяются `String` и `Option[String]`.

Для сравнения: попытка проделать то же самое в Scala 3 завершится ошибкой компиляции:

```
scala> "hello" == Option("hello") // (в Scala 3)
1 | "hello" == Option("hello")
  | ~~~~~
  | Values of types String and Option[String] cannot be
  | compared with == or !=
```

Это улучшение в безопасности Scala 3 достигается за счет новой возможности под названием *многостороннее равенство*. Оно заключается в том, что компилятор по-особому обращается с методами `==` и `!=`. Определения этих методов, показанные в листинге 23.10, не изменились в Scala 3 по сравнению со Scala 2. Поменялось лишь поведение компилятора: в Scala 3 универсальное сравнение преобразуется в многостороннее.

Листинг 23.10. Методы == и != в Scala 2 и Scala 3

```
// На примере класса Any:
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
```

Чтобы понять, как работает многостороннее равенство в Scala 3, будет полезно рассмотреть детали реализации универсального равенства в Scala 2. Вот как оно было устроено на уровне JVM: когда компилятор Scala 2 обнаруживал вызов == и !=, он сначала проверял, принадлежат ли сравниваемые значения к простым типам Java. Если да, то компилятор генерировал специальный байт-код для эффективной проверки этих простых типов на равенство. В противном случае, если один из типов был простым, а другой — нет, компилятор генерировал код для упаковки простого значения. В итоге оба операнда имели ссылочные типы. Затем компилятор генерировал код, который сначала определял, равен ли левый операнд null. Если да, то сгенерированный код проводил ту же проверку для правого операнда, чтобы получить результат типа Boolean. Это гарантировало, что вызовы == и != не могут сгенерировать NullPointerException. В противном случае сгенерированный код вызывал из левого операнда, который, как уже было установлено, не равнялся null, метод equals, передавая ему правый операнд.

В Scala 3 компилятор выполняет точно такие же действия, но сначала проверяет, допускается ли это сравнение. Для этого он ищет given-экземпляр класса типов с именем CanEqual. Вот его определение:

```
sealed trait CanEqual[-L, -R]
```

Трейт CanEqual принимает два параметра типов, L и R¹. L — это тип левого операнда проверки на равенство, а R — тип правого операнда. CanEqual не предоставляет никакого метода для непосредственной проверки на равенство двух объектов типа L и R, поскольку в Scala 3 эта операция по-прежнему выполняется за счет методов == и !=. Если вкратце, то вместо того, чтобы предоставлять операцию проверки типов L и R на равенство, как можно было бы ожидать от обычного класса типов, CanEqual просто *дает разрешение* на проведение этой операции с помощью == и !=.

¹ Класс типов обычно означает множество типов, для которых доступны given-экземпляры трейта, принимающего один параметр типа, но CanEqual можно считать трейтом, определяющим множество, которое состоит из пар типов. Например, проверка String и Option[String] на равенство не скомпилируется в Scala 3, так как тип (String, Option[String]) не входит в множество, из которого состоит класс типов CanEqual.

Как описывалось в разделе 18.6, знак минус рядом с параметром типа означает, что трейт `CanEqual` является контравариантным как по `L`, так и по `R`. Ввиду этой контравариантности `CanEqual[Any, Any]` является подтипом любого другого типа, `CanEqual[L, R]`, независимо от `L` или `R`. В результате экземпляр `CanEqual[Any, Any]` можно использовать для выдачи разрешения на проверку на равенство любых двух типов. Например, если `given`-экземпляр `CanEqual[Int, Int]` нужен, чтобы разрешить проверку на равенство двух значений `Int`, `given`-экземпляра `CanEqual[Any, Any]` будет достаточно, так как `CanEqual[Any, Any]` является подтипом `CanEqual[Int, Int]`. Благодаря этому факту `CanEqual` является запечатанным трейтом всего с одним экземпляром, который имеет универсально применимый тип `CanEqual[Any, Any]`. Этот объект носит имя `derived` и объявлен в объекте-компаньоне `CanEqual`:

```
object CanEqual:
  object derived extends CanEqual[Any, Any]
```

Следовательно, чтобы предоставить `given`-экземпляр `CanEqual[L, R]`, независимо от того, что собой представляют типы `L` и `R`, вы должны использовать один и только один экземпляр `CanEqual`, `CanEqual.derived`.

В целях обратной совместимости компилятор Scala 3 позволяет проводить некоторые проверки на равенство по умолчанию, даже если для нужного типа *недоступен* `given`-экземпляр `CanEqual`. Компилятор позволяет проверить, равны ли значения типов `L` и `R`, даже в случае отсутствия `given`-экземпляра типа `CanEqual[L, R]`, если выполняется любое из следующих условий.

1. `L` и `R` являются одним и тем же типом.
2. После выполнения *подъема* типа `L` является подтипом `R` или наоборот¹.
3. Для типов `L` и `R` не существует *рефлексивного* `given`-экземпляра `CanEqual`. Рефлексивными являются экземпляры, которые разрешают сравнение типа с самим собой, как в случае с `CanEqual[L, L]`.

Третье правило гарантирует, что, как только вы предоставите рефлексивный `given`-экземпляр `CanEqual`, позволив тем самым сравнивать тип с самим собой, этот тип больше нельзя будет сравнивать ни с каким другим типом — разве что существует `given`-экземпляр `CanEqual`, который позволяет это делать. В целях обратной совместимости Scala 3 фактически возвращается

¹ Чтобы поднять тип, компилятор заменяет ссылки на абстрактные типы в его ковариантных позициях их верхней границей и подставляет вместо типов уточнения, находящиеся в ковариантных позициях типа, их родителей.

к универсальному равенству по умолчанию при сравнении типов, для которых не было определено рефлексивных экземпляров `CanEqual`.

Scala 3 предоставляет `given`-экземпляры для нескольких типов из стандартной библиотеки, включая рефлексивные экземпляры для строк. Вот почему проверка `String` и `Option[String]` на равенство по умолчанию запрещена. `Given`-экземпляры `CanEqual[String, String]`, предоставляемого стандартной библиотекой, достаточно для того, чтобы компилятор Scala 3 по умолчанию не позволял проверять на равенство `String` и `Option[String]`.

Это поведение по умолчанию дает возможность беспрепятственного перехода со Scala 2 на Scala 3, так как в существующем пользовательском коде, который переносится из Scala 2, нет никаких экземпляров `CanEqual` для этих типов. Представьте, к примеру, что ваш имеющийся проект на Scala 2 содержит класс `Apple`, определенный следующим образом:

```
case class Apple(size: Int)
```

И где-то в вашем коде происходит сравнение двух яблок:

```
val appleTwo = Apple(2)
val appleTwoToo = Apple(2)
appleTwo == appleTwoToo // true
```

Это сравнение по умолчанию будет компилироваться и работать в Scala 3, так как левая и правая стороны имеют один и тот же тип. Однако в Scala 3 компилятор по умолчанию все так же позволяет сравнивать типы, для которых не существует рефлексивных `given`-экземпляров `CanEqual`, поэтому следующая нежелательная проверка на равенство по-прежнему успешно компилируется:

```
case class Orange(size: Int)
val orangeTwo = Orange(2)
appleTwo == orangeTwo // false
```

Это сравнение `Apple` и `Orange`, скорее всего, является ошибочным, так как оно всегда возвращается `false`. Чтобы выполнить полную проверку работоспособности всех сравнений на равенство в Scala 3, даже если для задействованных типов не было определено рефлексивных `given`-экземпляров, вы можете включить «строгое равенство». Для этого нужно либо передать компилятору параметр командной строки `-language:strictEquality`, либо добавить следующую инструкцию импорта в исходный файл:

```
import scala.language.strictEquality
```

После включения строгого равенства сравнение яблок и апельсинов будет заканчиваться ожидаемой ошибкой компиляции:

```
scala> appleTwo == orangeTwo
1 | appleTwo == orangeTwo
  | ~~~~~
  | Values of types Apple and Orange cannot be
  | compared with == or !=
```

К сожалению, вместе с этим теперь возникает другая, *нежелательная* ошибка компиляции, связанная с корректным сравнением двух яблок:

```
scala> appleTwo == appleTwoToo
1 | appleTwo == appleTwoToo
  | ~~~~~
  | Values of types Apple and Apple cannot be
  | compared with == or !=
```

Чтобы сделать возможным это сравнение в режиме строгого равенства, нужно предоставить экземпляр `CanEqual`, который разрешает проверять яблоки на равенство с другими яблоками. Для этого в объекте-компаньоне `Apple` можно определить явный `given`-экземпляр, как показано в листинге 23.11 (хотя это идет вразрез с философией языка).

Листинг 23.11. Явно определенный (но не идиоматический) провайдер `CanEqual`

```
case class Apple(size: Int)
object Apple:
  given canEq: CanEqual[Apple, Apple] = CanEqual.derived
```

Вместо этого лучше указать, что вы хотите *вывести* для своего экземпляра `Apple` экземпляр класса типов `CanEqual`, как показано в листинге 23.12.

Листинг 23.12. Предоставление `CanEqual` с помощью инструкции `derives`

```
case class Apple(size: Int) derives CanEqual // идиоматически
```

Этот идиоматический подход основан на *выводе класса типов*, который позволяет делегировать определение `given`-экземпляра класса типов члену с именем `derived` в объекте-компаньоне этого класса. Это заставляет компилятор вставить заданный провайдер (как тот, что был показан в листинге 23.11) в объект-компаньон `Apple`.

Об инструкции `derives` можно было бы рассказывать и дальше, так как большинство методов `derived` генерируют экземпляры класса типов с помощью метапрограммирования на этапе компиляции.

Теперь благодаря тому, что вы определили экземпляр `CanEqual[Apple, Apple]` с использованием инструкции `derives`, компилятор позволит вам сравнивать яблоки в режиме строгого равенства:

```
appleTwo == appleTwoToo // тоже true
```

23.5. Неявные преобразования

Неявные преобразования были первой неявной конструкцией в Scala. Они создавались для того, чтобы помочь сделать код яснее и избавиться от шаблонных приведений типов. Например, стандартная библиотека Scala определяет неявное преобразование `Int` в `Long`. Если передать `Int` методу, который ожидает `Long`, компилятор автоматически приведет тип `Int` к `Long`, не требуя явного вызова функции преобразования, такой как `toLong`. Поскольку любое значение `Int` можно безопасно привести к значению `Long` и оба этих типа представляют целые числа в дополнительном коде, это неявное преобразование может облегчить чтение исходного кода за счет устранения повторяющихся участков.

Однако со временем неявные преобразования вышли из моды, так как они могли делать код не только яснее за счет удаления шаблонных участков, но и более запутанным ввиду отсутствия четкости. В Scala были добавлены другие конструкции, которые являются более удачными альтернативами неявных преобразований, включая методы расширения и контекстные параметры. В Scala 3 для неявных преобразований осталось всего несколько сценариев применения. И хотя они по-прежнему поддерживаются, для их использования нужно импортировать флаг переключения возможностей, иначе компилятор выдаст предупреждение.

Вот как они работают. Если компилятор Scala приходит к выводу, что указанный тип не отвечает ожидаемому, он начинает искать неявное преобразование, которое сможет исправить ошибку выбора кандидата. Иными словами, каждый раз, когда компилятор встречает `X` там, где требуется `Y`, он ищет неявное преобразование, которое приводит `X` к `Y`. Представьте, к примеру, что у вас есть крошечный тип¹ для представления поля «улица» в почтовом адресе:

```
case class Street(value: String)
```

¹ Крошечные типы обсуждались в разделе 17.4.

И у вас имеется экземпляр этого класса:

```
val street = Street("123 Main St")
```

В этом случае вы не сможете инициализировать переменную типа `String` с помощью `Street`:

```
scala> val streetStr: String = street
1 | val streetStr: String = street
  | ~~~~~
  |
  | Found: (street : Street)
  | Required: String
```

Вместо этого вам придется вручную привести `Street` к `String` путем вызова `street.value`:

```
val streetStr: String = street.value // 123 Main St
```

Этот код легко понять, но у вас может быть ощущение того, что вызов `value` из `String` для преобразования значения в `String` является шаблонным и малоинформативным кодом. Поскольку тип `Street` всегда можно безопасно привести к типу `String`, который лежит в его основе, вам, возможно, захочется предоставить неявное преобразование из `Street` в `String`. В Scala 3 для этого нужно определить `given`-экземпляр типа `Conversion[Street, String]`¹, который является потомком типа функции `Street => String`. Вот его определение:

```
abstract class Conversion[-T, +U] extends (T => U):
  def apply(x: T): U
```

Поскольку у трейта `Conversion` есть всего один абстрактный метод, для определения экземпляра зачастую можно использовать функциональный литерал SAM². Следовательно, неявное приведение `Street` к `String` можно определить так:

```
given streetToString: Conversion[Street, String] = _.value
```

Чтобы не получать предупреждения во время компиляции при использовании неявных преобразований, для их включения следует либо передать компилятору параметр `-language:implicitConversions`, либо локально указать следующую инструкцию импорта:

```
import scala.language.implicitConversions
```

¹ В Scala 3 неявное преобразование можно также определить с помощью `implicit def`, чтобы сохранить совместимость со Scala 2. В будущем этот подход может быть признан устаревшим.

² Методы SAM были описаны в разделе 8.9.

Включив неявные преобразования, вы можете написать следующее (при условии, что гивен `streetToString` находится в области видимости в качестве единого идентификатора):

```
val streetStr: String = street
```

Здесь компилятор встречает `Street` в контексте, в котором должен быть указан тип `String`, и воспринимает это как обычную ошибку типизации. Но прежде, чем сдаваться, он ищет неявное преобразование `Street` в `String`. В данном случае он находит `streetToString`. После этого компилятор автоматически вставляет `streetToString` в приложение. При этом внутри код принимает следующий вид:

```
val streetStr: String = streetToString(street)
```

Это *неявное* преобразование в буквальном смысле этого слова. Здесь нет явного запроса на приведение типов. Вместо этого вы поместили метод `streetToString` как доступное неявное преобразование, разместив его в области видимости и определив его как `given`. В результате компилятор будет автоматически использовать его каждый раз, когда `Street` необходимо привести к `String`.

Если же вы хотите определить неявное преобразование, позаботьтесь о том, чтобы оно всегда было подходящим. Например, приведение `Double` к `Int` неявным образом может вызвать недоумение, поскольку автоматическая потеря точности значения является сомнительной идеей. Поэтому мы на самом деле рекомендуем не преобразование. Намного логичнее двигаться в обратном направлении: от более конкретного типа к более общему. Например, `Int` можно преобразовать в `Double` без потери точности, поэтому неявное приведение `Int` к `Double` имеет смысл. На самом деле именно это и происходит. Объект `scala.Predef`, который автоматически импортируется любой программой на Scala, определяет неявные преобразования «меньших» числовых типов в «большие», в том числе и приведение `Int` к `Double`.

Вот почему в Scala значения `Int` можно хранить в переменных типа `Double`. В системе типов для этого не предусмотрено отдельного правила; это просто применение неявного преобразования¹.

¹ Тем не менее компилятор Scala относится к этому преобразованию по-особому, переводя его в специальный байт-код `i2d`. Благодаря этому скомпилированный образ получается таким же, как и в Java.

23.6. Пример использования класса типов: сериализация JSON

В разделе 23.1 мы упоминали сериализацию в качестве примера поведения, применимого к типам, которые в остальном не имеют ничего общего и, следовательно, являются хорошими кандидатами на попадание в класс типов. Напоследок в этой главе мы хотели бы проиллюстрировать использование класса типов для поддержки сериализации в JSON. Чтобы не усложнять этот пример, мы проигнорируем десериализацию, хотя обычно оба эти процесса реализуются в одной и той же библиотеке.

JSON — широко используемый формат обмена данными между клиентами на JavaScript и серверными приложениями¹. Он определяет форматы для представления строк, чисел, логических значений, массивов и объектов. Таким образом, все, что вы хотите сериализовать в JSON, должно быть выражено в одном из этих пяти типов данных. Строки JSON выглядят как строковые литералы Scala, такие как `"tennis"`. Числа JSON, представляющие целочисленные значения, аналогичны литералам `Int` в Scala, таким как `10`. Логическое значение JSON может быть равно либо `true`, либо `false`. Объект JSON — это набор пар «ключ — значение», разделенных запятыми и заключенных в фигурные скобки; ключ представляет собой строковое имя. Массив JSON — это список типов данных JSON, разделенных запятыми и заключенных в квадратные скобки. В JSON также определено значение `null`. Вот пример объекта, содержащего по одному члену каждого из остальных четырех типов JSON, плюс член `null`:

```
{
  "style": "tennis",
  "size": 10,
  "inStock": true,
  "colors": ["beige", "white", "blue"],
  "humor": null
}
```

В этом примере мы сериализуем значения `String` Scala в строки JSON, `Int` и `Long` в числа JSON, `Boolean` в логические значения JSON, `List` в массивы JSON и несколько других типов в объекты JSON. Необходимость сериализации типов из стандартной библиотеки Scala, таких как `Int`, подчеркивает то, насколько трудно было бы решить эту задачу путем примеси трейта в класс,

¹ JSON расшифровывается как JavaScript Object Notation.

который вы хотите сериализовать. Вы можете определить такой трейт и назвать его, скажем, `JsonSerializable`. Он может предоставлять метод `toJson`, который генерирует текст JSON для этого объекта. Затем вы могли бы пришивать `JsonSerializable` в свои собственные классы и реализовывать метод `toJson`. Однако с такими типами, как `String`, `Int`, `Long`, `Boolean` или `List`, это не работает, так как их нельзя изменять.

Подход на основе класса типов лишен этой проблемы. Вы можете определить иерархию классов, целиком ориентированную на сериализацию объектов абстрактного типа `T` в JSON, не требуя при этом, чтобы классы, которые вы хотите сериализовать, наследовали общий супертрейт. Вместо этого можно определить `given`-экземпляр трейта класса типов для каждого типа, предназначенного для сериализации в JSON. Такой трейт, с именем `JsonSerializer`, показан в листинге 23.13. Он принимает один параметр типа, `T`, и предлагает метод `serialize`, который берет экземпляр `T` и преобразует его в строку JSON.

Листинг 23.13. Класс типов для сериализации в JSON

```
trait JsonSerializer[T]:
  def serialize(o: T): String
```

Чтобы у ваших пользователей была возможность вызывать метод `toJson` из сериализуемых классов, можно определить метод расширения. Как уже обсуждалось в разделе 22.5, подходящим местом для размещения этого метода является трейт самого класса типов. Если вы выберете этот вариант, метод `toJson` будет доступен в типе `T` всегда, когда `JsonSerializer[T]` находится в области видимости. Трейт `JsonSerializer`, улучшенный с помощью этого метода расширения, показан в листинге 23.14.

Листинг 23.14. Класс типов для сериализации в JSON с методом расширения

```
trait JsonSerializer[T]:
  def serialize(o: T): String

  extension (a: T)
    def toJson: String = serialize(a)
```

Следующим шагом было бы логично определить `given`-экземпляры класса типов для `String`, `Int`, `Long` и `Boolean`. Подходящим местом для их размещения будет объект-компаньон `JsonSerializer`, поскольку, как описывалось в разделе 21.2, компилятор ищет в нем нужный `given`-экземпляр, если его не удалось найти в области видимости. Эти `givens` можно определить так, как показано в листинге 23.15.

Листинг 23.15. Объект-компаньон сериализатора JSON с гивенами

```
object JsonSerializer:
  given stringSerializer: JsonSerializer[String] with
    def serialize(s: String) = s"\$s\"
  given intSerializer: JsonSerializer[Int] with
    def serialize(n: Int) = n.toString
  given longSerializer: JsonSerializer[Long] with
    def serialize(n: Long) = n.toString
  given booleanSerializer: JsonSerializer[Boolean] with
    def serialize(b: Boolean) = b.toString
```

Импорт метода расширения

Вам может пригодиться возможность импортировать метод расширения, добавляющий метод `toJson` в любые типы `T`, для которых доступен `JsonSerializer[T]`. Метод расширения, определенный в листинге 23.14, на это не способен, так как он делает `toJson` доступным для `T` только в случае, если `JsonSerializer[T]` находится в *области видимости*. В противном случае он не сработает, даже если `JsonSerializer[T]` присутствует в объекте-компаньоне для `T`. Чтобы упростить импорт метода расширения, можете поместить его в объект-одиночку, такой как показан в листинге 23.16. Этот метод содержит инструкцию `using`, которая требует, чтобы гивен `JsonSerializer[T]` был доступен для типа `T`, к которому этот метод применяется.

Листинг 23.16. Метод расширения для удобного импорта

```
object toJsonMethods:
  extension [T](a: T)(using jser: JsonSerializer[T])
    def toJson: String = jser.serialize(a)
```

Имея в своем распоряжении объект `toJsonMethods`, вы можете поэкспериментировать с сериализаторами в REPL. Вот несколько примеров их использования:

```
import toJsonMethods.*
"tennis".toJson // "tennis"
10.toJson      // 10
true.toJson    // true
```

Будет полезно сравнить два метода расширения: один в объекте `toJsonMethods` из листинга 23.16, а другой — в трейте `JsonSerializer` из листинга 23.14. Метод расширения `toJsonMethods` принимает `JsonSerializer[T]` в качестве параметра `using`, а метод расширения в `JsonSerializer` этого не делает, так как он по определению является *членом* `JsonSerializer[T]`. Таким образом, если `toJson` в `toJsonMethods` вызывает `serialize` из пере-

данной ссылки `JsonSerializer` с именем `jser`, то метод `toJson` в трейте `JsonSerializer` вызывает `serialize` из `this`.

Сериализация объектов предметной области

Теперь представьте, что вам нужно сериализовать в JSON экземпляры определенных классов в модели вашей предметной области, включая адресную книгу, показанную в листинге 23.17. Эта книга содержит список контактов, у каждого из которых есть произвольное количество адресов и телефонных номеров (от 0 и больше)¹.

Листинг 23.17. Классы-образцы для адресной книги

```
case class Address(  
  street: String,  
  city: String,  
  state: String,  
  zip: Int  
)  
  
case class Phone(  
  countryCode: Int,  
  phoneNumber: Long  
)  
  
case class Contact(  
  name: String,  
  addresses: List[Address],  
  phones: List[Phone]  
)  
  
case class AddressBook(contacts: List[Contact])
```

Строка JSON для адресной книги формируется из строк JSON ее вложенных объектов. Таким образом, чтобы сгенерировать строку JSON для адресной книги, каждый из ее вложенных объектов должен поддерживать преобразование в формат JSON. Например, каждый экземпляр `Contact` в поле `contacts` должен быть представлен в формате JSON этого контакта. Каждый экземпляр `Address` контакта должен быть преобразован в JSON этого адреса. Следовательно, для сериализации `AddressBook` необходимо сериализовать в JSON каждый объект, из которого состоит адресная книга. Поэтому будет логично определить сериализаторы для всех объектов предметной области.

¹ Для атрибутов этих классов было бы лучше определить крошечные типы, как описывалось в разделе 17.4. Но, чтобы не усложнять этот пример, мы будем использовать типы `String` и `Int`.

Хорошим местом размещения `given`-экземпляров `JsonSerializer` для ваших объектов предметной области являются их объекты-компаньоны. В листинге 23.18 показано, как вы можете, к примеру, определить сериализаторы для `Address` и `Phone`. В методах `serialize` мы импортируем и используем метод расширения `toJson` из объекта `ToJsonMethods`, показанного в листинге 23.16, но переименовываем его в `asJson`. Это необходимо, чтобы избежать конфликта с одноименным методом расширения `toJson`, унаследованным от `JsonSerializer` (см. листинг 23.14).

Листинг 23.18. Сериализаторы JSON для `Address` и `Phone`

```
object Address:
  given addressSerializer: JsonSerializer[Address] with
    def serialize(a: Address) =
      import ToJsonMethods.{toJson as asJson}
      s""|{
        | "street": ${a.street.asJson},
        | "city": ${a.city.asJson},
        | "state": ${a.state.asJson},
        | "zip": ${a.zip.asJson}
      }"".stripMargin

object Phone:
  given phoneSerializer: JsonSerializer[Phone] with
    def serialize(p: Phone) =
      import ToJsonMethods.{toJson as asJson}
      s""|{
        | "countryCode": ${p.countryCode.asJson},
        | "phoneNumber": ${p.phoneNumber.asJson}
      }"".stripMargin
```

Сериализация списков

Два других объекта предметной области, `Contact` и `AddressBook`, содержат списки. Поэтому для их сериализации было бы полезно иметь общую процедуру преобразования типов `List Scala` в массивы JSON. Массив JSON представляет собой список типов данных JSON, разделенных запятыми и заключенных в квадратные скобки, поэтому `List[T]` можно сериализовать для любого типа `T`, при условии существования `JsonSerializer[T]`. В листинге 23.19 показан `given JsonSerializer` для списков, который будет генерировать массив JSON из `List`, если для типа элементов списка существует `JsonSerializer`.

Листинг 23.19. `Given`-сериализатор JSON для списков

```
object JsonSerializer:
  // гивены для строк, целых чисел и логических значений...
  given listSerializer[T](using
```


вложенный объект не реализует `Serializable`, вы получите исключение на этапе выполнения. Эта ошибка может произойти в обоих языках, но если в Java она происходит во время работы программы, то в Scala благодаря классам типов она проявляется на этапе компиляции.

Напоследок стоит отметить, что возможность вызова `toJson` в теле функции, переданной в `map` ("`toJson`" в `t => t.toJson`), объясняется наличием в области видимости гивена `JsonSerializer[T]`: анонимного параметра `using`, переданного в `listSerializer`. Метод расширения, который используется в этом случае, объявлен в самом трейте `JsonSerializer`, представленном в листинге 23.14.

Собираем все вместе

Теперь, имея в своем распоряжении способ сериализации списков, вы можете применить его в сериализаторах для `Contact` и `AddressBook`. Это показано в листинге 23.20. Как и прежде, при импорте метода расширения `toJson` нужно переименовать в `asJson`, чтобы избежать конфликта имен.

Листинг 23.20. Given-сериализаторы JSON для `Contact` и `AddressBook`

```
object Contact:
  given contactSerializer: JsonSerializer[Contact] with
    def serialize(c: Contact) =
      import ToJsonMethods.{toJson as asJson}
      s""""{
        | "name": ${c.name.asJson},
        | "addresses": ${c.addresses.asJson},
        | "phones": ${c.phones.asJson}
        |}"""".stripMargin

object AddressBook:
  given addressBookSerializer: JsonSerializer[AddressBook] with
    def serialize(a: AddressBook) =
      import ToJsonMethods.{toJson as asJson}
      s""""{
        | "contacts": ${a.contacts.asJson}
        |}"""".stripMargin
```

У нас все готово для сериализации адресной книги в JSON. В качестве примера возьмем экземпляр `AddressBook`, показанный в листинге 23.21, на который ссылается переменная `addressBook`. Импортировав из `ToJsonMethods` метод расширения `toJson`, вы сможете сериализовать эту адресную книгу с помощью вызова:

```
addressBook.toJson
```

Результат в формате JSON показан в листинге 23.22.

Листинг 23.21. AddressBook

```
val addressBook =
    AddressBook(
        List(
            Contact(
                "Bob Smith",
                List(
                    Address(
                        "12345 Main Street",
                        "San Francisco",
                        "CA",
                        94105
                    ),
                    Address(
                        "500 State Street",
                        "Los Angeles",
                        "CA",
                        90007
                    )
                )
            ),
            List(
                Phone(
                    1,
                    5558881234
                ),
                Phone(
                    49,
                    5558413323
                )
            )
        )
    )
)
```

Листинг 23.22. Адресная книга, представленная в формате JSON

```
{
  "contacts": [{
    "name": "Bob Smith",
    "addresses": [{
      "street": "12345 Main Street",
      "city": "San Francisco",
      "state": "CA",
      "zip": 94105
    }, {
      "street": "500 State Street",
      "city": "Los Angeles",
      "state": "CA",
```

```
    "zip": 90007
  }],
  "phones": [{
    "countryCode": 1,
    "phoneNumber": 5558881234
  }, {
    "countryCode": 49,
    "phoneNumber": 5558413323
  }]
}]
}
```

Конечно, настоящая библиотека для работы с JSON была бы намного сложнее того, что вы увидели в этом примере. Вам, скорее всего, следовало бы воспользоваться средствами метапрограммирования Scala, чтобы автоматизировать генерацию экземпляров `JsonSerializer` за счет вывода класса типов.

Резюме

В этой главе вы познакомились с классами типов и рассмотрели несколько примеров. Классы типов — это основополагающий способ реализации специального полиморфизма в Scala. Тот факт, что для классов типов в Scala предусмотрен синтаксический сахар в виде границ контекста, говорит о том, насколько важным является данный подход к проектированию в этом языке. Вам встречалось несколько способов применения классов типов: для главных методов, безопасных проверок на равенство, неявных преобразований и сериализации JSON. Надеемся, эти примеры дали вам представление о том, в каких ситуациях классы типов являются подходящим архитектурным решением. В следующей главе мы сменим тему и подробно рассмотрим библиотеку коллекций Scala.

24

Углубленное изучение коллекций

В Scala включена весьма элегантная и эффективная библиотека коллекций. На первый взгляд API коллекций представляется незаметным, однако вызванные им изменения в вашем стиле программирования могут быть весьма существенными. Зачастую это похоже на работу на высоком уровне с основными строительными блоками программы, представляющими собой скорее коллекции, чем их элементы. Этот новый стиль программирования требует некоторой адаптации. К счастью, ее облегчает ряд привлекательных свойств коллекций Scala. Они просты в использовании, лаконичны в описании, безопасны, быстры в работе и универсальны.

- **Простота использования.** Небольшого словаря, содержащего от 20 до 50 методов, вполне достаточно для решения основного набора задач всего за пару операций. Не нужно морочить голову сложными циклическими структурами или рекурсиями. Стабильные коллекции и операции без побочных эффектов означают, что вам не следует опасаться случайного повреждения существующих коллекций новыми данными. Взаимовлияние итераторов и обновления коллекций исключено.
- **Лаконичность.** То, что раньше занимало от одного до нескольких циклов, теперь можно выразить всего одним словом. Можно выполнять функциональные операции, задействуя упрощенный синтаксис, и без особых усилий комбинировать операции таким образом, чтобы в результате получалось нечто похожее на обычные алгебраические формулы.
- **Безопасность.** Чтобы разобраться в этом вопросе, нужен определенный опыт. Природа коллекций Scala с их статической типизацией и функциональностью означает, что подавляющее большинство потенциальных

ошибок отлавливается еще во время компиляции. Это достигается благодаря следующему:

- сами операции над коллекциями используются довольно широко, а следовательно, прошли проверку временем;
- использование операций над коллекциями делает ввод и вывод явным в виде параметров функций и результатов;
- эти явные входные и выходные данные являются предметом для статической проверки типов.

Суть заключается в том, что большинство случаев неверного использования кода проявится в виде ошибок типа. И вовсе не редкость, когда программы, состоящие из нескольких сотен строк кода, запускаются с первой же попытки.

- **Скорость.** Операции с коллекциями, имеющиеся в библиотеках, уже настроены и оптимизированы. В результате этого использование коллекций обычно отличается высокой эффективностью. Тщательно настроенные структуры данных и операции могут улучшить ситуацию, но в то же время, принимая решения, далекие от оптимальных, можно ее значительно ухудшить. Более того, коллекции были адаптированы под параллельную обработку на многоядерных системах. Параллельно обрабатываемые коллекции поддерживают те же самые операции, что и последовательно обрабатываемые, поэтому изучать новые операции и переписывать код не нужно. Последовательно обрабатываемые коллекции можно превратить в параллельно обрабатываемые, просто вызвав метод `par`.
- **Универсальность.** Коллекции реализуют одни и те же операции над любым типом там, где в этих операциях есть определенный смысл. Следовательно, используя сравнительно небольшой словарь операций, вы приобретаете множество возможностей. Например, концептуально строка является последовательностью символов. Следовательно, в коллекциях Scala строки поддерживают все операции с последовательностями. То же самое справедливо и для массивов.

В этой главе мы даем углубленное описание API имеющихся в Scala классов коллекций с точки зрения их использования. Краткий тур по библиотекам коллекций мы совершили в главе 15. В этой главе нам предстоит поучаствовать в более подробном путешествии, в ходе которого мы покажем все классы коллекций и все определенные в них методы, то есть представим все сведения, необходимые для использования коллекций Scala.

24.1. Изменяемые и неизменяемые коллекции

Как вам уже известно, в целом коллекции в Scala подразделяются на изменяемые и неизменяемые. Изменяемые могут обновляться или расширяться на месте. Это значит, вы можете изменять, добавлять или удалять элементы коллекции как побочный эффект. Неизменяемые коллекции, напротив, всегда сохраняют неизменный вид. Но все же есть операции, имитирующие добавление, удаление или обновление, но каждая из таких операций возвращает новую коллекцию, оставляя старую без изменений.

Все классы коллекций находятся в пакете `scala.collection` или в одном из его подпакетов: `mutable`, `immutable` и `generic`. Большинство классов коллекций, востребованных в клиентском коде, существуют в трех вариантах, которые имеют разные характеристики в смысле возможности изменения. Эти три варианта находятся в пакетах `scala.collection`, `scala.collection.immutable` и `scala.collection.mutable`.

Коллекция в пакете `scala.collection.immutable` гарантированно неизменяемая для всех. Такая коллекция после создания всегда остается неизменной. Поэтому можно полагаться на то, что многократные обращения к одному и тому же значению коллекции в разные моменты времени всегда будут давать коллекцию с теми же элементами.

Коллекция в пакете `scala.collection.mutable` известна тем, что имеет ряд операций, изменяющих коллекцию на месте. Эти операции позволяют вам создавать код для самостоятельного изменения коллекции. Но при этом нужно четко понимать, что существует вероятность обновления из других частей исходного кода, и защищаться от нее.

Коллекции в пакете `scala.collection` могут быть как изменяемыми, так и неизменяемыми. Например, `scala.collection.IndexedSeq[T]` является супертрейтом как для `scala.collection.immutable.IndexedSeq[T]`, так и для его изменяемого брата `scala.collection.mutable.IndexedSeq[T]`. В целом корневые коллекции в пакете `scala.collection` поддерживают трансформирующие операции, которые влияют на целую коллекцию, такие как `map` и `filter`. Неизменяемые коллекции в пакете `scala.collection.immutable` обычно дополняются операциями добавления и удаления отдельных значений, а изменяемые коллекции в пакете `scala.collection.mutable` дополняются некоторыми модифицирующими операциями с побочными эффектами.

Между корневыми и неизменяемыми коллекциями есть еще одно различие: клиенты неизменяемых коллекций получают гарантии того, что никто не

сможет внести в коллекцию изменения, а клиенты корневых коллекций знают только то, что не могут изменить коллекцию самостоятельно. Даже если статический тип такой коллекции не предоставляет операций для ее изменения, все же существует вероятность того, что в процессе выполнения программы она получит тип изменяемой коллекции, предоставив другим клиентам возможность вносить в нее изменения.

По умолчанию в Scala всегда выбираются неизменяемые коллекции. Например, если просто написать `Set` без какого-либо префикса или ничего не импортируя, то будет получено неизменяемое множество, а если написать `Iterable` — то неизменяемый объект с возможностью обхода его элементов, поскольку имеются привязки по умолчанию, импортируемые из пакета `scala`. Чтобы получить изменяемые версии по умолчанию, следует явно указать `collection.mutable.Set` или `collection.mutable.Iterable`.

Последний пакет в иерархии коллекций — `collection.generic`. В нем содержатся строительные блоки для абстрагирования поверх конкретных коллекций. Впрочем, постоянные пользователи коллекций должны ссылаться на классы в пакете `generic` только в крайних случаях.

24.2. Согласованность коллекций

Наиболее важные классы коллекций показаны на рис. 24.1.

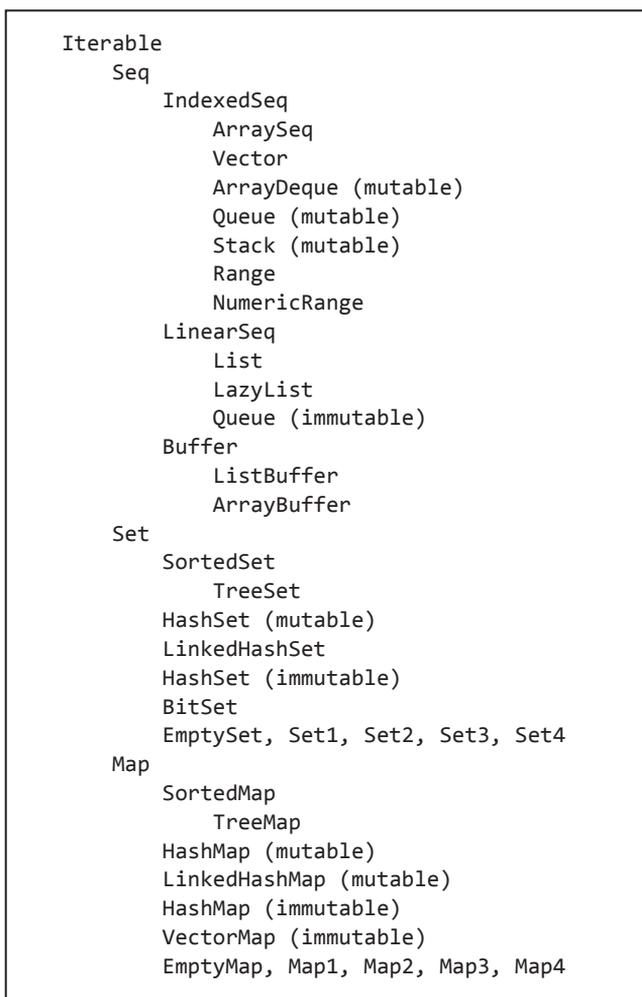
Эти классы имеют много общего. Например, каждая разновидность коллекции может быть создана с использованием единообразного синтаксиса, заключающегося в записи названия класса коллекции, за которым стоят элементы данной коллекции:

```
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25, "z" -> 26)
Set(Color.Red, Color.Green, Color.Blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
```

Тот же принцип применяется и к конкретным реализациям коллекций:

```
List(1, 2, 3)
HashMap("x" -> 24, "y" -> 25, "z" -> 26)
```

Методы `toString` для всех коллекций выводят информацию, аналогичную показанной ранее, с именем типа, за которым следуют значения элементов

**Рис. 24.1.** Иерархия коллекций

коллекции, заключенные в круглые скобки. Все коллекции поддерживают API, предоставляемый `Iterable`, но все их методы возвращают собственный класс, а не корневой класс `Iterable`. Например, у метода `map` класса `List` тип возвращаемого значения `List`, у метода `map` класса `Set` тип возвращаемого значения `Set`. То есть статический возвращаемый тип этих методов абсолютно точен:

```

List(1, 2, 3).map(_ + 1) // List(2, 3, 4): List[Int]
Set(1, 2, 3).map(_ * 2) // Set(2, 4, 6): Set[Int]

```

Равенство для всех классов коллекций также организовано единообразно: более подробно этот вопрос рассматривается в разделе 24.12.

Большинство классов, показанных на рис. 24.1, существуют в трех вариантах: корневом, изменяемом и неизменяемом (`root`, `mutable` и `immutable`). Единственное исключение — трейт `Buffer`, существующий только в виде изменяемой коллекции.

Далее в главе все эти классы мы рассмотрим поочередно.

24.3. Трейт `Iterable`

На вершине иерархии коллекций находится трейт `Iterable[A]`, где `A` — тип элементов коллекции. Все методы в этом трейте определены в терминах абстрактного метода `iterator`, который возвращает элементы коллекции один за другим.

```
def iterator: Iterator[A]
```

Классам коллекций, реализующим `Iterable`, нужно определить только этот метод, а все остальные методы могут быть унаследованы из `Iterable`.

В `Iterable` также определяется много конкретных методов, перечисленных в табл. 24.1. Их можно разбить на следующие категории.

- **Операции итерирования `foreach`, `grouped`, `sliding`** перебирают все элементы коллекции в порядке, который определяет ее итератор. Методы `grouped` и `sliding` возвращают итераторы, которые, в свою очередь, возвращают не отдельные элементы, а, скорее, подпоследовательности элементов исходной коллекции. Максимальный размер этих подпоследовательностей указывается в качестве аргумента для этих методов. Метод `grouped` делит полученные элементы на инкременты, а `sliding` формирует по ним скользящее окно. Разницу между ними должен наглядно продемонстрировать следующий код:

```
val xs = List(1, 2, 3, 4, 5)

val git = xs.grouped(3) // an Iterator[List[Int]]
git.next()             // List(1, 2, 3)
git.next()             // List(4, 5)
val sit = xs.sliding(3) // an Iterator[List[Int]]
sit.next()             // List(1, 2, 3)
sit.next()             // List(2, 3, 4)
sit.next()             // List(3, 4, 5)
```

- **Сложение ++** (псевдоним: `concat`) складывает две коллекции вместе или добавляет все элементы итератора в коллекцию.
- **Операции отображения `map`, `flatMap` и `collect`** создают новую коллекцию путем применения некой функции к элементам коллекции.
- **Преобразования `toIndexedSeq`, `toIterable`, `toList`, `toMap`, `toSeq`, `toSet` и `toVector`** превращают коллекцию `Iterable` в неизменяемую. Все эти преобразования возвращают объект-получатель, если он уже соответствует требуемому типу коллекции. Например, применение `toList` к списку выдаст сам список. Методы `toArray` и `toBuffer` возвращают новую изменяемую коллекцию, даже если объект-получатель соответствует. Метод `to` можно использовать для преобразования в любую другую коллекцию.
- **Операция копирования `copyToArray`**. Как следует из названия, копирует элементы коллекции в массив.
- **Операции с размером `isEmpty`, `nonEmpty`, `size`, `knownSize`, `sizeCompare` и `sizeIs`** имеют отношение к размеру коллекции. Чтобы вычислить количество элементов коллекции, в некоторых случаях может потребоваться совершить обход, например в случае `List`. В других случаях коллекция может содержать бесконечное количество элементов, например `LazyList.from(0)`. Методы `knownSize`, `sizeCompare` и `sizeIs` предоставляют информацию о количестве элементов, обходя как можно меньшее количество элементов.
- **Операции извлечения элементов `head`, `last`, `headOption`, `lastOption` и `find`** выбирают первый или последний элемент коллекции или же первый элемент, соответствующий условию. Однако следует заметить, что не все коллекции имеют четко определенное значение первого и последнего. Например, `HashSet` может хранить элементы в соответствии с их хеш-ключами, и порядок их следования от запуска к запуску может изменяться. В таком случае для разных запусков программы первый элемент `HashSet` также может быть разным. Коллекция считается *упорядоченной*, если всегда выдает свои элементы в одном и том же порядке. Большинство коллекций упорядочены, однако некоторые, такие как `HashSet`, таковыми не являются, отказ от упорядочения позволяет им быть более эффективными. Упорядочение зачастую необходимо, чтобы получать воспроизводимые тесты и способствовать отладке. Поэтому коллекции `Scala` предоставляют упорядоченные альтернативы для всех типов коллекций. Например, упорядоченная альтернатива для `HashSet` — `LinkedHashSet`.
- **Операции извлечения подколлекций `takeWhile`, `tail`, `init`, `slice`, `take`, `drop`, `filter`, `dropWhile`, `filterNot` и `withFilter`** возвращают подколлекцию, определяемую диапазоном индексов или предикатом.

- **Операции подразделения** `groupBy`, `groupMap`, `groupMapReduce`, `splitAt`, `span`, `partition` и `partitionMap` разбивают элементы переданной коллекции на несколько подколлекций.
- **Операции тестирования элементов** `exists`, `forall` и `count` тестируют элементы коллекции на соответствие заданному предикату.
- **Операции свертки** `foldLeft`, `foldRight`, `reduceLeft`, `reduceRight` применяют бинарные операции к следующим друг за другом элементам.
- **Операции специализированных свертков** `sum`, `product`, `min` и `max` работают с коллекциями конкретных типов (`numeric` или `comparable`).
- **Операции со строками** `mkString` и `addString` предоставляют альтернативные способы преобразования коллекции в строку.
- **Операция представления** `view` — это коллекция, которая вычисляется лениво. Больше о представлениях вы узнаете из раздела 24.13.

Таблица 24.1. Операции в трейте `Iterable`

Что	Что делает
Абстрактный метод	
<code>xs.iterator</code>	Итератор, который возвращает каждый элемент в <code>xs</code>
Итерирование:	
<code>xs.foreach(f)</code>	Применяет функцию <code>f</code> к каждому элементу <code>xs</code> . Вызов <code>f</code> нужен только для получения побочных эффектов; по сути, <code>foreach</code> отбрасывает любой результат функции <code>f</code>
<code>xs.grouped(size)</code>	Итератор, который возвращает «блоки» коллекции фиксированного размера
<code>xs.sliding(size)</code>	Итератор, который возвращает скользящее окно по элементам этой коллекции
Сложение	
<code>xs ++ ys</code> (или <code>xs.concat(ys)</code>)	Коллекция, состоящая из элементов <code>xs</code> и <code>ys</code> . Элемент <code>ys</code> — это коллекция <code>IterableOnce</code> , то есть <code>Iterable</code> или <code>Iterator</code>
Отображения	
<code>xs.map(f)</code>	Коллекция, получающаяся в результате применения функции <code>f</code> к каждому элементу в <code>xs</code>

Что	Что делает
<code>xs.flatMap(f)</code>	Коллекция, получающаяся в результате применения функции <code>f</code> , результатом которой является коллекция, к каждому элементу в <code>xs</code> и объединяющая результаты
<code>xs.collect(f)</code>	Коллекция, получающаяся в результате применения частично примененной функции <code>f</code> к каждому элементу в <code>xs</code> , для которого она определена, и объединяющая результаты
Преобразования	
<code>xs.toArray</code>	Превращает коллекцию в массив
<code>xs.toList</code>	Превращает коллекцию в список
<code>xs.toIterable</code>	Превращает коллекцию в итерируемую коллекцию
<code>xs.toSeq</code>	Превращает коллекцию в последовательность
<code>xs.toIndexedSeq</code>	Превращает коллекцию в проиндексированную последовательность
<code>xs.toSet</code>	Превращает коллекцию в множество
<code>xs.toMap</code>	Превращает коллекцию пар «ключ — значение» в отображение
<code>xs.to(SortedSet)</code>	Обобщенная операция преобразования, которая принимает в качестве параметра фабрику коллекций
Копирование	
<code>xs.copyToArray(arr, s, len)</code>	Копирует максимум <code>len</code> элементов в массиве <code>arr</code> , начиная с индекса <code>s</code> . Последние два аргумента необязательны
Получение информации о размере	
<code>xs.isEmpty</code>	Проверяет, является ли коллекция пустой
<code>xs.nonEmpty</code>	Проверяет, содержит ли коллекция элементы
<code>xs.size</code>	Количество элементов в коллекции
<code>xs.knownSize</code>	Количество элементов, если его можно вычислить за постоянное время. В противном случае <code>-1</code>
<code>xs.sizeCompare(ys)</code>	Возвращает отрицательное значение, если коллекция <code>xs</code> короче <code>ys</code> , положительное, если длиннее, и <code>0</code> , если обе коллекции имеют одинаковый размер. Работает даже для бесконечных коллекций

Таблица 24.1 (продолжение)

Что	Что делает
<code>xs.sizeIs < 42</code> , <code>xs.sizeIs != 42</code> , и т. д.	Сравнивает размер коллекции с заданным значением, перебирая как можно меньше элементов
Извлечение элементов	
<code>xs.head</code>	Первый элемент коллекции (или какой-нибудь элемент, если порядок следования элементов не определен)
<code>xs.headOption</code>	Первый элемент <code>xs</code> в <code>Option</code> -значении или <code>None</code> , если <code>xs</code> пуст
<code>xs.last</code>	Последний элемент коллекции (или какой-нибудь элемент, если порядок следования элементов не определен)
<code>xs.lastOption</code>	Последний элемент <code>xs</code> в <code>Option</code> -значении параметра или <code>None</code> , если <code>xs</code> пуст
<code>xs.find(p)</code>	<code>Option</code> -значение, содержащее первый элемент в <code>xs</code> , удовлетворяющий условию <code>p</code> , или <code>None</code> , если такого элемента нет
Создание подколлекций	
<code>xs.tail</code>	Вся коллекция, за исключением <code>xs.head</code>
<code>xs.init</code>	Вся коллекция, за исключением <code>xs.last</code>
<code>xs.slice(from, to)</code>	Коллекция, состоящая из элементов в некотором диапазоне индексов <code>xs</code> (от <code>from</code> включительно до <code>to</code> не включительно)
<code>xs.take(n)</code>	Коллекция, состоящая из первых <code>n</code> элементов <code>xs</code> (или <code>n</code> каких-либо произвольных элементов, если порядок следования элементов не определен)
<code>xs.drop(n)</code>	Вся коллекция <code>xs</code> , за вычетом тех элементов, которые получаются при использовании выражения <code>xs take n</code>
<code>xs.takeWhile(p)</code>	Самый длинный префикс элементов в коллекции, каждый из которых соответствует условию <code>p</code>
<code>xs.dropWhile(p)</code>	Коллекция без самого длинного префикса элементов, каждый из которых соответствует условию <code>p</code>
<code>xs.takeRight(n)</code>	Коллекция, состоящая из заключительных <code>n</code> элементов <code>xs</code> (или произвольных <code>n</code> элементов, если порядок не определен)

Что	Что делает
<code>xs.dropRight(n)</code>	Вся коллекция, кроме <code>xs.takeRight n</code>
<code>xs.filter(p)</code>	Коллекция, состоящая только из тех элементов <code>xs</code> , которые соответствуют условию <code>p</code>
<code>xs.withFilter(p)</code>	Нестрогий фильтр коллекции. Все операции в отношении фильтруемых элементов будут применяться только к тем элементам <code>xs</code> , для которых условие <code>p</code> вычисляется в <code>true</code>
<code>xs.filterNot(p)</code>	Коллекция, состоящая из тех элементов <code>xs</code> , которые не соответствуют условию <code>p</code>
Слияния	
<code>xs.zip(ys)</code>	Итерируемые пары соответствующих элементов из <code>xs</code> и <code>ys</code>
<code>xs.lazyZip(ys)</code>	Значение, предоставляющее методы для поэлементной работы с коллекциями <code>xs</code> и <code>ys</code> . См. раздел 14.9
<code>xs.zipAll(ys, x, y)</code>	Итерируемые пары соответствующих элементов из <code>xs</code> и <code>ys</code> ; если одна из последовательностей короче, то расширяется за счет добавления элементов <code>x</code> или <code>y</code>
<code>xs.zipWithIndex</code>	Итерируемые пары элементов из <code>xs</code> с их индексами
Разбиения	
<code>xs.splitAt(n)</code>	Разбивает <code>xs</code> по позиции на пару коллекций (<code>xs.take(n)</code> , <code>xs.drop(n)</code>)
<code>xs.span(p)</code>	Разбивает <code>xs</code> в соответствии с предикатом на пару коллекций (<code>xs.takeWhile(p)</code> , <code>xs.dropWhile(p)</code>)
<code>xs.partition(p)</code>	Разбивает <code>xs</code> на пару коллекций, в одной из которых находятся все элементы, удовлетворяющие условию <code>p</code> , а в другой — все элементы, не удовлетворяющие этому условию (<code>xs.filter(p)</code> , <code>xs.filterNot(p)</code>)
<code>xs.partitionMap(f)</code>	Преобразует каждый элемент <code>xs</code> в значение <code>Either[X, Y]</code> и разбивает результат на две коллекции: одна с элементами из <code>Left</code> , а другая с элементами из <code>Right</code>
<code>xs.groupBy(f)</code>	Разбивает <code>xs</code> на отображение коллекций в соответствии с функцией дискриминатором <code>f</code>

Таблица 24.1 (продолжение)

Что	Что делает
<code>xs.groupMap(f)(g)</code>	Превращает <code>xs</code> в отображение коллекций в соответствии с функцией-дискриминатором <code>f</code> и применяет функцию преобразования <code>g</code> к каждому элементу каждой коллекции
<code>xs.groupMapReduce(f)(g)(h)</code>	Превращает <code>xs</code> в отображение коллекций в соответствии с функцией-дискриминатором <code>f</code> , применяет функцию преобразования <code>g</code> к каждому элементу каждой коллекции и сводит каждую коллекцию к единственному значению, объединяя ее элементы с помощью функции <code>h</code>
Состояния элементов	
<code>xs.forall(p)</code>	Булево значение, показывающее, соблюдается ли условие <code>p</code> для всех элементов <code>xs</code>
<code>xs.exists(p)</code>	Булево значение, показывающее, соблюдается ли условие <code>p</code> для каких-либо элементов <code>xs</code>
<code>xs.count(p)</code>	Количество элементов в <code>xs</code> , удовлетворяющих условию <code>p</code>
Свертки	
<code>xs.foldLeft(z)(op)</code>	Применяет бинарную операцию <code>op</code> к последовательным элементам <code>xs</code> , продвигаясь слева направо, начиная с <code>z</code>
<code>xs.foldRight(z)(op)</code>	Применяет бинарную операцию <code>op</code> к последовательным элементам <code>xs</code> , продвигаясь справа налево, начиная с <code>z</code>
<code>xs.reduceLeft(op)</code>	Применяет бинарную операцию <code>op</code> к последовательным элементам <code>xs</code> непустой коллекции <code>xs</code> , продвигаясь слева направо
<code>xs.reduceRight(op)</code>	Применяет бинарную операцию <code>op</code> к последовательным элементам <code>xs</code> непустой коллекции <code>xs</code> , продвигаясь справа налево
Специализированные свертки	
<code>xs.sum</code>	Сумма числовых элементов коллекции <code>xs</code>
<code>xs.product</code>	Произведение числовых элементов коллекции <code>xs</code>
<code>xs.min</code>	Минимальное значение упорядочиваемых элементов коллекции <code>xs</code>
<code>xs.max</code>	Максимальное значение упорядочиваемых элементов коллекции <code>xs</code>

Что	Что делает
Строки	
<code>xs addString (b, start, sep, end)</code>	Добавляет строку к строковому буферу <code>b</code> типа <code>StringBuilder</code> со всеми элементами <code>xs</code> с разделителями <code>sep</code> , заключенными между строками <code>start</code> и <code>end</code> . Аргументы <code>start</code> , <code>sep</code> и <code>end</code> являются необязательными
<code>xs mkString (start, sep, end)</code>	Превращает коллекцию в строку со всеми элементами <code>xs</code> с разделителями <code>sep</code> , заключенными между строками <code>start</code> и <code>end</code> . Аргументы <code>start</code> , <code>sep</code> и <code>end</code> являются необязательными
Представления	
<code>xs.view</code>	Создает представление <code>xs</code>

Подкатегории Iterable

В иерархии наследования ниже `Iterable` находятся три трейта: `Seq`, `Set` и `Map`. Трейты `Seq` и `Map` объединяет то, что они реализуют трейт `PartialFunction`¹ с методами `apply` и `isDefinedAt`. Но способы реализации `PartialFunction` у каждого трейта свои.

Для последовательностей `apply` — позиционное индексирование, в котором элементы всегда нумеруются с нуля. То есть `Seq(1, 2, 3)(1) == 2`. Для множеств `apply` — проверка на принадлежность. Например, `Set('a', 'b', 'c')('b') == true`, а `Set()('a') == false`. И наконец, для отображений `apply` — средство выбора. Например, `Map('a' -> 1, 'b' -> 10, 'c' -> 100)('b') == 10`.

Более подробно каждый из этих видов коллекций мы рассмотрим в следующих трех разделах.

24.4. Трейты последовательностей Seq, IndexedSeq и LinearSeq

Трейт `Seq` представляет последовательности. Последовательностью называется разновидность итерируемой коллекции, у которой есть длина и все элементы которой имеют фиксированные индексированные позиции, отсчет

¹ Частично определенные функции описаны в разделе 13.7.

которых начинается с нуля. Операции с последовательностями (сведены в табл. 24.2) разбиваются на следующие категории.

- **Операции индексирования и длины** `apply`, `isDefinedAt`, `length`, `indices`, `lengthCompare` и `lengthIs`. Для `Seq` операция `apply` означает индексирование, поэтому последовательность типа `Seq[T]` является частично примененной функцией, которая получает аргумент типа `Int` (индекс) и выдает элемент последовательности типа `T`. Иными словами, `Seq[T]` расширяет `PartialFunction[Int, T]`. Элементы последовательности индексируются от нуля до длины последовательности `length` за вычетом единицы. Метод `length`, применяемый в отношении последовательностей, — псевдоним общего для коллекций метода `size`. Метод `lengthCompare` позволяет сравнивать длины двух последовательностей, даже если одна из них имеет бесконечную длину. Метод `lengthIs` — псевдоним метода `sizeIs`.
- **Операции поиска индекса** `indexOf`, `lastIndexOf`, `indexOfSlice`, `lastIndexOfSlice`, `indexWhere`, `lastIndexWhere`, `segmentLength` и `prefixLength` возвращают индекс элемента, равного заданному значению или соответствующего некоему условию.
- **Операции добавления** `+`: (псевдоним: `prepend`), `++`: (псевдоним: `prependAll`), `:+` (псевдоним: `append`), `:+:` (псевдоним: `appendAll`) и `padTo` возвращают новые последовательности, получаемые путем добавления к началу или концу последовательности.
- **Операции обновления** `updated` и `patch` возвращают новую последовательность, получаемую путем замены некоторых элементов исходной последовательности.
- **Операции сортировки** `sorted`, `sortWith` и `sortBy` сортируют элементы последовательности в соответствии с различными критериями.
- **Операции реверсирования** `reverse` и `reverseIterator` обрабатывают или возвращают элементы последовательности в обратном порядке, с последнего до первого.
- **Операции сравнения** `startsWith`, `endsWith`, `contains`, `corresponds`, `containsSlice` и `search` определяют соотношение двух последовательностей или ищут элемент в последовательности.
- **Операции над множествами** `intersect`, `diff`, `distinct` и `distinctBy` выполняют над элементами двух последовательностей операции, подобные операциям с множествами, или удаляют дубликаты.

Для изменяемой последовательности предлагается дополнительный метод добавления `update` с побочным эффектом, который позволяет элементам

последовательности обновляться. Вспомним: в главе 3 мы уже говорили, что синтаксис вида `seq(idx) = elem` — не более чем сокращение для выражения `seq.update(idx, elem)`. Обратите внимание на разницу между методами `update` и `updated`. Первый изменяет элемент последовательности на месте и доступен только для изменяемых последовательностей. Второй доступен для всех последовательностей и всегда вместо изменения исходной возвращает новую последовательность.

Таблица 24.2. Операции в трейте Seq

Что	Что делает
Индексирование и длина	
<code>xs(i)</code> (или после раскрытия <code>xs.apply(i)</code>)	Элемент <code>xs</code> с индексом <code>i</code>
<code>xs.isDefinedAt(i)</code>	Проверяет, содержится ли <code>i</code> в <code>xs.indices</code>
<code>xs.length</code>	Длина последовательности (то же самое, что и <code>size</code>)
<code>xs.lengthCompare(len)</code>	Возвращает отрицательное значение <code>Int</code> , если длина <code>xs</code> меньше <code>len</code> , положительное, если больше, и <code>0</code> , если они равны. Работает даже для бесконечных коллекций <code>xs</code>
<code>xs.indices</code>	Диапазон индексов <code>xs</code> от <code>0</code> до <code>xs.length - 1</code>
Поиск индекса	
<code>xs.indexOf(x)</code>	Индекс первого элемента в <code>xs</code> , равного значению <code>x</code> (существует несколько вариантов)
<code>xs.lastIndexOf(x)</code>	Индекс последнего элемента в <code>xs</code> , равного значению <code>x</code> (существует несколько вариантов)
<code>xs.indexOfSlice(ys)</code>	Первый индекс <code>xs</code> при условии, что следующие друг за другом элементы, начинающиеся с элемента с этим индексом, составляют последовательность <code>ys</code>
<code>xs.lastIndexOfSlice(ys)</code>	Последний индекс <code>xs</code> при условии, что следующие друг за другом элементы, начинающиеся с элемента с этим индексом, составляют последовательность <code>ys</code>
<code>xs.indexWhere(p)</code>	Индекс первого элемента в <code>xs</code> , удовлетворяющего условию <code>p</code> (существует несколько вариантов)
<code>xs.segmentLength(p, i)</code>	Длина самого длинного непрерывного сегмента элементов в <code>xs</code> , начинающегося с <code>xs(i)</code> , который удовлетворяет условию <code>p</code>
Добавления	
<code>x +: xs</code> (или <code>xs.prepended(x)</code>)	Новая последовательность со значением <code>x</code> , добавленным в начало <code>xs</code>

Таблица 24.2 (продолжение)

Что	Что делает
<code>ys ++: xs</code> (или <code>xs.prependAll(ys)</code>)	Новая последовательность, состоящая из всех элементов <code>ys</code> , добавленных в начало <code>xs</code>
<code>xs :+ x</code> (или <code>xs.appended(x)</code>)	Новая последовательность со значением <code>x</code> , добавленным в конец <code>xs</code>
<code>xs :++ ys</code> (или <code>xs.appendedAll(ys)</code>)	Новая последовательность, состоящая из всех элементов <code>ys</code> , добавленных в конец <code>xs</code> . То же самое, что <code>xs ++ ys</code>
<code>xs.padTo(len, x)</code>	Последовательность, получающаяся при добавлении значения <code>x</code> к <code>xs</code> , до тех пор, пока длина не достигнет значения <code>len</code>
Обновления	
<code>xs.patch(i, ys, r)</code>	Последовательность, получающаяся путем замены <code>r</code> элементов последовательности <code>xs</code> , начиная с <code>i</code> , элементами последовательности <code>ys</code>
<code>xs.updated(i, x)</code>	Копия <code>xs</code> , в которой элемент с индексом <code>i</code> заменяется значением <code>x</code>
<code>xs(i) = x</code> (или после раскрытия <code>xs.update(i, x)</code> , которая доступна только для изменяемых последовательностей <code>mutable.Seq</code>)	Изменяет значение элемента <code>xs</code> с индексом <code>i</code> на значение <code>x</code>
Сортировки	
<code>xs.sorted</code>	Новая последовательность, полученная путем сортировки элементов <code>xs</code> в порядке следования элементов типа <code>xs</code>
<code>xs.sortWith(lessThan)</code>	Новая последовательность, полученная путем сортировки элементов <code>xs</code> с использованием в качестве операции сравнения <code>lessThan</code> (Меньше чем)
<code>xs.sortBy(f)</code>	Новая последовательность, полученная путем сортировки элементов <code>xs</code> . Сравнение двух элементов выполняется путем применения к ним функции <code>f</code> и сравнения результатов
Реверсирования	
<code>xs.reverse</code>	Последовательность из элементов <code>xs</code> , следующих в обратном порядке
<code>xs.reverseIterator</code>	Итератор, выдающий все элементы <code>xs</code> в обратном порядке

Что	Что делает
Сравнения	
<code>xs.sameElements(ys)</code>	Проверяет, содержат ли <code>xs</code> и <code>ys</code> одинаковые элементы в одном и том же порядке
<code>xs.startsWith(ys)</code>	Проверяет, не начинается ли <code>xs</code> с последовательности <code>ys</code> (имеется несколько вариантов)
<code>xs.endsWith(ys)</code>	Проверяет, не заканчивается ли <code>xs</code> последовательностью <code>ys</code> (имеется несколько вариантов)
<code>xs.contains(x)</code>	Проверяет, имеется ли в <code>xs</code> элемент, равный <code>x</code>
<code>xs.search(x)</code>	Проверяет, содержит ли отсортированная коллекция <code>xs</code> элемент, равный <code>x</code> , и делает это потенциально эффективнее, чем <code>xs.contains x</code>
<code>xs.containsSlice(ys)</code>	Проверяет, имеется ли в <code>xs</code> непрерывная последовательность, равная <code>ys</code>
<code>xs.corresponds(ys)(p)</code>	Проверяет, удовлетворяют ли соответствующие элементы <code>xs</code> и <code>ys</code> бинарному предикату <code>p</code>
Операции над множествами	
<code>xs.intersect(ys)</code>	Пересечение множеств, состоящих из элементов последовательностей <code>xs</code> и <code>ys</code> , которое сохраняет порядок следования элементов в <code>xs</code>
<code>xs.diff(ys)</code>	Разность множеств, состоящих из элементов последовательностей <code>xs</code> и <code>ys</code> , которое сохраняет порядок следования элементов в <code>xs</code>
<code>xs.distinct</code>	Часть последовательности <code>xs</code> , не содержащая дубликатов
<code>xs.distinctBy(f)</code>	Подпоследовательность <code>xs</code> , в которой после применения преобразующей функции <code>f</code> нет повторяющихся элементов

У каждого трейта `Seq` есть два подтрейта: `LinearSeq` и `IndexedSeq`. Они не добавляют никаких новых операций, но каждый из них предлагает разные характеристики производительности. У линейной последовательности (`linear sequence`) есть эффективные операции `head` и `tail`, а у индексированной — эффективные операции `apply`, `length` и (если последовательность изменяемая) `update`. В `List` зачастую применяется линейная последовательность, как и в `LazyList`. Представители двух часто используемых индексированных последовательностей — `Array` и `ArrayBuffer`. Класс `Vector` обеспечивает весьма интересный компромисс между индексированным и последовательным доступом. У него практически постоянные линейные издержки как на

индексированный, так и на последовательный доступ. Поэтому векторы служат хорошей основой для смешанных схем доступа, в которых используется как индексированный, так и последовательный доступ. Более подробно мы рассмотрим векторы в разделе 24.7.

Изменяемый подтрейт `IndexedSeq` добавляет операции для преобразования имеющихся элементов. В отличие от `map` и `sort`, которые доступны в `Seq`, эти операции, представленные в табл. 24.3, не возвращают новый экземпляр коллекции.

Таблица 24.3. Операции в трейте `mutable.IndexedSeq`

Что	Что делает
Добавления	
<code>xs.mapInPlace(f)</code>	Преобразует все элементы <code>xs</code> , применяя к каждому из них функцию <code>f</code>
<code>xs.sortInPlace()</code>	Сортирует элементы <code>xs</code> на месте
<code>xs.sortInPlaceBy(f)</code>	Сортирует элементы <code>xs</code> на месте и в соответствии с порядком, который определяется путем применения функции <code>f</code> к каждому элементу
<code>xs.sortInPlaceWith(c)</code>	Сортирует элементы <code>xs</code> на месте в соответствии с функцией сравнения <code>c</code>

Буферы

Важная подкатегория изменяемых последовательностей — буферы. Они позволяют не только обновлять существующие элементы, но и вставлять и удалять элементы, а также с высокой эффективностью добавлять новые элементы в конец буфера. Принципиально новые методы, поддерживаемые буфером, — `+=` (псевдоним: `append`) и `++=` (псевдоним: `appendAll`) для добавления элементов в конец буфера, `+=` (псевдоним: `prepend`) и `++=` (псевдоним: `prependAll`) для добавления элементов в начало буфера, `insert` и `insertAll` для вставки элементов, а также `remove`, `-=` (псевдоним: `subtractOne`) и `---=` (псевдоним: `subtractAll`) для удаления элементов. Все эти операции сведены в табл. 24.4.

Наиболее часто используются две реализации буферов: `ListBuffer` и `ArrayBuffer`. Исходя из названий, `ListBuffer` базируется на классе `List` и поддерживает высокоэффективное преобразование своих элементов в список, а `ArrayBuffer` базируется на массиве и может быть быстро превращен в массив. Наметки реализации `ListBuffer` мы показали в разделе 1.2.

Таблица 24.4. Операции в трейте Buffer

Что	Что делает
Добавления	
<code>buf += x</code> (или <code>buf.append(x)</code>)	Добавляет элемент <code>x</code> в конец <code>buf</code> и возвращает в качестве результата сам <code>buf</code>
<code>buf ++= xs</code> (или <code>buf.appendAll(xs)</code>)	Добавляет в конец буфера все элементы <code>xs</code>
<code>x +=: buf</code> (или <code>buf.prepend(x)</code>)	Добавляет элемент <code>x</code> в начало буфера
<code>xs ++=: buf</code> (или <code>buf.prependAll(xs)</code>)	Добавляет в начало буфера все элементы <code>xs</code>
<code>buf.insert(i, x)</code>	Вставляет элемент <code>x</code> в то место в буфере, на которое указывает индекс <code>i</code>
<code>buf.insertAll(i, xs)</code>	Вставляет все элементы <code>xs</code> в то место в буфере, на которое указывает индекс <code>i</code>
<code>buf.padToInPlace(n, x)</code>	Добавляет в буфер элементы <code>x</code> , пока общее количество его элементов не достигнет <code>n</code>
Удаления	
<code>buf -= x</code> (или <code>buf.subtractOne(x)</code>)	Удаляет из буфера элемент <code>x</code>
<code>buf --= x</code> (или <code>buf.subtractAll(xs)</code>)	Удаляет из буфера все элементы <code>xs</code>
<code>buf.remove(i)</code>	Удаляет из буфера элемент с индексом <code>i</code>
<code>buf.remove(i, n)</code>	Удаляет из буфера <code>n</code> элементов, начиная с элемента с индексом <code>i</code>
<code>buf.trimStart(n)</code>	Удаляет из буфера первые <code>n</code> элементов
<code>buf.trimEnd(n)</code>	Удаляет из буфера последние <code>n</code> элементов
<code>buf.clear()</code>	Удаляет из буфера все элементы
Замена:	
<code>buf.patchInPlace(i, xs, n)</code>	Заменяет (максимум) <code>n</code> элементов буфера элементами из <code>xs</code> , начиная с индекса <code>i</code>
Копирование	
<code>buf.clone()</code>	Новый буфер с теми же элементами, что и в <code>buf</code>

24.5. Множества

Коллекции `Set` — это итерируемые `Iterable`-коллекции, которые не содержат повторяющихся элементов. Общие операции над множествами сведены в табл. 24.5, в табл. 24.6 показаны операции для неизменяемых множеств, а в табл. 24.7 — операции для изменяемых множеств. Операции разбиты на следующие категории.

- **Проверки `contains`, `apply` и `subsetOf`.** Метод `contains` показывает, содержит ли множество заданный элемент. Метод `apply` для множества является аналогом `contains`, поэтому `set(elem)` — то же самое, что и `set contains elem`. Следовательно, множества могут также использоваться в качестве тестовых функций, возвращающих `true` для содержащихся в них элементов, например:

```
val fruit = Set("apple", "orange", "peach", "banana")
fruit("peach") // true
fruit("potato") // false
```

- **Добавления + (псевдоним: `incl`) и ++ (псевдоним: `concat`)** добавляют в множество один и более элементов, возвращая в качестве результата новое множество.
- **Удаления - (псевдоним: `excl`) и -- (псевдоним: `removedAll`)** удаляют из множества один и более элементов, возвращая новое множество.
- **Операции над множествами** для объединения, пересечения и разности множеств. Существуют в двух формах: текстовом и символьном. К текстовым относятся версии `intersect`, `union` и `diff`, а к символьным — `&`, `|` и `&~`. Оператор `++`, наследуемый `Set` из `Iterable`, может рассматриваться в качестве еще одного псевдонима `union` или `|`, за исключением того, что `++` получает `IterableOnce`-аргумент, а `union` и `|` получают множества.

Таблица 24.5. Операции в трейте `Set`

Что	Что делает
Проверки	
<code>xs.contains(x)</code>	Проверяет, является ли <code>x</code> элементом <code>xs</code>
<code>xs(x)</code>	Делает то же самое, что и <code>xs.contains(x)</code>
<code>xs.subsetOf(ys)</code>	Проверяет, является ли <code>xs</code> подмножеством <code>ys</code>
Удаления	
<code>xs.empty</code>	Пустое множество того же класса, что и <code>xs</code>

Что	Что делает
Бинарные операции	
<code>xs & ys</code> (или <code>xs.intersect(ys)</code>)	Пересечение множеств <code>xs</code> и <code>ys</code>
<code>xs ys</code> (или <code>xs.union(ys)</code>)	Объединение множеств <code>xs</code> и <code>ys</code>
<code>xs & ~ ys</code> (или <code>xs.diff(ys)</code>)	Разность множеств <code>xs</code> и <code>ys</code>

Неизменяемые множества предлагают методы добавления и удаления элементов путем возвращения новых множеств, которые сведены в табл. 24.6.

Таблица 24.6. Операции в трейте `immutable.Set`

Что	Что делает
Добавления	
<code>xs + x</code> (или <code>xs.incl(x)</code>)	Множество, содержащее все элементы <code>xs</code> и элемент <code>x</code>
<code>xs ++ ys</code> (или <code>xs.concat(ys)</code>)	Множество, содержащее все элементы <code>xs</code> и все элементы <code>ys</code>
Удаления	
<code>xs - x</code> (или <code>xs.excl(x)</code>)	Множество, содержащее все элементы <code>xs</code> , кроме <code>x</code>
<code>xs -- ys</code> (или <code>xs.removedAll(ys)</code>)	Множество, содержащее все элементы <code>xs</code> , кроме элементов множества <code>ys</code>

У изменяемых множеств есть методы, которые добавляют, удаляют и обновляют элементы, которые сведены в табл. 24.7.

Таблица 24.7. Операции в трейте `mutable.Set`

Что	Что делает
Добавления	
<code>xs += x</code> (или <code>xs.addOne(x)</code>)	Добавляет элемент <code>x</code> в множество <code>xs</code> как побочный эффект и возвращает само множество <code>xs</code>
<code>xs ++= ys</code> (или <code>xs.addAll(ys)</code>)	Добавляет все элементы <code>ys</code> в множество <code>xs</code> как побочный эффект и возвращает само множество <code>xs</code>

Таблица 24.7 (окончание)

Что	Что делает
<code>xs.add(x)</code>	Добавляет элемент <code>x</code> в <code>xs</code> и возвращает <code>true</code> , если <code>x</code> прежде не был в множестве, или <code>false</code> , если уже был
Удаления	
<code>xs -= x</code> (или <code>xs.subtractOne(x)</code>)	Удаляет элемент <code>x</code> из множества <code>xs</code> как побочный эффект и возвращает само множество <code>xs</code>
<code>xs -= ys</code> (или <code>xs.subtractAll(ys)</code>)	Удаляет все элементы <code>ys</code> из множества <code>xs</code> как побочный эффект и возвращает само множество <code>xs</code>
<code>xs.remove(x)</code>	Удаляет элемент <code>x</code> из <code>xs</code> и возвращает <code>true</code> , если <code>x</code> прежде уже был в множестве, или <code>false</code> , если его прежде там не было
<code>xs.filterInPlace(p)</code>	Сохраняет только те элементы в <code>xs</code> , которые удовлетворяют условию <code>p</code>
<code>xs.clear()</code>	Удаляет из <code>xs</code> все элементы
Обновление	
<code>xs(x) = b</code> (или после раскрытия <code>xs.update(x, b)</code>)	Если аргумент <code>b</code> типа <code>Boolean</code> имеет значение <code>true</code> , то добавляет <code>x</code> в <code>xs</code> , в противном случае удаляет <code>x</code> из <code>xs</code>
Клонирование	
<code>xs.clone()</code>	Возвращает новое изменяемое множество с такими же элементами, как и в <code>xs</code>

Операция `s += elem` в качестве побочного эффекта добавляет `elem` во множество `s` и в качестве результата возвращает измененное множество. По аналогии с этим `s -= elem` удаляет элемент `elem` из множества и возвращает в качестве результата измененное множество. Помимо `+=` и `-=`, есть также операции над несколькими элементами `++=` и `--=`, которые добавляют или удаляют все элементы `Iterable` или итератора.

Выбор в качестве имен методов `+=` и `-=` означает, что очень похожий код может работать как с изменяемыми, так и с неизменяемыми множествами. Рассмотрим сначала следующий интерпретатор, в котором используется неизменяемое множество `s`:

```
var s = Set(1, 2, 3)
s += 4
s = 2
s // Set(1, 3, 4)
```

В этом примере в отношении `var`-переменной типа `immutable.Set` используются методы `+=` и `-=`. Согласно объяснениям, которые были даны в шаге 10 главы 3, инструкции вида `s += 4` — это сокращенная форма записи для `s = s + 4`. Следовательно, в их выполнении участвует еще один метод `+`, применяемый в отношении множества `s`, а затем результат присваивается переменной `s`. А теперь рассмотрим аналогичную работу в интерпретаторе с изменяемым множеством:

```
val s = collection.mutable.Set(1, 2, 3)
s += 4 // Set(1, 2, 3, 4)
s = 2  // Set(1, 3, 4)
s      // Set(1, 3, 4)
```

Конечный эффект очень похож на предыдущий диалог с интерпретатором: начинаем мы с множеством `Set(1, 2, 3)`, а заканчиваем с множеством `Set(1, 3, 4)`. Но даже притом что инструкции выглядят такими же, как и раньше, они выполняют несколько иные действия. Теперь инструкция `s += 4` вызывает метод `+=` в отношении значения `s`, которое представляет собой изменяемое множество, выполняя изменения на месте. Аналогично этому инструкция `s -= 2` теперь вызывает в отношении этого же множества метод `-=`.

Сравнение этих двух диалогов позволяет выявить весьма важный принцип. Зачастую можно заменить изменяемую коллекцию, хранящуюся в `val`-переменной, неизменяемой коллекцией, хранящейся в `var`-переменной, и наоборот. Это работает по крайней мере до тех пор, пока нет псевдонимов ссылок на коллекцию, позволяющих заметить, обновилась она на месте или была создана новая коллекция.

Изменяемые множества также предоставляют в качестве вариантов `+=` и `-=` методы `add` и `remove`. Разница в том, что методы `add` и `remove` возвращают булев результат, показывающий, возымела ли операция эффект над множеством.

В текущей реализации по умолчанию изменяемого множества его элементы хранятся с помощью хеш-таблицы. В реализации по умолчанию неизменяемых множеств используется представление, которое адаптируется к количеству элементов множества. Пустое множество представляется в виде простого объекта-одиночки. Множества размером до четырех элементов представляются в виде одиночного объекта, сохраняющего все элементы как поля. Все неизменяемые множества, имеющие большие размеры, реализуются в виде сжатых хеш-массивов из сопоставленных префиксных деревьев¹.

¹ Префиксные деревья на основе сжатых хеш-массивов описываются в разделе 24.7.

Последствия применения таких вариантов представления заключаются в том, что для множеств небольших размеров с количеством элементов, не превышающим четырех, неизменяемые множества получаются более компактными и более эффективными в работе, чем изменяемые. Поэтому, если предполагается, что множество будет небольшим, попробуйте сделать его неизменяемым.

24.6. Отображения

Коллекции типа `Map` представляют собой `Iterable`-коллекции, состоящие из пар «ключ — значение», которые также называются отображениями или ассоциациями. Объект `Predef` в Scala предлагает неявное преобразование, позволяющее использовать запись вида *ключ* `->` *значение* в качестве альтернативы синтаксиса для пары вида (*ключ*, *значение*). Таким образом, выражение для инициализации `Map("x" -> 24, "y" -> 25, "z" -> 26)` означает абсолютно то же самое, что и выражение `Map(("x", 24), ("y", 25), ("z", 26))`, но читается легче.

Основные операции над отображениями, сведенные в табл. 24.8, похожи на аналогичные операции над множествами. Неизменяемые отображения поддерживают дополнительные операции добавления и удаления, которые возвращают новые отображения, как показано в табл. 24.9. Изменяемые отображения дополнительно поддерживают операции, перечисленные в табл. 24.10. Операции над отображениями разбиваются на следующие категории.

- **Операции поиска `apply`, `get`, `getOrElse`, `contains` и `isDefinedAt`** превращают отображения в частично примененные функции от ключей к значениям. Основной метод поиска для отображений выглядит так:

```
def get(key): Option[Value]
```

Операция `m.get(key)` проверяет, содержит ли отображение ассоциацию для заданного ключа. Будучи в наличии, такая ассоциация возвращает значение ассоциации в виде объекта типа `Some`. Если такой ключ в отображении не определен, то `get` возвращает `None`. В отображениях также определяется метод `apply`, возвращающий значение, непосредственно ассоциированное с заданным ключом, без его инкапсуляции в `Option`. Если ключ в отображении не определен, то выдается исключение.

- **Добавления и обновления + (псевдоним: `updated`), ++ (псевдоним: `concat`) `updateWith` и `updatedWith`** позволяют добавлять к отображению новые привязки или изменять уже существующие.

- **Удаления** - (псевдоним: `removed`) и `--` (псевдоним: `removedAll`) позволяют удалять привязки из отображения.
- **Операции создания подколлекций** `keys`, `keySet`, `keysIterator`, `valuesIterator` и `values` возвращают по отдельности ключи и значения отображений в различных формах.
- **Преобразования** `filterKeys` и `mapValues` создают новое отображение путем фильтрации и преобразования привязок существующего отображения.

Таблица 24.8. Операции в трейте `Map`

Что	Что делает
Поиск	
<code>ms.get(k)</code>	Значение <code>Option</code> , связанное с ключом <code>k</code> в отображении <code>ms</code> , или <code>None</code> , если ключ не найден
<code>ms(k)</code> (или после раскрытия <code>ms apply k</code>)	Значение, связанное с ключом <code>k</code> в отображении <code>ms</code> , или выдает исключение, если ключ не найден
<code>ms.getOrElse(k, d)</code>	Значение, связанное с ключом <code>k</code> в отображении <code>ms</code> , или значение по умолчанию <code>d</code> , если ключ не найден
<code>ms.contains(k)</code>	Проверяет, содержится ли в <code>ms</code> отображение для ключа <code>k</code>
<code>ms.isDefinedAt(k)</code>	То же, что и <code>contains</code>
Создание подколлекций	
<code>ms.keys</code>	<code>Iterable</code> -коллекция, содержащая каждый ключ, имеющийся в <code>ms</code>
<code>ms.keySet</code>	Множество, содержащее каждый ключ, имеющийся в <code>ms</code>
<code>ms.keysIterator</code>	Итератор, выдающий каждый ключ, имеющийся в <code>ms</code>
<code>ms.values</code>	<code>Iterable</code> -коллекция, содержащая каждое значение, связанное с ключом в <code>ms</code>
<code>ms.valuesIterator</code>	Итератор, выдающий каждое значение, связанное с ключом в <code>ms</code>
Преобразования	
<code>ms.view.filterKeys(p)</code>	Представление отображения, содержащее только те отображения в <code>ms</code> , в которых ключ удовлетворяет условию <code>p</code>
<code>ms.view.mapValues(f)</code>	Представление отображения, получающееся в результате применения функции <code>f</code> к каждому значению, связанному с ключом в <code>ms</code>

Таблица 24.9. Операции в трейте `immutable.Map`

Что	Что делает
Добавления и обновления	
<code>ms + (k -> v)</code> (или <code>ms.updated(k, v)</code>)	Отображение, содержащее все ассоциации <code>ms</code> , а также ассоциацию <code>k -> v</code> ключа <code>k</code> со значением <code>v</code>
<code>ms ++= kvs</code> (или <code>ms.concat(kvs)</code>)	Отображение, содержащее все ассоциации <code>ms</code> , а также все пары «ключ — значение» из <code>kvs</code>
<code>ms.updatedWith(k)(f)</code>	Отображение с добавлением, обновлением или удалением привязки для ключа <code>k</code> . Функция <code>f</code> принимает в качестве параметра значение, связанное в настоящий момент с ключом <code>k</code> (или <code>None</code> , если такой привязки нет), и возвращает новое значение (или <code>None</code> для удаления привязки)
Удаления	
<code>ms - k</code> (или <code>ms.removed(k)</code>)	Отображение, содержащее все ассоциации <code>ms</code> , за исключением тех, которые относятся к ключу <code>k</code>
<code>ms -- ks</code> (или <code>ms.removedAll(ks)</code>)	Отображение, содержащее все ассоциации <code>ms</code> , за исключением тех, ключи которых входят в <code>ks</code>

Таблица 24.10. Операции в трейте `mutable.Map`

Что	Что делает
Добавления и обновления	
<code>ms(k) = v</code> (или после раскрытия <code>ms.update(k, v)</code>)	Добавляет в качестве побочного эффекта ассоциацию ключа <code>k</code> со значением <code>v</code> к отображению <code>ms</code> , перезаписывая все ранее имевшиеся ассоциации <code>k</code>
<code>ms += (k -> v)</code>	Добавляет в качестве побочного эффекта ассоциацию ключа <code>k</code> со значением <code>v</code> к отображению <code>ms</code> и возвращает само отображение <code>ms</code>
<code>ms ++= kvs</code>	Добавляет в качестве побочного эффекта все ассоциации, имеющиеся в <code>kvs</code> , к <code>ms</code> и возвращает само отображение <code>ms</code>
<code>ms.put(k, v)</code>	Добавляет к <code>ms</code> ассоциацию ключа <code>k</code> со значением <code>v</code> и возвращает как <code>Option</code> любое значение, ранее связанное с <code>k</code>
<code>ms.getOrElseUpdate(k, d)</code>	Если ключ <code>k</code> определен в отображении <code>ms</code> , то возвращает связанное с ним значение. В противном случае обновляет <code>ms</code> ассоциацией <code>k -> d</code> и возвращает <code>d</code>

Что	Что делает
<code>ms.updateWith(k)(f)</code>	Добавляет, обновляет или удаляет ассоциацию с ключом <code>k</code> . Функция <code>f</code> принимает в качестве параметра значение, которое в настоящий момент связано с <code>k</code> (или <code>None</code> , если такой ассоциации нет), и возвращает новое значение (или <code>None</code> при удалении ассоциации)
Удаления	
<code>ms -= k</code>	Удаляет в качестве побочного эффекта ассоциацию с ключом <code>k</code> из <code>ms</code> и возвращает само отображение <code>ms</code>
<code>ms --- ks</code>	Удаляет в качестве побочного эффекта все ассоциации из <code>ms</code> с ключами, имеющимися в <code>ks</code> , и возвращает само отображение <code>ms</code>
<code>ms.remove(k)</code>	Удаляет все ассоциации с ключом <code>k</code> из <code>ms</code> и возвращает как <code>Option</code> любое значение, ранее связанное с <code>k</code>
<code>ms.filterInPlace(p)</code>	Сохраняет в <code>ms</code> только те ассоциации, у которых ключ удовлетворяет условию <code>p</code>
<code>ms.clear()</code>	Удаляет из <code>ms</code> все ассоциации
Преобразование и клонирование	
<code>ms.mapValuesInPlace(f)</code>	Выполняет преобразование всех связанных значений в отображении <code>ms</code> с помощью функции <code>f</code>
<code>ms.clone()</code>	Возвращает новое изменяемое отображение с такими же ассоциациями, как и в <code>ms</code>

Операции добавления и удаления для отображений — зеркальные отражения таких же операций для множеств. Неизменяемое отображение может быть преобразовано с помощью операций `+`, `-` и `updated`. Для сравнения: изменяемое отображение `m` можно обновить «на месте» двумя способами: `m(key) = value` и `m += (key -> value)`. Изменяемые отображения также поддерживают вариант `m.put(key, value)`, который возвращает значение `Option`, содержащее то, что прежде ассоциировалось с ключом, или `None`, если ранее такой ключ в отображении отсутствовал.

Метод `getOrElseUpdate` пригодится для обращения к отображениям там, где они действуют в качестве кэша. Скажем, у вас есть весьма затратное вычисление, запускаемое путем вызова функции `f`:

```
def f(x: String) =
  println("taking my time.")
  Thread.sleep(100)
  x.reverse
```

Далее предположим, что у `f` нет побочных эффектов, поэтому ее повторный вызов с тем же самым аргументом всегда будет выдавать тот же самый результат. В таком случае можно сберечь время, сохранив ранее вычисленные привязки аргумента и результата выполнения `f` в отображении, и вычислять результат выполнения `f`, только если результат для аргумента не был найден в отображении. Можно сказать, что отображение — это *кэш* для вычислений функции `f`:

```
val cache = collection.mutable.Map[String, String]()
```

Теперь можно создать более эффективную кэшированную версию функции `f`:

```
scala> def cachedF(s: String) = cache.getOrElseUpdate(s, f(s))
def cachedF(s: String): String
```

```
scala> cachedF("abc")
taking my time.
val res16: String = cba
```

```
scala> cachedF("abc")
val res17: String = cba
```

Обратите внимание: второй аргумент `getOrElseUpdate` — это аргумент, передаваемый по имени. Следовательно, показанное ранее вычисление `f("abc")` выполняется лишь в том случае, если методу `getOrElseUpdate` потребуется значение его второго аргумента, что происходит именно тогда, когда его первый аргумент не найден в кэширующем отображении. Вы могли бы также непосредственно реализовать `cachedF`, используя только основные операции с отображениями, но для этого понадобится дополнительный код:

```
def cachedF(arg: String) =
  cache.get(arg) match
    case Some(result) => result
    case None =>
      val result = f(arg)
      cache(arg) = result
      result
```

24.7. Конкретные классы неизменяемых коллекций

В Scala на выбор предлагается множество конкретных классов неизменяемых коллекций. Друг от друга они отличаются реализуемыми трейтами (отображения, множества, последовательности) тем, могут ли они быть бес-

конечными, и скоростью выполнения различных операций. Начнем с обзора наиболее востребованных типов неизменяемых коллекций.

Списки

Списки относятся к конечным неизменяемым последовательностям. Они обеспечивают постоянное время доступа к своим первым элементам, а также ко всей остальной части списка и имеют постоянное время выполнения операций `cons` для добавления нового элемента в начало списка. Многие другие операции имеют линейную зависимость времени выполнения от длины списка. Более подробно списки рассматриваются в главах 14 и 1.

Ленивые списки

Ленивые списки похожи на списки, за исключением того, что их элементы вычисляются лениво (или отложено). Вычисляться будут только запрошенные элементы. Поэтому ленивые списки могут быть бесконечно длинными. Во всем остальном они имеют такие же характеристики производительности, что и списки.

В то время как списки конструируются с помощью оператора `::`, ленивые конструируются с помощью похожего оператора `#::`. Пример ленивого списка, содержащего целые числа 1, 2 и 3, выглядит следующим образом:

```
scala> val str = 1 #:: 2 #:: 3 #:: LazyList.empty
val str: scala.collection.immutable.LazyList[Int] =
  LazyList(<not computed>)
```

«Голова» этого ленивого списка — 1, а «хвост» — 2 и 3. Но никакие элементы здесь не выводятся, поскольку список еще не вычислен! Ленивые списки объявлены как вычисляющиеся лениво, и метод `toString`, вызванный в отношении такого списка, заботится о том, чтобы не навязывать лишние вычисления.

Далее показан более сложный пример. В нем вычисляется ленивый список, содержащий последовательность чисел Фибоначчи с заданными двумя числами. Каждый элемент последовательности Фибоначчи есть сумма двух предыдущих элементов в серии:

```
scala> def fibFrom(a: Int, b: Int): LazyList[Int] =
  a #:: fibFrom(b, a + b)
def fibFrom: (a: Int, b: Int)LazyList[Int]
```

Эта функция кажется обманчиво простой. Понятно, что первый элемент последовательности — a , за ним стоит последовательность Фибоначчи, начинающаяся с b , затем — элемент со значением $a + b$. Самое сложное здесь — вычислить данную последовательность, не вызвав бесконечную рекурсию. Если функция вместо `#::` использует оператор `::`, то каждый вызов функции будет приводить к еще одному вызову, что выльется в бесконечную рекурсию. Но, поскольку применяется оператор `#::`, правая часть выражения не вычисляется до тех пор, пока не будет востребована.

Вот как выглядят первые несколько элементов последовательности Фибоначчи, начинающейся с двух заданных чисел:

```
scala> val fibs = fibFrom(1, 1).take(7)
val fibs: scala.collection.immutable.LazyList[Int] =
  LazyList(<not computed>)
```

```
scala> fibs.toList
val res23: List[Int] = List(1, 1, 2, 3, 5, 8, 13)
```

Неизменяемые `ArraySeq`

Списки очень эффективны в алгоритмах, которые работают исключительно с начальными элементами. Получение, добавление и удаление начала списка занимает постоянное время. А вот получение или изменения последующих элементов — это операции линейного времени, зависящие от длины списка. В результате список может быть не самым оптимальным вариантом для алгоритмов, которые не ограничиваются обработкой начальных элементов.

Последовательный массив (`ArraySeq`) — это неизменяемая последовательность на основе приватного массива, которая решает проблему неэффективного произвольного доступа в списках. Последовательные массивы позволяют получить любой элемент коллекции за постоянное время. Благодаря этому вы можете не ограничиваться работой только с начальными элементами. Время получения элемента не зависит от его местоположения, и потому `ArraySeq` может оказаться эффективней списков в некоторых алгоритмах.

С другой стороны, тип `ArraySeq` основан на `Array`, поэтому добавление элементов в начало занимает линейное время, а не постоянное, как в случае со списками. Более того, на всякое добавление или обновление одного элемента в `ArraySeq` уходит линейное время, поскольку при этом копируется весь внутренний массив.

Векторы

`List` и `ArraySeq` — структуры данных, эффективные для одних вариантов использования и неэффективные для других. Например, добавление элемента в начало `List` занимает постоянное время, а в `ArraySeq` — линейное. С другой стороны, индексированный доступ занимает постоянное время в `ArraySeq` и линейное в `List`.

Вектор обеспечивает хорошую производительность для всех своих операций. Доступ и обновление любых элементов вектора занимает «эффективно постоянное время», как описано ниже. Эти операции выполняются медленнее, чем получение начала списка или чтение элементов в последовательном массиве, но время их выполнения остается постоянным. В результате алгоритмы, использующие векторы, не должны ограничиваться доступом или обновлением только к началу последовательности. Они могут обращаться к элементам и обновлять их в произвольном месте и поэтому могут быть гораздо более удобными в написании.

Создаются и изменяются векторы точно так же, как и любые другие последовательности:

```
val vec = scala.collection.immutable.Vector.empty
val vec2 = vec :+ 1 :+ 2 // Vector(1, 2)
val vec3 = 100 +: vec2  // Vector(100, 1, 2)
vec3(0)                // 100
```

Для представления векторов используются широкие неглубокие деревья. Каждый узел дерева содержит до 32 элементов вектора или до 32 других узлов дерева. Векторы, содержащие до 32 элементов, могут быть представлены одним узлом. Векторы с количеством элементов до $32 \cdot 32 = 1024$ могут быть представлены в виде одного направления. Двух переходов от корня дерева к конечному элементу достаточно для векторов, имеющих до 2^{15} элементов, трех — для векторов с 2^{20} , четырех — для векторов с 2^{25} элементов, а пяти — для векторов с количеством элементов до 2^{30} . Следовательно, для всех векторов разумного размера выбор элемента требует до пяти обычных доступов к массиву. Именно это мы имели в виду, когда написали «эффективно постоянное время».

Векторы неизменяемы, следовательно, внести в элемент вектора изменение на месте невозможно. Но метод `updated` позволяет создать новый вектор, отличающийся от заданного только одним элементом:

```
val vec = Vector(1, 2, 3)
vec.updated(2, 4) // Vector(1, 2, 4)
vec              // Vector(1, 2, 3)
```

Последняя строка данного кода показывает, что вызов метода `updated` никак не повлиял на исходный вектор `vec`. Функциональное обновление векторов также занимает «эффективно постоянное время». Обновить элемент в середине вектора можно с помощью копирования узла, содержащего элемент, и каждого указывающего на него узла, начиная с корня дерева. Это значит, функциональное обновление создает от одного до пяти узлов, каждый из которых содержит до 32 элементов или поддеревьев. Конечно, это гораздо затратнее обновления на месте в изменяемом массиве, но все же намного дешевле копирования всего вектора.

Векторы сохраняют разумный баланс между быстрым произвольным доступом и быстрыми произвольными функциональными обновлениями, поэтому в настоящий момент они представляют исходную реализацию неизменяемых индексированных последовательностей:

```
collection.immutable.IndexedSeq(1, 2, 3) // Vector(1, 2, 3)
```

Неизменяемые очереди

В очередях используется принцип «первым пришел, первым вышел». Упрощенная реализация неизменяемых очередей рассматривалась в главе 18. Создать пустую неизменяемую очередь можно следующим образом:

```
val empty = scala.collection.immutable.Queue[Int]()
```

Добавить элемент в неизменяемую очередь можно с помощью метода `enqueue`:

```
val has1 = empty.enqueue(1) // Queue(1)
```

Чтобы добавить в очередь сразу несколько элементов, следует вызвать метод `enqueueAll` с коллекцией в качестве его аргументов:

```
val has123 = has1.enqueueAll(List(2, 3)) // Queue(1, 2, 3)
```

Чтобы удалить элемент из начала очереди, следует воспользоваться методом `dequeue`:

```
scala> val (element, has23) = has123.dequeue
val element: Int = 1
has23: scala.collection.immutable.Queue[Int] = Queue(2, 3)
```

Обратите внимание: `dequeue` возвращает пару, состоящую из удаленного элемента и оставшейся части очереди.

Диапазоны

Диапазон представляет собой упорядоченную последовательность целых чисел с одинаковым интервалом между ними. Например, 1, 2, 3 является диапазоном точно так же, как и 5, 8, 11, 14. Чтобы создать в Scala диапазон, следует воспользоваться предопределенными методами `to` и `by`. Рассмотрим несколько примеров:

```
1 to 3          // Range(1, 2, 3)
5 to 14 by 3    // Range(5, 8, 11, 14)
```

Если нужно создать диапазон, исключая его верхний предел, то следует вместо метода `to` воспользоваться методом-помощником `until`:

```
1 until 3 // Range(1, 2)
```

Диапазоны представлены постоянным объемом памяти, поскольку могут быть определены всего тремя числами: их началом, их концом и значением шага. Благодаря такому представлению большинство операций над диапазонами выполняется очень быстро.

Сжатые коллекции НАМТ

Хеш-извлечения (hash-tries¹) — стандартный способ эффективной реализации неизменяемых множеств и отображений. Сжатые коллекции НАМТ (хеш-массивы сопоставленных префиксных деревьев)² — разновидность хеш-извлечений в JVM, которая оптимизирует размещение элементов и следит за тем, чтобы деревья были представлены каноничным и компактным образом.

Их представление похоже на векторы тем, что в них также используются деревья, где у каждого узла имеется 32 элемента, или 32 поддерева, но выбор осуществляется на основе хеш-кода. Например, самые младшие пять разрядов хеш-кода ключа используются в целях поиска заданного ключа в отображении для выбора первого поддерева, следующие пять — для выбора следующего поддерева и т. д. Процесс выбора прекращается, как только

¹ Trie происходит от слова retrieval (извлечение) и произносится как «три» или «трай».

² *Steindorfer M.J., Vinju J.J.* Optimizing hash-array mapped tries for fast and lean immutable JVM collections // ACM SIGPLAN Notices, volume 50, pages 783–800. ACM, 2015.

у всех элементов, хранящихся в узле, будет хеш-код, отличающийся от всех остальных в разрядах, выбранных до сих пор. Таким образом, не обязательно используются все разряды хеш-кода.

Хеш-извлечения позволяют достичь хорошего баланса между достаточно быстрым поиском и достаточно эффективными функциональными вставками (+) и удалениями (-). Именно поэтому в Scala они положены в основу реализаций по умолчанию неизменяемых отображений и множеств. Фактически для неизменяемых множеств и отображений, содержащих менее пяти элементов, в Scala предусматривается дальнейшая оптимизация. Множества и отображения, содержащие от одного до четырех элементов, хранятся как отдельные объекты, содержащие эти элементы (или в случае с отображениями — пары «ключ — значение») в виде полей. Пустое изменяемое множество и пустое изменяемое отображение во всех случаях являются объектами-одиночками — не нужно дублировать для них место хранения, поскольку пустые неизменяемые множество или отображение всегда будут оставаться пустыми.

Красно-черные деревья

Форма сбалансированных двоичных деревьев — красно-черные деревья, в которых одни узлы обозначены как красные, а другие — как черные. Операции над ними, как и над любыми другими сбалансированными двоичными деревьями, гарантированно завершаются за время, экспоненциально зависящее от размера дерева.

Scala предоставляет реализации множеств и отображений, во внутреннем представлении которых используется красно-черное дерево. Доступ к ним осуществляется по именам `TreeSet` и `TreeMap`:

```
val set = collection.immutable.TreeSet.empty[Int]
set + 1 + 3 + 3 // TreeSet(1, 3)
```

Кроме того, красно-черные деревья в Scala — стандартный механизм реализации `SortedSet`, поскольку предоставляют весьма эффективный итератор, возвращающий все элементы множества в отсортированном виде.

Неизменяемые битовые множества

Битовое множество представляет коллекцию небольших целых чисел в виде битов большего целого числа. Например, битовое множество, содержащее

3, 2 и 0, будет представлено в двоичном виде как целое число 1101, которое в десятичном виде является числом 13.

Во внутреннем представлении битовые множества используют массив из 64-разрядных Long-значений. Первое Long-значение в массиве предназначено для целых чисел от 0 до 63, второе — от 64 до 127 и т. д. Таким образом, битовые множества очень компактны, поскольку самое большое целое число в множестве чуть меньше нескольких сотен или около того.

Операции над битовыми множествами выполняются очень быстро. Проверка на присутствие занимает постоянное время. На добавление элемента в множество уходит время, пропорциональное количеству Long-значений в массиве битового множества, которых обычно немного. Некоторые примеры использования битового множества выглядят следующим образом:

```
val bits = scala.collection.immutable.BitSet.empty
val moreBits = bits + 3 + 4 + 4 // BitSet(3, 4)
moreBits(3) // true
moreBits(0) // false
```

Векторные отображения

VectorMap (векторное отображение) представляет отображение с использованием как вектора (для ключей), так и HashMap. Оно предоставляет итератор, возвращающий все элементы в том порядке, в котором они были вставлены.

```
import scala.collection.immutable.VectorMap
val vm = VectorMap.empty[Int, String]
val vm1 = vm + (1 -> "one") // VectorMap(1 -> one)
val vm2 = vm1 + (2 -> "two") // VectorMap(1 -> one, 2 -> two)
vm2 == Map(2 -> "two", 1 -> "one") // true
```

В первых строчках видно, что содержимое VectorMap сохраняет порядок вставки, а последняя строчка показывает, что векторные отображения можно сравнивать с другими видами отображений и в ходе этого сравнения не учитывается порядок следования элементов.

Списочные отображения

Списочное отображение представляет собой отображение в виде связанного списка пар «ключ — значение». Как правило, операции над списочными отображениями могут потребовать перебора всего списка. В связи с этим такие операции занимают время, пропорциональное размеру отображения.

Списочные отображения не нашли в Scala широкого применения, поскольку работа со стандартными неизменяемыми отображениями почти всегда выполняется быстрее. Единственно возможное положительное отличие наблюдается в том случае, если по какой-то причине отображение сконструировано так, что первые элементы списка выбираются гораздо чаще других элементов.

```
val map = collection.immutable.ListMap(1 -> "one", 2 -> "two")
map(2) // "two"
```

24.8. Конкретные классы изменяемых коллекций

Изучив наиболее востребованные из имеющихся в стандартной библиотеке Scala классы неизменяемых коллекций, рассмотрим классы изменяемых коллекций.

Буферы массивов

Буферы массивов уже встречались нам в разделе 15.1. В таком буфере содержатся массив и его размер. Большинство операций над буферами массивов выполняются с такой же скоростью, что и над массивами, поскольку эти операции просто обращаются к обрабатываемому массиву и вносят в него изменения. Кроме того, у буфера массива могут быть данные, весьма эффективно добавляемые в его конец. Добавление записи в буфер массива занимает амортизированно постоянное время. Из этого следует, что буферы массивов хорошо подходят для эффективного создания больших коллекций, когда новые элементы всегда добавляются в конец. Вот как выглядят некоторые примеры их применения:

```
val buf = collection.mutable.ArrayBuffer.empty[Int]
buf += 1 // ArrayBuffer(1)
buf += 10 // ArrayBuffer(1, 10)
buf.toArray // Array(1, 10)
```

Буферы списков

В разделе 15.1 нам встречались и буферы списков. Буфер списка похож на буфер массива, за исключением того, что внутри него используется не массив, а связанный список. Если сразу после создания буфер предполагается

превращать в список, то лучше воспользоваться буфером списка, а не буфером массива. Вот как выглядит соответствующий пример¹:

```
val buf = collection.mutable.ListBuffer.empty[Int]
buf += 1 // ListBuffer(1)
buf += 10 // ListBuffer(1, 10)
buf.toList // List(1, 10)
```

Построители строк

По аналогии с тем, что буфер массива используется для создания массивов, а буфер списка — для создания списков, построитель строк применяется для создания строк. Построители строк используются настолько часто, что заранее импортируются в пространство имен по умолчанию. Создать их можно с помощью простого выражения `new StringBuilder`:

```
val buf = new StringBuilder
buf += 'a' // a
buf += "bcdef" // abcdef
buf.toString // abcdef
```

ArrayDeque

`ArrayDeque` — изменяемая последовательность, которая поддерживает эффективное добавление элементов в начало и в конец. Внутри она использует массив изменяемого размера. Если вам нужно вставлять элементы в начало или в конец буфера, то используйте `ArrayDeque` вместо `ArrayBuffer`.

Очереди

Наряду с неизменяемыми очередями в `Scala` есть и изменяемые. Используются они аналогично, но для добавления элементов вместо `enqueue` применяются операторы `+=` и `++=`. Кроме того, метод `dequeue` будет просто удалять из изменяемой очереди головной элемент и возвращать его. Примеры использования даны ниже:

```
val queue = new scala.collection.mutable.Queue[String]
queue += "a" // Queue(a)
```

¹ Появляющийся в ответах интерпретатора в этом и в нескольких других примерах раздела `buf.type` является синглтон-типом. Как сказано в разделе 7.6, `buf.type` означает, что переменная содержит именно тот объект, на который указывает `buf`.

```
queue += List("b", "c") // Queue(a, b, c)
queue                // Queue(a, b, c)
queue.dequeue        // a
queue                // Queue(b, c)
```

Стеки

Scala предоставляет изменяемый стек. Ниже представлен пример:

```
val stack = new scala.collection.mutable.Stack[Int]
stack.push(1) // Stack(1)
stack        // Stack(1)
stack.push(2) // Stack(2, 1)
stack        // Stack(2, 1)
stack.top    // 2
stack        // Stack(2, 1)
stack.pop    // 2
stack        // Stack(1)
```

Обратите внимание: в Scala нет поддержки неизменяемых стеков, поскольку данная функциональность уже реализована в списках. Операция `push` над неизменяемым стеком аналогична выражению `a ::` для списка. Операция `pop` — то же самое, что вызов `head` и `tail` для списка.

Изменяемые `ArraySeq`

Последовательный массив — это изменяемая последовательность фиксированного размера, которая хранит свои элементы внутри `Array[AnyRef]`. В Scala он реализован в виде класса `ArraySeq`.

Данный класс обычно используется в случаях, когда вам нужен производительный массив и притом необходимо создавать обобщенные экземпляры последовательности с элементами, тип которых не известен заранее и не может быть получен во время выполнения с помощью `ClassTag`. С этими проблемами, которые присущи массивам, вы познакомитесь чуть позже, в разделе 24.9.

Хеш-таблицы

Хеш-таблица сохраняет свои элементы в образующем ее массиве, помещая каждый в позицию в массиве, определяемую хеш-кодом этого элемента. На добавление элемента в хеш-таблицу всегда уходит одно и то же время,

если только в массиве нет еще одного элемента с точно таким же хеш-кодом. Поэтому, пока помещенные в хеш-таблицу элементы имеют хорошее распределение хеш-кодов, работа с ней выполняется довольно быстро. По этой причине типы изменяемых отображений и множеств по умолчанию в Scala основаны на хеш-таблицах.

Хеш-множества и хеш-отображения используются точно так же, как и любые другие множества или отображения. Ниже представлены некоторые простые примеры:

```
val map = collection.mutable.HashMap.empty[Int,String]
map += (1 -> "make a web site")
           // Map(1 -> make a web site)
map += (3 -> "profit!")
           // Map(1 -> make a web site, 3 -> profit!)
map(1)      // make a web site
map.contains(2) // false
```

Конкретный порядок обхода элементов хеш-таблицы не гарантируется. Выполняется простой обход элементов массива, на котором основана хеш-таблица, в порядке расположения его элементов. Чтобы получить гарантированный порядок обхода, следует воспользоваться *связанными* хеш-отображением или множеством вместо обычных. Связанные хеш-отображение или множество почти аналогичны обычным хеш-отображению или множеству, но включают связанный список элементов в порядке их добавления. Обход элементов такой коллекции всегда выполняется в том же порядке, в котором они добавлялись в нее изначально.

Слабые хеш-отображения

Слабое хеш-отображение представляет собой особую разновидность хеш-отображения, в которой сборщик мусора не следует по ссылкам от отображения к хранящимся в нем ключам. Это значит, ключ и связанное с ним значение исчезнут из отображения, если на данный ключ нет другой ссылки. Слабые хеш-отображения используются для решения таких задач, как кэширование, когда нужно повторно задействовать результат затратной функции в случае повторного вызова функции в отношении того же самого ключа. Если ключи и результаты функции хранятся в обычном хеш-отображении, то оно может бесконечно разрастись и никакие ключи никогда не станут мусором. Этой проблемы удастся избежать с помощью слабого хеш-отображения. Как только объект ключа становится недоступным, связанная с ним запись удаляется из такого отображения. Слабые хеш-отображения реализованы

в Scala в виде обертки положенной в их основу Java-реализации `java.util.WeakHashMap`.

Совместно используемые отображения

К совместно используемому отображению могут обращаться сразу несколько потоков. В дополнение к обычным операциям с `Map` это отображение предоставляет атомарные операции (табл. 24.11).

Таблица 24.11. Операции в трейте `concurrent.Map`

Что	Что делает
<code>m.putIfAbsent(k, v)</code>	Добавляет привязку «ключ — значение» <code>k -> v</code> , кроме тех случаев, когда <code>k</code> уже определен в <code>m</code>
<code>m.remove(k, v)</code>	Удаляет запись для ключа <code>k</code> , если он в данный момент отображен на значение <code>v</code>
<code>m.replace(k, old, new)</code>	Заменяет значение, связанное с <code>k</code> , новым значением <code>new</code> , если ранее с ключом было связано значение <code>old</code>
<code>m.replace(k, v)</code>	Заменяет значение, связанное с <code>k</code> , значением <code>v</code> , если ранее этот ключ был связан с каким-либо значением

Трейт `scala.concurrent.Map` определяет интерфейс для изменяемых отображений с поддержкой конкурентного доступа. Стандартная библиотека предлагает две реализации этого трейта. Первая — это `java.util.concurrent.ConcurrentMap` из Java, которую можно автоматически превратить в отображение языка Scala, используя стандартные для Java/Scala операции приведения коллекций (эти преобразования будут описаны в разделе 24.16). Вторая реализация, `TrieMap`, основана на НАМТ без блокировок.

Изменяемые битовые множества

Изменяемое битовое множество похоже на неизменяемое, но отличается тем, что может быть изменено на месте. По сравнению с неизменяемым оно работает при обновлениях немного эффективнее, поскольку неизменные `Long`-значения копировать не нужно. Пример использования выглядит так:

```
val bits = scala.collection.mutable.BitSet.empty
bits += 1 // BitSet(1)
bits += 3 // BitSet(1, 3)
bits    // BitSet(1, 3)
```

24.9. Массивы

Массивы в Scala — особая разновидность коллекции. С одной стороны, массивы Scala в точности соответствуют массивам Java. То есть Scala-массив `Array[Int]` представлен как Java-массив `int[]`, `Array[Double]` — как `double[]`, а `Array[String]` — как `String[]`. Но вместе с тем массивы Scala предоставляют гораздо больше, чем их Java-аналоги. Во-первых, массивы Scala могут быть *обобщенными*. То есть можно воспользоваться массивом `Array[T]`, где `T` является параметром типа или абстрактным типом. Во-вторых, массивы Scala совместимы со Scala-последовательностями, то есть туда, где требуется `Seq[T]`, можно передавать `Array[T]`. И наконец, массивы Scala также поддерживают все операции с последовательностями. Приведем несколько практических примеров:

```
val a1 = Array(1, 2, 3)
val a2 = a1.map(_ * 3)           // Array(3, 6, 9)
val a3 = a2.filter(_ % 2 != 0) // Array(3, 9)
a3.reverse                     // Array(9, 3)
```

Если учесть, что массивы Scala представлены точно так же, как массивы Java, то как эти дополнительные свойства могут поддерживаться в Scala?

Ответ заключается в систематическом использовании неявных преобразований. Массив не может претендовать на то, чтобы *быть* последовательностью, поскольку тип данных, представляющих настоящий массив, не является подтипом типа `Seq`. Вместо этого там, где массив будет использоваться в качестве последовательности `Seq`, он будет неявно обернут в подкласс класса `Seq`. Имя этого подкласса — `scala.collection.mutable.ArraySeq`. Вот как это работает:

```
val seq: collection.Seq[Int] = a1 // ArraySeq(1, 2, 3)
val a4: Array[Int] = seq.toArray // Array(1, 2, 3)
a1 eq a4                // false
```

Сеанс работы с интерпретатором показывает, что массивы совместимы с последовательностями благодаря неявному преобразованию из `Array` в `ArraySeq`. Преобразование в обратном направлении, из `ArraySeq` в `Array`, можно выполнить с использованием метода `toArray`, который определен в классе `Iterable`. В последней строке приведенного ранее диалога с интерпретатором показано, что заключенный в оболочку массив при его изъятии оттуда с помощью метода `toArray` возвращает копию исходного массива.

Есть еще одно неявное преобразование, применимое к массивам. Оно просто «добавляет» к массивам все методы, применимые к последовательностям, но сами массивы в последовательности не превращает. «Добавляет» означает,

что массив заворачивается в другой объект типа `ArrayOps`, который поддерживает все методы работы с последовательностями. Обычно этот `ArrayOps`-объект существует весьма непродолжительное время — он становится недоступен после вызова метода для работы с последовательностью, и его место хранения может быть использовано повторно. Современные виртуальные машины зачастую вообще избегают создания таких объектов.

Разница между этими двумя неявными преобразованиями массивов показана в следующем примере:

```
val seq: collection.Seq[Int] = a1 // ArraySeq(1, 2, 3)
seq.reverse // ArraySeq(3, 2, 1)
val ops: collection.ArrayOps[Int] = a1 // Array(1, 2, 3)
ops.reverse // Array(3, 2, 1)
```

Как видите, при вызове метода `reverse` в отношении объекта `seq` типа `ArraySeq` опять будет получен объект типа `ArraySeq`. Это не противоречит здравому смыслу, поскольку массивы `ArraySeq`, заключенные в оболочку, относятся к типам `Seq`, а вызов метода `reverse` в отношении любого `Seq`-объекта вновь даст `Seq`-объект. В то же время вызов метода `reverse` в отношении значения `ops` класса `ArrayOps` приведет к возвращению значения типа `Array`, а не `Seq`.

Приведенный ранее пример с `ArrayOps` был надуманным, предназначенным лишь для демонстрации разницы с `ArraySeq`. В обычной ситуации вы бы не стали определять значение класса `ArrayOps`, а просто вызвали бы в отношении массива метод из класса `Seq`:

```
a1.reverse // Array(3, 2, 1)
```

Объект `ArrayOps` вставляется автоматически в ходе неявного преобразования. Следовательно, показанная выше строка кода эквивалентна следующей строке, где метод `intArrayOps` является преобразованием, которое неявно вставлялось в предыдущем примере:

```
intArrayOps(a1).reverse // Array(3, 2, 1)
```

Возникает вопрос, касающийся способа выбора компилятором `intArrayOps` в показанной выше строке среди других неявных преобразований в `ArraySeq`. Ведь оба преобразования отображают массив на тип, поддерживающий метод `reverse`, указанный во введенном коде. Ответ на данный вопрос — уровни приоритета этих двух преобразований. У преобразования `ArrayOps` более высокий приоритет, чем у `ArraySeq`. Первое преобразование определено в объекте `Predef`, а второе — в классе `scala.LowPriorityImplicits`, являющемся суперклассом для `Predef`. Неявные преобразования в подклассах

и подобъектах имеют приоритет над неявными преобразованиями в базовых классах. Следовательно, если применимы оба преобразования, то будет выбрано то, которое определено в `Predef`. Очень похожая схема, рассмотренная в разделе 21.7, работает для строк.

Теперь вы знаете, что массивы совместимы с последовательностями и могут поддерживать все операции, применяемые к последовательностям. А как насчет обобщенности? В Java вы не можете воспользоваться записью `T[]`, где `T` представляет собой параметр типа. А как же тогда представлен имеющийся в Scala тип `Array[T]`? Фактически такой обобщенный массив, как `Array[T]`, может во время выполнения программы стать любым из восьми примитивных типов массивов Java: `byte[]`, `short[]`, `char[]`, `int[]`, `long[]`, `float[]`, `double[]`, `boolean[]` — или же стать массивом объектов. Единственный общий тип, охватывающий во время выполнения программы все эти типы, — `AnyRef` (или его аналог `java.lang.Object`), следовательно, это именно тот тип, на который компилятор Scala отображает `Array[T]`. Во время выполнения программы, когда происходит обращение к элементу массива типа `Array[T]` или обновление этого элемента, производится ряд проверок на соответствие типам. Благодаря этому определяется действительный тип массива, а затем выполняется корректная операция уже над Java-массивом. Проверки на соответствие типам несколько замедляют операции над массивами. Можно ожидать, что обращения к обобщенным массивам будут в три-четыре раза медленнее обращений к простым массивам или массивам объектов. Следовательно, если требуется максимально высокая производительность, то предпочтение следует отдавать конкретным, а не обобщенным массивам.

Но одного представления типа обобщенного массива недостаточно, должен существовать также способ *создания* обобщенных массивов. А это еще более сложная задача, которая требует от вас оказать некоторую помощь. В качестве примера рассмотрим попытку создания метода, работающего с обобщениями и создающего массив:

```
// Неправильно!  
def evenElems[T](xs: Vector[T]): Array[T] =  
  val arr = new Array[T]((xs.length + 1) / 2)  
  for i <- 0 until xs.length by 2 do  
    arr(i / 2) = xs(i)  
  arr
```

Метод `evenElems` возвращает новый массив, состоящий из тех элементов используемого в качестве аргумента вектора `xs`, которые находятся в нем на четных позициях. В первой строке тела метода `evenElems` создается получаемый в результате массив, имеющий тот же тип элементов, что и аргумент. Следовательно, в зависимости от фактического параметра типа для `T` это

может быть `Array[Int]`, или `Array[Boolean]`, или массив из других примитивных типов Java, или же массив какого-нибудь ссылочного типа. Но все эти типы имеют во время выполнения программы различные представления, поэтому возникает вопрос: как среда выполнения Scala собирается выбирать из них нужное? По сути, сделать это, основываясь на имеющейся информации, она не может, поскольку фактический тип, который соответствует параметру типа `T`, во время выполнения кода затирается. Поэтому при попытке скомпилировать показанный ранее код будет получено следующее сообщение об ошибке:

```
2 | val arr = new Array[T]((xs.length + 1) / 2)
  |                                     ^
  |                                     No ClassTag available for T
```

Здесь вам следует помочь компилятору, предоставив подсказку времени выполнения о том, какой параметр типа у `evenElems`. Эта подсказка принимает форму *тега класса* типа `scala.reflect.ClassTag`. Тег класса описывает заданный *исполняемый класс*, предоставляя исчерпывающую информацию о нем конструктору массива.

Во многих случаях компилятор может создавать тег класса самостоятельно. Именно так обстоит дело с конкретными типами наподобие `Int` или `String`. То же распространяется и на некоторые обобщенные типы наподобие `List[T]`, где для выстраивания предположения об исполняемом классе информации вполне достаточно; в данном примере исполняемым классом будет `List`.

Для полностью обобщенных случаев обычно практикуется передача тега класса с помощью контекстного ограничителя, рассмотренного в разделе 23.2. А вот как, используя этот ограничитель, можно исправить показанное ранее определение:

```
// Этот код работает
import scala.reflect.ClassTag
def evenElems[T: ClassTag](xs: Vector[T]): Array[T] =
  val arr = new Array[T]((xs.length + 1) / 2)
  for i <- 0 until xs.length by 2 do
    arr(i / 2) = xs(i)
  arr
```

В этом новом определении компилятор при создании `Array[T]` ищет тег класса для параметра типа `T`, то есть будет искать неявное значение типа `ClassTag[T]`. Если такое значение будет найдено, то тег класса будет использован для создания массива нужного вида. В противном случае вы увидите сообщение об ошибке, похожее на показанное ранее.

Вот как выглядит диалог с интерпретатором, в котором используется метод `evenElems`:

```
evenElems(Vector(1, 2, 3, 4, 5)) // Array(1, 3, 5)
evenElems(Vector("this", "is", "a", "test", "run")) // Array(this, a, run)
```

В обоих случаях компилятор Scala автоматически создает тег класса для типа элемента (сначала `Int`, потом `String`) и передает его неявному параметру метода `evenElems`. Компилятор может сделать то же самое для всех конкретных типов, но не способен на это, если сам аргумент является еще одним параметром типа без признака класса. Например, следующий код не пройдет компиляцию:

```
scala> def wrap[U](xs: Vector[U]) = evenElems(xs)
1 | def wrap[U](xs: Vector[U]) = evenElems(xs)
  |                                     ^
  |                                     No ClassTag available for U
```

Здесь метод `evenElems` получает тег класса для параметра типа `U`, но ничего не находит. Разумеется, решение в данном случае — потребовать еще один неявный тег класса для `U`. Код, представленный ниже, уже пройдет компиляцию:

```
def wrap[U: ClassTag](xs: Vector[U]) = evenElems(xs)
```

Этот пример показывает также, что контекстное ограничение в определении `U` — краткая форма неявного параметра, названного здесь `evidence$1` и имеющего тип `ClassTag[U]`.

24.10. Строки

Как и массивы, строки не являются последовательностями в прямом смысле слова, но могут быть в них преобразованы и вдобавок поддерживают все операции с последовательностями. Ниже приводятся примеры операций, которые могут вызываться в отношении строк:

```
val str = "hello"
str.reverse           // olleh
str.map(_.toUpperCase) // HELLO
str.drop(3)           // lo
str.slice(1, 4)       // ell
val s: Seq[Char] = str // hello
```

Эти операции поддерживаются двумя неявными преобразованиями, рассмотренными в разделе 23.5. Первое, имеющее более низкий уровень приоритета, отображает класс `String` на класс `WrappedString`, являющийся подклассом

`immutable.IndexedSeq`. Это преобразование было применено в последней строке предыдущего примера, в котором строка была преобразована в значение типа `Seq`. Другое преобразование, с более высоким уровнем приоритета, отображает строку на объект `StringOps`, который добавляет к строкам все методы, применяемые к неизменяемым последовательностям. В предыдущем примере это преобразование было неявно вставлено в вызовы методов `reverse`, `map`, `drop` и `slice`.

24.11. Характеристики производительности

Как показали все предыдущие разъяснения, разным типам коллекций свойственны различные характеристики производительности. Именно это обстоятельство становится главной причиной выбора конкретного типа коллекции из множества других типов. Характеристики производительности некоторых наиболее востребованных операций над коллекциями сведены в табл. 24.12 и 24.13.

Таблица 24.12. Характеристики производительности типов последовательностей

Операции	head	tail	apply	update	prepend	append	insert
Неизменяемые							
List	C	C	L	L	C	L	—
LazyList	C	C	L	L	C	L	—
ArraySeq	C	L	C	L	L	L	—
Vector	eC	eC	eC	eC	eC	eC	—
Queue	aC	aC	L	L	L	C	—
Range	C	C	C	—	—	—	—
String	C	L	C	L	L	L	—
Изменяемые							
ArrayBuffer	C	L	C	C	L	aC	L
ListBuffer	C	L	L	L	C	C	L
StringBuilder	C	L	C	C	L	aC	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C	—	—	—
Stack	C	L	L	L	C	L	L
Array	C	L	C	C	—	—	—
ArrayDeque	C	L	C	C	aC	aC	L

Таблица 24.13. Характеристики производительности типов множеств и отображений

Операции	lookup	add	remove	min
Неизменяемые				
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC ^a
VectorMap	eC	eC	aC	L
ListMap	L	L	L	L
Изменяемые				
HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC ^a

^a Если биты плотно упакованы.

Записи в этих двух таблицах расшифровываются следующим образом:

- C операция занимает постоянное (короткое) время;
- eC операция занимает эффективно постоянное время, но это может зависеть от некоторых допущений, таких как максимальная длина вектора или распределение хеш-ключей;
- aC операция занимает амортизированное постоянное время. Некоторые вызовы операции могут занимать больше времени, но если выполняется множество операций, то берется только постоянное среднее время, затрачиваемое на одну операцию;
- Log на операцию уходит время, пропорциональное логарифму размера коллекции;
- L операция имеет линейный характер, то есть на нее уходит время, пропорциональное размеру коллекции;
- операция не поддерживается.

В табл. 24.12 рассматриваются как неизменяемые, так и изменяемые типы последовательностей со следующими операциями:

- head выбор первого элемента последовательности;
- tail создание новой последовательности, содержащей все элементы, за исключением первого;
- apply индексирование;

- update функциональное обновление (с помощью метода `updated`) для неизменяемых последовательностей, обновление с побочными эффектами (с помощью метода `update`) для изменяемых;
- prepend добавление элемента в начало последовательности. Если последовательность неизменяемая, то данная операция приводит к созданию новой последовательности. Если изменяемая, то существующая последовательность изменяется;
- append добавление элемента в конец последовательности. Если последовательность неизменяемая, то данная операция приводит к созданию новой последовательности. Если изменяемая, то существующая последовательность изменяется;
- insert вставка элемента в произвольную позицию последовательности. Поддерживается только для изменяемых последовательностей.

В табл. 24.13 рассматриваются как неизменяемые, так и изменяемые типы множеств и отображений со следующими операциями:

- lookup проверка на наличие элемента в множестве или выбор значения, связанного с ключом;
- add добавление нового элемента в множество или новой пары «ключ — значение» в отображение;
- remove удаление элемента из множества или ключа из отображения;
- min наименьший элемент множества или наименьший ключ отображения.

24.12. Равенство

В библиотеках коллекций соблюдается единообразный подход к равенству и хешированию. В первую очередь идея заключается в разбиении коллекций на категории: множества, отображения и последовательности. Коллекции из разных категорий всегда не равны. Например, коллекция `Set(1, 2, 3)` не равна коллекции `List(1, 2, 3)` даже притом, что они содержат одни и те же элементы. В то же время внутри одной и той же категории коллекции равны лишь при условии, что содержат одинаковые элементы (для последовательностей — одинаковые элементы, в одинаковом порядке), например `List(1, 2, 3) == Vector(1, 2, 3)` и `HashSet(1, 2) == TreeSet(2, 1)`.

При проверке равенства неважно, является коллекция изменяемой или неизменяемой. Для изменяемых коллекций равенство просто зависит от содер-

жащихся в ней элементов на момент выполнения проверки на равенство. Это значит, что в разное время изменяемая коллекция может быть равна разным коллекциям в зависимости от того, какие элементы были добавлены или удалены. Данное обстоятельство может стать ловушкой в случае использования изменяемых коллекций в качестве ключа в хеш-отображении. Например:

```
import collection.mutable.{HashMap, ArrayBuffer}
val buf = ArrayBuffer(1, 2, 3)
val map = HashMap(buf -> 3) // Map((ArrayBuffer(1, 2, 3),3))
map(buf) // 3
buf(0) += 1
map(buf)
// java.util.NoSuchElementException: key not found:
//   ArrayBuffer(2, 2, 3)
```

В данном примере выбор, сделанный в последней строке, скорее всего, завершится сбоем, поскольку хеш-код массива `xs` в предпоследней строке изменился. Поэтому при поиске на основе хеш-кода будет отыскиваться другое место, отличное от того, в котором был сохранен `xs`.

24.13. Представления

Коллекции имеют всего несколько методов, которые конструируют новые коллекции. В качестве примеров можно привести `map`, `filter` и `++`. Такие методы мы называем *преобразователями*, поскольку они получают как минимум одну коллекцию в качестве объекта-получателя и создают в результате своей работы другую.

Преобразователи можно реализовать двумя основными способами — строгим и нестрогим (или ленивым). Строгий преобразователь создает новую коллекцию со всеми ее элементами. А нестрогий создает только заместитель получаемой в результате коллекции, а ее элементы конструируются по требованию.

В качестве примера нестрогого преобразователя рассмотрим следующую реализацию ленивой операции `map`:

```
def lazyMap[T, U](col: Iterable[T], f: T => U) =
  new Iterable[U]:
    def iterator = col.iterator.map(f)
```

Обратите внимание на то, что `lazyMap` создает новый объект типа `Iterable`, не прибегая к обходу всех элементов заданной коллекции `coll`. Вместо этого заданная функция `f` применяется к элементам итератора `iterator` новой коллекции по мере их востребованности.

По умолчанию коллекции в Scala — строгие во всех своих проявлениях, за исключением `lazyList`, в которой все методы преобразования реализованы лениво. Но существует систематический подход для превращения каждой коллекции в ленивую и *наоборот*, основанный на представлениях коллекций. *Представление* — это особая разновидность коллекции, которая изображает какую-либо основную коллекцию, но реализует все ее преобразователи лениво.

Для перехода от коллекции к ее представлению можно воспользоваться методом `view`. Если `xs` — некая коллекция, то `xs.view` создает точно такую же коллекцию, но с ленивой реализацией всех преобразователей. Перейти обратно от представления к строгой коллекции можно с помощью операции приведения с фабрикой строгих коллекций в качестве параметра.

В качестве примера предположим, что имеется вектор `Int`-значений, в отношении которого нужно последовательно применить две функции `map`:

```
val v = Vector((1 to 10)*)
// Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

v.map(_ + 1).map(_ * 2)
// Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

В последней инструкции выражение `v.map(_ + 1)` создает новый вектор, который второй вызов `map(_ * 2)` преобразует в третий вектор. Во многих ситуациях создание промежуточного результата из первого вызова `map` представляется неэкономным. В надуманном примере быстрее было бы воспользоваться одним вызовом `map` в сочетании с двумя функциями, `(_ + 1)` и `(_ * 2)`. При наличии двух функций, доступных в одном и том же месте, это можно сделать самостоятельно. Но довольно часто последовательные преобразования структур данных выполняются в различных программных модулях. Объединение этих преобразований может разрушить модульность. Более универсальный способ избавления от промежуточных результатов заключается в том, что сначала вектор превращается в представление, затем к представлению применяются все преобразования, после чего оно опять преобразуется в вектор:

```
(v.view.map(_ + 1).map(_ * 2)).toVector
// Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Мы вновь поочередно выполним эту последовательность операций:

```
scala> val vv = v.view
val vv: scala.collection.IndexedSeqView[Int] =
  IndexedSeqView(<not computed>)
```

Применение `v.view` выдает значение типа `IndexedSeqView`, то есть лениво вычисляемую `IndexedSeq`-последовательность. Как и в случае с `LazyList`, применение `toString` к представлениям не приводит к вычислению их элементов. Вот почему элементы `vv` выводятся на экран как `<not computed>`.

Применение первого метода `map` к представлению дает следующий результат:

```
scala> vv.map(_ + 1)
val res13: scala.collection.IndexedSeqView[Int] =
  IndexedSeqView(<not computed>)
```

Результат выполнения `map` — другое значение `IndexedSeqView[Int]`. По сути, это оболочка, которая *записывает* тот факт, что `map` с функцией `(_ + 1)` нужно применить к вектору `v`. Но этот метод `map` не применяется, пока не будет принудительно создано представление. Теперь применим к последнему результату второй метод `map`:

```
scala> res13.map(_ * 2)
val res14: scala.collection.IndexedSeqView[Int] =
  IndexedSeqView(<not computed>)
```

И наконец, принудительное получение последнего результата приводит к следующему диалогу:

```
scala> res14.toVector
val res15: Seq[Int] =
  Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Обе сохраненные функции, `(_ + 1)` и `(_ * 2)`, применяются как часть выполнения операции `to`, и создается новый вектор. При таком способе промежуточные структуры данных не нужны.

Операции преобразования, применяемые к представлению, не создают новую структуру данных. Вместо этого они возвращают объект `Iterable`, итератор которого является результатом применения операции преобразования к итератору исходной коллекции.

Потребность в использовании представлений обусловлена производительностью. Вы видели, что с помощью переключения коллекции на представление удалось избежать создания промежуточных результатов. Такая экономия может сыграть весьма важную роль. В качестве еще одного примера рассмотрим задачу поиска первого палиндрома в списке слов. Палиндромом называется слово, которое читается в обратном порядке точно так же, как и в прямом. Необходимые для этого определения имеют следующий вид:

```
def isPalindrome(x: String) = x == x.reverse
def findPalindrome(s: Iterable[String]) = s.find(isPalindrome)
```

Теперь предположим, что имеется весьма длинная последовательность слов и нужно найти палиндром в первом ее миллионе слов. Можно ли повторно воспользоваться определением `findPalindrome`? Разумеется, можно создать следующий код:

```
findPalindrome(words.take(1000000))
```

Он неплохо разделяет два аспекта, заключающихся в получении первого миллиона слов последовательности и поиска в них палиндрома. Но у этого решения есть недостаток: всегда будет создаваться промежуточная последовательность, состоящая из миллиона слов, даже если первое ее слово уже является палиндромом. Следовательно, потенциально получается, что 999 999 слов копируются в промежуточный результат, не подвергаясь последующей проверке. Многие программисты откажутся от этого и напишут собственную специализированную версию поиска палиндрома в некоем заданном префиксе последовательности аргументов. Но с представлениями этого делать не придется. Нужно просто воспользоваться следующим кодом:

```
findPalindrome(words.view.take(1000000))
```

Здесь также неплохо разрешен конфликт интересов, но вместо последовательности из миллиона элементов будет создан отдельный легковесный объект представления. Таким образом, вам не придется выбирать между производительностью и модульностью.

После того как вы увидели все эти интересные примеры использования представлений, у вас может возникнуть вопрос: а зачем вообще нужны строгие коллекции? Одна из причин состоит в том, что сравнение производительности не всегда бывает в пользу ленивых коллекций. Для коллекций меньших размеров добавленные издержки на формирование и применение замыканий в представлениях зачастую выше, чем выгоды, получаемые за счет того, что в них не применяются промежуточные структуры данных. Возможно, еще более существенной причиной является то, что вычисления в представлениях могут создавать серьезные помехи, если у отложенных операций есть побочные эффекты.

Вот пример, о который обожглись многие пользователи Scala версий до 2.8. В этих версиях тип `Range` был ленивым, поэтому вел себя, по сути, как представление. Программисты пробовали создавать такие вот акторы¹:

```
val actors = for i <- 1 to 10 yield actor { ??? }
```

¹ Библиотека акторов Scala устарела, но этот исторический прием все еще актуален.

А потом удивлялись, почему впоследствии ни один из акторов не выполнялся, даже если метод `actor` должен создавать и запускать актор из следующего за ним кода, заключенного в фигурные скобки. Чтобы понять, почему ничего не происходит, вспомним, что показанное ранее выражение `for` эквивалентно применению метода `map`, показанного ниже:

```
val actors = (1 to 10).map(i => actor { ??? })
```

Поскольку ранее диапазон, созданный выражением `(1 to 10)`, вел себя подобно представлению, результатом выполнения `map` опять было представление. То есть элементы не вычислялись, а следовательно, акторы не создавались! Они создавались бы с помощью принудительного вычисления всего диапазона, но далеко не очевидно, что это нужно для того, чтобы заставить акторов сделать их работу.

Чтобы избежать подобных сюрпризов, коллекции Scala в версии 2.8 получили более простые правила. Все коллекции, за исключением ленивых списков и представлений, являются строгими. Перейти от строгой коллекции к ленивой можно только через метод представления `view`. Единственный способ перейти обратно — применить метод `to`. Следовательно, акторы, определенные ранее, в Scala 2.8 поведут себя ожидаемым образом, то есть будут созданы и запущены десять акторов. Чтобы вернуться к прежнему парадоксальному поведению, придется добавить явный вызов метода `view`:

```
val actors = for i <- (1 to 10).view yield actor { ??? }
```

В целом представления — весьма эффективный инструмент, позволяющий увязать соображения эффективности с соображениями модульности. Но чтобы не путаться в тонкостях отложенных вычислений, применение представлений лучше ограничить чисто функциональным кодом, в котором у преобразований коллекций нет побочных эффектов. И лучше избегать смешивания представлений и операций, создающих новые коллекции, если у них к тому же есть побочные эффекты.

24.14. Итераторы

Итератор — это не коллекция, а, скорее, способ поочередного обращения к элементам коллекции. Двумя базовыми операциями итератора являются `next` и `hasNext`. Вызов `it.next()` вернет следующий элемент итератора и продвинет итератор дальше. Затем при повторном вызове `next` в отношении того же итератора будет выдан элемент, следующий за тем, который был возвращен ранее. Если возвращать станет нечего, то вызов `next`

приведет к генерации исключения `NoSuchElementException`. Определить, остались ли еще элементы в коллекции, можно с помощью метода `hasNext` класса `Iterator`.

Наиболее простой способ выполнить последовательный перебор всех элементов, возвращаемых итератором, — использовать цикл `while`:

```
while it.hasNext do
  println(it.next())
```

Итераторы в Scala также предоставляют аналоги большинства методов, имеющихся в трейтах `Iterable` и `Seq`. Например, они предоставляют метод `foreach`, который выполняет заданную процедуру в отношении каждого элемента, возвращенного итератором. При использовании `foreach` показанный ранее цикл можно сократить до следующего кода:

```
it.foreach(println)
```

Как и всегда, в качестве альтернативного синтаксиса для выражений, использующих `foreach`, `map`, `filter` и `flatMap`, можно воспользоваться выражением `for`. То есть еще одним способом вывести на экран все элементы, возвращенные итератором, мог бы быть следующий код:

```
for elem <- it do println(elem)
```

Между методом `foreach`, применяемым в отношении итераторов, и таким же методом, применяемым к коллекциям, допускающим обход элементов, есть существенная разница: при вызове в отношении итератора `foreach` оставит итератор в состоянии завершения его работы. Поэтому еще один вызов `next` в отношении того же самого итератора закончится неудачно и приведет к генерации исключения `NoSuchElementException`. Напротив, при вызове в отношении коллекции `foreach` оставляет количество элементов в коллекции без изменений, если только переданная функция не добавляет или не удаляет элементы, чего делать не рекомендуется, поскольку это легко может привести к неожиданным результатам.

Другие операции, общие для `Iterator` и `Iterable`, обладают таким же свойством оставлять итератор в состоянии завершения работы. Например, итераторы предоставляют метод `map`, возвращающий новый итератор:

```
val it = Iterator("a", "number", "of", "words")
val lit = it.map(_.length)
it.hasNext // true
lit.foreach(println) // prints 1, 6, 2, 5
it.hasNext // false
```

Как видите, после вызова `map` итератор `it` не перешел в конец, а вот итератор, полученный в результате вызова `it.map`, можно перебрать до конца.

Еще один пример — метод `dropWhile`, который может использоваться для поиска первого элемента итератора, имеющего определенное свойство. Например, для поиска в ранее показанном итераторе первого слова, в котором как минимум два символа, можно задействовать следующий код:

```
val it = Iterator("a", "number", "of", "words")
val dit = it.dropWhile(_.length < 2)
dit.next() // number
it.next()  // of
```

Еще раз обратите внимание на то, что `it` был изменен вызовом `dropWhile`: теперь `it` указывает на второе слово "number" в списке. Фактически `it` и результат `res`, полученный при выполнении `dropWhile`, вернут одинаковую последовательность элементов.

Есть только одна стандартная операция, `duplicate`, позволяющая повторно использовать один и тот же итератор:

```
val (it1, it2) = it.duplicate
```

Вызов метода `duplicate` возвращает два итератора, каждый из которых возвращает точно такие же элементы, что и итератор `it`. Оба итератора работают независимо друг от друга, продвижение одного из них совершенно не влияет на состояние другого. В отличие от этого исходный итератор `it` продвигается при вызове `duplicate` в самый конец и становится непригодным для дальнейшего использования.

В целом итераторы ведут себя как коллекции, *если вы никогда не обращаетесь к итератору еще раз после вызова метода на нем*. Библиотеки коллекций Scala делают это явным с помощью абстракции под названием `IterableOnce`, которая является обобщающим супертрейтом для `Iterable` и `Iterator`. Судя по названию, объекты типа `IterableOnce` допускают обход своих элементов хотя бы однократно, но состояние таких объектов после обхода не определено. Если объект типа `IterableOnce` фактически относится к типу `Iterator`, то после обхода будет указывать на конец коллекции, если же относится к типу `Iterable`, то не изменит своего состояния. Чаще всего `IterableOnce` используется как тип аргумента для методов, который могут получать в качестве аргумента `Iterator` или `Iterable`. Примером может послужить метод добавления `++` в трейте `Iterable`. Он получает параметр типа `IterableOnce`, позволяя добавлять элементы, получаемые как из коллекции типа `Iterator`, так и из `Iterable`.

Все операции над итераторами сведены в табл. 24.14.

Таблица 24.14. Операции в трейте `Iterator`

Что	Что делает
Абстрактные методы	
<code>it.next()</code>	Возвращает следующий элемент в итераторе и продвигается за него
<code>it.hasNext</code>	Возвращает <code>true</code> , если может вернуть еще элемент
Вариации	
<code>it.buffered</code>	Буферизованный итератор, возвращающий все элементы <code>it</code>
<code>it.grouped(size)</code>	Итератор, выдающий элементы, возвращаемые <code>it</code> , блоками фиксированного размера
<code>it.sliding(size)</code>	Итератор, выдающий элементы, возвращаемые <code>it</code> , в виде скользящего по коллекции окна фиксированного размера
Копирование	
<code>it.toArray(arr, s, l)</code>	Копирует не более одного элемента, возвращенного <code>it</code> , в массив <code>arr</code> , начиная с индекса <code>s</code> . Последние два аргумента являются необязательными
Дублирование	
<code>it.duplicate</code>	Пара итераторов, каждый из которых независимо от другого возвращает все элементы <code>it</code>
Добавления	
<code>it ++ jt</code>	Итератор, выдающий все элементы, возвращаемые итератором <code>it</code> , а затем все элементы, возвращаемые итератором <code>jt</code>
<code>it.padTo(len, x)</code>	Итератор, выдающий все элементы <code>it</code> , а затем копии <code>x</code> до тех пор, пока не будет полностью достигнута длина <code>len</code>
Отображения	
<code>it.map(f)</code>	Итератор, получаемый в результате применения функции к каждому элементу, возвращаемому <code>it</code>
<code>it.flatMap(f)</code>	Итератор, получаемый в результате применения возвращающей итератор функции <code>f</code> к каждому элементу <code>it</code> и добавления результатов

Что	Что делает
<code>it.collect(f)</code>	Итератор, получаемый в результате использования частично примененной функции <code>f</code> к каждому элементу <code>it</code> , для которого она определена, и сбора результатов
Преобразования	
<code>it.toArray</code>	Собирает возвращаемые <code>it</code> элементы в массив
<code>it.toList</code>	Собирает возвращаемые <code>it</code> элементы в список
<code>it.toIterable</code>	Собирает возвращаемые <code>it</code> элементы в коллекцию <code>Iterable</code>
<code>it.toSeq</code>	Собирает возвращаемые <code>it</code> элементы в последовательность
<code>it.toIndexedSeq</code>	Собирает возвращаемые <code>it</code> элементы в индексированную последовательность
<code>it.toSet</code>	Собирает возвращаемые <code>it</code> элементы в множество
<code>it.toMap</code>	Собирает возвращаемые <code>it</code> пары «ключ — значение» в отображение
<code>it to SortedSet</code>	Обобщенная операция преобразования, принимающая в качестве параметра фабрику коллекций
Информация о размере	
<code>it.isEmpty</code>	Проверяет, пуст ли итератор (противоположность <code>hasNext</code>)
<code>it.nonEmpty</code>	Проверяет, содержит ли итератор элементы (псевдоним метода <code>hasNext</code>)
<code>it.size</code>	Количество элементов, возвращенных <code>it</code> (после этой операции <code>it</code> окажется в самом конце!)
<code>it.length</code>	То же самое, что и <code>it.size</code>
<code>it.knownSize</code>	Количество элементов, если его можно узнать без изменения состояния итератора; в противном случае — 1
Индексированный поиск с извлечением элемента	
<code>it.find(p)</code>	<code>Option</code> -значение, содержащее первый возвращаемый <code>it</code> элемент, удовлетворяющий условию <code>p</code> , или <code>None</code> при отсутствии таких элементов (итератор продвигается за найденный элемент или, если не найден ни один элемент, оказывается в конечной позиции)
<code>it.indexOf(x)</code>	Индекс первого возвращенного <code>it</code> элемента, равного <code>x</code> (итератор продвигается за найденный элемент)

Таблица 24.14 (продолжение)

Что	Что делает
<code>it.indexWhere(p)</code>	Индекс первого возвращенного <code>it</code> элемента, удовлетворяющего условию <code>p</code> (итератор продвигается за найденный элемент)
Подытераторы	
<code>it.take(n)</code>	Итератор, выдающий первые <code>n</code> элементов итератора <code>it</code> (<code>it</code> продвинется за позицию <code>n</code> -го элемента или же окажется в конечной позиции, если в <code>it</code> содержится меньше <code>n</code> элементов)
<code>it.drop(n)</code>	Итератор, начинающий выборку с элемента <code>it</code> с позиции <code>(n + 1)</code> (<code>it</code> продвинется до этой же позиции)
<code>it.slice(m, n)</code>	Итератор, выдающий блок элементов, возвращенный из <code>it</code> , начинающийся с <code>m</code> -го элемента и заканчивающийся перед <code>n</code> -м элементом
<code>it.takeWhile(p)</code>	Итератор, выдающий элементы из <code>it</code> , пока условие <code>p</code> вычисляется в <code>true</code>
<code>it.dropWhile(p)</code>	Итератор, пропускающий элементы из <code>it</code> , пока условие <code>p</code> вычисляется в <code>true</code> , и выдающий остаток
<code>it.filter(p)</code>	Итератор, выдающий все элементы из <code>it</code> , удовлетворяющие условию <code>p</code>
<code>it.withFilter(p)</code>	То же самое, что и <code>it filter p</code> . Эта операция необходима для использования итераторов в выражениях <code>for</code>
<code>it.filterNot(p)</code>	Итератор, выдающий все элементы из <code>it</code> , не удовлетворяющие условию <code>p</code>
<code>it.distinct</code>	Итератор, выдающий все элементы из <code>it</code> , не включая дубликаты
Деление	
<code>it.partition(p)</code>	Разбивает <code>it</code> на два итератора, один из которых возвращает из <code>it</code> все элементы, удовлетворяющие условию <code>p</code> , а другой — все элементы, не удовлетворяющие этому условию
Состояния элементов	
<code>it.forall(p)</code>	Булево значение, показывающее, соблюдается ли условие <code>p</code> для всех элементов, возвращаемых <code>it</code>
<code>it.exists(p)</code>	Булево значение, показывающее, соблюдается ли условие <code>p</code> для какого-либо из элементов, возвращаемых <code>it</code>

Что	Что делает
<code>it.count(p)</code>	Количество элементов в <code>it</code> , удовлетворяющих условию <code>p</code>
Свертки	
<code>it.foldLeft(z)(op)</code>	Применяет бинарную операцию <code>op</code> к соседним элементам, возвращаемым <code>it</code> , проходя коллекцию слева направо и начиная с <code>z</code>
<code>it.foldRight(z)(op)</code>	Применяет бинарную операцию <code>op</code> к соседним элементам, возвращаемым <code>it</code> , проходя коллекцию справа налево и начиная с <code>z</code>
<code>it.reduceLeft(op)</code>	Применяет бинарную операцию <code>op</code> к соседним элементам, возвращаемым непустым итератором <code>it</code> , проходя коллекцию слева направо
<code>it.reduceRight(op)</code>	Применяет бинарную операцию <code>op</code> к соседним элементами, возвращаемым непустым итератором <code>it</code> , проходя коллекцию справа налево
Специализированные свертки	
<code>it.sum</code>	Сумма значений числовых элементов, выдаваемых итератором <code>it</code>
<code>it.product</code>	Произведение значений числовых элементов, выдаваемых итератором <code>it</code>
<code>it.min</code>	Минимальное значение упорядочиваемых элементов, выдаваемых итератором <code>it</code>
<code>it.max</code>	Максимальное значение упорядочиваемых элементов, выдаваемых итератором <code>it</code>
Слияния	
<code>it.zip jt</code>	Итератор пар соответствующих элементов, выдаваемых итераторами <code>it</code> и <code>jt</code>
<code>it.zipAll(jt, x, y)</code>	Итератор пар соответствующих элементов, выдаваемых итераторами <code>it</code> и <code>jt</code> , причем тот итератор, что короче, расширяется, чтобы соответствовать более длинному итератору путем добавления элементов <code>x</code> или <code>y</code>
<code>it.zipWithIndex</code>	Итератор пар, состоящих из элементов, возвращаемых <code>it</code> , и их индексов
Обновление	
<code>it.patch(i, jt, r)</code>	Итератор, получаемый из <code>it</code> путем замены <code>r</code> элементов, начиная с позиции <code>i</code> , итератором вставки <code>jt</code>

Таблица 24.14 (окончание)

Что	Что делает
Сравнение	
<code>it.sameElements(jt)</code>	Проверяет, не возвращают ли итераторы <code>it</code> и <code>jt</code> одни и те же элементы в одном и том же порядке. Обратите внимание: после этой операции следует отбросить как <code>it</code> , так и <code>jt</code>
Строки	
<code>it.addString(b, start, sep, end)</code>	Добавляет строку к <code>StringBuilder b</code> , которая показывает все элементы, возвращаемые <code>it</code> , между разделителями <code>sep</code> и заключена в строковые значения <code>start</code> и <code>end</code> . Аргументы <code>start</code> , <code>sep</code> и <code>end</code> являются необязательными
<code>it.mkString(start, sep, end)</code>	Преобразует итератор в строку, показывающую все элементы, возвращаемые <code>it</code> , между разделителями <code>sep</code> , и заключенную в строковые значения <code>start</code> и <code>end</code> . Аргументы <code>start</code> , <code>sep</code> и <code>end</code> являются необязательными

Буферизованные итераторы

Порой бывает необходим итератор, способный заглянуть вперед, позволяя проинспектировать следующий возвращаемый элемент, не продвигаясь за него. Рассмотрим, к примеру, задачу пропуска начальных пустых строк итератора, который возвращает последовательность строк. Возможно, возникнет соблазн создать нечто похожее на метод, показанный ниже:

```
// Этот код работать не будет
def skipEmptyWordsNOT(it: Iterator[String]) =
  while it.next().isEmpty do {}
```

Но, присмотревшись, можно понять, что этот код неработоспособен: разумеется, он будет пропускать начальные пустые строки, но также продвинет `it` за первую непустую строку!

Решить эту задачу можно с помощью буферизованного итератора, экземпляра трейта `BufferedIterator`, который является подтрейтом трейта `Iterator` и предоставляет еще один метод по имени `head`. Вызов `head` в отношении буферизованного итератора приведет к возвращению его первого элемента, но без продвижения итератора. При использовании буферизованного итератора код для пропуска пустых слов может выглядеть следующим образом:

```
def skipEmptyWords(it: BufferedIterator[String]) =
  while it.head.isEmpty do it.next()
```

Каждый итератор может быть преобразован в буферизованный итератор путем вызова своего метода `buffered`. Пример его использования выглядит так:

```
val it = Iterator(1, 2, 3, 4)
val bit = it.buffered
bit.head // 1
bit.next() // 1
bit.next() // 2
```

Следует заметить, что вызов `head` в отношении буферизованного итератора `bit` не приводит к изменению позиции итератора. Поэтому последующий вызов, `bit.next()`, возвращает то же самое значение, что и `bit.head`.

24.15. Создание коллекций с нуля

Вам уже попался синтаксис наподобие `List(1, 2, 3)`, который создает список из трех целых чисел, и `Map('A' -> 1, 'C' -> 2)`, который создает отображение с двумя привязками. Фактически это универсальная возможность коллекций Scala. Можно взять любое имя коллекции и указать после него в круглых скобках список элементов. В результате получится новая коллекция с заданными элементами. Ниже представлены еще примеры:

```
Iterable()           // пустая коллекция
List()              // пустой список
List(1.0, 2.0)      // список с элементами 1.0, 2.0
Vector(1.0, 2.0)    // вектор с элементами 1.0, 2.0
Iterator(1, 2, 3)   // итератор, возвращающий три целых числа
Set(dog, cat, bird) // множество из трех животных
HashSet(dog, cat, bird) // хеш-множество из тех же животных
Map('a' -> 7, 'b' -> 0) // отображение символов на целые числа
```

«Скрытно» каждая из показанных ранее строк является вызовом метода `apply` определенного объекта. Например, третья из этих строк раскрывается в следующий код:

```
List.apply(1.0, 2.0)
```

Здесь показан вызов метода `apply`, принадлежащего объекту-компаньону класса `List`. Этот метод получает произвольное число аргументов и создает из них список. Каждый класс коллекций в библиотеке Scala располагает

объектом-компаньоном с таким же методом `apply`. И неважно, представлена конкретная реализация классом коллекции, как в случае с `List`, `LazyList` или `Vector`, или же трейтом, как в случае с `Seq`, `Set` или `Iterable`. В последнем случае вызов `apply` приведет к созданию некой исходной реализации трейта. Рассмотрим ряд примеров:

```
scala> List(1, 2, 3)
val res17: List[Int] = List(1, 2, 3)

scala> Iterable(1, 2, 3)
val res18: Iterable[Int] = List(1, 2, 3)

scala> mutable.Iterable(1, 2, 3)
val res19: scala.collection.mutable.Iterable[Int] =
  ArrayBuffer(1, 2, 3)
```

Помимо `apply`, в каждом объекте-компаньоне определен и элемент `empty`, возвращающий пустую коллекцию. Поэтому вместо `List()` можно воспользоваться кодом `List.empty`, вместо `Map()` задействовать код `Map.empty` и т. д.

Кроме того, потомки трейтов `Seq` и `Set` в своих объектах-компаньонах предоставляют другие факторные операции, которые сведены в табл. 24.15. Вкратце это:

- `concat`, которая конкатенирует произвольное количество коллекций;
- `fill` и `tabulate`, которые создают одно- или многомерные коллекции заданной размерности, инициализированные каким-либо выражением или функцией составления таблицы;
- `range`, которая создает коллекции целых чисел с какой-либо постоянной длиной шага;
- `iterate` и `unfold`, которые создают последовательность, получающуюся из многократного применения функции к начальному элементу или состоянию.

Таблица 24.15. Фабричные методы для трейтов `Seq` и `Set`

Что	Что делает
<code>C.empty</code>	Пустая коллекция
<code>C(x, y, z)</code>	Коллекция, состоящая из элементов <code>x</code> , <code>y</code> и <code>z</code>
<code>C.concat(xs, ys, zs)</code>	Коллекция, получаемая конкатенацией элементов коллекций <code>xs</code> , <code>ys</code> и <code>zs</code>

Что	Что делает
<code>C.fill(n)(e)</code>	Коллекция длиной n , где каждый элемент вычисляется выражением e
<code>C.fill(m, n)(e)</code>	Коллекция коллекций размерностью $m \times n$, где каждый элемент вычисляется выражением e (существует также в более высоких измерениях)
<code>C.tabulate(n)(f)</code>	Коллекция длиной n , где элемент по каждому индексу i вычисляется путем вызова $f(i)$
<code>C.tabulate(m, n)(f)</code>	Коллекция коллекций размерностью $m \times n$, где элемент по каждому индексу (i, j) вычисляется путем вызова $f(i, j)$ (существует также в более высоких измерениях)
<code>C.range(start, end)</code>	Коллекция целых чисел $start \dots end - 1$
<code>C.range(start, end, step)</code>	Коллекция целых чисел, начинающаяся со $start$ и наращиваемая с шагом $step$ до значения end , исключая само это значение
<code>C.iterate(x, n)(f)</code>	Коллекция длиной n с элементами $x, f(x), f(f(x)), \dots$
<code>C.unfold(init)(f)</code>	Коллекция, которая использует функцию f для вычисления своего следующего элемента и состояния, начиная с состояния $init$

24.16. Преобразования между коллекциями Java и Scala

Как и в Scala, в Java есть богатая библиотека коллекций. Обе библиотеки имеют много общего. Например, и в той и в другой есть такие категории, как итераторы, итерируемые коллекции, множества, отображения и последовательности. Но есть и важные различия. В частности, в библиотеках Scala уделяется намного больше внимания неизменяемым коллекциям и предоставляется куда больше операций, выполняющих преобразование коллекции в новую коллекцию.

Иногда может понадобиться выполнить преобразование из одной среды в другую. Например, нужно обратиться к уже существующей Java-коллекции, как если бы это была Scala-коллекция. Или же следует передать одну из коллекций Scala методу Java, который ожидает получения Java-аналога. Сделать это не составит никакого труда, так как Scala предлагает в объекте

`CollectionConverters` удобные преобразования между всеми основными типами коллекций. В частности, двунаправленные преобразования имеются между следующими типами:

<code>Iterator</code>	↔	<code>java.util.Iterator</code>
<code>Iterator</code>	↔	<code>java.util.Enumeration</code>
<code>Iterable</code>	↔	<code>java.lang.Iterable</code>
<code>Iterable</code>	↔	<code>java.util.Collection</code>
<code>mutable.Buffer</code>	↔	<code>java.util.List</code>
<code>mutable.Set</code>	↔	<code>java.util.Set</code>
<code>mutable.Map</code>	↔	<code>java.util.Map</code>

Чтобы включить эти преобразования, нужно просто импортировать их:

```
scala> import jdk.CollectionConverters.*
```

Это делает возможным преобразования между соответствующими коллекциями Scala и Java с помощью методов расширения `asScala` и `asJava`:

```
scala> import collection.mutable.*
```

```
scala> val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3).asJava
val jul: java.util.List[Int] = [1, 2, 3]
```

```
scala> val buf: Seq[Int] = jul.asScala
val buf: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)
```

```
scala> val m: java.util.Map[String, Int] =
      HashMap("abc" -> 1, "hello" -> 2).asJava
m: java.util.Map[String,Int] = {hello=2, abc=1}
```

Внутренний механизм этих преобразований работает за счет создания объекта-обертки, пересылающего все операции базовому объекту коллекции. Поэтому коллекции при преобразовании между Java и Scala никогда не копируются. Интересной особенностью является то, что при круговом преобразовании из, скажем, Java-типа в соответствующий Scala-тип и обратно, в тот же Java-тип, будет получен точно такой же объект коллекции, который имелся в самом начале.

Есть также ряд других востребованных Scala-коллекций, которые могут быть преобразованы в Java-типы, но для которых нет соответствующих преобразований в обратном направлении. К ним относятся:

<code>Seq</code>	⇒	<code>java.util.List</code>
<code>mutable.Seq</code>	⇒	<code>java.util.List</code>
<code>Set</code>	⇒	<code>java.util.Set</code>
<code>Map</code>	⇒	<code>java.util.Map</code>

Поскольку в Java изменяемые и неизменяемые коллекции по типам не различаются, преобразование из, скажем, `collection.immutable.List` выдаст коллекцию `java.util.List`. При всех попытках применить операции по внесению изменений в отношении этой коллекции будет генерироваться исключение `UnsupportedOperationException`, например:

```
scala> val jul: java.util.List[Int] = List(1, 2, 3)
val jul: java.util.List[Int] = [1, 2, 3]

scala> jul.add(7)
java.lang.UnsupportedOperationException
    at java.util.AbstractList.add(AbstractList.java:131)
```

Резюме

Теперь вы получили более детальное представление об использовании коллекций Scala. В них применен подход, предоставляющий вам целый ряд не просто полезных специализированных методов, но по-настоящему эффективных строительных блоков. Сочетание двух или трех таких строительных блоков позволит провести множество полезных вычислений. Эффективность стиля, принятого в библиотеке, наиболее ярко проявляется благодаря имеющемуся в Scala облегченному синтаксису для функциональных литералов, а также благодаря тому, что сам язык предоставляет множество типов коллекций, сохраняющих постоянство и неизменяемость. В следующей, заключительной главе мы рассмотрим утверждения и модульное тестирование.

25

Утверждения и тесты

Утверждения и тесты — два важных способа проверки правильности поведения созданных вами программных средств. В данной главе мы покажем несколько вариантов для их создания и запуска, доступных в Scala.

25.1. Утверждения

Утверждения в Scala создаются в виде вызовов predefined метода `assert`¹. Выражение `assert(condition)` выдает ошибку `AssertionError`, если *условие* `condition` не соблюдается. Существует также версия `assert`, использующая два аргумента: выражение `assert(condition, explanation)` тестирует *условие*. При его несоблюдении оно выдает ошибку `AssertionError`, в сообщении о которой содержится заданное *объяснение* `explanation`. Тип объяснения — `Any`, поэтому в качестве объяснения может передаваться любой объект. Чтобы поместить в `AssertionError` строковое объяснение, метод `assert` будет вызывать в отношении этого объекта метод `toString`. Например, в метод по имени `above`, класса `Element` и показанный в листинге 10.13, `assert` можно поместить после вызовов `widen`, чтобы убедиться в одинаковой ширине расширенных элементов. Этот вариант показан в листинге 25.1.

Листинг 25.1. Использование утверждения

```
def above(that: Element): Element =  
  val this1 = this widen that.width
```

¹ Метод `assert` определен в объекте-одиночке `Predef`, элементы которого автоматически импортируются в каждый исходный файл программы на языке Scala.

```
val that1 = that widen this.width
assert(this1.width == that1.width)
elem(this1.contents ++ that1.contents)
```

Выполнить ту же задачу можно и по-другому: проверить ширину в конце метода `widen`, непосредственно перед возвращением значения. Этого можно добиться, сохранив результат в `val`-переменной, выполняя утверждение применительно к результату с последующим указанием `val`-переменной, чтобы результат возвращался в том случае, если утверждение было успешно подтверждено. Но, как показано в листинге 25.2, это можно сделать более выразительно, воспользовавшись довольно удобным методом `ensuring` из объекта-одиночки `Predef`.

Листинг 25.2. Использование метода `ensuring` для утверждения результата выполнения функции

```
private def widen(w: Int): Element =
  if w <= width then
    this
  else {
    val left = elem(' ', (w - width) / 2, height)
    var right = elem(' ', w - width - left.width, height)
    left beside this beside right
  } ensuring (w <= _.width)
```

Благодаря неявному преобразованию метод `ensuring` может использоваться с любым результирующим типом. В данном коде все выглядит так, словно `ensuring` вызван в отношении результата выполнения метода `widen`, имеющего тип `Element`. Однако на самом деле `ensuring` вызван в отношении типа, в который `Element` был автоматически преобразован. Метод `ensuring` получает один аргумент, функцию-предикат, которая берет результирующий тип, возвращает булево значение и передает результат предикату. Если предикат возвращает `true`, то метод `ensuring` возвращает результат, в противном случае метод выдаст ошибку `AssertionError`.

В данном примере предикат имеет вид `w <= _.width`. Знак подчеркивания является заместителем для одного аргумента, который передается предикату, а именно результата типа `Element`, получаемого от метода `widen`. Если ширина, переданная в виде `w` методу `widen`, меньше ширины результата типа `Element` или равна ей, то предикат будет вычислен в `true` и результатом `ensuring` будет объект типа `Element`, в отношении которого он был вызван. Данное выражение в методе `widen` последнее, поэтому его результат типа `Element` и будет значением, возвращаемым самим методом `widen`.

Утверждения могут включаться и выключаться с помощью флагов командной строки JVM `-ea` и `-da`. Когда флаги включены, каждое утверждение

служит небольшим тестом, использующим фактические данные, вычисленные при выполнении программы. Далее мы сфокусируемся на написании внешних тестов, которые предоставляют собственные тестовые данные и выполняются независимо от приложения.

25.2. Тестирование в Scala

Scala содержит много опций для тестирования, начиная с хорошо известного инструментария Java, такого как JUnit и TestNG, и заканчивая инструментарием, написанным на Scala, например ScalaTest, specs2 и ScalaCheck. В оставшейся части главы мы дадим краткий обзор этого инструментария. Начнем со ScalaTest.

ScalaTest — наиболее гибкая среда тестирования в Scala, это средство можно легко настроить на решение различных задач. Гибкость ScalaTest означает, что команды могут использовать тот стиль тестирования, который более полно отвечает их потребностям. Например, для команд, знакомых с JUnit, наиболее удобным станет стиль AnyFunSuite. Пример показан в листинге 25.3.

Листинг 25.3. Написание тестов с использованием AnyFunSuite

```
import org.scalatest.funsuite.AnyFunSuite
import Element.elem

class ElementSuite extends AnyFunSuite:

  test("elem result should have passed width") {
    val ele = elem('x', 2, 3)
    assert(ele.width == 2)
  }
```

Центральная концепция в ScalaTest — *набор*, то есть коллекция, тестов. *Тестом* может являться любой код с именем, который может быть запущен и завершен успешно или неуспешно, отложен или отменен. Центральный блок композиции в ScalaTest — трейт *Suite*. В нем объявляются методы жизненного цикла, определяющие исходный способ запуска тестов, который можно переопределить под нужные способы написания и запуска тестов.

В ScalaTest предлагаются *стилевые трейты*, расширяющие *Suite* и переопределяющие методы жизненного цикла для поддержания различных стилей тестирования. В этой среде также предоставляются *примешиваемые трейты*, которые переопределяют методы жизненного цикла таким образом,

чтобы те отвечали конкретным потребностям тестирования. Классы тестов определяются сочетанием стиля `Suite` и примешиваемых трейтов, а тестовые наборы — путем составления экземпляров `Suite`.

Пример стиля тестирования — `AnyFunSuite`, расширенный тестовым классом, показанным в листинге 25.3. Слово `Fun` в `AnyFunSuite` означает функцию, а `test` является определенным в `AnyFunSuite` методом, который вызывается первичным конструктором `ElementSuite`. Вы указываете название теста в виде строки в круглых скобках, а сам код теста помещаете в фигурные скобки. Код теста является функцией, передаваемой методу `test` в виде параметра по имени, которая регистрирует его для последующего выполнения.

Среда `ScalaTest` интегрирована в широко используемые инструментальные средства сборки, такие как `sbt` и `Maven`, и в IDE, например `IntelliJIDEA` и `Eclipse`. `Suite` можно запустить и непосредственно через приложение `ScalaTest Runner` или из интерпретатора `Scala`, просто вызвав в отношении его метод `execute`. Соответствующий пример выглядит так:

```
scala> (new ElementSuite).execute()  
ElementSuite:  
- elem result should have passed width
```

Все стили `ScalaTest`, включая `AnyFunSuite`, предназначены для стимуляции написания специализированных тестов с осмысленными названиями. Кроме того, все стили создают вывод, похожий на спецификацию, которая может облегчить общение между заинтересованными сторонами. Выбранным вами стилем предписывается только то, как будут выглядеть объявления ваших тестов. Все остальное в `ScalaTest` работает одинаково, независимо от выбранного стиля¹.

25.3. Информативные отчеты об ошибках

В тесте, показанном в листинге 25.3, предпринимается попытка создать элемент шириной, равной 2, и высказывается утверждение, что ширина получившегося элемента действительно равна 2. Если утверждение не подтвердится, то отчет об ошибке будет включать имя файла и номер строки с неоправдавшимся утверждением, а также информативное сообщение об ошибке:

¹ Более подробную информацию о `ScalaTest` можно найти на сайте www.scalatest.org.

```
scala> val width = 3
width: Int = 3
```

```
scala> assert(width == 2)
org.scalatest.exceptions.TestFailedException:
  3 did not equal 2
```

Чтобы обеспечить содержательные сообщения об ошибках при неверных утверждениях, ScalaTest в ходе компиляции анализирует выражения, переданные каждому вызову утверждения. Если вы хотите увидеть более подробную информацию о неверных утверждениях, то можете воспользоваться имеющимся в ScalaTest средством Diagrams, сообщения об ошибках которого показывают схему выражения, переданного утверждению `assert`:

```
scala> assert(List(1, 2, 3).contains(4))
org.scalatest.exceptions.TestFailedException:
```

```
  assert(List(1, 2, 3).contains(4))
  |   |   |   |   |   |   |
  |   1  2  3 false  4
  List(1, 2, 3)
```

Имеющиеся в ScalaTest методы `assert` не делают разницы в сообщениях об ошибках между фактическим и ожидаемым результатами. Они просто показывают, что левый операнд не равен правому, или показывают значения на схеме. Если нужно подчеркнуть различия между фактическим и ожидаемым результатами, то можно воспользоваться имеющимся в ScalaTest альтернативным методом `assertResult`:

```
assertResult(2) {
  ele.width
}
```

С помощью данного выражения показывается, что от кода в фигурных скобках ожидается результат 2. Если в результате выполнения этого кода получится 3, то в отчете об ошибке тестирования будет показано сообщение `Expected 2, but got 3` (Ожидалось 2, но получено 3).

При необходимости проверить, выдает ли метод ожидаемое исключение, можно воспользоваться имеющимся в ScalaTest методом `assertThrows`:

```
assertThrows[IllegalArgumentException] {
  elem('x', -2, 3)
}
```

Если код в фигурных скобках выдает не то исключение, которое ожидалось, или вообще не выдает его, то метод `assertThrows` тут же завершит свою

работу и выдаст исключение `TestFailedException`. А отчет об ошибке будет содержать сообщение с полезной для вас информацией:

```
Expected IllegalArgumentException to be thrown,  
but NegativeArraySizeException was thrown.
```

Но если код завершится внезапной генерацией экземпляра переданного класса исключения, то метод `assertThrows` выполнится нормально. При необходимости провести дальнейшее исследование ожидаемого исключения можно вместо `assertThrows` воспользоваться методом перехвата `intercept`. Он работает аналогично методу `asassertThrows`, за исключением того, что при генерации ожидаемого исключения `intercept` возвращает это исключение:

```
val caught =  
    intercept[ArithmeticException] {  
        1 / 0  
    }  
  
assert(caught.getMessage == "/ by zero")
```

Короче говоря, имеющиеся в `ScalaTest` утверждения стараются выдать полезные сообщения об ошибках, способные помочь вам диагностировать и устранить проблемы в коде.

25.4. Использование тестов в качестве спецификаций

В стиле тестирования при *разработке через реализацию поведения* (behavior-driven development, BDD) основной упор делается на написание легко воспринимаемых человеком спецификаций расширенного поведения кода и сопутствующих тестов, которые проверяют, свойственно ли коду такое поведение. В `ScalaTest` включены несколько трейтов, содействующих этому стилю тестирования. Пример использования одного такого трейта, `AnyFlatSpec`, показан в листинге 25.4.

Листинг 25.4. Спецификация и тестирование поведения с помощью `AnyFlatSpec`

```
import org.scalatest.flatspec.AnyFlatSpec  
import org.scalatest.matchers.should.Matchers  
import Element.elem  
  
class ElementSpec extends AnyFlatSpec, Matchers:
```

```

"A UniformElement" should
  "have a width equal to the passed value" in {
    val ele = elem('x', 2, 3)
    ele.width should be (2)
  }

it should "have a height equal to the passed value" in {
  val ele = elem('x', 2, 3)
  ele.height should be (3)
}

it should "throw an IAE if passed a negative width" in {
  an [IllegalArgumentException] should be thrownBy {
    elem('x', -2, 3)
  }
}

```

При использовании `AnyFlatSpec` тесты создаются в виде *директив спецификации*. Сначала в виде строки пишется название тестируемого *субъекта* ("A UniformElement" в листинге 25.4), затем `should`, или `must`, или `can` (что означает «обязан», или «должен», или «может» соответственно), потом строка, обозначающая характер поведения, требуемого от субъекта, а затем ключевое слово `in`. В фигурных скобках, стоящих после `in`, пишется код, тестирующий указанное поведение. В последующих директивах, чтобы сослаться на самый последний субъект, можно написать `it`. При выполнении `AnyFlatSpec` этот трейт будет запускать каждую директиву спецификации в виде теста `ScalaTest`. `AnyFlatSpec` (и другие специфицирующие трейты, которые есть в `ScalaTest`) генерирует вывод, который при запуске читается как спецификация. Например, вот на что будет похож вывод, если запустить `ElementSpec` из листинга 25.4 в интерпретаторе:

```

scala> (new ElementSpec).execute()
A UniformElement
- should have a width equal to the passed value
- should have a height equal to the passed value
- should throw an IAE if passed a negative width

```

В листинге 25.4 также показан имеющийся в `ScalaTest` предметно-ориентированный язык (domain-specific language, DSL) *выявления соответствий*. Примешиванием трейта `Matchers` можно создавать утверждения, которые при чтении больше похожи на естественный язык. Имеющийся в `ScalaTest` DSL-язык предоставляет множество средств выявления соответствий, кроме этого, позволяет определять новые предоставленные пользователем средства выявления соответствий, содержащие сообщения об ошибках. Такие средства, показанные в листинге 25.4, включают синтаксис `should be` и `an [...]` `should be thrownBy: ...`. Как вариант, если предпочтение отдается глаголу

must, а не should, то можно примешать MustMatchers. Например, примешивание MustMatchers позволит вам создавать следующие выражения:

```
result must be >= 0
map must contain key 'c'
```

Если последнее утверждение не подтвердится, то будет показано сообщение об ошибке следующего вида:

```
Map('a' -> 1, 'b' -> 2) did not contain key 'c'
```

Среда тестирования specs2 — средство с открытым кодом, написанное на Scala Эриком Торреборре (Eric Torreborre), — также поддерживает BDD-стиль тестирования, но с другим синтаксисом. Например, specs2 можно использовать для создания теста, показанного в листинге 25.5.

Листинг 25.5. Спецификация и тестирование поведения с использованием среды specs2

```
import org.specs2.*
import Element.elem

object ElementSpecification extends Specification:
  "A UniformElement" should {
    "have a width equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.width must be_==(2)
    }
    "have a height equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.height must be_==(3)
    }
    "throw an IAE if passed a negative width" in {
      elem('x', -2, 3) must
        throwA[IllegalArgumentException]
    }
  }
}
```

В specs2, как и в ScalaTest, существует DSL-язык выявления соответствий. Некоторые примеры работы средств выявления соответствий, имеющихся в specs2, показаны в листинге 25.5 в строках, содержащих `must be_==` и `must throwA`¹. Среда specs2 можно использовать в автономном режиме, но она также интегрируется со ScalaTest и JUnit, поэтому specs2-тесты можно запускать и с этими инструментами.

¹ Программное средство specs2 можно загрузить с сайта specs2.org.

Одной из основных идей BDD является то, что тесты могут помогать обмениваться мнениями людям, принимающим решения о характере поведения программных средств, людям, разрабатывающим программные средства, и людям, определяющим степень завершенности и работоспособности программных средств. В этом ключе могут применяться любые стили, имеющиеся в ScalaTest или specs2, однако в ScalaTest есть специально разработанный для этого стиль AnyFeatureSpec. Пример его использования показан в листинге 25.6.

Листинг 25.6. Использование тестов для содействия обмену мнениями среди всех заинтересованных сторон

```
import org.scalatest.*
import org.scalatest.featurespec.AnyFeatureSpec

class TVSetSpec extends AnyFeatureSpec, GivenWhenThen:

  Feature("TV power button") {
    Scenario("User presses power button when TV is off") {
      Given("a TV set that is switched off")
      When("the power button is pressed")
      Then("the TV should switch on")
      pending
    }
  }
}
```

Стиль AnyFeatureSpec разработан для того, чтобы направить в нужное русло обсуждений предназначение программных средств: вам следует выявить специфические *требования*, а затем дать им точное определение в виде *скриптов*. Сосредоточиться на переговорах об особенностях отдельно взятых скриптов помогают методы Given, When и Then, предоставляемые трейтом GivenWhenThen. Вызов pending в самом конце показывает: и тест, и само поведение не реализованы, имеется лишь спецификация. Как только будут реализованы все тесты и конкретные действия, тесты будут пройдены и требования можно будет посчитать выполненными.

25.5. Тестирование на основе свойств

Еще одним полезным средством тестирования для Scala является ScalaCheck — среда с открытым кодом, созданная Рикардом Нильсоном (Rickard Nilsson). Она позволяет указывать свойства, которыми должен обладать тестируемый код. Для каждого свойства ScalaCheck создает данные и выдает утверждения, проверяющие наличие тех или иных свойств. Пример использования ScalaCheck из ScalaTest AnyWordSpec, в который примешан трейт ScalaCheckPropertyChecks, показан в листинге 25.7.

Листинг 25.7. Тестирование на основе свойств с помощью ScalaCheck

```
import org.scalatest.wordspec.AnyWordSpec
import org.scalatestplus.scalacheck.ScalaCheckPropertyChecks
import org.scalatest.matchers.must.Matchers.*
import Element.elem

class ElementSpec extends AnyWordSpec,
  ScalaCheckPropertyChecks:
  "elem result" must {
    "have passed width" in {
      forAll { (w: Int) =>
        whenever (w > 0) {
          elem('x', w % 100, 3).width must equal (w % 100)
        }
      }
    }
  }
}
```

`AnyWordSpec` — класс стиля, имеющийся в `ScalaTest`. Трейт `ScalaCheckPropertyChecks` предоставляет несколько методов `forAll`, позволяющих смешивать тесты на основе проверки наличия свойств с традиционными тестами на основе утверждений или выявления соответствий. В данном примере проверяется наличие свойства, которым должен обладать фабричный метод `elem`. Свойства `ScalaCheck` выражены в виде значений функций, получающих в качестве параметров данные, необходимые для утверждений о наличии свойств. Эти данные будут генерироваться `ScalaCheck`. В свойстве, показанном в листинге 25.7, данными является целое число `w`, которое представляет ширину. Внутри тела функции показан следующий код:

```
whenever (w > 0) {
  elem('x', w % 100, 3).width must equal (w % 100)
}
```

Директива `whenever` указывает на то, что при каждом вычислении левостороннего выражения в `true` правостороннее также должно быть `true`. Таким образом, в данном случае выражение в блоке должно быть `true` всякий раз, когда `w` больше нуля. А правостороннее будет выдавать `true`, если ширина, переданная фабричному методу `elem`, будет равна ширине объекта `Element`, возвращенного фабричным методом.

При таком небольшом объеме кода `ScalaCheck` сгенерирует несколько пробных значений, проверяя каждое из них в поисках значения, для которого свойство не соблюдается. Если свойство соблюдается для каждого значения, испытанного с помощью `ScalaCheck`, то тест будет пройден. В противном случае он будет завершен с генерацией исключения

`TestFailedException`, которое содержит информацию, включающую значение, вызвавшее сбой.

25.6. Подготовка и проведение тестов

Во всех средах, упомянутых в текущей главе, имеется механизм для подготовки и проведения тестов. В данном разделе будет дан краткий обзор того подхода, который используется в `ScalaTest`. Чтобы получить подробное описание любой из этих сред, нужно обратиться к их документации.

Подготовка больших наборов тестов в `ScalaTest` проводится путем вложения `Suite`-наборов в `Suite`-наборы. При выполнении `Suite`-набор запустит не только свои тесты, но и все тесты вложенных в него `Suite`-наборов. Вложенные `Suite`-наборы, в свою очередь, выполняют тесты вложенных в них `Suite`-наборов и т. д. Таким образом, большой набор тестов будет представлен деревом `Suite`-объектов. При выполнении в этом дереве корневого `Suite`-объекта будут выполнены все имеющиеся в нем `Suite`-объекты.

Наборы можно вкладывать вручную или автоматически. Чтобы вложить их вручную, вам нужно либо переопределить метод `nestedSuites` в своих `Suite`-классах, либо передать предназначенные для вложения `Suite`-объекты в конструктор класса `Suites`, который для этих целей предоставляет `ScalaTest`. Для автоматического вложения имена пакетов передаются имеющемуся в `ScalaTest` средству `Runner`, которое определит `Suite`-объекты автоматически, вложит их ниже корневого `Suite` и выполнит корневой `Suite`.

Имеющееся в `ScalaTest` приложение `Runner` можно вызвать из командной строки или через такие средства сборки, как `sbt` или `maven`. Самый распространенный способ запуска `ScalaTest`, скорее всего, с помощью `sbt`¹. Например, для запуска тестового класса, показанного в листинге 25.6, с помощью `sbt` создайте новый каталог, поместите тестовый класс в файл с именем `TVSetSpec.scala` в подкаталог `src/test/scala` и добавьте следующий файл `build.sbt` в новый каталог:

```
name := "ThankYouReader!"

scalaVersion := "3.0.0"

libraryDependencies += "org.scalatest" %% "scalatest" %
    "3.2.9" % "test"
```

¹ Установить `sbt` вы можете с сайта <https://www.scala-sbt.org/>.

Затем вы можете войти в оболочку `sbt`, набрав `sbt`:

```
$ sbt
[info] welcome to sbt 1.5.2 (AdoptOpenJDK Java 1.8.0_262)
...
sbt:ThankYouReader!>
```

Вам будет выдан запрос на ввод названия проекта, в данном случае `ThankYouReader!` Если вы введете `test`, он скомпилирует и запустит тестовый класс. Результат показан на рис. 25.1.

```
sbt:ThankYouReader!> test
[info] TVSetSpec:
[info] Feature: TV power button
[info]   Scenario: User presses power button when TV is off (pending)
[info]     Given a TV set that is switched off
[info]     When the power button is pressed
[info]     Then the TV should switch on
[info] Run completed in 297 milliseconds.
[info] Total number of tests run: 0
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 0, canceled 0, ignored 0, pending 1
[info] No tests were executed.
[success] Total time: 2 s, completed May 29, 2021 7:11:00 PM
sbt:ThankYouReader!>
```

Рис. 25.1. Вывод, полученный при запуске `org.scalatest.run`

Резюме

В этой главе мы показали примеры примешивания утверждений непосредственно в рабочий код, а также способы их внешней записи в тестах. Вы увидели, что программисты, работающие на Scala, могут воспользоваться преимуществами таких популярных средств тестирования от сообщества Java, как JUnit, и более новыми средствами, разработанными исключительно для Scala, например `ScalaTest`, `ScalaCheck` и `specs2`. И утверждения, встроенные в код, и внешние тесты могут помочь повысить качество ваших программных продуктов.

Глоссарий

Алгебраический тип данных (algebraic data type). Тип, определяемый путем предоставления нескольких альтернатив, у каждой из которых есть собственный конструктор. Обычно предоставляется возможность его декомпозиции через сопоставление с образцом. Эта концепция встречается в языках спецификаций и функциональных языках программирования. В Scala алгебраические типы данных можно эмулировать с помощью `case`-классов.

Альтернатива (alternative). Ветвь выражения `match`. Имеет вид «`case pattern => выражение`». Альтернативу также могут называть *вариантом* (case).

Аннотация (annotation). Встречается в исходном коде и закрепляется за какой-то частью синтаксиса. Аннотации обрабатываются компьютером, поэтому их фактически можно использовать для расширения Scala.

Анонимная функция (anonymous function). Альтернативное название функционального литерала.

Анонимный класс (anonymous class). Синтетический подкласс, сгенерированный компилятором Scala из выражения `new`, в котором вслед за именем класса или трейта идут фигурные скобки. В них содержится тело анонимного подкласса, которое может быть пустым. Но если имя, указанное после выражения `new`, ссылается на трейт или класс с абстрактными членами, то эти члены необходимо конкретизировать внутри фигурных скобок, определяющих тело анонимного подкласса.

Аргумент (argument). При вызове функции каждому ее параметру соответствует передаваемый *аргумент*. Параметр — это переменная, которая ссылается на аргумент. Аргумент — это объект, который передается во время вызова. Кроме того, приложения могут принимать аргументы (командной строки), содержащиеся в массиве `Array[String]`, который передается методам `main` объектов-одиночек.

Блок (block). Одно или несколько выражений или объявлений, заключенных в фигурные скобки. При вычислении блока все его выражения и объявления обрабатываются по очереди, после чего блок возвращает в качестве результата значение последнего выражения. Блоки обычно используются в качестве тел функции, выражений `for`, циклов `while` и в любых других местах, где нужно сгруппировать какое-то количество инструкций. Если говорить более формально, то блок — это конструкция для инкапсуляции, в которой видны только побочные эффекты и итоговое значение. Таким образом, фигурные скобки, в которых определяется класс или объект, не формируют блок, поскольку поля и методы (определенные внутри этих скобок) видны снаружи. Такие фигурные скобки составляют *шаблон*.

Вариантность (variance). Параметр типа для класса или трейта можно помечать с помощью аннотации *вариантности*: либо как *ковариантный* (+), либо как *контравариантный* (-). Такие аннотации определяют, как у обобщенного класса или трейта работает отношение наследования. Например, обобщенный класс `List` ковариантен в своем параметре типа, поэтому `List[String]` является подтипом `List[Any]`. По умолчанию, если не указывать аннотации + или -, то параметры типов *ковариантны*.

Виртуальная машина Java (Java Virtual Machine, JVM). *Среда выполнения*, в которой работают программы на языке Scala.

Возврат (return). В программах на языке Scala функция *возвращает* значение. Вы можете называть его *результатом* функции. Кроме того, можно сказать, что функция *приводит к* значению. Результат любой функции в Scala — объект.

Вспомогательная функция (helper function). Предназначена для предоставления каких-либо возможностей для одной или нескольких других функций поблизости. Вспомогательные функции часто реализуются как локальные.

Вспомогательный конструктор (auxiliary constructor). Внутри фигурных скобок с определением класса можно указывать дополнительные конструкторы, которые выглядят как определения методов с именем `this`, но без результирующего типа.

Вспомогательный метод (helper method). Вспомогательная функция, являющаяся членом класса. Вспомогательные методы зачастую являются приватными.

Выдача (yield). Выражение может *выдавать* (yield) результат. Ключевое слово `yield` обозначает результат выражения `for`.

Вызов (invoke). Вы можете *вызвать* метод, функцию или замыкание с аргументами, то есть при выполнении их тела будут использованы заданные аргументы.

Выражение (expression). Любой фрагмент кода в Scala дает какой-либо результат. Можно сказать, что результат *вычисляется из* выражения или выражение *возвращает* значение.

Выражение генератора (generator expression). Генерирует последовательность значений в выражении `for`. Например, в `for(i <- 1 to 10)` выражением генератора выступает `1 to 10`.

Выражение фильтра (filter expression). Булево выражение, которое идет за инструкцией `if` в выражении `for`. В `for(i <- 1 to 10; if i % 2 == 0)` фильтрующим выражением выступает `i % 2 == 0`.

Генератор (generator). Определяет именованное значение и последовательно присваивает ему результаты выражения `for`. Например, в `for(i <- 1 to 10)` генератором выступает `i <- 1 to 10`. Значение справа от `<-` — это *выражение генератора*.

Замыкание (closure). Функциональный объект, который захватывает свободные переменные и как будто замыкается вокруг переменных, видимых в момент его создания.

Значение (value). Результат любого вычисления или выражения в Scala. При этом все значения в Scala являются объектами. Значение, в сущности, — это образ объекта в памяти (в куче JVM или стеке).

Императивный стиль (imperative style). В этом стиле программирования акцент делается на том, в какой последовательности выполняются операции, чтобы их побочные эффекты проявились в правильном порядке. Этот стиль характеризуется итерацией с циклами, изменением данных без копирования и методами с побочными эффектами. Эта парадигма доминирует в таких языках, как C, C++, C# и Java, контрастируя с *функциональным стилем*.

Инвариант (invariant). Термин имеет два значения. Это может быть свойство, которое всегда остается истинным при условии, что структура данных имеет правильный формат. Например, инвариантом отсортированного двоичного дерева может быть требование, согласно которому каждый узел должен быть упорядочен перед своим правым подузлом, если таковой имеется. Термин «*инвариант*» также иногда

служит синонимом нонварианта: «класс `Array` является инвариантным в своем параметре типа».

Инициализация (initialize). При определении переменной в исходном коде Scala вы должны *инициализировать* ее с помощью какого-либо объекта.

Инструкция (statement). Выражение, определение или импорт, то есть действия, которые могут быть заданы в шаблоне или блоке исходного кода Scala.

Карринг (currying). Способ написания функций с несколькими списками параметров. Например, `def f(x: Int)(y: Int)` — это каррированная функция с двумя списками параметров. Чтобы применить каррированную функцию, ей нужно передать несколько списков аргументов, как в `f(3)(4)`. Но мы можем также выполнить *частичное применение* каррированной функции, как в `f(3)`.

Класс (class). Определяется с помощью ключевого слова `class` и может быть либо абстрактным, либо конкретным. Во время создания экземпляра класса его можно параметризовать с указанием типов и значений. В выражении `new Array[String](2)` создается экземпляр класса `Array`, а тип получаемого значения — `Array[String]`. Класс, который принимает параметры типов, называется *конструктором типа*. У типа тоже может быть класс — например, классом типа `Array[String]` является `Array`.

Класс-компаньон (companion class). Класс с тем же именем, что и у объекта-синглтона, определенного в том же исходном файле. Это класс-компаньон объекта-одиночки.

Ковариантность (covariant). Аннотацию *ковариантности* можно применить к параметру типа в классе или трейте, указав перед ним знак плюс (+). В результате класс или трейт формирует ковариантные взаимоотношения с аннотированным параметром типа (направленные в ту же сторону). Например, если класс `List` является ковариантным в своем первом параметре типа, то `List[String]` будет подтипом `List[Any]`.

Композиция примесей (mixin composition). Процесс примешивания трейтов в классы или другие трейты. От традиционного множественного наследования *композиция примесей* отличается тем, что тип ссылки `super` неизвестен в момент определения трейта и определяется каждый раз, когда трейт примешивается в класс или в другой трейт.

Конструктор типа (type constructor). Класс или трейт, принимающий параметры типов.

Контравариантность (contravariant). Аннотацию *контравариантности* можно применить к любому параметру типа в классе или трейте, указав перед ним знак минус (-). В результате класс или трейт формирует контравариантные взаимоотношения с аннотированным параметром типа (направленные в противоположную от него сторону). Например, если класс `Function1` является контравариантным в своем первом параметре типа, то `Function1[Any, Any]` будет подтипом `Function1[String, Any]`.

Литерал (literal). `1`, "Один" и `(x: Int) => x + 1` — это все примеры *литералов*. Это краткое описание объекта, которое в точности отражает структуру созданного объекта.

Локальная переменная (local variable). `val` или `var`, определенные внутри блока. Параметры функции, несмотря на свою схожесть с локальными переменными, таковыми не считаются; их называют просто параметрами или переменными.

Локальная функция (local function). Функция, определенная внутри блока. Тогда как функция, определенная как член класса, трейта или объекта-одиночки, называется *методом*.

Метапрограммирование (meta-programming). Вид программирования, в котором программы принимают на вход другие программы. Компиляторы и инструменты наподобие `scaladoc` являются метапрограммами. Средства метапрограммирования необходимы для работы с *аннотациями*.

Метод (method). Функция, которая является членом какого-то класса, трейта или объекта-одиночки.

Метод без параметров (parameterless method). Это функция без параметров, которая является членом класса, трейта или объекта-одиночки.

Множественные определения (multiple definitions). Одно и то же выражение может быть присвоено в *нескольких определениях*, если используется синтаксис вида `val v1, v2, v3 = expr`.

Модификатор (modifier). Ключевое слово, которое каким-то образом ограничивает определение класса, трейта, поля или метода. Например, модификатор `private` говорит о том, что определяемый класс, трейт, поле или метод является приватным.

Недоступность (unreachable). На уровне Scala объекты могут становиться недоступными; в этом случае среда выполнения может освободить память, которую они занимают. Недоступность вовсе не означает от-

сутствие ссылок. Ссылочные типы (экземпляры `AnyRef`) реализуются в виде объектов, размещенных в куче JVM. Когда экземпляр ссылочного типа становится недоступным, на него и в самом деле ничего не ссылается, что позволяет сборщику мусора его освободить. Типы значений (экземпляры `AnyVal`) реализуются как с помощью примитивных типов, так и в виде типов-обертки (таких, как `java.lang.Integer`), которые находятся в куче. Экземпляры типов значений могут упаковываться (превращаться из примитивного значения в объект-обертку) и распаковываться (превращаться из объекта-обертки в примитивное значение) на протяжении существования переменной, которая на них ссылается. Если экземпляр типа значения, который в настоящий момент представлен объектом-оберткой в куче JVM, становится недоступным, то на него и в самом деле ничего не ссылается, что позволяет сборщику мусора его освободить. Но если в настоящий момент значение имеет примитивный тип и становится недоступным, то это не означает, что на него больше ничего не ссылается, поскольку он больше не существует в виде объекта в куче JVM. Среда выполнения может освободить память, занятую недоступными объектами. Но если тип `Int`, к примеру, реализован во время выполнения как примитивное значение `int` из языка Java и занимает какую-то память в стековом фрейме выполняющегося метода, то эта память освобождается, только когда фрейм достается из стека при завершении метода. Память для ссылочных типов, таких как `String`, может быть освобождена сборщиком мусора JVM после того, как они станут недоступными.

Неизменяемость (immutable). Объект является *неизменяемым*, если после того, как он был создан, его значение нельзя изменить никаким видимым для клиентов образом. Объекты могут быть неизменяемыми, но не обязательно.

Нонвариант (nonvariant). Параметр типа класса или трейта по умолчанию является *нонвариантом*. Когда этот параметр меняется, класс или трейт не производит подтип. Например, класс `Array` является нонвариантом в своем параметре типа, поэтому `Array[String]` нельзя считать ни подтипом, ни супертипом `Array[Any]`.

Обобщенный класс (generic class). Класс, принимающий параметры типа. Например, класс `scala.List` принимает параметр типа, поэтому является обобщенным.

Обобщенный трейт (generic trait). Трейт, принимающий параметры типа. Например, трейт `scala.collection.Set` принимает параметр типа, поэтому является обобщенным.

Образец (паттерн) (pattern). В выражении `match образца` указывается после каждого ключевого слова `case`, но перед *ограждением образца* (или символом `=>`).

Объект без ссылки (unreferenced). См. *Недоступность*.

Объект-компаньон (companion object). Объект-одиночка с тем же именем, что и у класса, определенного в том же исходном файле. Объекты-компаньоны и классы-компаньоны имеют доступ к приватным членам друг друга. Кроме того, любое неявное приведение типов, определенное в объекте-компаньоне, будет находиться в области видимости кода, в котором используется класс.

Объект-одиночка (singleton object). Объект, в определении которого указано ключевое слово `object`. У каждого объекта-одиночки есть один и только один экземпляр. Если он находится в одном исходном файле с одноименным классом, то его называют *объектом-компаньоном* данного класса. При этом класс является его *классом-компаньоном*. Объект-одиночка, у которого нет класса-компаньона, является *самостоятельным объектом*.

Объявление (declare). Вы можете *объявить* абстрактное поле, метод или тип, указав имя сущности, но не ее реализацию. Ключевое различие между объявлением и определением в том, что последнее, в отличие от первого, создает реализацию именованной сущности.

Ограждение образца (pattern guard). В выражении `match` за образцом может следовать его *ограждение*. Например, в `case x if x % 2 == 0 => x + 1` ограждением *образца* выступает `if x % 2 == 0`. Блок `case` с ограждением *образца* выбирается только в случае соответствия *образцу* и если его ограждение возвращает `true`.

Ограничение типа (type constraint). Некоторые аннотации выступают *ограничениями типа*; это значит, что они накладывают дополнительные ограничения на то, какие значения охватывает тип. Например, `@positive` может ограничивать 32-битный целочисленный тип `Int` положительными значениями. Ограничения типов проверяются не стандартным компилятором Scala, а его подключаемым модулем или дополнительным инструментом.

Операция (operation). В Scala каждая *операция* представляет собой вызов метода. Методы могут вызываться с помощью *синтаксиса оператора*; например, в выражении `b + 2` знак `+` — это *оператор*.

Определение (define). В программе на языке Scala *определение* состоит из имени и реализации. Вы можете определять классы, трейты, объекты-

одиночки, поля, методы, локальные функции, локальные переменные и т. д. Определение всегда подразумевает какую-то реализацию, поэтому абстрактные члены не определяются, а *объявляются*.

Параметр (parameter). Функции могут принимать любое количество *параметров* (от нуля и больше). У каждого параметра есть имя и тип. Если сравнивать аргументы и параметры, то первые ссылаются непосредственно на объекты, которые передаются при вызове функции, а вторые являются переменными, которые ссылаются на эти передаваемые аргументы.

Параметр типа (type parameter). Параметр обобщенного класса или метода, в качестве которого должен быть указан тип. Например, класс `List` определен как `class List[T] { ...`, а метод `identity` объекта `Predef` имеет определение `def identity[T](x:T) = x`. В обоих случаях `T` — это параметр типа.

Параметр, передаваемый по значению (by-value parameter). Параметр, перед типом которого *не* указано `a =>`, например `(x: Int)`. Аргумент, относящийся к такому параметру, вычисляется перед вызовом метода. Это альтернатива передаче параметров *по имени*.

Параметр, передаваемый по имени (by-name parameter). Параметр, перед типом которого указано `a =>`, например `(x: => Int)`. Аргумент, относящийся к такому параметру, вычисляется не перед вызовом метода, а каждый раз, когда в методе на него ссылаются *по имени*. Если параметр не передается по имени, то передается *по значению*.

Параметрическое поле (parametric field). Поле, определенное как параметр класса.

Первичный конструктор (primary constructor). Главный конструктор класса, который вызывает конструктор суперкласса, при необходимости инициализирует поля с использованием переданных значений и выполняет любой высокоуровневый код, определенный в фигурных скобках класса. Поля инициализируются только для тех параметров, которые не были переданы в суперконструктор; исключения составляют поля, которые не используются в теле класса и, следовательно, могут быть опущены.

Перекрывание (shadow). Новое определение локальной переменной *перекрывает* переменную с тем же именем, находящуюся в наружной области видимости.

Переменная (variable). Именованная сущность, которая ссылается на объект. Переменная может быть определена как `val` или `var`. В обоих случаях

при определении требуется инициализация, но только `var` позже можно будет назначить ссылку на другой объект.

Переназначаемость (reassignable). Переменная может быть *переназначаемой*. Значения `var` переназначаемые, а `val` — нет.

Подкласс (subclass). Класс является *подклассом* всех своих суперклассов и трейтов.

Подстановочный тип (wildcard type). Включает ссылки на неизвестные переменные типов. Например, `Array[_]` является подстановочным типом. Это массив, о типе элементов которого ничего не известно.

Подтип (subtype). Компилятор Scala позволяет использовать вместо типа любые его *подтипы*. Если классы и трейты не принимают никаких параметров типов, то отношение подтипа является зеркальным отражением отношения подкласса. Например, если класс `Cat` является подклассом абстрактного класса `Animal` и ни один из них не принимает параметры типов, то тип `Cat` будет подтипом `Animal`. Точно так же если трейт `Apple` является подтрейтом трейта `Fruit` и ни один из них не принимает параметры типов, то тип `Apple` будет подтипом `Fruit`. Но если классы и трейты принимают параметры типов, то в силу вступает вариантность. Например, поскольку абстрактный класс `List` является ковариантным в своем единственном параметре типа (то есть объявлен как `List[+A]`), `List[Cat]` будет подтипом `List[Animal]`, а `List[Apple]` — подтипом `List[Fruit]`. Эти отношения подтипов существуют, несмотря на то что классом каждого из этих типов выступает `List`. Для сравнения: поскольку класс `Set` не является ковариантным в своем параметре типа (объявлен как `Set[A]`, без знака плюс), `Set[Cat]` *не* будет подтипом `Set[Animal]`. Подтип должен правильно реализовывать контракты своих супертипов с соблюдением принципа подстановки Лисков, но компилятор проверяет это только на уровне типа.

Подтрейт (subtrait). Трейт является *подтрейтом* всех своих супертрейтов.

Получатель (receiver). Получатель вызова метода — это переменная, выражение или объект, для которого вызывается метод.

Предикат (predicate). Функция, возвращающая тип `Boolean`.

Применение (apply). Вы можете *применить* метод, функцию или замыкание к аргументам, то есть вызвать их для этих аргументов.

Примесь (mixin). Так называют трейт, который используется в композиции примесей. Иными словами, `Nat` в `trait Nat` является обычным трей-

том, а в `new Cat extends AnyRef with Hat` его можно назвать примесью. Этот термин можно использовать как глагол. Например, вы можете *примешать* трейты в классы или другие трейты.

Принцип единообразного доступа (uniform access principle). Согласно этому принципу для доступа к переменным и функциям, у которых нет параметров, должен использоваться один и тот же синтаксис. Scala поддерживает данный принцип, не позволяя указывать скобки при вызове функций без параметров. Благодаря этому вместо определения функции без параметров можно подставить `val` и *наоборот*, не затрагивая клиентский код.

Присваивание (assign). Вы можете *присвоить* объект переменной. После этого переменная будет ссылаться на данный объект.

Процедура (procedure). Функция, возвращающая тип `Unit`, которая выполняется исключительно для получения побочных эффектов.

Прямой подкласс (direct subclass). Класс, являющийся *прямым наследником* своего суперкласса.

Прямой суперкласс (direct superclass). Класс, непосредственной производной которого является другой класс или трейт, то есть ближайший класс, который находится сверху от него в иерархии наследования. Если класс `Parent` указан в необязательной инструкции `extends` класса `Child`, то это значит, что `Parent` является прямым суперклассом `Child`. Если в инструкции `extends` класса `Child` указан трейт, то данный трейт будет прямым суперклассом `Child`. Если у `Child` нет инструкции `extends`, его прямым родителем является `AnyRef`. Прямой суперкласс может принимать параметры типов — например, класс `Child` может расширять `Parent[String]`; в этом случае прямым суперклассом `Child` по-прежнему будет `Parent`, а не `Parent[String]`. С другой стороны, `Parent[String]` будет прямым *супертипом* `Child`. Чтобы узнать больше о различиях между классами и типами, см. *Супертип*.

Равенство (equality). Отношение между значениями, выраженное как `==` (при отсутствии уточняющих параметров). См. также *Равенство ссылок*.

Равенство ссылок (reference equality). Означает, что две ссылки идентифицируют один и тот же объект Java. В случае со ссылочными типами равенство ссылок можно определить с помощью вызова `eq` в `AnyRef` (в Java-программах для определения равенства ссылок можно использовать `==`, но тоже только для ссылочных типов).

Результат (result). Выражения в программах на языке Scala дают *результат*. Результат любого выражения в Scala — объект.

Результирующий (в Scala), возвращаемый (в Java) тип (result type). Результирующий/возвращаемый тип метода — тип значения, которое возвращается в результате вызова этого метода.

Рекурсия (recursive). Функция, которая вызывает саму себя, называется *рекурсивной*. Если этот вызов происходит только в последнем выражении функции, то она является хвостовой рекурсией.

Самостоятельный объект (standalone object). Объект-одиночка без класса-компаньона.

Свободная переменная (free variable). Переменная называется *свободной*, если используется в выражении, но не объявлена внутри него. Например, в выражении функционального литерала $(x: \text{Int}) \Rightarrow (x, y)$ используются обе переменные, x и y , но только y является свободной, так как не определена внутри этого выражения.

Связанная переменная (bound variable). Переменная, которая определена и используется внутри выражения, является его *связанной переменной*. Например, в выражении функционального литерала $(x: \text{Int}) \Rightarrow (x, y)$ используются две переменные, x и y , но только x является связанной, поскольку определена в выражении как Int и выступает единственным аргументом функции, описанной этим выражением.

Селектор (selector). Значение, с которым сопоставляются образцы в выражении `match`. Например, в `s match { case _ => }` селектором выступает `s`.

Сериализация (serialization). Объект можно *сериализовать* в поток байтов, который затем может быть сохранен в файлы или передан по сети. Позже поток байтов можно будет *десериализовать*, даже находясь на другом компьютере, и получить объект, который идентичен сериализованному оригиналу.

Сигнатура (signature). Сокращенный вариант понятия «*сигнатура типа*».

Сигнатура типа (type signature). Сигнатура типа метода определяет его имя, а также количество, порядок и типы его параметров (если таковые имеются) и возвращаемый тип. Сигнатура типа класса, трейта или объекта-синглтона определяет его имя, сигнатуры типов всех его членов и конструкторов, а также отношения наследования и применения, которые в нем объявлены.

Синтетический класс (synthetic class). Не пишется вручную программистом, а генерируется автоматически компилятором.

Скрипт (script). Файл с высокоуровневыми определениями и выражениями, который можно запускать непосредственно с помощью команды `scala`,

без предварительной компиляции. Скрипт должен заканчиваться выражением, а не определением.

Слабоструктурированные данные (semi-structured data). Данные XML являются слабоструктурированными. Они структурированы лучше, чем плоский двоичный или текстовый файл, но при этом уступают полноценным структурам данных в языках программирования.

Собственный тип (self type). Собственный тип трейта — это предполагаемый тип `this`, получателя, который будет использоваться внутри трейта. Любой конкретный класс, который примешивается в трейт, должен иметь тип, соответствующий собственному типу трейта. Чаще всего собственные типы используются для разбиения крупных классов на несколько трейтов (см. главу 7).

Создание экземпляра (instantiate). Создать экземпляр класса означает создать на основе класса новый объект. Эта операция происходит только во время выполнения.

Среда выполнения (runtime). Виртуальная машина Java (Java Virtual Machine, JVM), в которой выполняется программа на языке Scala. *Среда выполнения* включает в себя как виртуальную машину, соответствующую спецификации JVM, так и библиотеки Java API вместе со стандартными библиотеками Scala API. Словосочетание «*во время выполнения*» (run time) означает, что программа выполняется. Существует также время компиляции.

Ссылаться (refers). Переменная в выполняемой программе на языке Scala всегда *ссылается* на какой-то объект. Даже если этой переменной присвоить `null`, на концептуальном уровне она будет ссылаться на объект `Null`. Во время выполнения объект может быть реализован в виде объекта Java или значения примитивного типа, но Scala позволяет программистам рассуждать о выполнении своего кода на более высоком уровне абстракции. См. также *Ссылка*.

Ссылка (reference). Абстракция для указателя в Java, которая однозначно идентифицирует объект, размещенный в куче JVM. Переменные ссылочных типов хранят ссылки на объекты, поскольку ссылочные типы (экземпляры `AnyRef`) реализованы в виде объектов Java, находящихся в куче JVM. Для сравнения: переменные с типом значения могут хранить ссылку (на тип-обертку), а могут и нет (когда объект представлен примитивным значением). В целом переменные в Scala *ссылаются* на объекты. Термин «ссылаться» более абстрактный, чем «хранить ссылку». Если переменная типа `scala.Int` в настоящий момент представлена в виде примитивного значения `int` из Java,

то все равно ссылается на объект `Int`, хотя никаких ссылок при этом не используется.

Ссылочная прозрачность (referential transparency). Свойство функций, которые не зависят от временного контекста и не имеют побочных эффектов. Если взять конкретные входные данные, то вызов ссылочно прозрачной функции можно заменить ее результатом, не меняя семантику программы.

Ссылочный тип (reference type). Подкласс `AnyRef`. Во время выполнения экземпляры ссылочных типов всегда находятся в куче JVM.

Статический тип (static type). См. *Тип*.

Суперкласс (superclass). Суперклассом класса являются его прямой суперкласс, прямой суперкласс прямого суперкласса и так далее вплоть до `Any`.

Супертип (supertype). Тип является супертипом по отношению ко всем своим подтипам.

Супертрейт (supertrait). Супертрейты класса или трейта (если таковые имеются) включают все трейты, напрямую примешанные в класс или трейт или любые его суперклассы, а также супертрейты этих трейтов.

Тип (type). У всех переменных и выражений в программе на языке Scala есть *тип*, известный во время компиляции. Он ограничивает значения, на которые может ссылаться переменная и которые может возвращать выражение во время выполнения. Тип переменной или выражения можно также называть *статическим типом*, если необходимо подчеркнуть его отличие от *типа времени выполнения*. Иными словами, понятие «тип» само по себе является статическим. Тип отличается от класса, поскольку параметризованный класс может формировать много разных типов. Например, `List` — это класс, а не тип. `List[T]` — тип со свободным параметром типа. `List[Int]` и `List[String]` — это тоже типы (их называют *образующими типами*, поскольку у них нет свободных параметров типов). У типа может быть класс или трейт. Например, классом типа `List[Int]` является `List`, а трейтом типа `Set[String]` — `Set`.

Тип времени выполнения (runtime type). Тип объекта во время выполнения. Для сравнения: *статическим* называют тип выражения во время компиляции. Большинство типов времени выполнения представляют собой типы классов без параметров типов. Например, тип времени выполнения `"Hi"` является строкой, а `(x: Int) => x + 1` — `Function1`.

Для проверки типов времени выполнения можно использовать `isInstanceOf`.

Тип значения (value type). Любой подкласс `AnyVal`, такой как `Int`, `Double` или `Unit`. Этот термин имеет смысл на уровне исходного кода Scala. Во время выполнения экземпляры типов значений, соответствующие примитивным типам Java, могут быть реализованы в виде значений примитивных типов или экземпляров типов-обертки, таких как `java.lang.Integer`. На протяжении существования экземпляра типа значения среда выполнения может превращать его из примитивного типа в тип-обертку и обратно (то есть упаковывать и распаковывать).

Тип, зависящий от пути (path-dependent type). Тип наподобие `swiss.cow.Food`, где `swiss.cow` — это *путь*, составляющий ссылку на объект. Смысл типа зависит от пути, по которому вы к нему обращаетесь. Например, `swiss.cow.Food` и `fish.Food` — это разные типы.

Трейт (trait). Определяется с помощью ключевого слова `trait` и представляет собой нечто похожее на абстрактный класс, который не может принимать никаких значений. Его можно «примешивать» в классы или другие трейты с помощью процедуры под названием «композиция примесей». Трейт, примешанный в класс или другой трейт, называют *примесью*. Трейт может быть параметризован с использованием одного или нескольких типов; в этом случае формируется новый тип. Например, `Set` — трейт, который принимает один параметр типа, в то время как `Set[Int]` — это тип. Можно сказать, что `Set` является трейтом типа `Set[Int]`.

Уточняющий тип (refinement type). Тип, который формируется за счет присваивания значений членам базового типа внутри его фигурных скобок. Эти члены уточняют типы, присутствующие в базовом типе. Например, тип «животное, которое ест траву» можно выразить как `Animal { type SuitableFood = Grass }`.

Фильтр (filter). Инstrukция `if` в выражении `for`, за которой идет булево выражение. В `for(i <- 1 to 10; if i % 2 == 0)` фильтром выступает `if i % 2 == 0`. Значение справа от `if` — это *выражение фильтра*.

Функциональное значение (function value). Функциональный объект, который можно вызывать, как любую другую функцию. Класс функционального значения расширяет один из трейтов `FunctionN` (например, `Function0`, `Function1`) из пакета `scala` и обычно выражается в исходном коде с помощью синтаксиса *функциональных литералов*. Функциональное значение вызывается, когда срабатывает его метод

`apply`. Функциональное значение, захватывающее свободные переменные, является *замыканием*.

Функциональный литерал (function literal). Функция без имени в исходном коде Scala, описанная с помощью синтаксиса функциональных литералов. Например, `(x: Int, y: Int) => x + y`.

Функциональный стиль (functional style). В этом стиле программирования акцент делается на функциях и вычислении результатов, а порядок выполнения операций играет второстепенную роль. Характерные черты этого стиля — передача функциональных значений в методы с циклами, неизменяемые данные и методы без побочных эффектов. Эта парадигма доминирует в таких языках, как Haskell и Erlang, контрастируя с *императивным стилем*.

Функция (function). Функцию можно *вызвать* со списком аргументов для получения какого-либо результата. У функции есть список параметров, тело и возвращаемый тип. Функции, являющиеся членами класса, трейта или объекта-сигнлтона, называются *методами*. Функции, определенные внутри других функций, называются *локальными*. Функции, возвращающие тип `Unit`, называются *процедурами*. Анонимные функции в исходном коде называются *функциональными литералами*. Во время выполнения для функционального литерала создается объект, называемый *функциональным значением*.

Функция без параметров (parameterless function). Функция, которая не принимает параметров и определяется без использования пустых скобок. При вызове таких функций можно не указывать скобки. Это соответствует *принципу единообразного доступа*, что позволяет поменять `def` на `val`, не модифицируя клиентский код.

Функция первого класса (first-class function). Scala поддерживает *функции первого класса*. Это значит, вы можете выразить функцию в виде *функционального литерала* (как, например, в `(x: Int) => x + 1`) или объекта, который называют *функциональным значением*.

Характеристика for (for comprehension). Альтернативное название выражения `for`.

Хвостовая рекурсия (tail recursive). Возникает, когда функция вызывает саму себя только в своей последней операции.

Целевая типизация (target typing). Разновидность выведения типов, которая учитывает, какой тип ожидается в итоге. Например, в `nums.filter(x => x > 0)` компилятор Scala определяет, что `x` — это тип эле-

ментов `nums`, поскольку метод `filter` вызывает функцию для каждого элемента `nums`.

Частично примененная функция (partially applied function). Функция, которая используется в выражении с неполным списком своих аргументов. Например, если функция `f` имеет тип `Int => Int => Int`, то `f` и `f(1)` будут *частично примененными функциями*.

Член (member). Любой именованный элемент шаблона класса, трейта или объекта-синглтона. Чтобы обратиться к члену, нужно указать имя его владельца, точку и затем его простое имя. Например, поля и методы верхнего уровня, определенные в классе, являются членами этого класса. Трейт, определенный внутри класса, является его членом. Тип, определенный в классе с помощью ключевого слова `type`, является членом этого класса. Класс является членом пакета, в котором он определен. Тогда как локальную переменную или функцию нельзя считать членом окружающего ее блока.

Шаблон (template). Тело класса, трейта или объекта-одиночки. Определяет сигнатуру типа, поведение и начальное состояние класса, трейта или объекта.

Экземпляр (instance). Экземпляр класса — объект, понятие, которое существует только во время выполнения программы.

Библиография

- [Abe96] *Abelson, Harold and Gerald Jay Sussman*. Structure and Interpretation of Computer Programs. The MIT Press, second edition, 1996.
- [Aho86] *Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman*. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.
- [Bay72] *Bayer, Rudolf*. “Symmetric binary B-Trees: Data structure and maintenance algorithms.” *Acta Informatica*, 1 (4): 290–306, 1972.
- [Blo08] *Bloch, Joshua*. Effective Java Second Edition. Addison-Wesley, 2008.
- [DeR75] *DeRemer, Frank and Hans Kron*. “Programming-in-the large versus programming-in-the-small.” In Proceedings of the international conference on Reliable software, pages 114–121. ACM, New York, NY, USA, 1975. doi:<http://doi.acm.org/10.1145/800027.808431>.
- [Dij70] *Dijkstra, Edsger W*. “Notes on Structured Programming.”, April 1970. Circulated privately. Available at <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF> as EWD249 (accessed June 6, 2008).
- [Eck98] *Eckel, Bruce*. Thinking in Java. Prentice Hall, 1998.
- [Emi07] *Emir, Burak, Martin Odersky, and John Williams*. “Matching Objects With Patterns.” In Proc. ECOOP, Springer LNCS, pages 273–295. July 2007.
- [Eva03] *Evans, Eric*. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.
- [Fow04] *Fowler, Martin*. “Inversion of Control Containers and the Dependency Injection pattern.” January 2004. Available on the web at <http://martinfowler.com/articles/injection.html> (accessed August 6, 2008).
- [Gam95] *Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides*. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [Goe06] *Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Homes, and Doug Lea*. Java Concurrency in Practice. Addison Wesley, 2006.

- [Jav] The Java Tutorials: Creating a GUI with JFC/Swing. Available on the Web at <http://java.sun.com/docs/books/tutorial/uiswing>.
- [Kay96] *Kay, Alan C.* “The Early History of Smalltalk.” In *History of Programming languages – II*, pages 511–598. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1. doi:<http://doi.acm.org/10.1145/234286.1057828>.
- [Kay03] *Kay, Alan C.* An Email to Stefan Ram on the Meaning of the Term “Object-oriented Programming”, July 2003. The Email is Published on the Web at http://www.purl.org/stefan_ram/pub/doc_kay_oop_en (accessed June 6, 2008).
- [Kri19] *Krikava, Filip, Heather Miller, and Jan Vitek.* “Scala Implicits are Everywhere: a Large-scale Study of the Use of Scala Implicits in the Wild.” In *Proceedings of the ACM on Programming Languages*, volume 3. ACM, 2019. doi:<https://doi.org/10.1145/3360589>.
- [Lan66] *Landin, Peter J.* “The Next 700 Programming Languages.” *Communications of the ACM*, 9 (3): 157–166, 1966.
- [Mey91] *Meyers, Scott.* *Effective C++*. Addison-Wesley, 1991.
- [Mey00] *Meyer, Bertrand.* *Object-Oriented Software Construction*. Prentice Hall, 2000.
- [Mor68] *Morrison, Donald R.* “PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric.” *J. ACM*, 15 (4): 514–534, 1968. ISSN 0004-5411. doi:<http://doi.acm.org/10.1145/321479.321481>.
- [Ode03] *Odersky, Martin, Vincent Cremet, Christine Röckl, and Matthias Zenger.* “A Nominal Theory of Objects with Dependent Types.” In *Proc. ECOOP’03*, Springer LNCS, pages 201–225. July 2003.
- [Ode05] *Odersky, Martin and Matthias Zenger.* “Scalable Component Abstractions.” In *Proceedings of OOPSLA*, pages 41–58. October 2005.
- [Ode11] *Odersky, Martin.* *The Scala Language Specification, Version 2.9*. EPFL, May 2011. Available on the Web at <http://www.scalalang.org/docu/manuals.html> (accessed April 20, 2014).
- [Ray99] *Raymond, Eric.* *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O’Reilly, 1999.
- [Rum04] *Rumbaugh, James, Ivar Jacobson, and Grady Booch.* *The Unified Modeling Language Reference Manual (2nd Edition)*. Addison-Wesley, 2004.
- [SPJ02] *Simon Peyton Jones, et al.* “Haskell 98 Language and Libraries, Revised Report.” Technical Report, <http://www.haskell.org/onlinereport>, 2002.

- [Ste99] *Steele, Jr., Guy L.* “Growing a Language.” *Higher-Order and Symbolic Computation*, 12: 221–223, 1999. Transcript of a Talk Given at OOPSLA 1998.
- [Ste15] *Steindorfer, Michael J and Jurgen J Vinju.* “Optimizing hash-array mapped tries for fast and lean immutable JVM collections.” In *ACM SIGPLAN Notices*, volume 50, pages 783–800. ACM, 2015.
- [Str00] *Strachey, Christopher.* “Fundamental Concepts in Programming Languages.” *Higher-Order and Symbolic Computation*, 13: 11–49, 2000.
- [Vaz07] *Vaziri, Mandana, Frank Tip, Stephen Fink, and Julian Dolby.* “Declarative Object Identity Using Relation Types.” In *Proc. ECOOP 2007*, pages 54–78. 2007.

Об авторах

Мартин Одерски, создатель языка Scala, — профессор в Федеральной политехнической школе Лозанны, Швейцария (EPFL), и основатель Lightbend, Inc. Работает над языками программирования и системами, в частности над темой совмещения объектно-ориентированного и функционального подходов. С 2001 года сосредоточен на проектировании, реализации и улучшении Scala. Внес вклад в разработку Java как соавтор обобщенных типов и создатель текущего эталонного компилятора `javac`. Мартину было присвоено звание действительного члена ACM.

Лекс Спун — разработчик программного обеспечения в компании Square Inc.¹, создающей простое в использовании программное обеспечение для бизнеса и мобильных платежей. Занимался Scala на протяжении двух лет в ходе постдокторантуры в EPFL. Помимо Scala, участвовал в разработке самых разнообразных языков, включая динамический язык Smalltalk, научный язык X10 и логический язык CodeQL.

Билл Веннерс — президент Artima, Inc., занимающейся консалтингом, курсами, книгами и инструментами для работы со Scala. Автор книги *Inside the Java Virtual Machine* про архитектуру и внутреннее устройство платформы Java. Билл представляет сообщество в Scala Center и является ведущим разработчиком и проектировщиком фреймворка тестирования `ScalaTest` и библиотеки `Scalactic`, предназначенной для функционального и объектно-ориентированного программирования.

Фрэнк Соммерс — основатель и президент компании Autospaces Inc., предоставляющей решения для автоматизации рабочих процессов в сфере финансовых услуг. Фрэнк ежедневно работает с языком Scala уже свыше двенадцати лет.

¹ С 1 декабря 2021 года компания называется Block.

Мартин Одерски, Лекс Спун, Билл Веннерс, Фрэнк Коммерс

Scala. Профессиональное программирование

5-е издание

Перевел с английского *А. Павлов*

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературные редакторы	<i>А. Аверьянов, Н. Хлебина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, Е. Павлович</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 19.05.22. Формат 70х100/16. Бумага офсетная. Усл. п. л. 49,020. Тираж 500. Заказ 0000.