Функциональное программирование

Кей Хорстман

SCALA

для нетерпеливых

ДМК ПРЕСС
издательство

Кей Хостманн

Scala для нетерпеливых

Cay S. Hostmann

Scala for the Impatient

Addison-Wesley

Кей Хостманн

Scala для нетерпеливых

Москва, 2013

УДК 004.432.42Scala
ББК 32.973-018.1
Х84

Хостманн К.
Х84 Scala для нетерпеливых. – М.: ДМК Пресс, 2013. – 408 с.: ил.

ISBN 978-5-94074-920-2

Книга в сжатой форме описывает, что можно делать на языке Scala, и как это делать. Кей Хорстманн (Cay Horstmann), основной автор всемирного бестселлера «Core Java™», дает быстрое и практическое введение в язык программирования, основанное на примерах программного кода. Он знакомит читателя с концепциями языка Scala и приемами программирования небольшими «порциями», что позволяет быстро осваивать их и применять на практике. Практические примеры помогут вам пройти все стадии компетентности, от новичка до эксперта.

Издание предназначено для программистов разной квалификации, как знакомых с языком Scala, так и впервые изучающих языки функционального программирования.

УДК 004.432.42Scala
ББК 32.973-018.1

ISBN 978-0-321-77409-5 (анг.)
ISBN 978-5-94074-920-2 (рус.)

Моей жене, сделавшей эту книгу возможной,
и моим детям, сделавшим ее необходимой

Содержание

От издательства

При подготовке данной книги к печати, ее рукописи были переданы для обсуждения членам сообщества ru-scala (http://ru-scala.livejournal.com/), принявшим самое деятельное участие в ее улучшении.

Особую благодарность издательство выражает Руслану Шевченко, Виталию Морариану, Андрею Легкому и Ивану Федорову. Спасибо вам, друзья! Вашу помощь переоценить невозможно!

Предисловие

Когда я встретил Кея Хорстманна (Cay Horstmann) несколько лет тому назад, он сказал, что необходимо написать хорошую вводную книгу, описывающую язык Scala. Как раз перед этим вышла моя собственная книга, поэтому я, разумеется, спросил его, что в ней не так. Он ответил, что книга замечательная, но слишком большая – его студентам просто не хватает терпения прочитать все восемьсот страниц книги «Programming in Scala». Мне не оставалось ничего иного, как признать его правоту. И он вознамерился исправить ситуацию, написав книгу «Scala для нетерпеливых».

Я очень рад, что его книга наконец вышла, потому что она полностью соответствует своему названию. Она представляет собой весьма практичное введение в язык программирования Scala, описывает, в частности, чем этот язык отличается от Java, как преодолевать некоторые типичные проблемы, возникающие при его изучении, и как писать хороший программный код на языке Scala.

Scala – чрезвычайно выразительный и гибкий язык программирования. Он позволяет разработчикам библиотек использовать весьма сложные, высокоуровневые абстракции, чтобы пользователи этих библиотек, в свою очередь, могли легко и просто выражать свои мысли. В зависимости от того, с каким кодом вы столкнетесь, он может казаться очень простым или очень сложным.

Год назад я попытался дать некоторые разъяснения, определив ряд уровней для языка Scala и его стандартной библиотеки. Всего было выделено по три уровня для прикладных программистов и для создателей библиотек. Начальные уровни были просты в изучении, и их было вполне достаточно, чтобы начать писать программы. Знания, получаемые на средних уровнях, позволяют писать более выразительные и более функциональные программы, а библиотеки более гибкие в использовании. Освоив высшие уровни, программисты становятся экспертами, способными решать специализированные задачи. В то время я писал:

Я надеюсь, что это поможет начинающим решить, в каком порядке изучать темы, а учителям и авторам книг подскажет, в каком порядке представлять материал.

Книга Кея стала первой, где эта идея была воплощена в жизнь. Каждая глава помечена значком, обозначающим ее уровень, который сообщает читателю, насколько простой или сложной она является и на кого ориентирована – на разработчиков библиотек или прикладных программистов.

Как можно догадаться, первые главы представляют собой быстрое введение в основные возможности языка Scala. Но книга не останавливается на этом. Она также охватывает множество концепций «среднего» уровня и, наконец, доходит до описания весьма сложных тем, которые обычно не рассматриваются во вводных книгах, таких как создание парсер-комбинаторов или использование ограниченных продолжений. Метки, обозначающие уровень, могут служить руководством при выборе глав для чтения. И Кею с успехом удалось просто и доходчиво рассказать даже о самых сложных понятиях.

Мне настолько понравилась идея книги «Scala для нетерпеливых», что я предложил Кею и его редактору Грегу Доенчу (Greg Doench) выложить первую часть книги в свободный доступ на вебсайте Typesafe[1]. Они любезно согласились с моим предложением, за что я очень благодарен им. Теперь любой желающий может быстро обратиться к самому лучшему, на мой взгляд, компактному введению в язык Scala.

Мартин Одерски (Martin Odersky)
Январь 2012

---

[1] http://typesafe.com/resources/free-books. – *Прим. перев.*

Вступление

Развитие языков Java и C++ существенно замедлилось, и программисты, стремящиеся использовать самые современные технологии, обратили свои взоры на другие языки. Scala – весьма привлекательный выбор. Я считаю, что это самый интересный вариант для программистов, стремящихся вырваться за рамки Java или C++. Scala имеет выразительный синтаксис, который выглядит весьма свежо после приевшихся шаблонов Java. Программы на этом языке выполняются под управлением виртуальной машины Java, что открывает доступ к огромному количеству библиотек и инструментов. Он поддерживает функциональный стиль программирования, не отказываясь при этом от объектно-ориентированного стиля, давая возможность осваивать новые парадигмы постепенно. Интерпретатор дает возможность быстро опробовать свои идеи, что превращает изучение Scala в весьма увлекательное занятие. Наконец, язык Scala является статически типизированным языком, что дает компилятору возможность находить ошибки, а вам не тратить время на их поиск в работающей программе.

Я написал эту книгу для *нетерпеливых* читателей, желающих приступить к программированию на языке Scala немедленно. Я полагаю, что вы знакомы с Java, C# или C++, и потому не буду утруждать себя объяснением, что такое переменные, циклы или классы. Я не буду терпеливо перечислять все особенности языка, я не буду читать лекции о превосходстве одной парадигмы над другой, и я не заставлю вас продираться сквозь длинные искусственные примеры. Вместо этого вы будете получать необходимую информацию небольшими порциями, чтобы ее можно было быстро прочитать и вернуться к ней при необходимости.

Scala – сложный язык, но вы сможете эффективно использовать его, даже не зная всех его тонкостей. Мартин Одерски (Martin Odersky), создатель языка Scala, определил уровни владения языком для прикладных программистов и разработчиков библиотек, как показано в табл. П.1.

Таблица П.1. Уровни владения языком Scala

Прикладные программисты	Разработчики библиотек	Общий уровень владения языком
Начальный A1		Начальный
Переходный A2	Простой L1	Переходный
Эксперт A3	Сложный L2	Сложный
	Эксперт L3	Эксперт

Каждая глава (а иногда и отдельные разделы) помечены специальным значком, обозначающим уровень владения языком, необходимым для ее чтения. Главы следуют по возрастанию уровня сложности A1, L1, A2, L2, A3, L3. Даже если вы не планируете создавать собственные библиотеки, знание инструментов Scala, которыми пользуются разработчики библиотек, поможет вам эффективнее использовать чужие библиотеки.

Надеюсь, вам понравится изучать язык Scala с помощью этой книги. Если вы обнаружите ошибки или у вас появятся предложения по улучшению, посетите сайт http://horstmann.com/scala и оставьте свой комментарий. На этом сайте вы также найдете ссылку на файл архива, содержащего все примеры программного кода из этой книги.

Я очень благодарен Дмитрию Кирсанову (Dmitry Kirsanov) и Алине Кирсановой (Alina Kirsanova), превратившим мою рукопись в формате XHTML в замечательную книгу, позволив мне сконцентрироваться на содержимом и не отвлекаться на оформление. Любой автор скажет, насколько это здорово!

Книгу рецензировали: Адриан Кумиски (Adrian Cumiskey), Майк Дэвис (Mike Davis), Роб Диккенс (Rob Dickens), Даниэль Собрал (Daniel Sobral), Крейг Татарин (Craig Tataryn), Дэвид Уоленд (David Walend) и Уильям Уилер (William Wheeler). Спасибо вам за ваши комментарии и предложения!

Наконец, как всегда, хочу выразить признательность моему редактору Грегу Доенчу (Greg Doench) за то, что подал идею написать эту книгу, и за его поддержку в процессе работы.

Кей Хорстманн (Cay Horstmann)
Сан-Франциско, 2012

Об авторе

Кей Хорстманн (Cay S. Horstmann) – основной автор книги «Core Java™, Volumes I & II, Eighth Edition» (Sun Microsystems Press, 2008)[1], а также десятков других книг для профессиональных программистов и студентов факультетов информатики. Он является профессором информатики университета в Сан-Хосе и обладателем звания Java Champion.

---

[1] Кей С. Хорстманн, Г. Корнелл. Java 2. Библиотека профессионала. Основы. – Т. 1. – Вильямс, 2008. – ISBN: 978-5-8459-1378-4; Кей С. Хорстманн, Г. Корнелл. Java 2. Библиотека профессионала. Тонкости программирования. – Т. 2. – Вильямс, 2008. – ISBN: 978-5-8459-1482-8. – *Прим. перев.*

Глава 1. Основы

Темы, рассматриваемые в этой главе A1

- 1.1. Интерпретатор Scala.
- 1.2. Объявление значений и переменных.
- 1.3. Часто используемые типы.
- 1.4. Арифметика и перегрузка операторов.
- 1.5. Вызов функций и методов.
- 1.6. Метод apply.
- 1.7. Scaladoc.
- Упражнения.

В этой главе вы узнаете, как использовать язык Scala в качестве мощного карманного калькулятора, выполняя арифметические операции над числами в интерактивном режиме. Попутно здесь будет представлено множество важных понятий и идиом языка Scala. Вы также узнаете, как просматривать документацию Scaladoc.

В этом введении рассматриваются следующие основные темы:

- использование интерпретатора Scala;
- определение переменных с помощью объявлений `var` и `val`;
- числовые типы;
- использование операторов и функций;
- навигация по документации Scaladoc.

1.1. Интерпретатор Scala

Чтобы приступить к работе с интерпретатором Scala, необходимо:

- установить язык Scala;
- добавить путь к каталогу `scala/bin` в переменную окружения `PATH`;
- открыть окно терминала в своей операционной системе;
- ввести команду `scala` и нажать клавишу **Enter**.

Совет. Не любите пользоваться командной оболочкой? Другие способы запуска интерпретатора описываются на странице http://horstmann.com/scala/install.

Вводите следующие команды, завершая ввод нажатием клавиши **Enter**. Каждый раз интерпретатор будет выводить ответ. Например, если ввести `8 * 5 + 2` (как показано ниже), интерпретатор выведет ответ `42`.

```
scala> 8 * 5 + 2
res0: Int = 42
```

Результату назначается имя `res0`. Его можно использовать в последующих вычислениях:

```
scala> 0.5 * res0
res1: Double = 21.0
scala> "Hello, " + res0
res2: java.lang.String = Hello, 42
```

Как видите, интерпретатор также отображает тип результата – `Int`, `Double` и `java.lang.String` в примерах выше.

Вы можете вызывать методы. При некоторых способах запуска есть возможность использовать клавишу **Tab** для автоматического дополнения имен методов. Попробуйте ввести `res2.to` и нажать клавишу **Tab**. Если интерпретатор предложит варианты выбора, такие как:

```
toCharArray toLowerCase toString toUpperCase
```

это означает, что функция автоматического дополнения работает. Введите `U` и снова нажмите клавишу табуляции. Теперь вы должны получить единственный вариант:

```
res2.toUpperCase
```

Нажмите клавишу **Enter**, и на экране появится ответ. (Если функция автоматического дополнения не работает, придется ввести имя метода вручную.)

Попробуйте также понажимать клавиши со стрелками ↑ и ↓. В большинстве реализаций вы увидите прежде выполнявшиеся команды и сможете редактировать их. С помощью клавиш ←, → и **Del** измените последнюю команду на

```
res2.toLowerCase
```

Как видите, интерпретатор Scala способен прочитать выражение, вычислить его, вывести результат и прочитать следующее выражение. Такой порядок действий называется *цикл чтения–вычисления–вывода* (read-eval-print loop, REPL).

Строго говоря, программа `scala` *не является* интерпретатором. За кулисами она быстро компилирует введенные вами команды в байт-код и выполняет его в виртуальной машине Java. По этой причине большинство программистов на Scala предпочитают называть этот цикл «the REPL».

> **Совет.** Цикл REPL – ваш друг и помощник. Немедленная обратная связь поощряет эксперименты, и вы будете чувствовать себя увереннее, имея возможность немедленно получать результаты.

При этом очень хорошо иметь постоянно открытое окно редактора, чтобы можно было копировать в него удачные фрагменты кода для использования в будущем. Кроме того, экспериментируя с более сложными примерами, их можно сначала компоновать в редакторе, а затем копировать в окно REPL.

1.2. Объявление значений и переменных

Вместо имен `res0`, `res1` и т. д. можно определить собственные имена:

```
scala> val answer = 8 * 5 + 2
answer: Int = 42
```

Их можно использовать в последующих выражениях:

```
scala> 0.5 * answer
res3: Double = 21.0
```

Значение, объявленное с помощью `val`, в действительности является константой – ее значение нельзя изменить:

```
scala> answer = 0
<console>:6: error: reassignment to val
```

Чтобы объявить переменную, значение которой может изменяться, следует использовать ключевое слово `var`:

```
var counter = 0
counter = 1 // OK, переменные могут изменяться
```

В языке Scala предпочтительнее использовать `val`, если в дальнейшем не предполагается изменять значение. Самое удивительное для программистов на Java или C++, что в большинстве программ не требуется много переменных `var`.

Обратите внимание на отсутствие необходимости явно объявлять тип значения или переменной. Тип автоматически определяется из типа инициализирующего выражения. (Объявление значения или переменной без инициализации является ошибкой.)

Однако при необходимости тип можно объявить явно. Например:

```
val greeting: String = null
val greeting: Any = "Hello"
```

> **Примечание.** В языке Scala тип переменной или функции всегда указывается после имени этой переменной или функции. Это упрощает чтение объявлений сложных типов.

Так как мне часто приходится переключаться между языками Scala и Java, я замечаю, что мои пальцы автоматически набирают такие Java-объявления, как `String greeting`, поэтому мне приходится исправлять их на `greeting: String`. Это немного раздражает, но когда я работаю со сложными программами на языке Scala, мне нравится, что не нужно расшифровывать объявления в стиле языка C.

> **Примечание.** Возможно, кто-то уже заметил отсутствие точек с запятой после объявлений переменных и инструкций присваивания. В языке Scala точки с запятой необходимы, только если в одной строке присутствует несколько инструкций.

В одном объявлении можно объявить сразу несколько значений или переменных:

```
val xmax, ymax = 100 // xmax и ymax получат значение 100
var greeting, message: String = null
  // обе переменные, greeting и message, - строки со значением null
```

1.3. Часто используемые типы

Вы уже видели некоторые типы данных из языка Scala, такие как `Int` и `Double`. Как и в языке Java, в Scala имеются семь числовых типов: `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` и `Double`, – и один логический тип `Boolean`. Однако, в отличие от Java, все эти типы являются *классами*. В Scala нет никакой разницы между простыми типами и классами. Вы можете вызывать методы чисел, например:

```
1.toString() // Вернет строку "1"
```

или еще интереснее:

```
1.to(10)     // Вернет Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

(Класс `Range` будет рассматриваться в главе 13, а пока просто считайте его коллекцией чисел.)

В языке Scala нет необходимости использовать типы-обертки (wrapper types). Эту работу берет на себя компилятор, преобразуя простые типы в обертки и обратно. Например, создав массив чисел типа `Int`, в виртуальной машине вы получите массив `int[]`.

Как было показано в разделе 1.1 «Интерпретатор Scala», в случае со строками Scala опирается на класс `java.lang.String`. Однако расширяет этот класс более чем сотней дополнительных операций в классе `StringOps`. Например, метод `intersect` возвращает символы, общие для двух строк:

```
"Hello".intersect("World") // Вернет "lo"
```

В этом примере объект `"Hello"` типа `java.lang.String` неявно преобразуется в объект типа `StringOps`, и затем вызывается метод `intersect` класса `StringOps`.

Поэтому не забудьте заглянуть в описание класса `StringOps`, когда будете пользоваться документацией для Scala (см. раздел 1.7 «Scaladoc»).

Существуют также аналогичные классы `RichInt`, `RichDouble`, `RichChar` и т. д. Каждый из них имеет небольшой набор удобных методов для расширения на своих бедных родственников – `Int`, `Double` или `Char`. Метод `to`, представленный выше, в действительности является методом класса `RichInt`. В выражении

```
1.to(10)
```

значение 1 типа Int сначала преобразуется в объект типа RichInt, а затем вызывается метод to этого объекта.

Наконец, существуют классы BigInt и BigDecimal для вычислений с произвольной (но конечной) точностью. Они опираются на классы java.math.BigInteger и java.math.BigDecimal, но, как будет показано в следующем разделе, они намного удобнее, потому что допускают использование с обычными математическими операторами.

> **Примечание.** Для преобразований между числовыми типами в Scala используются методы, а не операция приведения типа. Например, 99.44.toInt вернет 99, а 99.toChar – 'c'. Разумеется, как и в языке Java, метод toString преобразует любой объект в строку.

Преобразовать строку цифр в число можно с помощью методов toInt и toDouble. Например, "99.44".toDouble вернет 99.44.

1.4. Арифметика и перегрузка операторов

Арифметические операторы в языке Scala действуют так же, как в Java или в C++:

```
val answer = 8 * 5 + 2
```

Операторы + - * / % выполняют свою обычную работу, как и поразрядные операторы & | ^ >> <<. Есть лишь один необычный аспект: эти операторы в действительности являются методами. Например,

```
a + b
```

это сокращенная форма записи

```
a.+(b)
```

Здесь + является именем метода. В языке Scala отсутствует глупое предубеждение против неалфавитно-цифровых символов в именах методов. Вы можете объявлять методы, содержащие почти любые символы в именах. Например, класс BigInt определяет метод с именем /%, который возвращает частное и остаток от деления.

В общем случае следующая форма записи

```
a method b
```

является сокращением от

```
a.method(b)
```

где `method` – это метод с двумя параметрами (один неявный и один – явный). Например, вместо

```
1.to(10)
```

можно записать

```
1 to 10
```

Используйте любую форму, которая будет проще для восприятия. Начинающие программисты на Scala по привычке придерживаются синтаксиса языка Java, и это хорошо. Разумеется, даже самые прожженные Java-программисты предпочтут использовать `a + b` вместо `a.+(b)`.

Между языком Scala, с одной стороны, и Java или C++ – с другой, существует одна весьма заметная разница. В языке Scala отсутствуют операторы `++` и `--`. Вместо них используются выражения `+=1` и `-=1`:

```
counter+=1 // Увеличит значение переменной counter, так как в Scala нет ++
```

Некоторые спрашивают, существуют ли какие-либо глубинные причины, объясняющие отсутствие оператора `++` в языке Scala. (Обратите внимание, что нельзя просто реализовать метод с именем `++`. Поскольку класс `Int` является неизменяемым, такой метод не сможет изменить целочисленное значение.) Разработчики языка Scala решили не вводить еще одно специальное правило, только чтобы сэкономить на нажатиях клавиш.

При работе с объектами типа `BigInt` и `BigDecimal` можно использовать обычные математические операторы:

```
val x: BigInt = 1234567890
x * x * x // Вернет 1881676371789154860897069000
```

Так намного лучше, чем в Java, где вы были бы вынуждены вызывать `x.multiply(x).multiply(x)`.

Примечание. В Java не поддерживается возможность перегрузки операторов, и разработчики Java утверждают, что это – благо, потому что препятствует появлению совершенно сумасшедших операторов, таких как `!@$&*`, которые могут сделать программу совершенно нечитаемой. Конечно, это глупо – программу точно так же можно сделать нечитаемой, выбирая сумасшедшие имена методов, такие как `qxywz`. Язык Scala позволяет определять операторы, полагаясь на ваше благоразумие при использовании этой возможности.

1.5. Вызов функций и методов

В дополнение к методам в языке Scala поддерживаются функции. В Scala имеются такие математические функции, как `min` или `pow`, пользоваться которыми намного проще, чем в Java, – для этого не требуется вызывать *статические методы* класса.

```
sqrt(2)     // Вернет 1.4142135623730951
pow(2, 4)   // Вернет 16.0
min(3, Pi) // Вернет 3.0
```

Математические функции определены в пакете `scala.math`. Их можно импортировать инструкцией

```
import scala.math._ // Символ _ в Scala - "групповой" символ, аналог * в Java
```

Примечание. При использовании пакета, имя которого начинается с префикса `scala.`, этот префикс можно опустить. Например, инструкция `import math._` эквивалентна инструкции `import scala.math._`, а вызов `math.sqrt(2)` эквивалентен вызову `scala.math.sqrt(2)`.

Инструкция `import` подробнее будет обсуждаться в главе 7. А пока просто используйте инструкцию `import packageName._`, когда вам потребуется импортировать какой-либо пакет.

В языке Scala отсутствуют статические методы, но в нем есть похожая особенность – *объекты-одиночки* (singleton objects), которые подробно будут рассматриваться в главе 6. Часто классы имеют *объекты-компаньоны* (companion object), чьи методы играют роль статических методов в Java. Например, объект-компаньон `BigInt` для класса `BigInt` имеет метод `probablePrime`, который генерирует случайное простое число с заданным количеством битов:

```
BigInt.probablePrime(100, scala.util.Random)
```

Попробуйте выполнить эту команду в REPL – вы получите число вида 1039447980491200275486540240713. Обратите внимание, что вызов BigInt.probablePrime напоминает вызов статического метода в Java.

> **Примечание.** Здесь Random – это объект-одиночка, генератор случайных чисел, определенный в пакете scala.util. Это одна из ситуаций, когда объекты-одиночки оказываются предпочтительнее классов. В Java часто встречается ошибка, когда для каждого случайного числа создается новый объект java.util.Random.

Вызовы методов без аргументов в языке Scala часто записываются без использования круглых скобок. Например, в API класса StringOps имеется метод distinct без (), возвращающий уникальные символы в строке. Его можно вызвать как

```
"Hello".distinct
```

Скобки обычно отсутствуют у методов без параметров, которые не изменяют объект. Подробнее об этом будет рассказываться в главе 5.

1.6. Метод apply

В языке принято использовать синтаксис, напоминающий вызовы функций. Например, если s – это строка, тогда выражение s(i) вернет *i*-й символ строки. (В C++ та же самая операция выполняется как s[i]; в Java – как s.charAt(i).) Попробуйте выполнить в REPL следующую команду:

```
"Hello"(4) // Вернет 'o'
```

Ее можно считать перегруженной формой оператора (). Однако в действительности она реализована как метод с именем apply. Например, в описании класса StringOps можно найти метод

```
def apply(n: Int): Char
```

То есть выражение "Hello"(4) фактически является краткой формой записи

```
"Hello".apply(4)
```

Заглянув в описание объекта-компаньона `BigInt`, можно увидеть методы `apply`, позволяющие преобразовывать строки или числа в объекты `BigInt`. Например, вызов

```
BigInt("1234567890")
```

является краткой формой записи

```
BigInt.apply("1234567890")
```

и возвращает новый объект `BigInt`, при этом *нет необходимости использовать ключевое слово new*. Например:

```
BigInt("1234567890") * BigInt("112358111321")
```

Использование метода `apply` объекта-компаньона является в языке Scala типичной идиомой конструирования объектов. Например, вызов `Array(1, 4, 9, 16)` вернет массив, созданный методом `apply` объекта-компаньона `Array`.

1.7. Scaladoc

Для исследования Java API программисты на Java пользуются системой генерации документации Javadoc. В Scala есть аналогичный инструмент – Scaladoc (рис. 1.1).

Навигация по документации в Scaladoc немного сложнее, чем в Javadoc. Классы в языке Scala обычно имеют намного больше вспомогательных методов, чем Java-классы. Некоторые методы используют пока незнакомые вам возможности. Наконец, описания некоторых особенностей рассказывают о том, как они реализованы, а не как ими пользоваться. (Разработчики языка Scala продолжают работать над улучшением внешнего вида Scaladoc, поэтому в будущем этот инструмент, возможно, станет более доступным для начинающих программистов.)

Ниже приводится несколько советов, касающихся навигации в Scaladoc, для тех, кто приступает к изучению языка.

Существует возможность работать с электронной документацией на сайте www.scala-lang.org/api, но лучше загрузить копию на странице www.scala-lang.org/downloads#api и установить ее локально.

В отличие от Javadoc, где список классов приводится в алфавитном порядке, классы в Scaladoc отсортированы по именам пакетов.

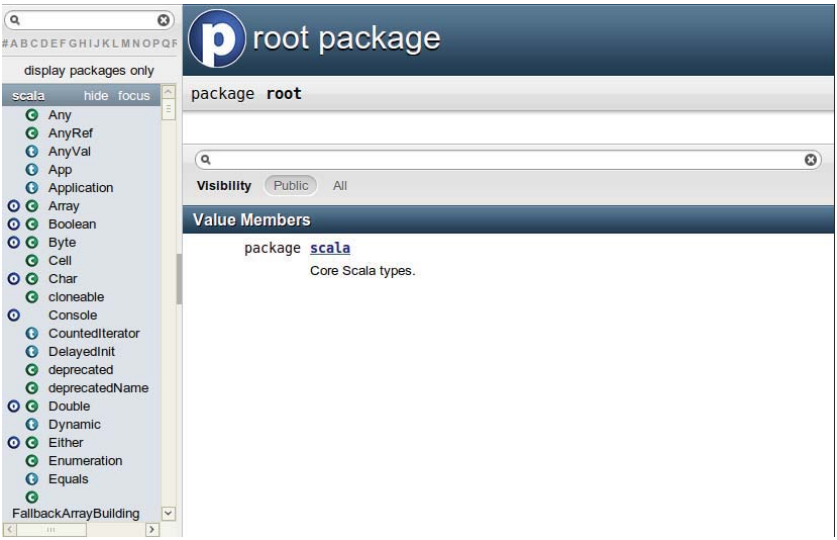


Рис. 1.1. Страница со статьей в окне Scaladoc

Если имя класса известно, а имя пакета нет, пользуйтесь фильтром в левом верхнем углу (рис. 1.2).

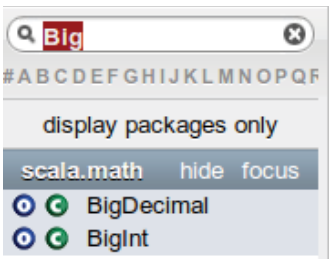


Рис. 1.2. Поле ввода фильтра в окне Scaladoc

Щелкните на значке **X**, чтобы очистить фильтр.

Обратите внимание на символы «O» и «C» рядом с именем каждого класса. Они позволяют исследовать класс (C) или объект-компаньон (O).

Документация Scaladoc может показаться слишком обширной. Поэтому примите следующие советы.

- Не забудьте заглянуть в описание классов `RichInt`, `RichDouble` и других, если потребуется выяснить, как работать с числовыми типами. Аналогично, если возникнут вопросы по работе со строками, загляните в описание класса `StringOps`.
- Математические функции сосредоточены в *пакете* `scala.math`, а не в каком-то классе.
- Иногда вам будут встречаться функции с забавными именами. Например, в `BigInt` имеется метод `unary_-`. Как будет показано

в главе 11, именно так определяется унарный оператор отрицания `-x`.

- Метод, отмеченный как `implicit`, реализует автоматическое преобразование. Например, объект `BigInt` автоматически выполняет преобразования из типов `int` и `long` в `BigInt`, когда это необходимо. Подробнее о неявных преобразованиях рассказывается в главе 21.
- Методы могут принимать функции в качестве параметров. Например, метод `count` класса `StringOps` требует функцию, возвращающую `true` или `false` для объекта `Char`, которая определяет, какие символы должны учитываться:

```
def count(p: (Char) => Boolean) : Int
```

Функции часто передаются методам в очень компактной форме записи. Например, вызов `s.count(_.isUpper)` вернет количество символов верхнего регистра. Более подробно стиль программирования будет рассматриваться в главе 12.

- Иногда вам будут встречаться классы, такие как `Range` или `Seq[Char]`. Как можно догадаться, они определяют диапазоны чисел и последовательности символов. Вы узнаете все об этих классах по мере погружения в изучение языка Scala.
- Пусть вас не смущает наличие огромного количества методов. В языке Scala принято предоставлять методы на все случаи жизни. Когда вам потребуется решить какую-либо проблему, просто поищите метод, который поможет в этом. Чаще всего вы будете обнаруживать методы, предназначенные для решения вашей задачи, а это значит, что вам не придется писать лишний программный код.
- Наконец, не пугайтесь, если столкнетесь с «колдовскими заклинаниями», такими как в классе `StringOps`:

```
def patch [B >: Char, That](from: Int, patch: GenSeq[B], replaced: Int)
(implicit bf: CanBuildFrom[String, B, That]): That
```

Просто игнорируйте их. Вы наверняка найдете более привычной версию:

```
def patch(from: Int, that: GenSeq[Char], replaced: Int): StringOps[A]
```

Если мысленно заменить `GenSeq[Char]` и `StringOps[A]` типом `String`, разобраться с этим методом будет намного проще. А кроме того, его легко можно опробовать в REPL:

```
"Harry".patch(1, "ung", 2) // Вернет "Hungry"
```

Упражнения

1. В окне Scala REPL введите `3.`, затем нажмите клавишу **Tab**. Какие методы могут быть вызваны?
2. В окне Scala REPL вычислите квадратный корень из 3, а затем возведите результат в квадрат. Насколько окончательный результат отличается от 3? (Подсказка: переменные `res` – ваши друзья.)
3. Переменные `res` – это значения `val` или настоящие переменные `var`?
4. Язык Scala позволяет умножать строки на числа – попробуйте выполнить выражение `"crazy" * 3` в REPL. Что получилось в результате? Где в Scaladoc можно найти ее описание?
5. Что означает выражение `10 max 2`? В каком классе определен метод `max`?
6. Используя число типа `BigInt`, вычислите 2^{1024}.
7. Что нужно импортировать для нахождения случайного простого числа вызовом метода `probablePrime(100, Random)` без использования каких-либо префиксов перед именами `probablePrime` и `Random`?
8. Один из способов создать файл или каталог со случайным именем состоит в том, чтобы сгенерировать случайное число типа `BigInt` и преобразовать его в систему счисления по основанию 36, в результате получится строка, такая как `"qsnvbevtomcj38o06kul"`. Отыщите в Scaladoc методы, которые можно было бы использовать для этого.
9. Как получить первый символ строки в языке Scala? А последний символ?
10. Что делают строковые функции `take`, `drop`, `takeRight` и `dropRight`? Какие преимущества и недостатки они имеют в сравнении с `substring`?

Глава 2. Управляющие структуры и функции

Темы, рассматриваемые в этой главе A1

- 2.1. Условные выражения.
- 2.2. Завершение инструкций.
- 2.3. Блочные выражения и присвоение.
- 2.4. Ввод и вывод.
- 2.5. Циклы.
- 2.6. Расширенные циклы for и генераторы for.
- 2.7. Функции.
- 2.8. Аргументы по умолчанию и именованные аргументы L1.
- 2.9. Переменное количество аргументов L1.
- 2.10. Процедуры.
- 2.11. Ленивые значения L1.
- 2.12. Исключения L1.
- Упражнения.

В этой главе вы узнаете, как реализовать условия, циклы и функции в языке Scala. Здесь вы встретитесь с фундаментальными различиями между Scala и другими языками программирования. В Java или C++ мы различаем *выражения* (expressions), такие как `3 + 4`, и *инструкции* (statements), например `if`. Выражение имеет значение; инструкция выполняет действие. В Scala практически все конструкции имеют значения, то есть являются выражениями. Это позволяет писать более короткие и более удобочитаемые программы.

Основные темы этой главы:

- выражение `if` имеет значение;
- блок имеет значение – значение последнего выражения;
- цикл `for` в Scala напоминает «расширенный» цикл `for` в Java;
- точки с запятой практически не нужны (в основном);
- тип `void` в Scala – это тип `Unit`;
- избегайте использования `return` в функциях;
- бойтесь пропажи `=` в определениях функций;

- исключения действуют так же, как в Java или C++, но для их перехвата используется синтаксис «сопоставления с образцом»;
- в Scala отсутствуют контролируемые исключения (checked exceptions).

2.1. Условные выражения

В языке Scala имеется конструкция `if/else` с тем же синтаксисом, что и в Java или C++. Однако в Scala `if/else` возвращает значение, а именно значение выражения, следующего за `if` или `else`. Например,

```
if (x > 0) 1 else -1
```

имеет значение `1` или `-1` в зависимости от значения `x`. Значение можно присвоить переменной:

```
val s = if (x > 0) 1 else -1
```

что равноценно следующей строке:

```
if (x > 0) s = 1 else s = -1
```

Однако первая форма предпочтительнее, потому что ее можно использовать для инициализации значения `val`. Во второй форме переменная `s` должна быть объявлена как `var`.

(Как уже упоминалось, точки с запятой почти не нужны в Scala – подробности смотрите в разделе 2.2 «Завершение инструкций».)

Для этих целей в Java и C++ имеется оператор `?:`. Выражение

```
x > 0 ? 1 : -1 // Java или C++
```

эквивалентно выражению `if (x > 0) 1 else -1` в Scala. Однако выражение `?:` не позволяет вставлять в него инструкции. Конструкция `if/else` в языке Scala объединяет возможности отдельных конструкций `if/else` и `?:` в Java и C++.

В Scala *каждое выражение имеет тип*. Например, выражение `if (x > 0) 1 else -1` имеет тип `Int`, потому что обе ветви имеют тип `Int`. Типом выражения, способного возвращать значения разных типов, такого как `if (x > 0) «positive» else -1`, является супертип для обеих

ветвей. В данном примере одна ветвь имеет тип `java.lang.String`, а другая – тип `Int`. Их общий *супертип* называется `Any`. (Подробнее об этом рассказывается в разделе 8.11 «Иерархия наследования в Scala».)

Если ветвь `else` отсутствует, как, например, ниже

```
if (x > 0) 1
```

может получиться, что инструкция `if` не будет иметь значения. Однако в Scala каждое выражение предполагает наличие *какого-либо* значения. Эта проблема элегантно была решена введением класса `Unit`, единственное значение которого записывается как `()`. Инструкция `if` без ветви `else` эквивалентна инструкции

```
if (x > 0) 1 else ()
```

Комбинацию `()` можно воспринимать как «пустое значение» и считать тип `Unit` аналогом типа `void` в Java или C++.

(Строго говоря, `void` означает отсутствие значения, тогда как `Unit` имеет единственное значение, означающее «нет значения». Если вам трудно осмыслить это, представьте разницу между пустым кошельком и кошельком с меткой «нет долларов».)

> **Примечание.** В языке Scala отсутствует инструкция `switch`, зато имеется более мощный механизм сопоставления с образцом, который будет представлен в главе 14. А пока просто используйте последовательности инструкций `if`.

> **Внимание.** В отличие от компилятора, оболочка REPL по умолчанию выполняет строку сразу после ее ввода. Например, если попытаться ввести
>
> ```
> if (x > 0) 1
> else if (x == 0) 0 else -1
> ```
>
> оболочка REPL выполнит `if (x > 0) 1` и выведет ответ. А следующая строка `else -1` собьет ее с толку.
> Если действительно необходимо разорвать строку перед `else`, используйте фигурные скобки[1]:

[1] Кстати, можно поступить проще, заключив многострочное выражение в фигурные скобки:

```
{
    if (x > 0) { 1
    } else if (x == 0) 0 else -1
}
```

– *Прим. ред.*

```
if (x > 0) { 1
} else if (x == 0) 0 else -1
```

Это необходимо делать только в REPL. В компилируемой программе парсер распознает предложение `else` на следующей строке.

Совет. Если потребуется скопировать блок кода в оболочку REPL, используйте режим вставки. Введите

```
:paste
```

Затем вставьте блок кода и нажмите комбинацию клавиш **Ctrl+K**[1]. После этого оболочка REPL выполнит вставленный блок целиком.

2.2. Завершение инструкций

В Java и C++ каждая инструкция завершается точкой с запятой. В Scala, так же как в JavaScript и некоторых других языках сценариев, точка с запятой не требуется, если она находится в конце строки. Точку с запятой можно также опускать перед `}`, `else` и везде, где из контекста очевидно, что достигнут конец инструкции.

Однако, если в одной строке находится несколько инструкций, они должны отделяться друг от друга точками с запятой. Например:

```
if (n > 0) { r = r * n; n -= 1 }
```

Точка с запятой необходима между выражениями `r = r * x` и `n -= 1`. Но после второго выражения точка с запятой не нужна, потому что за ним следует `}`.

Если потребуется перенести длинную инструкцию на другую строку, первая строка должна оканчиваться символом, который не может интерпретироваться как конец инструкции. Для этого прекрасно подойдет любой оператор:

```
s = s0 + (v - v0) * t + // Оператор + сообщает парсеру, что это не конец
0.5 * (a - a0) * t * t
```

На практике длинные выражения обычно необходимы для оформления вызова функции или метода, и в этом случае нет причин для беспокойства – после открывающей скобки `(` компилятор не будет

[1] В разных операционных системах для этой цели могут использоваться разные комбинации клавиш, например в Linux используется комбинация **Ctrl+D**. – *Прим. перев.*

воспринимать конец строки за конец инструкции, пока не встретит парную скобку).

По этой причине программисты на Scala часто предпочитают оформлять программный код в стиле Кернигана и Ритчи (Kernighan & Ritchie):

```
if (n > 0) {
     r = r * n
     n -= 1
}
```

Открывающая скобка { в конце строки явно свидетельствует, что инструкция будет продолжена на следующих строках.

Многим программистам, пришедшим из Java или C++, вначале неудобно без точек с запятой. Если вы предпочитаете ставить их – ставьте, это не возбраняется.

2.3. Блочные выражения и присвоение

В Java или C++ блочной инструкцией называется последовательность инструкций, заключенная в фигурные скобки {}. Используйте блочную инструкцию всякий раз, когда необходимо выполнить несколько операций в ветви условной инструкции или в теле цикла.

В языке Scala блок {} содержит последовательность *выражений* и сам считается выражением, результатом которого является результат последнего выражения.

Это может пригодиться для инициализации значений val, когда требуется выполнить более одного действия. Например:

```
val distance = { val dx = x - x0; val dy = y - y0; sqrt(dx * dx + dy * dy) }
```

Значением блока {} будет значение последнего выражения, выделенного жирным. Переменные dx и dy, необходимые только для хранения промежуточных результатов, надежно скрыты от остальной программы.

Инструкции присвоения в Scala не имеют значений, или, строго говоря, они имеют значение типа Unit. Напомню, что тип Unit эквивалентен типу void в Java и C++, с единственным значением, записываемым как ().

Блок, завершающийся инструкцией присвоения, такой как

```
{ r = r * n; n -= 1 }
```

имеет значение Unit. Это не проблема, но помните об этой особенности, когда будете определять функции – см. раздел 2.7 «Функции».

Поскольку инструкции присвоения возвращают значение Unit, их нельзя объединять в цепочки.

```
x = y = 1 // Неправильно
```

Значением выражения y = 1 является (), а в данном случае весьма маловероятно, что программист хотел присвоить Unit переменной x. (В Java и C++, напротив, инструкция присвоения возвращает присвоенное значение. В этих языках допускается объединять их в цепочки.)

2.4. Ввод и вывод

Чтобы вывести значение, используйте функцию print или println. Последняя добавляет символ перевода строки в конце. Например, пара инструкций

```
print("Answer: ")
println(42)
```

выведет то же, что и

```
println("Answer: " + 42)
```

Имеется также функция printf, принимающая строку описания формата в стиле языка C:

```
printf("Hello, %s! You are %d years old.\n", "Fred", 42)
```

Прочитать строку, введенную в консоли с клавиатуры, можно с помощью функции readLine. Чтобы прочитать число, логическое или символьное значение, используйте readInt, readDouble, readByte, readShort, readLong, readFloat, readBoolean или readChar. Метод readLine, в отличие от других, принимает строку приглашения к вводу:

```
val name = readLine("Your name: ")
print("Your age: ")
val age = readInt()
printf("Hello, %s! Next year, you will be %d.\n", name, age + 1)
```

2.5. Циклы

В Scala имеются такие же циклы `while` и `do`, как в Java и C++. Например:

```
while (n > 0) {
     r = r * n
     n -= 1
}
```

В Scala отсутствует прямой аналог цикла `for (инициализация; проверка; обновление)`. Если такой цикл потребуется, у вас есть два варианта на выбор – использовать цикл `while` или инструкцию `for`, как показано ниже:

```
for (i <- 1 to n)
     r = r * i
```

В главе 1 был показан метод `to` класса `RichInt`. Вызов `1 to n` вернет объект `Range`, представляющий числа в диапазоне от `1` до `n` (включительно).

Конструкция

```
for (i <- expr)
```

обеспечивает последовательное присвоение переменной `i` всех значений выражения `expr` справа от `<-`. Порядок присвоения зависит от типа выражения. Для коллекций, таких как `Range`, присвоит переменной `i` каждое значение по очереди.

> **Примечание.** Перед именем переменной в цикле `for` не требуется указывать `val` или `var`. Тип переменной соответствует типу элементов коллекции. Область видимости переменной цикла ограничивается телом цикла.

Для обхода элементов строки или массива зачастую нужно определить диапазон от 0 до $n - 1$. В этом случае используйте метод `until` вместо `to`. Он возвращает диапазон, не включающий верхнюю границу.

```
val s = "Hello"
var sum = 0
for (i <- 0 until s.length) // Последнее значение i в цикле s.length - 1
     sum += s(i)
```

В действительности в данном примере нет необходимости использовать индексы. Цикл можно выполнять непосредственно по символам:

```
var sum = 0
for (ch <- "Hello") sum += ch
```

В Scala циклы используются не так часто, как в других языках. Как будет показано в главе 12, значения в последовательностях зачастую можно обрабатывать, применяя функцию сразу ко всем элементам, для чего достаточно произвести единственный вызов метода.

Примечание. В Scala нет инструкций `break` или `continue` для преждевременного завершения цикла. Но как же быть, если это потребуется? Есть несколько вариантов:

1. Используйте логическую переменную управления циклом.
2. Используйте вложенные функции – при необходимости можно выполнить инструкцию `return` в середине функции.
3. Используйте метод `break` объекта `Breaks`:

```
import scala.util.control.Breaks._
breakable {
     for (...) {
          if (...) break; // выход из прерываемого блока
          ...
     }
}
```

Здесь передача управления за пределы цикла выполняется путем возбуждения и перехвата исключения, поэтому избегайте пользоваться этим механизмом, когда скорость выполнения критична.

2.6. Расширенные циклы for и for-генераторы

В предыдущем разделе была представлена базовая форма цикла `for`. Однако эта конструкция намного богаче, чем в Java или C++. В этом разделе описываются дополнительные возможности.

В заголовке цикла `for` допускается указывать несколько *генераторов* в форме *переменная* <- *выражение*, разделяя их точками с запятой. Например:

```
for (i <- 1 to 3; j <- 1 to 3) print((10 * i + j) + " ")
// Выведет 11 12 13 21 22 23 31 32 33
```

Каждый генератор может иметь *ограничитель* (guard) – логическое условие с предшествующим ему ключевым словом `if`:

```
for (i <- 1 to 3; j <- 1 to 3 if i != j) print((10 * i + j) + « «)
// Выведет 12 13 21 23 31 32
```

Обратите внимание на отсутствие точки с запятой перед `if`.

Допускается любое количество *определений*, вводящих переменные для использования внутри цикла:

```
for (i <- 1 to 3; from = 4 - i; j <- from to 3) print((10 * i + j) + « «)
// Выведет 13 22 23 31 32 33
```

Когда тело цикла начинается с инструкции `yield`, цикл будет конструировать коллекцию, добавляя в нее по одному элементу в каждой итерации:

```
for (i <- 1 to 10) yield i % 3
// Вернет Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

Такого рода циклы называют `for`-*генераторами* (`for`-comprehension).

Генерируемые коллекции по типу совместимы с первым генератором.

```
for (c <- "Hello"; i <- 0 to 1) yield (c + i).toChar
     // Вернет "HIeflmlmop"

for (i <- 0 to 1; c <- "Hello") yield (c + i).toChar
     // Вернет Vector('H', 'e', 'l', 'l', 'o', 'I', 'f', 'm', 'm', 'p')
```

> **Примечание.** При желании генераторы, ограничители и определения цикла `for` можно заключить в фигурные скобки и вместо точек с запятой использовать переводы строки:
>
> ```
> for { i <- 1 to 3
>       from = 4 - i
>       j <- from to 3 }
> ```

2.7. Функции

В дополнение к методам в Scala имеются функции. Метод оперирует объектом, а функция – нет. В C++ также есть функции, но в Java их приходится имитировать с помощью статических методов.

Чтобы определить функцию, нужно указать имя функции, *параметры* и *тело*, как показано ниже:

```
def abs(x: Double) = if (x >= 0) x else -x
```

Вы должны определить типы всех параметров. Однако, если функция не рекурсивная, определять тип возвращаемого значения не требуется. Компилятор Scala определяет тип возвращаемого значения по типу выражения справа от символа `=`.

Если тело функции содержит более одного выражения, используйте блок. Последнее выражение в блоке определяет значение, возвращаемое функцией. Например, следующая функция вернет значение `r` после цикла `for`.

```
def fac(n : Int) = {
    var r = 1
    for (i <- 1 to n) r = r * i
    r
}
```

В данном случае нет необходимости явно использовать ключевое слово `return`. В Scala допускается использовать `return` для немедленного выхода из функции, как в Java или C++, но такой способ редко используется.

> **Совет.** Даже при том, что нет ничего особенного в том, чтобы использовать `return` в именованных функциях (кроме семи лишних символов), лучше все-таки стараться привыкать жить без `return`. Очень скоро вы начнете использовать массу анонимных функций, но в них `return` не возвращает значение вызывающей программе, а выполняет выход во вмещающую, именованную функцию. Инструкцию `return` можно интерпретировать как своеобразную инструкцию `break` для функций и использовать ее только для преждевременного прерывания выполнения функций.

В рекурсивных функциях тип возвращаемого значения должен указываться обязательно. Например:

```
def fac(n: Int): Int = if (n <= 0) 1 else n * fac(n - 1)
```

Без определения типа возвращаемого значения компилятор Scala не сможет убедиться, что выражение `n * fac(n - 1)` имеет тип `Int`.

> **Примечание.** Некоторые языки программирования (такие как ML и Haskell) способны выводить тип рекурсивной функции с помощью алгоритма Хиндли-Милнера (Hindley-Milner). Однако он не надежен в объектно-ориентированных языках. Доработка алгоритма Хиндли-Милнера, чтобы обеспечить обработку подтипов, все еще находится на стадии исследований.

2.8. Аргументы по умолчанию и именованные аргументы L1

Существует возможность определять аргументы по умолчанию для функций, чтобы при вызове можно было не указывать их значения явно. Например:

```
def decorate(str: String, left: String = "[", right: String = "]") =
    left + str + right
```

Эта функция имеет два параметра, `left` и `right`, с аргументами по умолчанию `"["` и `"]"`.

Если вызвать ее как `decorate("Hello")`, она вернет `"[Hello]"`. Если вам не нравятся скобки по умолчанию, укажите свои собственные: `decorate("Hello", "<<<", ">>>")`.

Если при вызове число аргументов оказывается меньше числа параметров, применяются аргументы по умолчанию, начиная с конца. Например, в вызове `decorate("Hello", ">>>[")` будет использовано значение по умолчанию для параметра `right`, в результате будет получен результат: `">>>[Hello]"`.

При передаче аргументов можно также указывать имена параметров. Например:

```
decorate(left = "<<<", str = "Hello", right = ">>>")
```

В результате будет получена строка `"<<<Hello>>>"`. Обратите внимание, что именованные аргументы необязательно должны следовать в том же порядке, что и параметры.

Именованные аргументы могут сделать вызовы функций более удобочитаемыми. Они также могут пригодиться, когда функция имеет множество аргументов по умолчанию.

Допускается смешивать именованные и неименованные аргументы, указывая сначала неименованные:

```
decorate("Hello", right = "]<<<") // Вызовет decorate("Hello", "[", "]<<<")
```

2.9. Переменное количество аргументов L1

Иногда бывает удобно реализовать функцию, способную принимать переменное число аргументов. Следующий пример демонстрирует синтаксис объявления таких функций:

```
def sum(args: Int*) = {
    var result = 0
    for (arg <- args) result += arg
    result
}
```

Этой функции можно передать сколь угодно много аргументов.

```
val s = sum(1, 4, 9, 16, 25)
```

Функция принимает единственный параметр типа `Seq`, который будет обсуждаться в главе 13. А пока достаточно знать, что для доступа к каждому элементу можно использовать цикл `for`.

Такой функции нельзя передать уже имеющуюся последовательность значений. Следующий вызов оформлен неверно:

```
val s = sum(1 to 5) // Ошибка
```

Если вызывать функцию с одним аргументом, это должно быть единственное целое число, а не диапазон целых чисел. Чтобы исправить ошибку, необходимо сообщить компилятору, что параметр должен интерпретироваться как последовательность аргументов. Добавьте в конец `: _*`, как показано ниже:

```
val s = sum(1 to 5: _*)  // Интерпретировать 1 to 5
                         // как последовательность аргументов
```

Без этого синтаксиса не обойтись при определении рекурсивной функции с переменным числом аргументов:

```
def recursiveSum(args: Int*) : Int = {
     if (args.length == 0) 0
     else args.head + recursiveSum(args.tail : _*)
}
```

Здесь `head` – начальный элемент последовательности, а `tail` – все остальные элементы последовательности. Это также объект `Seq`, поэтому необходимо использовать `: _*` для преобразования его в последовательность аргументов.

> **Внимание.** При вызове Java-метода с переменным числом аргументов типа `Object`, такого как `PrintStream.printf` или `MessageFormat.format`, все простые типы необходимо будет преобразовать вручную. Например,
>
> ```
> val str = MessageFormat.format("The answer to {0} is {1}",
>      "everything", 42.asInstanceOf[AnyRef])
> ```
>
> Это относится к любым параметрам типа `Object`, и я упомянул об этом приеме здесь, потому что он наиболее часто встречается при использовании методов с переменным числом аргументов.

2.10. Процедуры

В Scala имеется специальная форма записи для объявления функций, не возвращающих значений. Если тело функции заключено в фигурные скобки *без предшествующего символа* `=`, тогда возвращаемое значение будет иметь тип `Unit`. Такие функции называются *процедурами*. Процедура не возвращает значение и вызывается исключительно ради побочного эффекта. Например, следующая процедура выведет строку в рамке

```
-------
|Hello|
-------
```

Поскольку процедура не возвращает никакого значения, символ `=` можно опустить.

```
def box(s : String) {          // Смотрите внимательнее: здесь нет  =
     val border = "-" * s.length + "--\n"
     println(border + "|" + s + "|\n" + border)
}
```

Некоторым (ко мне это не относится) не нравится краткий синтаксис объявления процедур, и они предлагают всегда явно указывать тип возвращаемого значения `Unit`:

```
def box(s : String): Unit = {
    ...
}
```

> **Внимание.** Краткий синтаксис объявления процедур может оказаться сюрпризом для программистов на Java и C++. Многие часто допускают ошибку, случайно забывая указать символ = в определениях функций. В этом случае в точке вызова функции будет получено сообщение об ошибке, говорящее, что тип `Unit` не допустим в данной точке.

2.11. Ленивые значения L1

Когда значение `val` объявляется как `lazy` (ленивое), его инициализация откладывается до первого обращения к нему. Например:

```
lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
```

(Операции с файлами будут обсуждаться в главе 9. А пока просто знайте, что этот вызов читает все символы из файла в строку.)

Если программа никогда не обратится к значению `words`, файл никогда не будет открыт. Чтобы убедиться в этом, попробуйте ввести эту команду в окне REPL, но укажите имя несуществующего файла. Вы увидите, что при выполнении инструкции никаких ошибок не произойдет. Однако при попытке обратиться к `words` будет выведено сообщение, что файл не найден.

Ленивые (lazy) значения удобно использовать, чтобы отложить дорогостоящие операции инициализации. Они также позволяют решить другие проблемы, связанные с инициализацией, такие как инициализация циклических зависимостей. Кроме того, они являются основой для разработки ленивых (lazy) структур данных – см. раздел 13.13 «Потоки».

Ленивые (lazy) значения можно считать чем-то средним между `val` и `def`. Сравните:

```
val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
    // Вычисляется немедленно, в момент определения words
```

```
lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
     // Вычисляется при первом обращении к words
def words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
     // Вычисляется всякий раз, когда происходит обращение к words
```

Примечание. Откладывание инициализации не дается бесплатно. Всякий раз, когда выполняется обращение к ленивому (lazy) значению, вызывается метод, который проверяет, потокобезопасным способом, было ли инициализировано значение.

2.12. Исключения

Исключения в языке Scala действуют так же, как в Java или C++. Когда возбуждается исключение, например

```
throw new IllegalArgumentException("x should not be negative")
```

вычисления прерываются, и среда времени выполнения пытается отыскать обработчик исключения `IllegalArgumentException`. Затем управление передается самому внутреннему такому обработчику. Если обработчик не будет найден, программа завершится.

Как и в Java, объект возбуждаемого исключения должен принадлежать подклассу `java.lang.Throwable`. Однако, в отличие от Java, в Scala отсутствуют «контролируемые» (checked) исключения – вам никогда не придется объявлять, что некоторый метод или функция может возбуждать исключение.

Примечание. В Java «контролируемые» (checked) исключения контролируются на этапе компиляции. Если метод может возбуждать исключение `IOException`, необходимо явно объявить это. Такой подход вынуждает программиста думать, где предусмотреть обработку этого исключения, что само по себе является достойным похвалы. К сожалению, это может приводить к созданию монструозных сигнатур методов, таких как `void doSomething() throws IOException, InterruptedException, ClassNotFoundException`. Многие программисты на Java терпеть не могут эту особенность и в результате либо перехватывают исключения слишком рано, либо используют слишком обобщенные классы исключений. Создатели Scala решили отказаться от контролируемых исключений, признавая, что полная проверка на этапе компиляции не всегда оправдана.

Выражение `throw` имеет специальный тип `Nothing`, что может пригодиться в выражениях `if/else`. Если одна из ветвей имеет тип

Nothing, типом всего выражения if/else становится тип другой ветви. Например:

```
if (x >= 0) { sqrt(x)
} else throw new IllegalArgumentException("x should not be negative")
```

Первая ветвь имеет тип Double, вторая – тип Nothing. Таким образом, все выражение if/else также имеет тип Double.

Синтаксис перехвата исключений основан на синтаксисе сопоставления с образцами (глава 14).

```
try {
    process(new URL("http://horstmann.com/fred-tiny.gif"))
} catch {
    case _: MalformedURLException => println("Bad URL: " + url)
    case ex: IOException => ex.printStackTrace()
}
```

Как и в Java или C++, более обобщенный тип исключений должен следовать за более специализированными.

Обратите внимание, что вместо имени переменной можно использовать _, если потом эта переменная не нужна.

Инструкция try/finally позволяет освободить занятые ресурсы независимо от наличия или отсутствия исключения. Например:

```
var in = new URL("http://horstmann.com/fred.gif").openStream()
try {
    process(in)
} finally {
    in.close()
}
```

Предложение finally выполнится независимо от того, возбудила ли исключение функция process. Поток чтения всегда будет закрыт.

Этот далеко не однозначный пример и поднимает несколько проблем.

- Что, если исключение возбудит конструктор URL или метод openStream? Тогда вход в блок try никогда не будет выполнен и никогда не будет выполнено предложение finally. В этом случае переменная in никогда не будет инициализирована, поэтому бессмысленно пытаться вызвать ее метод close.

- Почему бы не поместить `val in = new URL(...).openStream()` внутрь блока `try`? Тогда область видимости `in` не будет распространяться на предложение `finally`.
- Что, если исключение возбудит метод `in.close()`? Тогда исключение выйдет за пределы инструкции, замещая предыдущее. (Те же проблемы наблюдаются и в Java, и это не очень хорошо. В идеале старое исключение следовало бы присоединить к новому.)

Обратите внимание, что `try/catch` и `try/finally` имеют разные цели. Инструкция `try/catch` обрабатывает исключения, а инструкция `try/finally` предпринимает некоторые действия (обычно – освобождение ресурсов), если исключение не будет обработано. Их можно объединить в одну инструкцию `try/catch/finally` statement:

```
try { ... } catch { ... } finally { ... }
```

Это то же самое, что и

```
try { try { ... } catch { ... } } finally { ... }
```

Однако на практике эта комбинация редко оказывается полезной.

Упражнения

1. Сигнум числа равен 1, если число положительное, -1 – если отрицательное, и 0 – если оно равно нулю. Напишите функцию, вычисляющую это значение.
2. Какое значение возвращает пустой блок `{}`? Каков его тип?
3. Придумайте ситуацию, когда присвоение `x = y = 1` будет допустимым в Scala. (Подсказка: выберите подходящий тип для `x`.)
4. Напишите на языке Scala цикл, эквивалентный циклу на языке Java

```
for (int i = 10; i >= 0; i--) System.out.println(i);
```

5. Напишите процедуру `countdown(n: Int)`, которая выводит числа от *n* до 0.
6. Напишите цикл `for` для вычисления произведения кодовых пунктов Юникода всех букв в строке. Например, произведение символов в строке `"Hello"` равно `9415087488L`.

7. Решите предыдущее упражнение без применения цикла. (Подсказка: загляните в описание класса `StringOps` в Scaladoc.)
8. Напишите функцию `product(s : String)`, вычисляющую произведение, как описано в предыдущих упражнениях.
9. Сделайте функцию из предыдущего упражнения рекурсивной.
10. Напишите функцию, вычисляющую x^n, где n – целое число. Используйте следующее рекурсивное определение:
    - $x^n = y^2$, если n – четное и положительное число, где $y = x^{n/2}$.
    - $x^n = x \cdot x^{n-1}$, если n – нечетное и положительное число.
    - $x^0 = 1$.
    - $x^n = 1/x^{-n}$, если n – отрицательное число.

    Не используйте инструкцию `return`.

Глава 3. Работа с массивами

Темы, рассматриваемые в этой главе A1

- 3.1. Массивы фиксированной длины.
- 3.2. Массивы переменной длины: буферы.
- 3.3. Обход элементов массивов и буферов.
- 3.4. Преобразование массивов.
- 3.5. Типичные алгоритмы.
- 3.6. Расшифровываем Scaladoc.
- 3.7. Многомерные массивы.
- 3.8. Взаимодействие с Java.
- Упражнения.

В этой главе вы узнаете, как работать с массивами в Scala. Программисты на Java и C++ обычно выбирают массивы или родственные им структуры данных (такие как списки или векторы), когда требуется собрать воедино множество элементов. В Scala существуют другие возможности (см. главу 13), но сейчас, как мне кажется, вам не терпится погрузиться в массивы.

Основные темы этой главы:

- используйте тип `Array` для представления массивов фиксированной длины и тип `ArrayBuffer` для представления массивов переменной длины;
- не используйте `new`, когда создаете массив из начальных значений;
- используйте скобки `()` для доступа к элементам массива;
- используйте `for (elem <- arr)` для обхода элементов;
- используйте `for (elem <- arr if ...) ... yield ...` для преобразования в новый массив;
- массивы Scala и Java являются совместимыми; для совместимости с массивами `ArrayBuffer` используйте `scala.collection.JavaConversions`.

3.1. Массивы фиксированной длины

Если вам нужен массив, длина которого не будет меняться, используйте тип Array. Например:

```
val nums = new Array[Int](10)
     // Массив с десятью целыми числами, инициализированными нулями
val a = new Array[String](10)
     // Массив с десятью элементами, инициализированными значением null
val s = Array("Hello", "World")
     // Массив типа Array[String] с длиной 2 - тип выводится компилятором
     // Заметьте: при наличии начальных значений слово new не используется
s(0) = "Goodbye"
     // Массив Array("Goodbye", "World")
     // Для доступа к элементам используйте () вместо  []
```

Внутри JVM тип Array реализован как Java-массив. Массивы в предыдущем примере имеют тип java.lang.String[] внутри JVM. Массив с элементами типов Int, Double и других, эквивалентных простым типам в Java, является массивом примитивного типа. Например, Array(2, 3, 5, 7, 11) – это int[] в JVM.

3.2. Массивы переменной длины: буферы

В Java имеется тип ArrayList, а в C++ – тип vector, соответствующие массивам, которые могут увеличиваться и уменьшаться по требованию. В языке Scala для этого есть тип ArrayBuffer.

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
     // Или new ArrayBuffer[Int]
     // Пустой буфер, подготовленный для хранения целых чисел
b += 1
     // ArrayBuffer(1)
     // Добавление элемента в конец с помощью +=
b += (1, 2, 3, 5)
     // ArrayBuffer(1, 1, 2, 3, 5)
     // Добавление в конец нескольких элементов, заключенных в скобки
b ++= Array(8, 13, 21)
     // ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
     // С помощью оператора ++= можно добавить в конец любую коллекцию
```

```
b.trimEnd(5)
     // ArrayBuffer(1, 1, 2)
     // Удаление последних пяти элементов
```

Добавление или удаление элементов в конце массива – достаточно эффективная («с постоянным временем выполнения») операция.

Имеется также возможность вставлять и удалять элементы в произвольных позициях, но эти операции менее эффективны – все элементы правее указанной позиции должны сдвигаться. Например:

```
b.insert(2, 6)
     // ArrayBuffer(1, 1, 6, 2)
     // Вставка элемента в позицию с индексом 2
b.insert(2, 7, 8, 9)
     // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
     // Можно вставить любое количество элементов
b.remove(2)
     // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
     // ArrayBuffer(1, 1, 2)
     // Второй параметр определяет количество удаляемых элементов
```

Иногда нужно создать массив `Array`, но количество элементов заранее неизвестно. В этом случае сначала создайте буфер, а затем вызовите:

```
b.toArray
     // Array(1, 1, 2)
```

Обратное преобразование массива в буфер выполняется вызовом `a.toBuffer`.

3.3. Обход массивов и буферов

В Java и C++ имеется ряд синтаксических различий между массивами и списками/векторами. Язык Scala в этом отношении более однороден. В большинстве случаев можно использовать один и тот же код для работы с обоими типами.

Ниже показано, как выполнить обход элементов массива или буфера в цикле `for`:

```
for (i <- 0 until a.length)
     println(i + ": " + a(i))
```

Переменная `i` получает значения от `0` до `a.length - 1`.

Метод `until` класса `RichInt` возвращает все числа в указанном диапазоне, не включая верхней границы. Например:

```
0 until 10
    // Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Обратите внимание, что `0 until 10` фактически является вызовом метода `0.until(10)`.

Конструкция

```
for (i <- range)
```

обеспечивает последовательное присвоение переменной `i` всех значений диапазона. В данном случае переменная цикла `i` будет получать значения `0`, `1` и до (но не включая) `a.length`.

Чтобы обойти все элементы с четными индексами, можно использовать конструкцию

```
0 until (a.length, 2)
    // Range(0, 2, 4, ...)
```

Чтобы обойти элементы в обратном порядке, начинайте с конца массива:

```
(0 until a.length).reverse
    // Range(..., 2, 1, 0)
```

Если в теле цикла индексы не нужны, можно реализовать обход элементов непосредственно:

```
for (elem <- a)
    println(elem)
```

Эта конструкция очень напоминает «расширенный» цикл `for`[1] в Java или «основанный на диапазоне» цикл `for` в C++. Переменной `elem` последовательно присваиваются значения `a(0)`, `a(1)` и т. д.

[1] Имеется в виду цикл `foreach`. – *Прим. перев.*

3.4. Преобразование массивов

В предыдущих разделах было показано, что работа с массивами очень напоминает работу с массивами в Java или C++. Но в Scala имеются дополнительные возможности. Очень легко можно взять массив (или буфер) и преобразовать его некоторым способом. Такие преобразования не изменяют оригинального массива, а создают новый.

Для этого можно воспользоваться `for`-генератором (`for`-comprehension), как показано ниже:

```
val a = Array(2, 3, 5, 7, 11)
val result = for (elem <- a) yield 2 * elem
    // результат: Array(4, 6, 10, 14, 22)
```

Цикл `for (...) yield` создаст новую коллекцию того же типа, что и оригинал. Если в выражении участвует массив, на выходе получается другой массив. Если в выражении участвует буфер, именно буфер и создаст цикл `for (...) yield`.

Получившийся массив будет содержать значения выражения, следующего за `yield`, вычисляемого в каждой итерации.

Часто при обходе коллекции бывает необходимо обработать элементы, только соответствующие определенному условию. Этого можно добиться, добавив *ограничитель* (guard): выражение `if` внутри `for`. Следующий пример удваивает значения четных элементов и пропускает нечетные:

```
for (elem <- a if a % 2 == 0) yield 2 * elem
```

Имейте в виду, что в результате получается новая коллекция, – оригинальная коллекция не затрагивается.

> **Примечание.** Ту же задачу можно решить иначе:
>
> `a.filter(_ % 2 == 0).map(2 * _)`
>
> или даже
>
> `a.filter { _ % 2 == 0 } map { 2 * _ }`
>
> Некоторые программисты с опытом функционального программирования предпочтут использовать методы `filter` и `map`. Однако это лишь вопрос стиля – цикл `for` выполняет в точности ту же работу. Используйте тот способ, который считаете более простым.

Рассмотрим следующий пример. Дана последовательность целых чисел, из которой требуется удалить все отрицательные числа, кроме первого. Традиционное решение предполагает использование флага, который сбрасывается после обнаружения первого отрицательного числа.

```
var first = true
var n = a.length
var i = 0
while (i < n) {
     if (a(i) >= 0) i += 1
     else {
          if (first) { first = false; i += 1 }
          else { a.remove(i); n -= 1 }
     }
}
```

Однако такое решение выглядит некрасиво. Удалять элементы из буфера довольно неэффективно. Намного лучше просто скопировать неотрицательные числа в начало.

Сначала соберем все индексы:

```
var first = false
val indexes = for (i <- 0 until a.length if first || a(i) >= 0) yield {
     if (a(i) < 0) first = false; i
}
```

Затем переместим элементы в новые позиции и отсечем конец:

```
for (j <- 0 until indexes.length) a(j) = a(indexes(j))
a.trimEnd(a.length - indexes.length)
```

Обратите внимание, что гораздо лучше собрать все индексы вместе, чем просматривать их по одному.

3.5. Типичные алгоритмы

Часто утверждается, что значительную долю вычислений составляют суммирование и сортировка. К счастью, для решения таких задач в Scala имеются встроенные функции.

```
Array(1, 7, 2, 9).sum
     // 19
     // Может применяться и к буферам ArrayBuffer
```

Чтобы задействовать метод sum, элементы должны быть числового типа: целочисленные, вещественные или BigInteger/BigDecimal.

Аналогично методы min и max возвращают наименьший и наибольший элементы массива или буфера.

```
ArrayBuffer("Mary", "had", "a", "little", "lamb").max
    // "little"
```

Метод sorted сортирует массив или буфер и возвращает *новый* отсортированный массив или буфер, не изменяя оригинала:

```
val b = ArrayBuffer(1, 7, 2, 9)
val bSorted = b.sorted(_ < _)
    // b не изменился; bSorted - это ArrayBuffer(1, 2, 7, 9)
```

В качестве параметра методу передается функция сравнения – синтаксис описывается в главе 12.

Имеется также возможность отсортировать массив (но не буфер) на месте:

```
val a = Array(1, 7, 2, 9)
scala.util.Sorting.quickSort(a)
    // a - теперь Array(1, 2, 7, 9)
```

Чтобы использовать методы min, max и quickSort, типы элементов должны поддерживать операцию сравнения. Эту операцию поддерживают числа, строки и другие типы с трейтом (trait) Ordered.

Наконец, если потребуется вывести содержимое массива или буфера, метод mkString позволит указать разделитель для вывода между элементами. Второй вариант в примере ниже имеет параметры для префикса и окончания. Например:

```
a.mkString(" and ")
    // "1 and 2 and 7 and 9"
a.mkString("<", ",", ">")
    // "<1,2,7,9>"
```

Сравните с методом toString:

```
a.toString
    // "[I@b73e5"
    // Это - бесполезный метод toString из Java
```

```
b.toString
    // "ArrayBuffer(1, 7, 2, 9)"
    // Метод toString сообщает тип, что может пригодиться при отладке
```

3.6. Расшифровываем Scaladoc

Массивы и буферы имеют массу полезных методов, и было бы неплохо исследовать документацию Scala, чтобы получить представление об имеющихся возможностях.

> **Примечание.** Методы для класса `Array` перечислены в разделе `ArrayOps`. Технически массив преобразуется в объект `ArrayOps` перед любой применяемой операцией.

Поскольку Scala обладает более богатой системой типов, чем Java, в документации для Scala могут встретиться довольно странные синтаксические конструкции. К счастью, для успешной работы не требуется понимать все тонкости системы типов. Используйте табл. 3.1 как «дешифровальное кольцо».

Таблица 3.1. Дешифровальное кольцо для Scaladoc

Scaladoc	Объяснение
`def append(elems: A*): Unit`	Этот метод принимает *ноль или более* аргументов типа `A`. Например, `b.append(1, 7, 2, 9)` добавит четыре элемента в конец `b`
`def appendAll(xs: TraversableOnce[A]): Unit`	Параметр `xs` может быть любой коллекцией с трейтом (trait) `TraversableOnce`, наиболее типичным трейтом в иерархии коллекций в языке Scala. Другими распространенными трейтами, с которыми можно встретиться в Scaladoc, являются `Traversable` и `Iterable`. Все коллекции в языке Scala реализуют эти трейты, и различия между ними для пользователей библиотек носят академический характер. Просто при встрече считайте их признаком «некоторая коллекция». Однако трейт `Seq` требует реализации доступа к элементам по их индексам. Интерпретируйте его как «массив, список или строка»

Таблица 3.1. Дешифровальное кольцо для Scaladoc

Scaladoc	Объяснение
`def count(p: (A) => Boolean): Int`	Этот метод принимает *предикат*, функцию преобразования из типа `A` в тип `Boolean`. Подсчитывает количество элементов, для которых функция вернет `true`. Например, вызов `a.count(_ > 0)` вернет количество положительных элементов
`def += (elem: A): ArrayBuffer.this.type`	Этот метод возвращает `this`, что позволяет составлять цепочки вызовов, например: `b += 4 -= 5`. При работе с буфером `ArrayBuffer[A]` этот метод можно представлять как `def += (elem: A) : ArrayBuffer[A]`. Если кто-то сформирует подкласс класса `ArrayBuffer`, метод `+=` вернет объект этого подкласса
`def copyToArray[B >: A] (xs: Array[B]): Unit`	Обратите внимание, что функция копирует буфер `ArrayBuffer[A]` в массив `Array[B]`. Здесь `B` может быть *супертипом* `A`. Например, метод может скопировать `ArrayBuffer[Int]` в `Array[Any]`. При первом знакомстве просто игнорируйте фрагмент `[B >: A]` и замените `B` на `A`
`def max[B >: A] (implicit cmp: Ordering[B]): A`	`A` должен иметь супертип `B`, для которого существует «неявный» («implicit») объект типа `Ordering[B]`. Такие объекты упорядочения существуют для чисел, строк и других типов с трейтом `Ordered`, а также для классов, реализующих Java-интерфейс `Comparable`
`def padTo[B >: A, That](len: Int, elem: B) (implicit bf: CanBuildFrom [ArrayBuffer[A], B, That]): That`	Это объявление встречается, когда метод создает новую коллекцию. Пропустите ее и посмотрите на более простую альтернативу, в данном случае `def padTo (len: Int, elem: A) : ArrayBuffer[A]` В будущей версии Scaladoc такие объявления будут скрыты

3.7. Многомерные массивы

Как и в языке Java, многомерные массивы в Scala реализованы как массивы массивов. Например, двухмерный массив значений `Double` имеет тип `Array[Array[Double]]`. Создаются такие массивы с помощью метода `ofDim`:

```
val matrix = Array.ofDim[Double](3, 4) // Три строки, четыре столбца
```

Для доступа к элементам такого массива следует использовать две пары скобок:

```
matrix(row)(column) = 42
```

Имеется возможность создавать массивы со строками разной длины:

```
val triangle = new Array[Array[Int]](10)
for (i <- 0 until triangle.length)
     triangle(i) = new Array[Int](i + 1)
```

3.8. Взаимодействие с Java

Поскольку массивы в Scala реализованы как Java-массивы, их свободно можно передавать между программным кодом на Java и Scala.

При использовании Java-метода, принимающего или возвращающего `java.util.List`, в программном коде Scala можно было бы использовать Java-массив `ArrayList`, но это не самое лучшее решение. Вместо этого можно импортировать методы неявного преобразования из `scala.collection.JavaConversions`. После этого можно будет использовать буферы Scala в своем коде, которые автоматически будут обертываться списками Java при вызове Java-метода.

Например, класс `java.lang.ProcessBuilder` имеет конструктор с параметром `List<String>`. Ниже показано, как можно его вызвать из Scala:

```
import scala.collection.JavaConversions.bufferAsJavaList
import scala.collection.mutable.ArrayBuffer
val command = ArrayBuffer("ls", "-al", "/home/cay")
val pb = new ProcessBuilder(command) // Из Scala в Java
```

Буфер из языка Scala обертывается объектом Java-класса, реализующего интерфейс `java.util.List`.

Наоборот, когда Java-метод возвращает `java.util.List`, его можно автоматически преобразовать в `Buffer`:

```
import scala.collection.JavaConversions.asScalaBuffer
import scala.collection.mutable.Buffer
val cmd : Buffer[String] = pb.command() // из Java в Scala
```

```
// Нельзя использовать тип ArrayBuffer – обернутый объект гарантированно
// будет иметь тип Buffer
```

Если Java-метод вернет обернутый буфер Scala, тогда метод неявного преобразования извлечет оригинальный объект. В данном примере `cmd == command`.

Упражнения

1. Напишите фрагмент кода, который записывает в массив `a` целые числа в диапазоне от 0 (включительно) до `n` (исключая его).
2. Напишите цикл, меняющий местами смежные элементы в массиве целых чисел. Например, `Array(1, 2, 3, 4, 5)` должен стать `Array(2, 1, 4, 3, 5)`.
3. Повторите предыдущее упражнение, но создайте новый массив с переставленными элементами. Используйте выражение `for/yield`.
4. Дан массив целых чисел, создайте новый массив, в котором сначала будут следовать положительные значения из оригинального массива, в оригинальном порядке, а за ними отрицательные и нулевые значения, тоже в оригинальном порядке.
5. Как бы вы вычислили среднее значение элементов массива `Array[Double]`?
6. Как бы вы переупорядочили элементы массива `Array[Int]` так, чтобы они следовали в обратном отсортированном порядке? Как бы вы сделали то же самое с буфером `ArrayBuffer[Int]`?
7. Напишите фрагмент программного кода, выводящий значения всех элементов из массива, кроме повторяющихся. (Подсказка: загляните в Scaladoc.)
8. Перепишите пример в конце раздела 3.4 «Преобразование массивов». Соберите индексы отрицательных элементов, расположите их в обратном порядке, отбросьте последний индекс и вызовите `a.remove(i)` для каждого индекса. Сравните эффективность этого подхода с двумя подходами, представленными в разделе 3.4.
9. Создайте коллекцию всех часовых поясов, возвращаемых методом `java.util.TimeZone.getAvailableIDs` для Америки. Отбросьте префикс `"America/"` и отсортируйте результат.
10. Импортируйте `java.awt.datatransfer._` и создайте объект типа `SystemFlavorMap` вызовом

```
val flavors = SystemFlavorMap.getDefaultFlavorMap().
asInstanceOf[SystemFlavorMap]
```

Затем вызовите метод `getNativesForFlavor` с параметром `DataFlavor.imageFlavor` и получите возвращаемое значение как буфер Scala. (Зачем нужен этот непонятный класс? Довольно сложно найти пример использования `java.util.List` в стандартной библиотеке Java.)

Глава 4. Ассоциативные массивы и кортежи

Темы, рассматриваемые в этой главе A1

- 4.1. Конструирование отображений.
- 4.2. Доступ к значениям в отображениях.
- 4.3. Изменение значений в отображениях.
- 4.4. Обход элементов отображений.
- 4.5. Сортированные отображения.
- 4.6. Взаимодействие с Java.
- 4.7. Кортежи.
- 4.8. Функция zip.
- Упражнения.

Классическая программистская поговорка гласит: «Если вам дадут выбрать только одну структуру данных, выбирайте хеш-таблицы». *Хеш-таблицы*, или, говоря более обобщенно, *ассоциативные массивы*, являются одними из самых гибких структур данных. Как будет показано в этой главе, в языке Scala они особенно просты в использовании.

Ассоциативные массивы (`Map`) – это коллекции пар ключ/значение. Кроме того, в Scala определено общее понятие *кортежа* – агрегата из n объектов, необязательно одного типа. Пара – это обычный кортеж с $n = 2$. Кортежи с успехом могут использоваться везде, где необходимы агрегаты из двух или более значений. Их синтаксис коротко обсуждается в конце главы.

Основные темы этой главы:

- Scala имеет удобный синтаксис для создания отображений, обращений к ним и обхода их элементов;
- выбор между изменяемыми и неизменяемыми ассоциативными массивами;
- по умолчанию ассоциативные массивы создаются в виде хеш-таблиц, но есть возможность создавать их в виде деревьев;

- ассоциативные массивы легко можно передавать между Scala и Java;
- кортежи удобно использовать для агрегирования значений.

4.1. Конструирование отображений

Создать ассоциативный массив можно следующим образом:

```
val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
```

В результате будет создано неизменяемое отображение `Map[String, Int]`. Если потребуется создать изменяемое отображение:

```
val scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)
```

Если потребуется создать пустые ассоциативные массивы, необходимо выбрать реализацию ассоциативного массива и указать типы ключей и значений:

```
val scores = new scala.collection.mutable.HashMap[String, Int]
```

Ассоциативные массивы в языке Scala являются коллекциями *пар*. Пара – это просто группа из двух значений, необязательно одного типа, например: `("Alice", 10)`.

Пара создается оператором `->`. Значением выражения

```
"Alice" -> 10
```

является

```
("Alice", 10)
```

С тем же успехом ассоциативный массив можно создать так:

```
val scores = Map(("Alice", 10), ("Bob", 3), ("Cindy", 8))
```

Оператор `->` лишь немного проще воспринимается глазом, чем скобки. Но скобки создают впечатление, что ассоциативный массив является своеобразной функцией, которая отображает ключи в значения. Разница лишь в том, что функция вычисляет значения, а ассоциативный массив просто отыскивает их.

4.2. Доступ к значениям в ассоциативных массивах

В Scala аналогия между функциями и ассоциативными массивами особенно полная, потому что для поиска значений ключей используется нотация `()`.

```
val bobsScore = scores("Bob") // Подобно scores.get("Bob") в Java
```

Если ассоциативный массив не содержит искомого ключа, возбуждается исключение.

Проверить наличие ключа можно вызовом метода `contains`:

```
val bobsScore = if (scores.contains("Bob")) scores("Bob") else 0
```

Поскольку эта комбинация вызовов используется очень часто, для нее была создана сокращенная форма записи:

```
val bobsScore = scores.getOrElse("Bob", 0)
    // Если ассоциативный массив содержит ключ "Bob",
    // вернет значение; иначе вернет 0
```

Наконец, вызов `map.get(key)` вернет объект `Option`, имеющий либо значение типа `Some(`*значение ключа*`)`, либо `None`. Класс `Option` обсуждается в главе 14.

4.3. Изменение значений в ассоциативных массивах

Изменяемые ассоциативные массивы позволяют изменять значения ключей или добавлять новые с помощью `()` слева от знака `=`:

```
scores("Bob") = 10
    // Изменит значение существующего ключа "Bob" (предполагается, что
    // scores – изменяемый ассоциативный массив)
scores("Fred") = 7
    // Добавит новую пару ключ/значение в scores (предполагается, что
    // scores – изменяемый ассоциативный массив)
```

Для добавления нескольких ассоциаций можно также использовать оператор `+=`:

```
scores += ("Bob" -> 10, "Fred" -> 7)
```

Удалить ключ и ассоциированное с ним значение можно с помощью оператора -=:

```
scores -= "Alice"
```

Изменить *неизменяемый ассоциативный массив* невозможно, зато можно создать новый ассоциативный массив, содержащий необходимые изменения:

```
val newScores = scores + ("Bob" -> 10, "Fred" -> 7)
    // Новый ассоциативный массив
```

Ассоциативный массив `newScores` содержит те же ассоциации, что и `scores`, за исключением ключа `"Bob"`, значение которого было изменено, и нового ключа `"Fred"`.

Вместо сохранения результата в новом значении можно обновить переменную `var`:

```
var scores = ...
scores = scores + ("Bob" -> 10, "Fred" -> 7)
```

Аналогично, чтобы удалить ключ из неизменяемого ассоциативного массива, можно воспользоваться оператором -, возвращающим новый ассоциативный массив без ключа:

```
scores = scores - "Alice"
```

Можно было бы подумать, что создание новых ассоциативных массивов является неэффективным приемом, но это не так. Старый и новый ассоциативные массивы разделяют большую часть своей структуры. (Благодаря своей неизменяемости.)

4.4. Обход элементов ассоциативных массивов

Следующий, удивительно простой цикл выполняет обход всех пар ключ/значение в ассоциативном массиве:

```
for ((k, v) <- map) process k and v
```

Необычное здесь, что циклы `for` в Scala позволяют использовать сопоставление с образцом. (Подробнее об этом рассказывается в главе 14.) Таким способом можно получить ключ и значение каждой пары в ассоциативном массиве без лишних вызовов методов.

Если по каким-то причинам необходимо просто обойти ключи или значения, используйте методы `keySet` и `values`, как в Java. Метод `values` возвращает объект `Iterable`, который можно использовать в цикле `for`.

```
scores.keySet // Множество, такое как Set("Bob", "Cindy", "Fred", "Alice")
for (v <- scores.values) println(v)  // Выведет 10 8 7 10,
                                     // возможно в другом порядке
```

Чтобы получить инвертированный ассоциативный массив, просто создайте новый, поменяв местами ключи и значения:

```
for ((k, v) <- map) yield (v, k)
```

4.5. Сортированные ассоциативные массивы

При работе с ассоциативными массивами необходимо выбрать реализацию – в виде *хеш-таблицы* или в виде *сбалансированного дерева*. По умолчанию Scala создает хеш-таблицы. В отсутствие хорошей хеш-функции для ключей или при необходимости хранить ключи в отсортированном порядке можно выбрать реализацию ассоциативного массива в виде дерева.

Чтобы создать неизменяемое дерево вместо хеш-таблицы, используйте

```
val scores = scala.collection.immutable.SortedMap("Alice" -> 10,
     "Fred" -> 7, "Bob" -> 3, "Cindy" -> 8)
```

К сожалению, в Scala (по крайней мере, в версии 2.9) отсутствует реализация изменяемых деревьев. Лучшим выбором в данной ситуации является `TreeMap` из Java, как описывается в главе 13.

Совет. Если потребуется выполнить обход ключей в порядке их добавления, используйте `LinkedHashMap`. Например:

```
val months = scala.collection.mutable.LinkedHashMap("January" -> 1,
     "February" -> 2, "March" -> 3, "April" -> 4, "May" -> 5, ...)
```

4.6. Взаимодействие с Java

Если ассоциативный массив получен в результате вызова Java-метода, его можно преобразовать в соответствующий Scala-эквивалент, чтобы иметь возможность работать с ним с применением удобного API языка Scala. Это также пригодится, если вам доведется работать с изменяемыми деревьями, которые отсутствуют в Scala.

Просто добавьте инструкцию `import`:

```
import scala.collection.JavaConversions.mapAsScalaMap
```

Затем инициируйте преобразование, указав тип ассоциативного массива в языке Scala:

```
val scores: scala.collection.mutable.Map[String, Int] =
    new java.util.TreeMap[String, Int]
```

Кроме того, поддерживается преобразование `java.util.Properties` в `Map[String, String]`:

```
import scala.collection.JavaConversions.propertiesAsScalaMap
val props: scala.collection.Map[String, String] = System.getProperties()
```

Напротив, чтобы передать ассоциативный массив Scala в метод, принимающий ассоциативный массив Java, определите неявное обратное преобразование. Например:

```
import scala.collection.JavaConversions.mapAsJavaMap
import java.awt.font.TextAttribute._              // Импортировать ключи
val attrs = Map(FAMILY -> "Serif", SIZE -> 12) // ассоциативный массив Scala
val font = new java.awt.Font(attrs)    // Принимает ассоциативный массив Java
```

4.7. Кортежи

Ассоциативные массивы – это коллекции пар ключ/значение. Пары – это простейший случай *кортежей* (tuples), агрегатов значений разных типов.

Значение кортежа формируется заключением отдельных значений в *круглые скобки*. Например,

```
(1, 3.14, "Fred")
```

это кортеж типа

```
Tuple3[Int, Double, java.lang.String]
```

который также записывается как

```
(Int, Double, java.lang.String)
```

Если имеется кортеж, например,

```
val t = (1, 3.14, "Fred")
```

обратиться к его компонентам можно с помощью методов _1, _2, _3, например:

```
val second = t._2 // Присвоит переменной second значение 3.14
```

В отличие от строк и массивов, нумерация позиций компонентов в кортежах начинается с 1, а не с 0.

> **Примечание.** Выражение `t._2` можно записать как `t _2` (с пробелом вместо точки), но не как `t_2`.

Обычно для доступа к компонентам кортежа лучше использовать механизм сопоставления с образцом, например

```
val (first, second, third) = t  // Присвоит переменной first значение 1,
                                // second - 3.14, third - "Fred"
```

Вместо переменных, соответствующих ненужным компонентам, можно использовать _:

```
val (first, second, _) = t
```

С помощью кортежей удобно возвращать из функций несколько значений. Например, метод `partition` класса `StringOps` возвращает пару строк, содержащих символы, которые соответствуют и не соответствуют условию:

```
"New York".partition(_.isUpper) // Вернет пару ("NY", "ew ork")
```

4.8. Функция zip

Одна из причин использования кортежей – возможность объединения значений, которые должны обрабатываться вместе. Упаковка (zipping) значений в кортежи обычно выполняется с помощью метода `zip`. Например, следующий фрагмент

```
val symbols = Array("<", "-", ">")
val counts = Array(2, 10, 2)
val pairs = symbols.zip(counts)
```

создаст массив пар

```
Array(("<", 2), ("-", 10), (">", 2))
```

Пары могут обрабатываться совместно:

```
for ((s, n) <- pairs) Console.print(s * n) // Выведет <<---------->>
```

> **Совет.** Метод `toMap` преобразует коллекцию пар в ассоциативный массив.

Если имеется коллекция ключей и коллекция соответствующих им значений, их можно упаковать в ассоциативный массив:

```
keys.zip(values).toMap
```

Упражнения

1. Создайте ассоциативный массив с ценами на вещи, которые вы хотели бы приобрести. Затем создайте второй ассоциативный массив с теми же ключами и ценами с 10%-ной скидкой.
2. Напишите программу, читающую слова из файла. Используйте изменяемый ассоциативный массив для подсчета вхождений каждого слова. Для чтения слов используйте `java.util.Scanner`:

   ```
   val in = new java.util.Scanner(new java.io.File("myfile.txt"))
   while (in.hasNext()) process in.next()
   ```

   Или загляните в главу 9, где представлено более идиоматичное для Scala решение.
   В конце выведите все слова и их счетчики.

3. Выполните предыдущее упражнение, используя неизменяемый ассоциативный массив.
4. Выполните предыдущее упражнение, используя сортированный ассоциативный массив, чтобы слова выводились в отсортированном порядке.
5. Выполните предыдущее упражнение, используя `java.util.TreeMap`, адаптировав его для работы со Scala API.
6. Определите связанную хеш-таблицу, отображающую «Monday» в `java.util.Calendar.MONDAY`, и так далее для других дней недели. Продемонстрируйте обход элементов в порядке их добавления.
7. Выведите таблицу всех Java-свойств, таких как:

```
java.runtime.name              | Java(TM) SE Runtime Environment
sun.boot.library.path          | /home/apps/jdk1.6.0_21/jre/lib/i386
java.vm.version                | 17.0-b16
java.vm.vendor                 | Sun Microsystems Inc.
java.vendor.url                | http://java.sun.com/
path.separator                 | :
java.vm.name                   | Java HotSpot(TM) Server VM
```

   Для этого перед выводом таблицы нужно отыскать длину самого длинного ключа.
8. Напишите функцию `minmax(values: Array[Int])`, возвращающую пару, содержащую наименьшее и наибольшее значения.
9. Напишите функцию `lteqgt(values: Array[Int], v: Int)`, возвращающую тройку, содержащую счетчик значений меньших `v`, равных `v` и больших `v`.
10. Что произойдет, если попытаться упаковать две строки, такие как `"Hello".zip("World")`? Придумайте достаточно реалистичный случай использования.

Глава 5. Классы

Темы, рассматриваемые в этой главе A1

- 5.1. Простые классы и методы без параметров.
- 5.2. Свойства с методами доступа.
- 5.3. Свойства только с методами чтения.
- 5.4. Приватные поля объектов.
- 5.5. Свойства компонентов L1.
- 5.6. Дополнительные конструкторы.
- 5.7. Главный конструктор.
- 5.8. Вложенные классы L1.
- Упражнения.

В этой главе вы узнаете, как выглядят классы в Scala. Знакомые с классами Java и C++ не будут испытывать сложностей, а лаконичный синтаксис Scala придется им по вкусу.

Основные темы этой главы:

- для полей классов автоматически генерируются методы доступа (чтения и записи);
- для полей можно реализовать свои методы доступа, не изменяя клиентов класса – в соответствии с «принципом единообразия доступа»;
- для автоматического создания методов `getXxx`/`setXxx` компонентов JavaBean можно использовать аннотацию `@BeanProperty`;
- каждый класс имеет главный конструктор, «вплетенный» в определение класса; его параметры превращаются в поля класса; главный конструктор выполняет все инструкции в теле класса;
- дополнительные конструкторы являются необязательными и носят имя `this`.

5.1. Простые классы и методы без параметров

В простейшем виде классы Scala выглядят очень похожими на их эквиваленты в Java или C++:

```
class Counter {
     private var value = 0           // Поля должны инициализироваться
     def increment() { value += 1 }  // Методы по умолчанию общедоступные
     def current() = value
}
```

В Scala классы не объявляются как общедоступные (`public`). Файл с программным кодом на языке Scala может содержать множество классов, и все они будут общедоступными.

Чтобы использовать класс, необходимо создать объект, после чего можно вызывать его методы обычным способом:

```
val myCounter = new Counter // Или new Counter()
myCounter.increment()
println(myCounter.current)
```

При вызове методов без параметров (как данный) скобки можно опустить:

```
myCounter.current     // OK
myCounter.current()   // Тоже OK
```

Какую форму использовать? Хорошим стилем считается использовать `()` при вызове *методов-мутаторов* (mutator, изменяющих состояние объекта) и опускать их при вызове *методов-акцессоров* (accessor, не изменяющих состояния объекта).

Это соглашение иллюстрируется в наших примерах:

```
myCounter.increment()        // Скобки используются при вызове мутаторов
println(myCounter.current)   // Скобки не используются при вызове акцессоров
```

Можно обязать пользователей класса придерживаться этого стиля, объявив метод `current` без `()`:

```
class Counter {
     ...
```

```
    def current = value // Определение без ()
}
```

Теперь пользователь класса будет вынужден вызывать `myCounter.current` без скобок.

5.2. Свойства с методами доступа

При создании классов на языке Java обычно не принято использовать общедоступные поля:

```
public class Person { // Это - Java
    public int age;   // Такое объявление осуждается в Java
}
```

При наличии общедоступного поля любой сможет изменить поле `fred.age`, сделав Фреда младше или старше. Именно поэтому предпочтение отдается методам чтения (getter) и записи (setter):

```
public class Person {            // Это - Java
    private int age;
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

Пара методов чтения/записи, как в примере выше, часто называется *свойством*. Мы говорим, что класс `Person` имеет свойство `age`.

Чем лучше такой подход? Сам по себе – ничем. Любой сможет вызвать `fred.setAge(21)` и сделать Фреда 21-летним юнцом.

Но если это является проблемой, мы можем защититься от нее:

```
public void setAge(int newValue) { if (newValue > age) age = newValue; }
    // Нельзя сделать моложе
```

Использование методов чтения и записи предпочтительнее общедоступных методов, потому что они позволяют начать с простой семантики чтения/записи и развивать ее по мере необходимости.

> **Примечание.** Просто потому, что методы чтения и записи лучше общедоступных полей, вовсе не означает, что они хороши во всех случаях. Часто бывает нежелательно, чтобы клиент имел возможность изменять состояние объекта. В этом разделе я показываю, как реализуются свой-

ства в языке Scala, а выбор того или иного способа реализации остается за вами.

В Scala методы чтения и записи создаются автоматически (синтезируются) для каждого поля. В примере ниже определяется общедоступное поле:

```
class Person {
     var age = 0
}
```

Scala сгенерирует класс для JVM с *приватным* полем `age` и методами доступа (чтения и записи). Эти методы общедоступны, потому что поле `age` не было объявлено приватным (`private`). (Для приватного поля будут созданы приватные методы чтения и записи.)

В Scala методы чтения и записи получат имена `age` и `age_=`. Например:

```
println(fred.age) // Вызовет метод fred.age()
fred.age = 21      // Вызовет метод fred.age_=(21)
```

Примечание. Чтобы увидеть эти методы собственными глазами, скомпилируйте класс `Person` и посмотрите байткод с помощью `javap`:

```
$ scalac Person.scala
$ javap -private Person
Compiled from "Person.scala"
  public class Person extends java.lang.Object implements scala.
ScalaObject{
  private int age;
  public int age();
  public void age_$eq(int);
  public Person();
}
```

Как видите, компилятор создал методы `age` и `age_$eq`. (Символ `=` транслируется в `$eq`, потому что JVM не допускает наличия символа `=` в именах методов.)

Примечание. В Scala методы чтения и записи не получают имена `getXxx` и `setXxx`, но методы с этими именами играют ту же роль. В разделе 5.5 «Свойства компонентов» демонстрируется, как генерировать методы `getXxx` и `setXxx` в стиле Java, чтобы классы Scala могли взаимодействовать с инструментами Java.

Переопределить методы чтения и записи можно в любой момент. Например:

```
class Person {
     private var privateAge = 0 // Сделать приватным и переименовать

     def age = privateAge
     def age_=(newValue: Int) {
          if (newValue > privateAge) privateAge = newValue;
               // Нельзя сделать моложе
     }
}
```

Пользователь класса по-прежнему сможет обращаться к `fred.age`, но теперь ему не удастся омолодить Фреда:

```
val fred = new Person
fred.age = 30
fred.age = 21
println(fred.age) // 30
```

Примечание. Бертран Мейер (Bertrand Meyer), создатель известного языка Eiffel, сформулировал принцип единообразия доступа (Uniform Access Principle): «Все службы, обеспечиваемые неким модулем, должны быть доступны за счет универсальной системы обозначений, не зависящей от того, хранятся ли эти службы в памяти или вычисляются». В языке Scala код, вызывающий `fred.age`, не знает, как реализован элемент `age`, в виде метода или поля. (Разумеется, в JVM службы всегда реализуются в виде методов, либо синтезированных, либо созданных программистом.)

Совет. На первый взгляд может пугать то обстоятельство, что Scala генерирует методы чтения и записи для всех полей. Однако программист имеет возможность управлять этим процессом:

- если поле приватное, создаются приватные методы доступа;
- если поле объявлено как `val`, создается только метод чтения;
- чтобы методы не создавались, поле следует объявить как `private[this]` (см. раздел 5.4 «Приватные поля объектов»).

5.3. Свойства только с методами чтения

Иногда бывает желательно иметь свойство, доступное *только для чтения*, имеющее метод чтения и не имеющее метода записи. Если

значение свойства никогда не будет изменяться после создания объекта, объявите поле как `val`:

```
class Message {
     val timeStamp = new java.util.Date
     ...
}
```

Компилятор Scala создаст приватное поле и только метод чтения.

Однако иногда хотелось бы, чтобы свойство было недоступно клиенту для изменения, но могло изменяться некоторыми другими путями. Отличным примером может служить класс `Counter` из раздела 5.1 «Простые классы и методы без параметров». Концептуально счетчик имеет свойство `current`, значение которого изменяется вызовом метода `increment`, но это свойство не имеет метода записи.

Такое свойство нельзя реализовать на основе объявления `val` – значения `val` никогда не изменяются. Вместо этого можно сделать поле приватным и определить метод чтения:

```
class Counter {
     private var value = 0
     def increment() { value += 1 }
     def current = value // Без () в объявлении
}
```

Обратите внимание на отсутствие скобок `()` в определении метода чтения. Такое объявление *вынуждает* вызывать метод без скобок:

```
val n = myCounter.current // вызов myCounter.current() вызовет ошибку
```

Итак, на выбор имеются четыре варианта реализации свойств:

1. `var foo`: Scala синтезирует методы чтения и записи.
2. `val foo`: Scala синтезирует только метод чтения.
3. Вы определяете методы `foo` и `foo_=`.
4. Вы определяете метод `foo`.

Примечание. В Scala нельзя создать свойство, доступное только для записи (то есть свойство с методом записи и без метода чтения).

Совет. Встретившись с полем в классе Scala, помните, что это не то же самое, что поле класса в Java или C++. Это – приватное поле с методом чтения (для полей, объявленных как `val`) или с двумя методами доступа, чтения и записи (для полей, объявленных как `var`).

5.4. Приватные поля объектов

В Scala (так же как в Java или C++) метод может иметь доступ к приватным полям во *всех* объектах своего класса. Например:

```
class Counter {
     private var value = 0
     def increment() { value += 1 }

     def isLess(other : Counter) = value < other.value
          // имеет доступ к приватному полю другого объекта
}
```

Обращение к свойству `other.value` является вполне легальным, потому что `other` также является объектом класса `Counter`.

Scala позволяет еще больше ограничивать доступ с помощью квалификатора `private[this]`:

```
private[this] var value = 0 // Обращение someObject.value недопустимо
```

Теперь методы класса `Counter` смогут обращаться только к полю `value` текущего объекта, но не других объектов типа `Counter`. Такой режим доступа иногда называется *приватным для объекта* (object-private) и часто используется в некоторых объектно-ориентированных языках, таких как SmallTalk.

Для приватных полей класса Scala генерирует приватные методы чтения и записи. Однако для приватных полей объекта методы доступа не генерируются вообще.

> **Примечание.** Scala позволяет определять права доступа к отдельным классам. Квалификатор `private[ClassName]` определяет, что только методы этого класса имеют право доступа к данному полю. Здесь `ClassName` должно быть именем определяемого или внешнего класса. (См. раздел 5.8 «Вложенные классы», где обсуждаются внутренние классы.)

В данном случае будут сгенерированы вспомогательные методы доступа, позволяющие внешнему классу обращаться к полю. Эти методы будут общедоступными, потому что JVM не имеет механизмов управления доступом с такой точностью, и они получат имена, зависящие от реализации.

5.5. Свойства компонентов L1

Как было показано в предыдущих разделах, Scala автоматически генерирует методы доступа для определяемых вами полей. Однако эти имена отличаются от тех, что ожидают увидеть инструменты Java. Спецификация JavaBeans (www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html) определяет Java-свойство как пару методов `getFoo`/`setFoo` (или только метод `getFoo`, если свойство доступно лишь для чтения). Многие инструменты Java опираются на это соглашение об именовании.

Когда в программном коде на языке Scala поле помечается аннотацией `@BeanProperty`, для него автоматически генерируются такие методы. Например, фрагмент

```
import scala.reflect.BeanProperty
class Person {
    @BeanProperty var name: String = _
}
```

сгенерирует четыре метода:

1. `name: String`
2. `name_=(newValue: String): Unit`
3. `getName(): String`
4. `setName(newValue: String): Unit`

В табл. 5.1 показано, какие методы генерируются во всех возможных случаях.

> **Примечание.** Если поле определяется как параметр главного конструктора (см. раздел 5.7 «Главный конструктор») и желательно, чтобы для него были созданы методы доступа в соответствии со спецификацией JavaBeans, отметьте параметр конструктора аннотацией, как показано ниже:
>
> ```
> class Person(@BeanProperty var name: String)
> ```

Таблица 5.1. Методы, автоматически генерируемые для полей

Поле	Генерируемые методы	Когда используется
`val/var name`	`public name` `name_=` (только для `var`)	Для реализации общедоступного свойства на основе поля

`@BeanProperty val/var name`	`public name`
	`getName()`
	`name_=` (только для `var`)
	`setName(...)` (только для `var`)
		Для взаимодействия с компонентами JavaBeans

Таблица 5.1. Методы, автоматически генерируемые для полей (окончание)

Поле	Генерируемые методы	Когда используется
`private val/var name`	`private name` `name_=` (только для `var`)	Для ограничения доступности поля лишь данным классом. Используйте квалификатор `private`, только если свойство действительно не должно быть общедоступным
`private[this] val/var name`	Нет	Для ограничения доступности поля только данным объектом. Редко применяется на практике
`private[ClassName] val/var name`	Зависит от реализации	Для передачи прав доступа внешнему классу. Редко применяется на практике

5.6. Дополнительные конструкторы

Как в Java и C++, классы в языке Scala могут иметь произвольное количество конструкторов. Однако в Scala класс имеет один конструктор, более важный, чем все остальные, который называется *главным конструктором* (primary constructor). Кроме того, класс может иметь любое количество *дополнительных конструкторов* (auxiliary constructors).

Сначала обсудим дополнительные конструкторы, так как они проще. Они очень похожи на конструкторы в Java и C++, за исключением двух отличий.

1. Дополнительные конструкторы называются `this`. (В Java и C++ конструкторы имеют то же имя, что и класс, что может вызвать неудобство, если вы решите переименовать класс.)
2. Каждый дополнительный конструктор должен начинаться вызовом дополнительного конструктора, объявленного выше, или вызовом главного конструктора.

Следующий класс имеет два дополнительных конструктора.

```
class Person {
     private var name = ""
     private var age = 0

     def this(name: String) { // Дополнительный конструктор
```

```
        this()              // Вызов главного конструктора
        this.name = name
}
    def this(name: String, age: Int) { // Другой дополнительный конструктор
        this(name) // Вызов предыдущего дополнительного конструктора
        this.age = age
    }
}
```

С главным конструктором мы познакомимся в следующем разделе, а пока просто знайте, что класс, не определяющий главный конструктор явно, получает главный конструктор без аргументов.

Создать объект этого класса можно тремя способами:

```
val p1 = new Person              // Главный конструктор
val p2 = new Person("Fred")      // Первый дополнительный конструктор
val p3 = new Person("Fred", 42) // Второй дополнительный конструктор
```

5.7. Главный конструктор

В языке Scala каждый класс имеет главный конструктор. Главный конструктор не определяется, как метод `this`, а вплетается в определение класса.

1. Параметры главного конструктора перечисляются сразу вслед за именем класса.

```
class Person(val name: String, val age: Int) {
    // Параметры главного конструктора в (...)
    ...
}
```

Параметры главного конструктора автоматически превращаются в поля класса, которые инициализируются аргументами конструктора. В нашем примере `name` и `age` превратятся в поля класса `Person`. Вызов конструктора, такой как `new Person("Fred", 42)`, установит значения полей `name` и `age`.

Строка на языке Scala эквивалентна семи строкам на Java:

```
public class Person { // Это - Java
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
```

```
    }
    public String name() { return this.name; }
    public int age() { return this.age; }
    ...
}
```

2. Главный конструктор выполняет все инструкции в определении класса. Например, в следующем классе

```
class Person(val name: String, val age: Int) {
    println("Just constructed another person")
    def description = name + " is " + age + " years old"
}
```

инструкция `println` является частью главного конструктора. Она выполняется при создании каждого объекта.
Это удобно, когда необходимо настроить поле в процессе создания объекта. Например:

```
class MyProg {
    private val props = new Properties
    props.load(new FileReader("myprog.properties"))
        // Инструкция выше является частью главного конструктора
    ...
}
```

Примечание. Если после имени класса отсутствуют параметры, класс получит главный конструктор без параметров. Этот конструктор просто выполнит все инструкции, имеющиеся в теле класса.

Совет. Зачастую можно избавиться от дополнительных конструкторов, определяя аргументы по умолчанию в главном конструкторе. Например:

```
class Person(val name: String = "", val age: Int = 0)
```

Параметры главного конструктора могут определяться в любой форме из перечисленных в табл. 5.1. Например,

```
class Person(val name: String, private var age: Int)
```

объявляет и инициализирует поля

```
val name: String
private var age: Int
```

Параметры конструктора могут также определяться как параметры обычных методов, без `val` или `var`. Порядок обработки таких параметров зависит от того, как они используются внутри класса.

- Если параметр без `val` или `var` используется внутри хотя бы одного метода, он становится полем. Например,

```
class Person(name: String, age: Int) {
     def description = name + " is " + age + " years old"
}
```

  объявляет и инициализирует неизменяемые поля `name` и `age`, приватные для объекта.

  Такие поля эквивалентны объявлению `private[this] val` (см. раздел 5.4 «Приватные поля объектов»).
- В противном случае параметр не преобразуется в поле. Он интерпретируется как обычный параметр, доступный только в теле главного конструктора. (Строго говоря, это является оптимизацией, зависящей от конкретной реализации.)

В табл. 5.2 перечислено, какие поля и методы создаются для различных видов параметров главного конструктора.

Таблица 5.2. Поля и методы, генерируемые для параметров главного конструктора

Параметр главного конструктора	Генерируемые поля/методы
`name: String`	Поле, приватное для объекта. Если параметр не используется в других методах, поле `name` не создается
`private val/var name: String`	Приватное поле, приватные методы доступа
`val/var name: String`	Приватное поле, общедоступные методы доступа
`@BeanProperty val/var name: String`	Приватное поле, общедоступные в Scala и JavaBeans методы доступа

Если форма определения главного конструктора кажется вам обескураживающей, можете ее не использовать. Просто определите один или более дополнительных конструкторов обычным способом, но не забудьте добавить вызов `this()`, если конструктор не объединяется в цепочку с другим дополнительным конструктором.

Однако многим программистам нравится краткий синтаксис. Мартин Одерски (Martin Odersky) предлагает размышлять таким образом: в Scala классы принимают параметры, подобно обычным методам.

Примечание. Представляя параметры главного конструктора как параметры класса, параметры без `val` или `var` становятся более простыми для понимания. Область видимости таких параметров простирается до конца объявления класса. Поэтому вы можете использовать эти параметры в методах, и в этом случае компилятор автоматически преобразует их в поля.

Совет. Разработчики Scala высоко ценят каждое нажатие на клавиши, поэтому они позволяют объединять объявление класса с его главным конструктором. Читая определение класса на языке Scala, необходимо уметь выделять две части. Например, когда вы видите

```
class Person(val name: String) {
     var age = 0
     def description = name + " is " + age + " years old"
}
```

мысленно выделяйте определение класса:

```
class Person(val name: String) {
     var age = 0
     def description = name + " is " + age + " years old"
}
```

и определение конструктора:

```
class Person(val name: String) {
     var age = 0
     def description = name + " is " + age + " years old"
}
```

Примечание. Чтобы сделать главный конструктор приватным, добавьте ключевое слово `private`, как показано ниже:

```
class Person private(val id: Int) { ... }
```

В этом случае пользователь класса должен будет использовать дополнительный конструктор, чтобы создать объект `Person`.

5.8. Вложенные классы L1

В Scala допускается вкладывать все, что угодно, во все, что угодно. Можно определять функции внутри других функций и классы внутри других классов. Ниже приводится простейший пример такого определения классов.

```
import scala.collection.mutable.ArrayBuffer
class Network {
```

```
    class Member(val name: String) {
        val contacts = new ArrayBuffer[Member]
    }

    private val members = new ArrayBuffer[Member]

    def join(name: String) = {
        val m = new Member(name)
        members += m
        m
    }
}
```

Рассмотрим две группы людей:

```
val chatter = new Network
val myFace = new Network
```

В Scala каждый экземпляр получит собственный класс `Member`, точно так же, как каждый экземпляр получит собственное поле `members`. То есть `chatter.Member` и `myFace.Member` — это разные классы.

> **Примечание.** В языке Java все совсем иначе, где внутренний класс принадлежит внешнему классу.

В Scala используется более однородный подход. Например, чтобы создать новый внутренний объект, достаточно просто воспользоваться ключевым словом `new` с именем типа: `new chatter.Member`. В языке Java необходимо использовать более сложный синтаксис `chatter.new Member()`.

В примере с группами людей, представленном выше, имеется возможность принимать новых членов на собрании внутри группы, но нет возможности принимать их на собрании другой группы.

```
val fred = chatter.join("Fred")
val wilma = chatter.join("Wilma")
fred.contacts += wilma                  // OK
val barney = myFace.join("Barney") // Имеет тип myFace.Member
fred.contacts += barney
    // Нет, нельзя добавить myFace.Member в буфер элементов chatter.Member
```

Для групп людей такой порядок вещей имеет определенный смысл. Но если он вас не устраивает, решить проблему можно двумя способами.

Во-первых, тип `Member` можно объявить в другом месте. Для этого отлично подойдет объект-компаньон `Network`. (Объекты-компаньоны описываются в главе 6.)

```
object Network {
     class Member(val name: String) {
          val contacts = new ArrayBuffer[Member]
     }
}

class Network {
     private val members = new ArrayBuffer[Network.Member]
     ...
}
```

Во-вторых, можно использовать *проекцию типов* (type projection) `Network#Member`, которая означает: «член любой группы». Например:

```
class Network {
     class Member(val name: String) {
          val contacts = new ArrayBuffer[Network#Member]
     }
     ...
}
```

Этот прием можно использовать для получения возможности более точного управления, когда «внутренний класс определяется отдельно для каждого объекта» в отдельных местах в программе, но не повсюду. Более подробно вопросы проекции типов рассматриваются в главе 18.

Примечание. Во вложенном классе доступна ссылка `this`, указывающая на внешний класс, как `EnclosingClass.this`, аналогично языку Java. При желании можно определить псевдоним для этой ссылки, как показано ниже:

```
class Network(val name: String) { outer =>
     class Member(val name: String) {
          ...
          def description = name + « inside « + outer.name
     }
}
```

Строка `class Network { outer =>` создаст переменную `outer`, ссылающуюся на `Network.this`. Для этой переменной можно выбрать любое имя. На практике часто используется имя `self`, но оно может вносить путаницу при использовании внутри вложенных классов.

Этот синтаксис тесно связан с синтаксисом определения «собственного типа», с которым вы встретитесь в главе 18.

Упражнения

1. Усовершенствуйте класс `Counter` в разделе 5.1 «Простые классы и методы без параметров», так чтобы значение счетчика не превращалось в отрицательное число по достижении `Int.MaxValue`.
2. Напишите класс `BankAccount` с методами `deposit` и `withdraw` и свойством `balance`, доступным только для чтения.
3. Напишите класс `Time` со свойствами `hours` и `minutes`, доступными только для чтения, и методом `before(other: Time): Boolean`, который проверяет, предшествует ли время `this` времени `other`. Объект `Time` должен конструироваться как `new Time(hrs, min)`, где `hrs` – время в 24-часовом формате.
4. Перепишите класс `Time` из предыдущего упражнения так, чтобы внутри время было представлено количеством минут, прошедших с начала суток (между 0 и $24 \times 60 - 1$). Общедоступный интерфейс при этом *не должен* измениться. То есть эти изменения не должны оказывать влияние на клиентский код.
5. Создайте класс `Student` со свойствами в формате JavaBeans `name` (типа `String`) и `id` (типа `Long`), доступными для чтения/записи. Какие методы будут сгенерированы? (Используйте `javap` для проверки.) Сможете ли вы вызывать методы доступа в формате JavaBeans из программного кода на языке Scala? Необходимо ли это?
6. В классе `Person` из раздела 5.1 «Простые классы и методы без параметров» реализуйте главный конструктор, преобразующий отрицательное значение возраста в 0.
7. Напишите класс `Person` с главным конструктором, принимающим строку, которая содержит имя, пробел и фамилию, например: `new Person(«Fred Smith»)`. Сделайте свойства `firstName` и `lastName` доступными только для чтения. Должен ли параметр главного конструктора объявляться как `var`, `val` или как обычный параметр? Почему?
8. Создайте класс `Car` со свойствами, определяющими производителя, название модели и год производства, которые доступны только для чтения, и свойство с регистрационным номером автомобиля, доступное для чтения/записи. Добавьте четыре

конструктора. Все они должны принимать название производителя и название модели. При необходимости в вызове конструктора могут также указываться год и регистрационный номер. Если год не указан, он должен устанавливаться равным -1, а при отсутствии регистрационного номера должна устанавливаться пустая строка. Какой конструктор вы выберете в качестве главного? Почему?

9. Повторно реализуйте класс из предыдущего упражнения на Java, C# или C++ (по выбору). Насколько короче получился класс на языке Scala?
10. Взгляните на следующее определение класса:

```
class Employee(val name: String, var salary: Double) {
    def this() { this("John Q. Public", 0.0) }
}
```

Перепишите его так, чтобы он содержал явные определения полей и имел главный конструктор по умолчанию. Какое объявление вам нравится больше? Почему?

Глава 6. Объекты

Темы, рассматриваемые в этой главе A1

- 6.1. Объекты-одиночки.
- 6.2. Объекты-компаньоны.
- 6.3. Объекты, расширяющие классы или трейты.
- 6.4. Метод apply.
- 6.5. Объект, представляющий приложение.
- 6.6. Перечисления.
- Упражнения.

В этой короткой главе вы познакомитесь с особенностями использования конструкции `object`, когда необходимо ограничить приложение единственным экземпляром класса или выделить место для различных констант и функций.

Основные темы этой главы:

- использование конструкции `object` для группировки вспомогательных методов и создания объектов-одиночек;
- класс может иметь объект-компаньон с тем же именем;
- объекты могут расширять классы или трейты;
- метод `apply` объекта обычно используется для конструирования новых экземпляров класса-компаньона;
- избегайте использования метода `main`, используйте объект, расширяющий трейт `App`;
- перечисления можно реализовать за счет расширения объекта `Enumeration`.

6.1. Объекты-одиночки

В языке Scala отсутствуют статические методы и поля. Вместо этого используется конструкция `object`. Она определяет единственный экземпляр класса (singleton), обладающий всеми необходимыми качествами. Например:

```
object Accounts {
     private var lastNumber = 0
     def newUniqueNumber() = { lastNumber += 1; lastNumber }
}
```

Когда в приложении потребуется получить новый уникальный учетный номер, достаточно вызвать `Accounts.newUniqueNumber()`.

Конструктор такого объекта вызывается при первом к нему обращении. В данном примере конструктор `Accounts` выполняется при первом вызове метода `Accounts.newUniqueNumber()`. Если объект не используется, его конструктор не вызывается.

Объект обладает всеми характерными особенностями класса – он может даже наследовать другие классы или трейты (см. раздел 6.3 «Объекты, расширяющие классы или трейты»). Единственное исключение – такой объект не может иметь конструктор с параметрами.

Конструкция `object` в языке Scala используется в тех же случаях, что и объекты-одиночки (singleton object) в Java или C++:

- в качестве централизованного хранилища вспомогательных функций и/или констант;
- когда эффективно можно использовать только один экземпляр;
- когда для координации некоторой службы должен существовать только один экземпляр (шаблон проектирования «Singleton» – «Одиночка»).

> **Примечание.** Многие не воспринимают всерьез шаблон проектирования «Одиночка» («Singleton»). Scala дает вам в руки инструменты, а как их использовать, во благо или во вред, зависит от вас.

6.2. Объекты-компаньоны

В Java или C++ часто имеется класс, определяющий не только методы экземпляра, но и статические методы. В Scala добиться того же самого можно с помощью «объекта-компаньона» (companion object), имя которого совпадает с именем класса. Например:

```
class Account {
     val id = Account.newUniqueNumber()
     private var balance = 0.0
     def deposit(amount: Double) { balance += amount }
     ...
```

```
}

object Account { // Объект-компаньон
     private var lastNumber = 0
     private def newUniqueNumber() = { lastNumber += 1; lastNumber }
}
```

Класс и его объект-компаньон могут обращаться к приватным свойствам и методам друг друга. Они должны определяться в одном файле с исходными текстами.

Примечание. Объект-компаньон доступен из класса, но находится за пределами его области видимости. Например, класс `Account` должен вызывать метод объекта-компаньона, как `Account.newUniqueNumber()`, а не просто `newUniqueNumber()`.

Совет. В REPL необходимо определять вместе класс и его объект-компаньон в режиме вставки. Введите

```
:paste
```

А затем вставьте из буфера обмена определение класса и объекта-компаньона, после чего нажмите комбинацию **Ctrl**+**D**.

6.3. Объекты, расширяющие классы или трейты

Объект, созданный конструкцией `object`, может расширять класс и/или один или более трейтов (traits). В результате получается объект класса, расширяющий данный класс и/или трейты (traits) и обладающий всеми дополнительными особенностями, указанными в объявлении объекта.

Одним из применений таких объектов является создание совместно используемых объектов по умолчанию. Например, ниже приводится класс, объявляющий операции отмены.

```
abstract class UndoableAction(val description: String) {
     def undo(): Unit
     def redo(): Unit
}
```

По умолчанию эти операции «ничего не делают». Безусловно, в приложении достаточно иметь по одной такой операции.

```
object DoNothingAction extends UndoableAction(«Do nothing») {
     override def undo() {}
     override def redo() {}
}
```

Объект DoNothingAction может совместно использоваться везде, где необходимы эти операции по умолчанию.

```
val actions = Map("open" -> DoNothingAction, "save" -> DoNothingAction, ...)
     // Операции открытия и сохранения еще не реализованы
```

6.4. Метод apply

Иметь метод apply в объектах – обычное дело. Объект apply вызывается в выражениях вида

```
Object(arg1, ..., argN)
```

Обычно метод apply возвращает экземпляр класса компаньона.

Например, объект Array определяет метод apply, позволяющий создавать массивы с помощью таких выражений:

```
Array("Mary", "had", "a", "little", "lamb")
```

Почему бы просто не использовать конструктор? Возможность не использовать ключевое слово new может пригодиться, например, во вложенных выражениях, таких как

```
Array(Array(1, 7), Array(2, 9))
```

> **Внимание.** Очень легко спутать Array(100) и new Array(100). Первое выражение производит вызов apply(100), возвращающий Array[Int] с единственным элементом – целым числом 100. Второе выражение производит вызов конструктора this(100), возвращающий Array[Nothing] с сотней пустых элементов.

Ниже приводится пример определения метода apply:

```
class Account private (val id: Int, initialBalance: Double) {
     private var balance = initialBalance
     ...
```

```
}

object Account { // Объект-компаньон
     def apply(initialBalance: Double) =
     new Account(newUniqueNumber(), initialBalance)
     ...
}
```

Теперь объект, представляющий банковский счет, можно создавать так:

```
val acct = Account(1000.0)
```

6.5. Объект, представляющий приложение

Выполнение каждой программы на языке Scala должно начинаться с вызова метода `main` объекта, имеющего сигнатуру `Array[String] => Unit`:

```
object Hello {
     def main(args: Array[String]) {
          println("Hello, World!")
     }
}
```

Однако вместо создания метода `main` можно унаследовать трейт (trait) `App` и поместить код программы в тело конструктора:

```
object Hello extends App {
     println("Hello, World!")
}
```

Если потребуется организовать анализ *аргументов командной строки*, их можно получить из свойства `args`:

```
object Hello extends App {
     if (args.length > 0)
          println("Hello, " + args(0))
     else
          println("Hello, World!")
}
```

Если вызвать программу с параметром `scala.time`, по ее завершении в консоль будет выведено время выполнения программы.

```
$ scalac Hello.scala
$ scala -Dscala.time Hello Fred
Hello, Fred
[total 4ms]
```

Здесь не обошлось без некоторой доли волшебства. Трейт (trait) `App` наследует другой трейт, `DelayedInit`, который особым образом обрабатывается компилятором. Весь код инициализации из класса, наследующего этот трейт, перемещается в метод `delayedInit`. Метод `main` трейта `App` сохраняет аргументы командной строки, вызывает метод `delayedInit` и при необходимости выводит время выполнения.

> **Примечание.** В старых версиях Scala ту же роль играл трейт `Application`. Он обеспечивал перенос кода программы в статический инициализатор, который не оптимизировался динамическим компилятором. В современных версиях следует использовать трейт (trait) `App` вместо него.

6.6. Перечисления

В отличие от Java или C++, в языке Scala отсутствуют перечислимые типы. Однако в стандартной библиотеке имеется вспомогательный класс `Enumeration`, который можно использовать для создания *перечислений*.

Для этого достаточно создать объект `object`, наследующий класс `Enumeration`, и инициализировать значения перечисления вызовом метода `Value`. Например:

```
object TrafficLightColor extends Enumeration {
    val Red, Yellow, Green = Value
}
```

Здесь определяются три поля, `Red`, `Yellow` и `Green`, каждое из которых инициализируется вызовом метода `Value`. Это – сокращенная форма записи объявления

```
val Red = Value
val Yellow = Value
val Green = Value
```

Каждый вызов метода Value возвращает новый экземпляр вложенного класса с тем же именем Value.

При желании методу Value можно передать числовое значение, строковое имя или и то, и другое:

```
val Red = Value(0, "Stop")
val Yellow = Value(10)        // Строковое имя "Yellow"
val Green = Value("Go")       // Числовое значение 11
```

Если числовое значение не указано, будет присвоено значение на единицу больше предыдущего, начиная с нуля. В качестве строкового имени по умолчанию используется имя поля.

Теперь на значения перечисления можно ссылаться как TrafficLightColor.Red, TrafficLightColor.Yellow и т. д. Если это покажется слишком утомительным, можно воспользоваться инструкцией

```
import TrafficLightColor._
```

(Подробнее об *импортировании членов* классов или объектов рассказывается в главе 7.)

Не забывайте, что перечисление имеет тип TrafficLightColor.Value, а не TrafficLightColor – это тип объекта, хранящего значения. Некоторые рекомендуют определять псевдоним типа

```
object TrafficLightColor extends Enumeration {
     type TrafficLightColor = Value
     val Red, Yellow, Green = Value
}
```

Теперь перечисление будет иметь тип TrafficLightColor.TrafficLightColor, что можно использовать с выгодой только в случае применения инструкции import. Например:

```
import TrafficLightColor._
def doWhat(color: TrafficLightColor) = {
     if (color == Red) "stop"
     else if (color == Yellow) "hurry up"
     else "go"
}
```

Числовое значение элемента перечисления можно получить вызовом метода id, а строковое имя – вызовом метода toString.

Вызов `TrafficLightColor.values` возвращает множество всех значений:

```
for (c <- TrafficLightColor.values) println(c.id + ": " + c)
```

Наконец, имеется возможность отыскать элемент перечисления по его числовому идентификатору или строковому имени. Оба следующих вызова вернут объект `TrafficLightColor.Red`:

```
TrafficLightColor(0)        // Вызовет Enumeration.apply
TrafficLightColor.withName("Red")
```

Упражнения

1. Напишите объект `Conversions` с методами `inchesToCentimeters`, `gallonsToLiters` и `milesToKilometers`.
2. Предыдущую задачу трудно назвать объектно-ориентированной. Реализуйте общий суперкласс `UnitConversion` и определите объекты `InchesToCentimeters`, `GallonsToLiters` и `MilesToKilometers`, наследующие его.
3. Определите объект `Origin`, наследующий класс `java.awt.Point`. Почему это не самая лучшая идея? (Рассмотрите поближе методы класса `Point`.)
4. Определите класс `Point` с объектом-компаньоном, чтобы можно было конструировать экземпляры `Point`, как `Point(3, 4)`, без ключевого слова `new`.
5. Напишите приложение на языке Scala, используя трейт (trait) `App`, которое выводит аргументы командной строки в обратном порядке, разделяя их пробелами. Например, команда `scala Reverse Hello World` должна вывести `World Hello`.
6. Напишите перечисление, описывающее четыре масти игральных карт так, чтобы метод `toString` возвращал «♣», «♦», «♥» или «♠».
7. Реализуйте функцию для проверки масти карты, реализованной в предыдущем упражнении, которая проверяла бы принадлежность карты к красной масти.
8. Напишите перечисление, описывающее восемь углов куба RGB. В качестве числовых идентификаторов должны использоваться значения цвета (например, `0xff0000` — для `Red`).

Глава 7. Пакеты и импортирование

Темы, рассматриваемые в этой главе A1

- 7.1. Пакеты.
- 7.2. Правила видимости.
- 7.3. Объявления цепочек пакетов.
- 7.4. Объявления в начале файла.
- 7.5. Объекты пакетов.
- 7.6. Видимость пакетов.
- 7.7. Импортирование.
- 7.8. Импортирование возможно в любом месте.
- 7.9. Переименование и сокрытие членов.
- 7.10. Неявное импортирование.
- Упражнения.

В этой главе вы узнаете, как действуют инструкции `package` и `import` в языке Scala. Обе они не только более логичные, чем в Java, но и более гибкие.

Основные темы этой главы:

- пакеты могут вкладываться подобно классам;
- пути к пакетам – *не* абсолютные;
- объявление пакета вида `package x.y.z` оставляет невидимым содержимое промежуточных пакетов `x` и `x.y`;
- инструкции `package` без фигурных скобок в начале файла распространяются на весь файл;
- объект пакета может иметь функции и переменные;
- инструкции `import` могут импортировать пакеты, классы и объекты;
- инструкции `import` могут находиться где угодно;
- инструкции `import` могут переименовывать и скрывать члены пакетов;
- пакеты `java.lang`, `scala` и объект `Predef` импортируются всегда.

7.1. Пакеты

Пакеты в языке Scala служат той же цели, что и пакеты в Java или пространства имен в C++: управление именами в больших программах. Например, имя `Map` может встречаться в пакетах `scala.collection.immutable` и `scala.collection.mutable` без каких-либо конфликтов. Чтобы обратиться к любому из них, можно использовать полное имя класса `scala.collection.immutable.Map` или `scala.collection.mutable.Map`. При желании с помощью инструкции `import` можно определить более короткий псевдоним – см. раздел 7.7 «Импортирование».

Чтобы добавить элементы в пакет, их можно включить в инструкции `package`, как показано ниже:

```
package com {
     package horstmann {
          package impatient {
               class Employee
               ...
          }
     }
}
```

После этого класс с именем `Employee` будет доступен из любого места программы под именем `com.horstmann.impatient.Employee`.

В отличие от объекта или класса, пакет может быть определен в нескольких файлах. Предыдущий код, например, может находиться в файлах `Employee.scala`, а файл `Manager.scala` может содержать

```
package com {
     package horstmann {
          package impatient {
               class Manager
               ...
          }
     }
}
```

> **Примечание.** Между каталогом местонахождения файла и именем пакета не обязательно должна быть прямая связь. Файлы `Employee.scala` и `Manager.scala` не обязательно должны находиться в каталоге `com/horstmann/impatient`.

С другой стороны, в одном файле может быть объявлено несколько пакетов. Файл `Employee.scala` может содержать

```
package com {
     package horstmann {
          package impatient {
               class Employee
               ...
          }
     }
}

package org {
     package bigjava {
          class Counter
          ...
     }
}
```

7.2. Правила видимости

В Scala *правила видимости* для пакетов более последовательны, чем в Java. Пакеты в Scala могут *вкладываться*, как любые другие типы областей видимости. Допускается непосредственно обращаться к именам, находящимся во вмещающей области видимости. Например:

```
package com {
     package horstmann {
          object Utils {
               def percentOf(value: Double, rate: Double) = value * rate / 100
               ...
          }

          package impatient {
               class Employee {
                    ...
                    def giveRaise(rate: scala.Double) {
                         salary += Utils.percentOf(salary, rate)
                    }
               }
          }
     }
}
```

Обратите внимание на имя `Utils.percentOf`. Класс `Utils` объявлен в *родительском* пакете. Все, что определено в родительском пакете, находится в области видимости, поэтому нет необходимости использовать полное квалифицированное имя `com.horstmann.Utils.percentOf`. (Такое имя тоже можно использовать, если оно вам нравится – в конце концов, пакет `com` также находится в области видимости.)

Однако в бочке меда есть ложка дегтя. Взгляните:

```
package com {
  package horstmann {
    package impatient {
      class Manager {
        val subordinates = new collection.mutable.ArrayBuffer[Employee]
        ...
      }
    }
  }
}
```

Этот код пользуется тем преимуществом, что пакет `scala` всегда импортируется. Поэтому пакет `collection` фактически называется `scala.collection`.

А теперь представьте, что кто-то добавил следующий пакет, возможно, в другом файле:

```
package com {
    package horstmann {
        package collection {
            ...
        }
    }
}
```

Теперь класс `Manager` перестанет компилироваться. Компилятор будет безуспешно пытаться отыскать имя `mutable` внутри пакета `com.horstmann.collection`. Класс `Manager` предполагает использовать пакет `collection` из пакета `scala` верхнего уровня, а не из вложенного пакета `collection`, оказавшегося в доступной области видимости.

В языке Java эта проблема не возникает, потому что в нем всегда используются только абсолютные имена пакетов, начиная от корня иерархии пакетов. Но в Scala имена пакетов являются относительными, так же как имена вложенных классов. Однако при использовании вложенных классов подобная проблема обычно не возникает, потому

что весь код находится в одном файле, под контролем того, кто создал и сопровождает этот файл. Но границы пакетов открыты настежь. Любой может внести свой вклад в пакет в любой момент времени.

Одно из решений – использовать абсолютные имена пакетов, начиная с `_root_`, например:

```
val subordinates = new _root_.scala.collection.mutable.ArrayBuffer[Employee]
```

Другое решение – использовать объявления «цепочек» пакетов, как описывается в следующем разделе.

> **Примечание.** Большинство программистов используют полные пути в именах пакетов, без префикса `_root_`. Это безопасно, пока кто-нибудь не решит создать вложенный пакет с именем `scala`, `java`, `com`, `org` или подобным.

7.3. Объявления цепочек пакетов

Объявление `package` может содержать «цепочку» или сегмент пути, например:

```
package com.horstmann.impatient {
     // Члены пакетов com и com.horstmann здесь недоступны
     package people {
          class Person
          ...
     }
}
```

Такое объявление ограничивает круг доступных членов пакетов. Теперь пакет `com.horstmann.collection` больше не будет доступен под именем `collection`.

7.4. Объявления в начале файла

Вместо вложенных друг в друга объявлений, демонстрировавшихся до сих пор, можно поместить объявления `package` в начало файла, *без фигурных скобок*. Например:

```
package com.horstmann.impatient
package people

class Person
     ...
```

Такое объявление эквивалентно следующему:

```
package com.horstmann.impatient {
     package people {
          class Person
          ...
          // До конца файла
     }
}
```

Подобная форма объявления предпочтительнее, если весь программный код в файле принадлежит одному и тому же пакету (что встречается довольно часто).

Обратите внимание, что в примере выше все, находящееся в файле, принадлежит пакету `com.horstmann.impatient.people`, при этом пакет `com.horstmann.impatient` остается открытым, поэтому вы можете ссылаться на его содержимое.

7.5. Объекты пакетов

Пакет может содержать классы, объекты и трейты, но не определения функций или переменных. Это досадное ограничение обусловлено особенностями виртуальной машины Java. Было бы лучше, если бы имелась возможность добавлять в пакеты вспомогательные функции или константы непосредственно в пакет, а не в какой-то объект `Utils`. Объекты пакетов позволяют снять это ограничение.

Каждый пакет может иметь один *объект пакета*. Он объявляется в родительском пакете с тем же именем, что и дочерний пакет. Например:

```
package com.horstmann.impatient

package object people {
     val defaultName = "John Q. Public"
}

package people {
     class Person {
          var name = defaultName // Константа из пакета
     }
     ...
}
```

Обратите внимание, что имя значения `defaultName` не требуется определять полностью, потому что оно находится в том же пакете. В любом другом месте оно будет доступно, как `com.horstmann.impatient.people.defaultName`.

За кулисами объект пакета компилируется в класс JVM со статическими методами и полями, с именами `package.class` внутри пакета `package`. В примере выше это мог быть класс `com.horstmann.impatient.people.package` со статическим полем `defaultName`. (В JVM имя `package` можно использовать как имя класса.)

Отличная идея – использовать ту же схему именования для файлов с исходными текстами. Поместите объявление объекта пакета в файл `com/horstmann/impatient/people/package.scala`, и тогда любой, кто пожелает добавить свои функции или переменные в пакет, легко отыщет его.

7.6. Видимость внутри пакетов

В Java член класса, не объявленный как `public`, `private` или `protected`, будет доступен в пакете, содержащем класс. В Scala того же эффекта можно добиться с помощью квалификаторов. Следующий метод будет доступен в собственном пакете:

```
package com.horstmann.impatient.people

class Person {
     private[people] def description = "A person with name " + name
     ...
}
```

Область его видимости можно распространить на вмещающий пакет:

```
private[impatient] def description = "A person with name " + name
```

7.7. Импортирование

Операция импортирования позволяет определить короткие имена взамен длинных. Инструкция

```
import java.awt.Color
```

позволит использовать имя `Color` вместо `java.awt.Color`.

Это – единственная цель операции импортирования. Если вы обожаете длинные имена, импортирование вам не пригодится.

Все *члены пакета* можно импортировать как

```
import java.awt._
```

Это то же самое, что групповой символ * в Java. (В Scala символ * считается допустимым символом для использования в идентификаторах. Вы сможете объявить пакет com.horstmann.*.people, только ради всего святого не делайте так.)

Аналогично можно импортировать все *члены класса* или объекта.

```
import java.awt.Color._
val c1 = RED                // Color.RED
val c2 = decode("#ff0000")  // Color.decode
```

Это напоминает инструкцию import static в Java. Программисты на Java живут в страхе перед этим вариантом, но в Scala он используется достаточно часто.

Импортировав пакет, вы получаете доступ к вложенным в него пакетам с короткими именами. Например:

```
import java.awt._

def handler(evt: event.ActionEvent) { // java.awt.event.ActionEvent
    ...
}
```

Пакет event является членом пакета java.awt, и инструкция import перенесла его в область видимости.

7.8. Импортирование возможно в любом месте

В Scala инструкция import может располагаться в любом месте, не только в начале файла. Область видимости инструкции import простирается до конца вмещающего блока. Например:

```
class Manager {
    import scala.collection.mutable._
    val subordinates = new ArrayBuffer[Employee]
    ...
}
```

Это очень полезная особенность, особенно в совокупности с возможностью использования группового символа. Вероятность импортировать большое количество имен из разных источников всегда доставляла некоторое беспокойство. Фактически некоторые программисты на Java не любят групповой символ в инструкциях импортирования настолько, что никогда не используют его и позволяют своей интегрированной среде разработки (IDE) генерировать длинные списки импортируемых классов.

Помещая инструкции импортирования только туда, где это необходимо, можно существенно снизить риск конфликтов.

7.9. Переименование и сокрытие членов

Если необходимо импортировать несколько имен из пакета, используйте *селектор*, как показано ниже:

```
import java.awt.{Color, Font}
```

Синтаксис селекторов позволяет переименовывать импортируемые члены пакета:

```
import java.util.{HashMap => JavaHashMap}
import scala.collection.mutable._
```

Теперь имя `JavaHashMap` будет служить псевдонимом для `java.util.HashMap`, а простое имя `HashMap` будет соответствовать имени `scala.collection.mutable.HashMap`.

Селектор `HashMap => _` скроет `HashMap`, вместо того чтобы переименовать его. Это может пригодиться, только если импортируются другие члены с аналогичными именами:

```
import java.util.{HashMap => _, _}
import scala.collection.mutable._
```

Теперь имя `HashMap` однозначно будет соответствовать классу `scala.collection.mutable.HashMap`, потому что имя `java.util.HashMap` будет скрыто.

7.10. Неявный импорт

Любая программа на языке Scala неявно начинается с объявлений

```
import java.lang._
import scala._
import Predef._
```

Как и в программах на Java, пакет `java.lang` импортируется всегда. Следующий пакет, `scala`, тоже импортируется всегда, но особым образом. В отличие от других, этому пакету разрешено переопределять имена, импортированные предыдущими инструкциями `import`. Например, `scala.StringBuilder` затрет `java.lang.StringBuilder` без конфликта с ним.

Последним импортируется объект `Predef`. Он содержит довольно много полезных функций. (Их с тем же успехом можно было поместить в объект пакета `scala`, но объект `Predef` появился до того, как в Scala была реализована поддержка объектов пакетов.)

Поскольку пакет `scala` импортируется по умолчанию, вам никогда не придется писать имена пакетов, начинающиеся с префикса `scala`. Например,

```
collection.mutable.HashMap
```

дает тот же эффект, что и

```
scala.collection.mutable.HashMap
```

Упражнения

1. Напишите программу, на примере которой можно было убедиться, что

   ```
   package com.horstmann.impatient
   ```

   не то же самое, что и

   ```
   package com
   package horstmann
   package impatient
   ```

2. Напишите головоломку, которая смогла бы сбить с толку ваших коллег, программистов на Scala, использующую пакет `com`, не являющийся пакетом верхнего уровня.
3. Напишите пакет `random` с функциями `nextInt(): Int`, `nextDouble(): Double` и `setSeed(seed: Int): Unit`. Для генерации случайных чисел используйте линейный конгруэнтный генератор

   $next = previous \times a + b \bmod 2^n$,

   где $a = 1664525$, $b = 1013904223$ и $n = 32$.
4. Как вы думаете, почему создатели языка Scala реализовали синтаксис объектов пакетов, вместо того чтобы просто разрешить добавлять функции и переменные в пакет?
5. Что означает определение `private[com] def giveRaise(rate: Double)`? Есть ли в этом смысл?
6. Напишите программу, копирующую все элементы из Java-хеша в Scala-хеш. Используйте операцию импортирования для переименования обоих классов.
7. В предыдущем упражнении перенесите все инструкции `import` в самую внутреннюю область видимости, насколько это возможно.
8. Опишите эффект следующих инструкций:

   ```
   import java._
   import javax._
   ```

   Насколько правильно это решение?
9. Напишите программу, импортирующую класс `java.lang.System`, читающую имя пользователя из системного свойства `user.name`, пароль из объекта `Console` и выводящую сообщение в стандартный поток ошибок, если пароль недостаточно «секретный». В противном случае программа должна вывести приветствие в стандартный поток вывода. Не импортируйте ничего другого и не используйте полные квалифицированные имена (с точками).
10. Помимо `StringBuilder`, какие другие члены пакета `java.lang` переопределяет пакет `scala`?

Глава 8. Наследование

Темы, рассматриваемые в этой главе A1

- 8.1. Наследование классов.
- 8.2. Переопределение методов.
- 8.3. Проверка и приведение типов.
- 8.4. Защищенные поля и методы.
- 8.5. Создание суперклассов.
- 8.6. Переопределение полей.
- 8.7. Анонимные подклассы.
- 8.8. Абстрактные классы.
- 8.9. Абстрактные поля.
- 8.10. Порядок создания и опережающие определения L3.
- 8.11. Иерархия наследования в Scala.
- 8.12. Равенство объектов L1.
- Упражнения.

В этой главе вы узнаете о наиболее важных отличиях в механизме наследования между языком Scala и родственными ему языками C++ и Java.

Основные темы этой главы:

- ключевые слова `extends` и `final` действуют так же, как в Java;
- при переопределении методов необходимо использовать модификатор `override`;
- конструктор суперкласса можно вызывать только из главного конструктора;
- поля могут переопределяться.

8.1. Наследование классов

Наследование классов в Scala выполняется так же, как в Java, – с помощью ключевого слова `extends`:

```
class Employee extends Person {
    var salary: 0.0
    ...
}
```

Как и в Java, в подклассе определяются новые поля и методы и переопределяются методы суперкласса.

Как и в Java, класс можно объявить финальным с помощью ключевого слова `final`, после чего его уже нельзя будет унаследовать. В отличие от Java, можно объявлять финальными отдельные методы и поля, которые не должны переопределяться. (Дополнительные сведения о переопределении полей приводятся в разделе 8.6 «Переопределение полей».)

8.2. Переопределение методов

При переопределении неабстрактных методов в Scala *обязательно* следует использовать модификатор `override`. (Об абстрактных методах рассказывается в разделе 8.8 «Абстрактные классы».) Например:

```
public class Person {
    ...
    override def toString = getClass.getName + "[name=" + name + «]»
}
```

Модификатор `override` генерирует полезные *сообщения об ошибках* в следующих типичных ситуациях:

- когда допущена опечатка в имени переопределяющего метода;
- когда в переопределяющем методе по ошибке указан параметр неверного типа;
- когда в суперкласс вводится новый метод, имя которого конфликтует с именем метода в подклассе.

> **Примечание.** Последний случай является примером проблемы «уязвимости базового класса», когда изменение суперкласса без проверки всех его подклассов может стать источником проблем. Допустим, программист Алиса определила класс `Person`, и без ее ведома программист Боб определил подкласс `Student` с методом `id`, возвращающим идентификационный номер студента. Позднее Алиса также определила метод `id`, возвращающий государственный идентификационный номер человека. Когда Боб примет это изменение, правильность работы его программы (но не испытательных тестов, которыми пользуется Алиса) может нарушиться, потому что объекты `Student` теперь будут возвращать неожиданные идентификационные номера.

В Java эту проблему часто рекомендуется «решать», объявляя все методы финальными, если они явно не предусматривают возможности

переопределения. В теории выглядит неплохо, но программисты не любят лишаться возможности вносить в методы даже самые безобидные изменения (например, вывод записей в файл журнала). Именно поэтому в Java появилась дополнительная аннотация `@Override`.

Вызов метода суперкласса в Scala выполняется точно так же, как в Java, с помощью ключевого слова `super`:

```
public class Employee extends Person {
     ...
     override def toString = super.toString + "[salary=" + salary + «]»
}
```

Инструкция `super.toString` вызовет метод `toString` суперкласса, то есть метод `Person.toString`.

8.3. Проверка и приведение типов

Для проверки принадлежности объекта к заданному классу используется метод `isInstanceOf`. При положительном результате проверки можно воспользоваться методом `asInstanceOf`, чтобы преобразовать ссылку в ссылку на подкласс:

```
if (p.isInstanceOf[Employee]) {
     val s = p.asInstanceOf[Employee] // s имеет тип Employee
     ...
}
```

Выражение `p.isInstanceOf[Employee]` будет истинно, если `p` ссылается на объект класса `Employee` или его подкласс (такой как `Manager`).

Если `p` имеет значение `null`, проверка `p.isInstanceOf[Employee]` вернет `false` и вызов `p.asInstanceOf[Employee]` вернет `null`.

Если `p` ссылается не на экземпляр класса `Employee`, тогда вызов `p.asInstanceOf[Employee]` возбудит исключение.

Чтобы убедиться, что `p` ссылается на объект именно класса `Employee`, а не одного из его подклассов, следует использовать проверку

```
if (p.getClass == classOf[Employee])
```

Метод `classOf` определен в объекте `scala.Predef`, который импортируется всегда.

В табл. 8.1 перечислены соответствия между инструкциями проверки и приведения типов в Scala и Java.

Таблица 8.1. Проверка и приведение типов в Scala и Java

Scala	Java
`obj.isInstanceOf[Cl]`	`obj instanceof Cl`
`obj.asInstanceOf[Cl]`	`(Cl) obj`
`classOf[Cl]`	`Cl.class`

Однако для проверки и приведения типов обычно лучше использовать механизм сопоставления с образцами. Например:

```
p match {
    case s: Employee => ... // Обработать s как экземпляр Employee
    case _           => ... // p не является экземпляром Employee
}
```

Дополнительная информация приводится в главе 14.

8.4. Защищенные поля и методы

Как в Java или C++, имеется возможность объявлять поля и методы защищенными (`protected`). Такие члены будут доступны только подклассам и недоступны всем остальным.

В отличие от Java, защищенные члены *невидимы* в области пакета, которому принадлежит класс. (Если они должны быть видимы, следует использовать модификатор с именем пакета, как описывается в главе 7.)

Существует также вариант модификатора `protected[this]`, ограничивающий доступность к текущему объекту, по аналогии с вариантом `private[this]`, обсуждавшимся в главе 5.

8.5. Создание суперклассов

В главе 5 говорилось, что класс имеет один главный конструктор и может иметь произвольное количество дополнительных конструкторов, и что все дополнительные конструкторы должны начинаться с вызова предыдущего дополнительного или главного конструктора.

Как следствие дополнительные конструкторы никогда не вызывают конструктор суперкласса непосредственно.

В конечном итоге они вызывают главный конструктор подкласса. И только главный конструктор подкласса может вызвать конструктор суперкласса.

Не забывайте, что главный конструктор вплетается в определение класса. Вызов конструктора суперкласса также оказывается вплетенным в определение подкласса. Например:

```
class Employee(name: String, age: Int, val salary : Double) extends
     Person(name, age)
```

Это определение объявляет подкласс

```
class Employee(name: String, age: Int, val salary : Double) extends
     Person(name, age)
```

и главный конструктор, вызывающий конструктор суперкласса

```
class Employee(name: String, age: Int, val salary : Double) extends
Person(name, age)
```

Такое переплетение класса и конструктора обеспечивает очень краткую форму записи. Для простоты понимания параметры главного конструктора можно интерпретировать как параметры класса. Здесь класс `Employee` имеет три параметра: `name`, `age` и `salary`, два из которых «передаются» суперклассу.

Эквивалентный код на языке Java выглядит более пространно:

```
public class Employee extends Person { // Java
     private double salary;
     public Employee(String name, int age, double salary) {
          super(name, age);
          this.salary = salary;
     }
}
```

> **Примечание.** В языке Scala конструкторы никогда не вызывают `super(params)`, как в Java, чтобы вызвать конструктор суперкласса.

Классы Scala могут наследовать классы Java. В этом случае главный конструктор должен вызвать один из конструкторов суперкласса Java. Например:

```
class Square(x: Int, y: Int, width: Int) extends
     java.awt.Rectangle(x, y, width, width)
```

8.6. Переопределение полей

В главе 5 говорилось, что свойство в Scala состоит из приватного поля и методов акцессоров/мутаторов. Вы можете переопределить поле `val` (или определение `def` метода без параметров) другим полем `val` с тем же именем. Подкласс получит приватное поле и общедоступный метод чтения, при этом метод чтения переопределит соответствующий метод суперкласса. Например:

```
class Person(val name: String) {
     override def toString = getClass.getName + "[name=" + name + "]"
}

class SecretAgent(codename: String) extends Person(codename) {
     override val name = "secret"      // Скрыть имя ...
     override val toString = "secret"  // ... и имя класса
}
```

Этот пример показывает саму механику, но он довольно искусственный. Более типичный пример – переопределение абстрактных объявлений `def` объявлениями `val`, как показано ниже:

```
abstract class Person { Абстрактные классы рассматриваются в разделе 8.8
     def id: Int // Каждый имеет идентификационный номер,
                 // который вычисляется некоторым способом
     ...
}
class Student(override val id: Int) extends Person
     // Идентификационный номер студента просто передается конструктору
```

Обратите внимание на следующие ограничения (см. табл. 8.2):

- объявление `def` может переопределить только другое объявление `def`;
- объявление `val` может переопределить лишь другое объявление `val` или объявление `def` без параметров;
- объявление `var` может переопределить только абстрактное объявление `var` (см. раздел 8.8 «Абстрактные классы»).

> **Примечание.** В главе 5 говорилось, что предпочтительнее использовать поля `var`, потому что в этом случае всегда можно изменить реализацию и добавить пару методов чтения/записи. Однако это не оставляет выбора программистам, наследующим такой класс. Они не смогут переопределить поле `var` и добавить свою пару методов чтения/записи. Иными словами, если определить поле как `var`, все подклассы вынуждены будут использовать только его.

Таблица 8.2. Переопределение объявлений val, def и var

	Объявлением val	Объявлением def	Объявлением var
Переопределяется val	• подкласс получает приватное поле (с тем же именем, что и поле суперкласса, – это нормально); • метод чтения переопределяет метод чтения суперкласса	Ошибка	Ошибка
Переопределяется def	• подкласс получает приватное поле; • метод чтения переопределяет метод суперкласса	Как в Java	Объявление `var` может переопределить пару методов чтения/записи. Переопределение только метода чтения является ошибкой
Переопределяется var	Ошибка	Ошибка	Только если объявление `var` в суперклассе является абстрактным (см. раздел 8.8)

8.7. Анонимные подклассы

Как и в Java, имеется возможность создать экземпляр *анонимного* подкласса, если заключить его определение в блок, как показано ниже:

```
val alien = new Person("Fred") {
     def greeting = "Greetings, Earthling! My name is Fred."
}
```

Технически эта инструкция создаст объект *структурного типа* – подробности см. в главе 18. Получившийся тип обозначается как `Person{def greeting: String}`. Этот тип можно использовать в качестве типа параметра:

```
def meet(p: Person{def greeting: String}) {
     println(p.name + " says: " + p.greeting)
}
```

8.8. Абстрактные классы

Как и в Java, имеется возможность использовать ключевое слово `abstract` для обозначения класса, который не может использоваться для создания экземпляров, обычно из-за отсутствия определения одного или более методов. Например:

```
abstract class Person(val name: String) {
     def id: Int // Метод без тела - это абстрактный метод
}
```

Здесь мы говорим, что каждый человек имеет идентификационный номер, но не знаем, как он вычисляется. Каждый конкретный подкласс класса `Person` должен определить реализацию метода `id`. В Scala, в отличие от Java, абстрактные методы не требуется обозначать ключевым словом `abstract` – достаточно просто опустить тело. Как и в Java, класс хотя бы с одним абстрактным методом должен быть объявлен абстрактным.

При переопределении абстрактного метода в подклассе не требуется использовать ключевое слово `override`.

```
class Employee(name: String) extends Person(name) {
     def id = name.hashCode // ключевое слово override не требуется
}
```

8.9. Абстрактные поля

В дополнение к абстрактным методам класс может также содержать абстрактные поля. Абстрактное поле – это обычное поле, но без начального значения. Например:

```
abstract class Person {
val id: Int
     // Нет начального значения - абстрактное поле
     // с абстрактным методом чтения
var name: String
     // Другое абстрактное поле с абстрактными методами чтения/записи
}
```

Этот класс определяет абстрактные методы чтения для полей `id` и `name` и абстрактный метод записи для поля `name`. В сгенерированном Java-классе *поля отсутствуют*.

Конкретные подклассы должны реализовать *конкретные поля*, например:

```
class Employee(val id: Int) extends Person { // Подкласс с конкретным
                                             // свойством id
     var name = "" // и конкретным свойством name
}
```

Как и в случае с методами, в подклассах не требуется использовать ключевое слово `override` при определении поля, которое было абстрактным в суперклассе.

Абстрактное поле всегда можно определять без указания типа:

```
val fred = new Person {
     val id = 1729
     var name = "Fred"
}
```

8.10. Порядок создания и опережающие определения L3

При переопределении поля `val` в подклассе *и* использовании значения в конструкторе суперкласса получающееся поведение выглядит неочевидным.

Например. Некое существо (`Creature`) может воспринимать окружающий его мир своими органами чувств. Для простоты предположим, что существо живет в одномерном мире, а его восприятие окружающего мира можно описать целыми числами. По умолчанию существо видит на десять шагов вперед.

```
class Creature {
     val range: Int = 10
     val env: Array[Int] = new Array[Int](range)
}
```

Муравьи более близоруки:

```
class Ant extends Creature {
     override val range = 2
}
```

К сожалению, мы столкнулись с проблемой. Значение `range` используется в конструкторе суперкласса, а конструктор суперкласса выполняется перед конструктором подкласса. В данном случае происходит следующее:

1. Конструктор `Ant` вызывает конструктор `Creature` перед началом своих действий.
2. Конструктор `Creature` устанавливает поле `range` равным `10`.
3. Далее конструктор `Creature` инициализирует массив, вызывая метод чтения `range()`.
4. Этот метод переопределен и должен вернуть (пока неинициализированное) значение поля `range` класса `Ant`.
5. Метод `range` возвращает `0`. (То есть начальное значение любого целочисленного поля, определяемое при размещении объекта в памяти.)
6. Поле `env` становится массивом с длиной 0.
7. Управление возвращается в конструктор `Ant`, и выполняется попытка присвоить значение 2 полю `range`.

Даже при том, что со стороны кажется, что поле `range` получит значение 10 или 2, однако поле `env` будет инициализировано массивом с нулевой длиной. Этот пример наглядно демонстрирует, что вы не должны полагаться на значение `val` в теле конструктора.

В Java существует похожая проблема, когда конструктор суперкласса вызывает метод. Метод может быть переопределен в подклассе и не делать того, чего вы ожидаете. (Именно в этом и заключается наша проблема – выражение `range` вызывает метод чтения.)

Существует несколько ее решений.

- Объявить поле `val` как `final`. Это безопасное, но негибкое решение.
- Объявить `val` как `lazy` в суперклассе (см. главу 2). Это безопасное, но неэффективное решение.
- Использовать в подклассе синтаксис *опережающего определения* – см. ниже.

Синтаксис «опережающего определения» позволяет организовать инициализацию полей `val` в подклассах *до* того, как управление будет передано суперклассу. Синтаксис настолько неуклюж, что только мамочка сможет любить его. Суть его заключается в том, чтобы заключить поля `val` в блок, следующий за ключевым словом `extends`, как показано ниже:

```
class Bug extends {
     override val range = 3
} with Creature
```

Обратите внимание на ключевое слово `with` перед именем суперкласса. Обычно это ключевое слово используется с трейтами (см. главу 10).

Справа от оператора присвоения в опережающем определении допускается ссылаться только на предыдущие опережающие определения, на другие поля и методы класса ссылаться нельзя.

Совет. Для отладки проблем, связанных с порядком конструирования объектов, можно использовать флаг компилятора `-Xcheckinit`. Этот флаг генерирует код, возбуждающий исключение (вместо возврата значения по умолчанию) при обращении к неинициализированным полям.

Примечание. Проблема, связанная с порядком конструирования объектов, корнями уходит в дизайн языка Java, а именно в допустимость вызова методов подкласса из конструктора суперкласса. В языке C++, когда выполняется конструктор суперкласса, указатель на таблицу виртуальных функций объекта ссылается на таблицу суперкласса. Впоследствии в указатель записывается ссылка на таблицу подкласса. Благодаря этому в C++ исключается возможность изменять поведение конструктора через переопределение методов. Проектировщики языка Java посчитали эту особенность несущественной, и виртуальная машина Java не изменяет указатель на таблицу виртуальных методов в процессе создания объектов.

8.11. Иерархия наследования в Scala

На рис. 8.1 представлено дерево наследования классов в языке Scala. Классы, соответствующие простым типам Java, а также тип `Unit` наследуют класс `AnyVal`.

Все остальные классы являются подклассами класса `AnyRef`, который является аналогом класса `Object` в виртуальной машине Java или .NET.

Оба класса, `AnyVal` и `AnyRef`, наследуют класс `Any`, являющийся корнем дерева наследования.

Класс `Any` определяет методы `isInstanceOf`, `asInstanceOf`, а также методы сравнения и вычисления хеш-кода, с которыми мы познакомимся в разделе 8.12 «Равенство объектов».

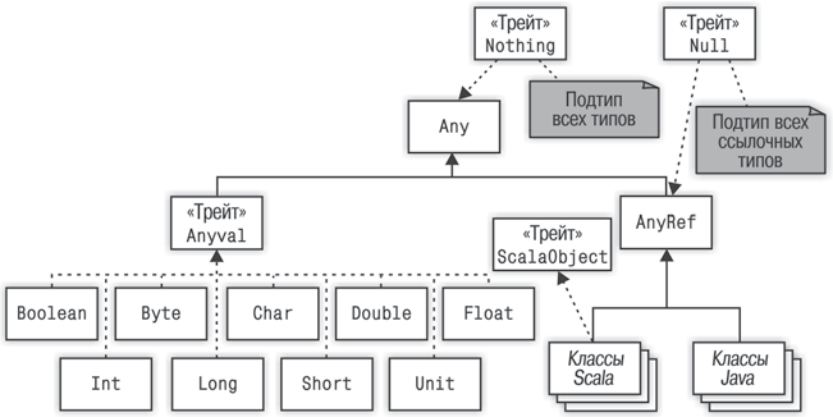


Рис. 8.1. Дерево наследования классов в языке Scala

Класс `AnyVal` не добавляет новых методов. Он служит лишь маркером типов-значений.

Класс `AnyRef` добавляет методы мониторинга `wait` и `notify/notifyAll` из класса `Object`. Он также добавляет метод `synchronized` с параметром-функцией. Этот метод является эквивалентом блока `synchronized` в Java. Например:

```
account.synchronized { account.balance += amount }
```

> **Примечание.** По аналогии с языком Java я рекомендую не использовать методы `wait`, `notify` и `synchronized` без веских на то причин, а пользоваться более высокоуровневыми конструкциями поддержки многопоточной модели выполнения.

Все классы в языке Scala реализуют интерфейс-маркер `ScalaObject`, не имеющий методов[1].

На другом конце иерархии находятся типы `Nothing` и `Null`.

`Null` – это тип, имеющий единственный экземпляр со значением `null`. Значение `null` можно присвоить любой ссылке и нельзя присвоить переменной никакого другого типа. Например, присвоить `null` переменной типа `Int` нельзя. В Java, напротив, значение `null` можно присвоить обертке `Integer`.

[1] В версии 2.10 этот интерфейс-маркер был удален как не несущий полезной нагрузки. – *Прим. перев.*

Тип Nothing не имеет экземпляров. Однако его можно использовать для реализации обобщенных конструкций. Например, пустой список Nil имеет тип List[Nothing], являющийся подтипом List[T] для любого T.

Внимание. Тип Nothing – это не то же самое, что void в Java или C++. В Scala тип void представлен типом Unit, единственным значением которого является (). Обратите внимание, что Unit не является супертипом какого-либо другого типа. Однако компилятор позволяет замещать любые значения значением (). Например:

```
def printAny(x: Any)   { println(x) }
def printUnit(x: Unit) { println(x) }
printAny("Hello")         // Выведет Hello
printUnit("Hello")
     // Заменит "Hello" на () и вызовет printUnit(()), который выведет ()
```

8.12. Равенство объектов L1

В Scala метод eq класса AnyRef проверяет, указывают ли две ссылки на один и тот же объект. Метод equals класса AnyRef вызывает eq. При реализации класса необходимо подумать о переопределении метода equals, чтобы обеспечить более естественную реализацию понятия равенства в вашей ситуации.

Например, при определении класса Item(val description: String, val price: Double) может оказаться желательным сравнивать два элемента по описанию (description) и цене (price). Ниже приводится соответствующая реализация метода equals:

```
final override def equals(other: Any) = {
     val that = other.asInstanceOf[Item]
     if (that == null) false
     else description == that.description && price == that.price
}
```

Примечание. Мы определили метод как финальный (final), потому что в общем случае достаточно сложно корректно унаследовать реализацию сравнения в подклассах. Проблема кроется в сохранении симметрии. Обычно бывает желательно, чтобы выражение a.equals(b) давало тот же результат, что и b.equals(a), даже когда b является экземпляром подкласса.

Внимание. Не забывайте определять метод equals с параметром типа Any. Следующее определение неверно:

```
final def equals(other: Item) = { ... }
```

Это другой метод, он не переопределяет метод equals класса AnyRef.

Переопределяя метод equals, не забудьте также переопределить метод hashCode[1]. Хеш-код должен вычисляться только на основе полей, используемых для сравнения. В примере с классом Item объедините хеш-коды полей:

```
final override def hashCode = 13*description.hashCode + 17*price.hashCode
```

> **Совет.** Переопределение методов equals и hashCode не является обязательным требованием. Для многих классов предпочтительнее считать разными разные объекты. Например, если никому не придет в голову считать одинаковыми два разных потока ввода/вывода или две радиокнопки.

В прикладных программах обычно не требуется вызывать eq или equals. Достаточно воспользоваться оператором ==. Для ссылочных типов он вызовет метод equals после того, как убедится, что ни один из операндов не равен null.

Упражнения

1. Определите класс CheckingAccount, наследующий класс BankAccount, который взимает $1 комиссионных за каждую операцию пополнения или списания.

```
class BankAccount(initialBalance: Double) {
     private var balance = initialBalance
     def deposit(amount: Double)  = { balance += amount; balance }
     def withdraw(amount: Double) = { balance -= amount; balance }
}
```

2. Определите класс SavingsAccount, наследующий класс BankAccount из предыдущего упражнения, который начисляет проценты каждый месяц (вызовом метода earnMonthlyInterest) и позволяет бесплатно выполнять три операции зачисления или списания каждый месяц. Метод earnMonthlyInterest должен сбрасывать счетчик транзакций.
3. Обратитесь к какой-нибудь книге по языку Java или C++, где приводится пример простой иерархии классов, возможно, вовлекающей работников, животных, геометрические фигуры или нечто подобное. Реализуйте этот пример на языке Scala.

---

[1] Иначе такие объекты будут некорректно работать с коллекциями, использующими хеш-значения своих элементов, такими как HashMap. – *Прим. перев.*

4. Определите абстрактный класс элемента `Item` с методами `price` и `description`. Определите подкласс простого элемента `SimpleItem`, представляющий элемент, цена и описание которого определяются в конструкторе. Используйте тот факт, что объявление `val` может переопределять `def`. Определите класс `Bundle` – пакет элементов, содержащий другие элементы. Его цена должна определяться как сумма цен элементов в пакете. Реализуйте также механизм добавления элементов в пакет и соответствующий метод `description`.
5. Спроектируйте класс точки `Point`, значения координат x и y которой передаются конструктору. Реализуйте подкласс точки с подписью `LabeledPoint`, конструктор которого будет принимать строку с подписью и значения координат x и y, например:

```
new LabeledPoint("Black Thursday", 1929, 230.07)
```

6. Определите абстрактный класс геометрической фигуры `Shape` с абстрактным методом `centerPoint` и подклассы прямоугольника и окружности, `Rectangle` и `Circle`. Реализуйте соответствующие конструкторы в подклассах и переопределите метод `centerPoint` в каждом подклассе.
7. Определите класс квадрата `Square`, наследующий класс `java.awt.Rectangle` и имеющий три конструктора: один, создающий квадрат по указанным координатам угла и ширине, другой, создающий квадрат с углом в точке (0, 0) с указанной шириной, и третий, создающий квадрат с углом в точке (0, 0) с шириной 0.
8. Скомпилируйте классы `Person` и `SecretAgent` из раздела 8.6 «Переопределение полей» и проанализируйте результаты компиляции с помощью javap. Сколько полей `name` вы обнаружили? Сколько методов чтения `name` вы обнаружили? Что они делают? (Подсказка: используйте ключи `-c` и `-private`.)
9. В классе `Creature` из раздела 8.10 «Порядок создания и опережающие определения» замените `val range` на `def`. Что произойдет, когда вы также будете использовать `def` в подклассе `Ant`? Что произойдет, если в подклассе использовать `val`? Почему?
10. Файл `scala/collection/immutable/Stack.scala` содержит определение класса

```
Stack[A] protected (protected val elems: List[A])
```

Объясните назначение ключевых слов `protected`. (Подсказка: вернитесь к обсуждению приватных конструкторов в главе 5.)

Глава 9. Файлы и регулярные выражения

Темы, рассматриваемые в этой главе A1

- 9.1. Чтение строк.
- 9.2. Чтение символов.
- 9.3. Чтение лексем и чисел.
- 9.4. Чтение из URL и других источников.
- 9.5. Чтение двоичных файлов.
- 9.6. Запись в текстовые файлы.
- 9.7. Обход каталогов.
- 9.8. Сериализация.
- 9.9. Управление процессами A2.
- 9.10. Регулярные выражения.
- 9.11. Группы в регулярных выражениях.
- Упражнения.

В этой главе вы узнаете, как решать наиболее типичные задачи обработки файлов, такие как чтение всех строк или слов из файлов или чтение файлов, содержащих числа.

Основные темы этой главы:

- `Source.fromFile(...).getLines.toArray` возвращает все строки из файла;
- `Source.fromFile(...).mkString` возвращает содержимое файла как строку;
- преобразование строк в числа выполняется методами `toInt` и `toDouble`;
- запись в текстовые файлы выполняется Java-классом `PrintWriter`;
- `"regex".r` — это объект `Regex`;
- регулярные выражения с обратными слешами и кавычками можно заключать в `"""..."""`;
- если регулярное выражение содержит группы, извлекать их содержимое можно с помощью синтаксиса

```
for (regex(var1, ...,varn) <- string).
```

9.1. Чтение строк

Прочитать все строки из файла можно с помощью метода `getLines` объекта `scala.io.Source`:

```
import scala.io.Source
val source = Source.fromFile("myfile.txt", "UTF-8")
    // Первый аргумент может быть строкой или объектом java.io.File
    // Определение кодировки можно опустить, если известно, что
    // кодировка файла совпадает с кодировкой по умолчанию в системе
val lineIterator = source.getLines
```

Результатом является итератор (см. главу 13). Его можно использовать для построчной обработки содержимого:

```
for (l <- lineIterator) process l
```

А также поместить все строки в массив или в буфер, вызвав метод `toArray` или `toBuffer` итератора:

```
val lines = source.getLines.toArray
```

Иногда бывает желательно просто прочитать все содержимое файла в одну строку. Эта операция выполняется еще проще:

```
val contents = source.mkString
```

> **Внимание.** Закончив использовать объект `Source`, не забудьте вызвать метод `close`.

9.2. Чтение символов

Для чтения отдельных символов из файла можно использовать объект `Source` непосредственно как итератор, поскольку класс `Source` расширяет `Iterator[Char]`:

```
for (c <- source) обработка c
```

Если потребуется извлечь символ, не передвигая позицию указателя в файле (подобно `istream::peek` в C++ или `PushbackInputStreamReader` в Java), вызовите метод `buffered` объекта `source`. После этого следую-

щий символ можно будет прочитать вызовом метода head без изменения позиции чтения в файле.

```
val source = Source.fromFile("myfile.txt", "UTF-8")
val iter = source.buffered
while (iter.hasNext) {
     if (iter.head is nice)
          обработать iter.next
     else
          ...
}
source.close()
```

С другой стороны, если файл не слишком большой, можно просто прочитать его содержимое в строку и обработать ее:

```
val contents = source.mkString
```

9.3. Чтение лексем и чисел

Ниже демонстрируется способ «на скорую руку» чтения лексем, разделенных пробелами:

```
val tokens = source.mkString.split("\\s+")
```

Преобразование строки в число можно выполнить с помощью метода toInt или toDouble. Например, если имеется файл, содержащий вещественные числа, их все можно прочитать в массив

```
val numbers = for (w <- tokens) yield w.toDouble
```

или

```
val numbers = tokens.map(_.toDouble)
```

> **Совет.** Помните, для обработки файлов, содержащих смесь из текста и чисел, всегда можно использовать класс java.util.Scanner.

Наконец, обратите внимание на возможность чтения чисел из консоли:

```
print("How old are you? ")
     // Объект Console импортируется по умолчанию, поэтому не требуется
```

```
    // указывать квалифицированные имена методов print и readInt
val age = readInt()
    // Или readDouble, или readLong
```

Внимание. Эти методы предполагают, что будет введена строка, содержащая единственное число, без начальных и завершающих пробелов. В противном случае будет возбуждено исключение `NumberFormatException`.

9.4. Чтение из URL и других источников

Объект `Source` содержит методы для чтения из источников, отличающихся от файлов:

```
val source1 = Source.fromURL("http://horstmann.com", "UTF-8")
val source2 = Source.fromString("Hello, World!")
    // Читает из указанной строки - удобно для отладки
val source3 = Source.stdin
    // Читает из стандартного ввода
```

Внимание. При чтении из URL необходимо заранее знать кодировку символов. Определить ее можно, например, из HTTP-заголовка. За дополнительной информацией обращайтесь по адресу www.w3.org/International/O-charset.

9.5. Чтение двоичных файлов

Scala не предусматривает собственных средств чтения двоичных файлов. Поэтому вам придется использовать стандартную библиотеку Java. Ниже показано, как прочитать весь файл в массив байтов:

```
val file = new File(filename)
val in = new FileInputStream(file)
val bytes = new Array[Byte](file.length.toInt)
in.read(bytes)
in.close()
```

9.6. Запись в текстовые файлы

Scala не имеет встроенной поддержки записи в файлы. Чтобы выполнить запись в текстовый файл, можно воспользоваться классом `java.io.PrintWriter`, например:

```
val out = new PrintWriter(«numbers.txt»)
for (i <- 1 to 100) out.println(i)
out.close()
```

Все работает как ожидается, кроме метода `printf`. При передаче числа методу `printf` компилятор будет преобразовывать его в тип `AnyRef`:

```
out.printf("%6d %10.2f",
    quantity.asInstanceOf[AnyRef], price.asInstanceOf[AnyRef]) // Ой
```

Поэтому используйте метод `format` класса `String`[1]:

```
out.print("%6d %10.2f".format(quantity, price))
```

> **Примечание.** Метод `printf` класса `Console` не страдает этой проблемой. Вы можете использовать
>
> `printf("%6d %10.2f", quantity, price)`
>
> для вывода сообщения в консоль.

9.7. Обход каталогов

В настоящее время в языке Scala не существует «официального» класса для обхода всех файлов в каталоге, или рекурсивного обхода дерева каталогов. В этом разделе мы рассмотрим пару альтернативных решений.

> **Примечание.** В предыдущей версии Scala имелись классы `File` и `Directory`. Их все еще можно отыскать в пакете `scala.tools.nsc.io`, в архиве `scala-compiler.jar`.

Совсем несложно самому написать функцию, возвращающую итератор для обхода всех подкаталогов в данном каталоге:

```
import java.io.File
def subdirs(dir: File): Iterator[File] = {
    val children = dir.listFiles.filter(_.isDirectory)
    children.toIterator ++ children.toIterator.flatMap(subdirs _)
}
```

[1] С расширением поддержки интерполяции строк в Scala 2.10 появилась возможность использовать более краткую форму записи: `out.print(f"$quantity%6d $price%10.2f")`. — *Прим. ред.*

С помощью этой функции можно выполнить обход всех подкаталогов:

```
for (d <- subdirs(dir)) обработать d
```

Если вы используете Java 7, можно воспользоваться методом `walkFileTree` класса `java.nio.file.Files`. Этот класс использует интерфейс `FileVisitor`. В Scala для определения операций предпочтение обычно отдается функциональным объектам, а не интерфейсам (даже при том, что в данном случае интерфейс обеспечивает более точное управление – за подробностями обращайтесь к документации в Javadoc). Следующее неявное преобразование адаптирует функцию к интерфейсу:

```
import java.nio.file._
implicit def makeFileVisitor(f: (Path) => Unit) = new SimpleFileVisitor[Path] {
  override def visitFile(p: Path, attrs: attribute.BasicFileAttributes) = {
    f(p)
    FileVisitResult.CONTINUE
  }
}
```

Теперь можно вывести содержимое всех подкаталогов вызовом

```
Files.walkFileTree(dir.toPath, (f: Path) => println(f))
```

Конечно, если нужно не просто вывести список файлов, в функции, передаваемой в метод `walkFileTree`, можно реализовать другие операции.

9.8. Сериализация

В Java *сериализация* используется для передачи объектов другим виртуальным машинам или для сохранения в кратковременной памяти. (Для организации долговременного хранения объектов сериализация может оказаться не самым оптимальным решением – довольно утомительно заниматься решением проблем с разными версиями объектов, которые неизбежно возникают в процессе продолжающейся разработки классов.)

Ниже показано, как объявить сериализуемый класс в Java и Scala.

Java:

```
public class Person implements java.io.Serializable {
     private static final long serialVersionUID = 42L;
     ...
}
```

Scala:

```
@SerialVersionUID(42L) class Person extends Serializable
```

Трейт `Serializable` объявлен в пакете `scala` и не требует импортирования.

> **Примечание.** Аннотацию `@SerialVersionUID` можно опустить, если вполне подходит значение ID по умолчанию.

Сериализация и десериализация выполняется как обычно:

```
val fred = new Person(...)
import java.io._
val out = new ObjectOutputStream(new FileOutputStream("/tmp/test.obj"))
out.writeObject(fred)
out.close()
val in = new ObjectInputStream(new FileInputStream("/tmp/test.obj"))
val savedFred = in.readObject().asInstanceOf[Person]
```

Коллекции в Scala поддерживают сериализацию, поэтому их можно использовать в качестве членов сериализуемых классов:

```
class Person extends Serializable {
     private val friends = new ArrayBuffer[Person]  // OK - ArrayBuffer
                                                    // сериализуется
     ...
}
```

9.9. Управление процессами A2

Для решения рутинных задач, таких как перемещение файлов из одного места в другое или управление множеством файлов, программисты традиционно используют сценарии командной оболочки. Язык командной оболочки позволяет легко определять подмножест-

ва файлов и направлять вывод одной программы на ввод другой. Однако, с точки зрения программирования, языки командных оболочек (shell language) оставляют желать лучшего.

Язык Scala проектировался как инструмент создания разнообразных программ, от самых простых до весьма массивных. Пакет `scala.sys.process` предоставляет вспомогательные средства для взаимодействия с программами командной оболочки. Вы можете писать сценарии на языке Scala, пользуясь всеми его возможностями.

Например:

```
import sys.process._
"ls -al .." !
```

Эта программа выполнит команду `ls -al ..`, которая выведет список файлов в родительском каталоге в стандартный вывод.

Пакет `sys.process` содержит неявное преобразование строк в `ProcessBuilder`. Оператор `!` *выполняет* объект `ProcessBuilder`.

Результатом оператора `!` является код завершения программы: `0` – если программа завершилась благополучно и ненулевое значение – в случае ошибки.

Если вместо `!` использовать `!!` , будет возвращен вывод программы в виде строки:

```
val result = "ls -al .." !!
```

С помощью оператора `#|` можно перенаправить вывод одной программы на ввод другой (pipe):

```
"ls -al .." #| "grep sec" !
```

> **Примечание.** Как видите, библиотека `process` использует команды операционной системы. В данном примере использованы команды `bash`, потому что эта командная оболочка доступна в Linux, Mac OS X и Windows.

Направить вывод в файл можно с помощью оператора `#>`:

```
"ls -al .." #> new File("output.txt") !
```

Чтобы выполнить запись в конец файла, используйте оператор `#>>`:

```
"ls -al .." #>> new File("output.txt") !
```

Чтобы выполнить ввод из файла, используйте `#<`:

```
"grep sec" #< new File("output.txt") !
```

Можно также реализовать ввод из URL:

```
"grep Scala" #< new URL("http://horstmann.com/index.html") !
```

Объединять процессы можно с помощью `p #&& q` (выполнить `q`, если `p` выполнился успешно) и `p #|| q` (выполнить `q`, если `p` завершился неудачей). Но, честно говоря, язык Scala лучше подходит для управления потоком выполнения, чем для создания сценариев, тогда почему бы не реализовать управление потоком выполнения в Scala?

> **Примечание.** Библиотека `process` использует знакомые операторы командной оболочки `| > >> < && ||`, но добавляет к ним приставку `#`, чтобы обеспечить им всем одинаковый приоритет.

Если потребуется воспользоваться библиотекой `process` в другом каталоге или с другими значениями переменных окружения, создайте объект `ProcessBuilder` с помощью метода `apply` объекта `Process`. Укажите команду, начальный каталог и последовательность (*name*, *value*) пар переменных окружения:

```
val p = Process(cmd, new File(dirName), ("LANG", "en_US"))
```

Затем выполните оператор `!`:

```
"echo 42" #| p !
```

9.10. Регулярные выражения

При обработке входных данных часто бывает необходимо использовать *регулярные выражения*. Класс `scala.util.matching.Regex` предоставляет простой способ их использования. Создать объект `Regex` можно с помощью метода `r` класса `String`:

```
val numPattern = "[0-9]+".r
```

Если регулярное выражение содержит обратные слеши или кавычки, можно воспользоваться синтаксисом неинтерпретируемых строк, `"""..."""`. Например:

```
val wsnumwsPattern = """\s+[0-9]+\s+""".r
    // Проще читается, чем "\\s+[0-9]+\\s+".r
```

Метод `findAllIn` возвращает итератор, выполняющий обход всех совпадений. Его можно использовать в цикле `for`:

```
for (matchString <- numPattern.findAllIn("99 bottles, 98 bottles"))
    обработка matchString
```

или превратить итератор в массив:

```
val matches = numPattern.findAllIn("99 bottles, 98 bottles").toArray
    // Array(99, 98)
```

Чтобы отыскать первое совпадение в строке, можно воспользоваться методом `findFirstIn`. В результате вы получите `Option[String]`. (Описание класса `Option` можно найти в главе 14.)

```
val m1 = wsnumwsPattern.findFirstIn("99 bottles, 98 bottles")
    // Some(" 98 ")
```

Проверить, находится ли совпадение в начале строки, можно с помощью `findPrefixOf`:

```
numPattern.findPrefixOf("99 bottles, 98 bottles")
    // Some(99)
wsnumwsPattern.findPrefixOf("99 bottles, 98 bottles")
    // None
```

Имеется возможность заменить первое совпадение или все совпадения:

```
numPattern.replaceFirstIn("99 bottles, 98 bottles", "XX")
    // "XX bottles, 98 bottles"
numPattern.replaceAllIn("99 bottles, 98 bottles", "XX")
    // "XX bottles, XX bottles"
```

9.11. Группы в регулярных выражениях

Группы используются для получения совпадений с подвыражениями в регулярных выражениях. Добавьте круглые скобки вокруг подвыражений, совпадения с которыми хотелось бы получить, например:

```
val numitemPattern = "([0-9]+) ([a-z]+)".r
```

Чтобы извлечь совпадения с группами, используйте объект регулярного выражения как «экстрактор» (см. главу 14), например:

```
val numitemPattern(num, item) = "99 bottles"
    // Присвоит num значение "99", а item – значение "bottles"
```

Если потребуется извлечь содержимое групп из нескольких совпадений, используйте инструкцию `for`:

```
for (numitemPattern(num, item) <- numitemPattern.findAllIn("99 bottles, 98
bottles"))
    обработать num и item
```

Упражнения

1. Напишите на языке Scala код, который размещает строки в файле в обратном порядке (последнюю делает первой и т. д.).
2. Напишите программу на языке Scala, которая читает файл с символами табуляции, заменяя их пробелами так, чтобы сохранить правильное расположение границ столбцов, и записывает результат в тот же файл.
3. Напишите фрагмент кода на Scala, который читает файл и выводит в консоль все слова, содержащие 12 или более символов. Дополнительные баллы начисляются тем, кто сможет сделать это в одной строке кода.
4. Напишите программу на Scala, которая читает текстовый файл, содержащий только вещественные числа, выводит сумму, среднее, максимальное и минимальное значения.
5. Напишите программу на Scala, которая записывает степени двойки и их обратные величины в файл с экспонентой от 0 до 20. Расположите числа в столбцах:

```
1    1
2    0.5
4    0.25
...  ...
```

6. Напишите регулярное выражение для поиска строк в кавычках `"как эта, возможно с \" или \\"` в программе на языке Java или

C++. Напишите программу на Scala, которая выводит все такие строки, найденные в файле с исходными текстами.

7. Напишите программу на Scala, которая читает текстовый файл и выводит все лексемы, не являющиеся вещественными числами. Используйте регулярное выражение.
8. Напишите программу на Scala, которая выводит атрибуты `src` всех тегов `img` в веб-странице. Используйте регулярные выражения и группы.
9. Напишите программу на Scala, которая подсчитывает количество файлов с расширением `.class` в указанном каталоге и во всех его подкаталогах.
10. Дополните пример сериализуемого класса `Person` возможностью сохранения коллекции друзей. Создайте несколько объектов `Person`, сделайте некоторые из них друзьями других и затем сохраните массив `Array[Person]` в файл. Прочитайте массив из файла и проверьте, не потерялись ли связи между друзьями.

Глава 10. Трейты

Темы, рассматриваемые в этой главе L1

- 10.1. Почему не поддерживается множественное наследование?
- 10.2. Трейты как интерфейсы.
- 10.3. Трейты с конкретными реализациями.
- 10.4. Объекты с трейтами.
- 10.5. Многоуровневые трейты.
- 10.6. Переопределение абстрактных методов в трейтах.
- 10.7. Трейты с богатыми интерфейсами.
- 10.8. Конкретные поля в трейтах.
- 10.9. Абстрактные поля в трейтах.
- 10.10. Порядок конструирования трейтов.
- 10.11. Инициализация полей трейтов.
- 10.12. Трейты, наследующие классы.
- 10.13. Собственные типы L2.
- 10.14. За кулисами.
- Упражнения.

В этой главе вы узнаете, как работать с трейтами (traits). Класс может наследовать один или более трейтов с целью использования их функциональных возможностей. Трейт, в свою очередь, может потребовать от класса реализовать определяемую им функциональность. Однако, в отличие от интерфейсов в Java, трейты в Scala могут предоставлять реализации по умолчанию этих особенностей, что делает их более привлекательными.

Основные темы этой главы:

- класс может реализовать произвольное количество трейтов;
- трейты могут потребовать от классов наличия определенных полей, методов или суперклассов;
- в отличие от интерфейсов в Java, трейт в Scala может предоставлять реализации полей и методов;
- при наследовании нескольких трейтов порядок имеет значение – трейт, чьи методы выполняются первыми, должен находиться в конце списка.

10.1. Почему не поддерживается множественное наследование?

В языке Scala, как и в Java, классы не могут наследовать множество суперклассов. На первый взгляд, это ограничение кажется неразумным. Почему бы не позволить классу наследовать сразу несколько классов? Некоторые языки программирования, в частности C++, поддерживают *множественное наследование*, но цена такой поддержки удивительно высока.

Множественное наследование прекрасно подходит, когда наследуемые классы не имеют *ничего общего*. Но, если классы имеют общие методы или поля, начинают возникать разнообразные проблемы. Ниже приводится типичный пример. Помощник преподавателя (teaching assistant) – это студент (student) и работник (employee) одновременно:

```
class Student {
     def id: String = ...
     ...
}

class Employee {
     def id: String = ...
     ...
}
```

Допустим, что есть возможность определить класс

```
class TeachingAssistant extends Student, Employee { // Недопустимо в Scala
     ...
}
```

К сожалению, такой класс `TeachingAssistant` наследует *два* метода `id`. Что вернет вызов `myTA.id`? Идентификатор студента? Идентификатор работника? Оба? (В C++ потребовалось бы переопределить метод `id`, чтобы прояснить намерения.)

Далее, допустим, что оба класса, `Student` и `Employee`, наследуют общий суперкласс `Person`:

```
class Person {
     var name: String = _
```

```
}

class Student extends Person { ... }
class Employee extends Person { ... }
```

Это ведет к проблеме *ромбовидного наследования* (diamond inheritance) (рис. 10.1). В классе `TeachingAssistant` нам требуется одно поле `name`, а не два. Как будут объединяться унаследованные поля? Как будет создаваться это поле? В C++ для решения этой проблемы используются «виртуальные базовые классы» – сложная и хрупкая особенность.

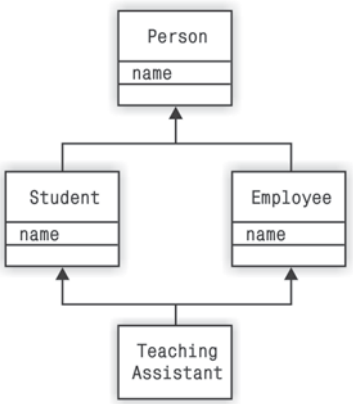


Рис. 10.1. При ромбовидном наследовании возникает проблема объединения общих полей

Создатели Java настолько старались избежать этих сложностей, что выбрали другой подход. Класс может наследовать только один суперкласс и реализовать любое количество *интерфейсов*, но интерфейсы могут иметь только абстрактные методы и не могут иметь полей.

На практике часто возникает необходимость реализовать некоторые методы в терминах других методов, но это невозможно сделать в интерфейсах Java. Поэтому в Java часто используется прием, когда сразу определяются интерфейс и абстрактный базовый класс, но такой подход может оказаться препятствием в будущем. Что, если вам потребуется унаследовать сразу два таких абстрактных базовых класса?

Вместо интерфейсов в Scala используются *трейты* (traits). Трейт может иметь не только абстрактные, но и конкретные методы, а класс может реализовать несколько трейтов сразу. Это решает проблему интерфейсов Java. В следующих разделах вы увидите, как разрешаются проблемы с конфликтующими методами и полями в языке Scala.

10.2. Трейты как интерфейсы

Начнем с чего-нибудь, более или менее знакомого. Трейт в языке Scala может действовать точно так же, как интерфейс в Java. Например:

```
trait Logger {
     def log(msg: String) // Абстрактный метод
}
```

Обратите внимание, что в данном случае не требуется объявлять метод абстрактным – нереализованные методы в трейтах автоматически становятся абстрактными.

Метод может быть реализован в подклассе:

```
class ConsoleLogger extends Logger {       // extends, не implements
     def log(msg: String) { println(msg) } // override не нужно
}
```

При переопределении абстрактных методов трейта не требуется указывать ключевое слово `override`.

> **Примечание.** В Scala отсутствует специальное ключевое слово, обозначающее, что класс реализует трейт. Как будет показано далее в этой главе, трейты более похожи на обычные классы, чем на интерфейсы Java.

Если потребуется унаследовать более одного трейта, дополнительные трейты можно добавить вслед за первым через ключевое слово `with`:

```
class ConsoleLogger extends Logger with Cloneable with Serializable
```

Здесь представлено наследование интерфейсов `Cloneable` и `Serializable` из библиотеки Java, исключительно чтобы продемонстрировать синтаксис. Все интерфейсы Java могут использоваться как трейты.

Как и в Java, классы в Scala могут унаследовать только один суперкласс, но произвольное количество трейтов.

> **Примечание.** Может показаться странным, что перед первым трейтом используется ключевое слово `extends`, а перед всеми остальными – `with`. Однако Scala не видит в этом ничего странного. Для Scala `Logger with Cloneable with Serializable` – это одна сущность, наследуемая классом.

10.3. Трейты с конкретными реализациями

В Scala не требуется, чтобы методы в трейтах были абстрактными. Например, наш класс `ConsoleLogger` можно превратить в трейт:

```
trait ConsoleLogger {
     def log(msg: String) { println(msg) }
}
```

Трейт `ConsoleLogger` предоставляет метод *с реализацией* – в данном случае этот метод выводит сообщение в консоль.

Ниже демонстрируется пример использования такого трейта:

```
class SavingsAccount extends Account with ConsoleLogger {
     def withdraw(amount: Double) {
          if (amount > balance) log("Insufficient funds")
          else balance -= amount
     }
     ...
}
```

Обратите внимание на отличие между Scala и Java. Класс `SavingsAccount` использует конкретную реализацию метода `log` из трейта `ConsoleLogger`. При использовании интерфейса Java подобное невозможно.

В Scala (и в других языках программирования, допускающих такую возможность) мы говорим, что функциональность `ConsoleLogger` «подмешивается» (mixed in) в класс `SavingsAccount`.

> **Примечание.** Возможно, термин «примеси» (или «добавки») пришел из мира приготовления мороженого, где слово «примеси» означает добавки, накладываемые в стаканчик с мороженым, прежде чем он будет передан клиенту, – стадия, которая может нравиться или не нравиться в зависимости от точки зрения.

Внимание. Трейты с конкретным поведением имеют один недостаток. Изменение такого трейта влечет за собой необходимость перекомпилировать все классы, куда подмешивается этот трейт.

10.4. Объекты с трейтами

Трейт можно добавить в отдельный объект на этапе его создания. Для демонстрации воспользуемся трейтом `Logged`, объявленным в стандартной библиотеке Scala. Он напоминает наш трейт `ConsoleLogger`, за исключением того, что не имеет конкретной реализации методов:

```
trait Logged {
     def log(msg: String) { }
}
```

Используем этот трейт в определении класса:

```
class SavingsAccount extends Account with Logged {
     def withdraw(amount: Double) {
          if (amount > balance) log("Insufficient funds")
          else ...
     }
     ...
}
```

Теперь ничего выводиться не будет, что может показаться бессмысленным. Но не торопитесь, такой прием позволяет «подмешать» более подходящий инструмент журналирования на этапе конструирования объекта. Стандартный трейт `ConsoleLogger` наследует трейт `Logged`:

```
trait ConsoleLogger extends Logged {
     override def log(msg: String) { println(msg) }
}
```

Этот трейт можно добавить *на этапе создания объекта*:

```
val acct = new SavingsAccount with ConsoleLogger
```

При вызове метода `log` в объекте `acct` будет вызываться метод `log` трейта `ConsoleLogger`.

Разумеется, в другой объект можно добавить другой трейт:

```
val acct2 = new SavingsAccount with FileLogger
```

10.5. Многоуровневые трейты

В класс или в объект можно добавить несколько трейтов, которые будут вызывать друг друга, начиная с *последнего*. Это может пригодиться, когда необходимо организовать преобразование значений в несколько этапов.

Ниже приводится простой пример. Нам может потребоваться добавлять текущее время и дату в каждое сообщение.

```
trait TimestampLogger extends Logged {
     override def log(msg: String) {
          super.log(new java.util.Date() + « « + msg)
     }
}
```

Кроме того, допустим, что нам необходимо укоротить слишком длинные сообщения:

```
trait ShortLogger extends Logged {
     val maxLength = 15 // Поля в трейтах описываются в разделе 10.8
     override def log(msg: String) {
          super.log(
               if (msg.length<= maxLength) msg
               else msg.substring(0, maxLength - 3) + "..."
          )
     }
}
```

Обратите внимание, что каждый из методов `log` передает измененное значение в вызов `super.log`.

В трейтах конструкция `super.log` имеет *иной* смысл, чем в классах. (Иначе трейты были бы бесполезны – они просто наследовали бы метод `log` трейта `Logged`, который ничего не делает.)

Конструкция `super.log` вызывает следующий трейт в иерархии, поэтому фактически вызываемый метод зависит от порядка следования трейтов. В общем случае обработка трейтов начинается с конца. (В разделе 10.10 «Порядок конструирования трейтов» описывается

возможность организации трейтов не в простую цепочку, а в произвольное дерево.)

Чтобы понять, почему порядок имеет значение, сравните следующие два примера:

```
val acct1 = new SavingsAccount with ConsoleLogger with
     TimestampLogger with ShortLogger
val acct2 = new SavingsAccount with ConsoleLogger with
     ShortLogger with TimestampLogger
```

При попытке списать со счета сумму, превышающую остаток, будет выведено сообщение

```
Sun Feb 06 17:45:45 ICT 2011 Insufficient...
```

Как видите, первым был вызван метод `log` трейта `ShortLogger`, а он, в свою очередь, вызвал `super.log` — метод `log` трейта `TimestampLogger`.

Однако аналогичная попытка с объектом `acct2` вернет сообщение

```
Sun Feb 06 1...
```

Здесь `TimestampLogger` находится в конце списка трейтов. Его метод `log` вызывается первым, и его результат усекается следующим трейтом.

> **Примечание.** При использовании трейтов нельзя заранее сказать, какой метод будет вызван инструкцией `super.someMethod`. Выбор метода зависит от порядка следования трейтов в объекте или классе, использующем их. Это делает ключевое слово `super` более гибким, чем простое наследование.

> **Совет.** Если потребуется точно определить, какой метод будет вызван, можно воспользоваться квадратными скобками: `super[ConsoleLogger].log(...)`. Указанный в скобках тип должен быть непосредственным супертипом – нельзя получить доступ к трейтам или классам, которые находятся дальше в дереве наследования.

10.6. Переопределение абстрактных методов в трейтах

Вернемся к нашему трейту `Logger`, где определена пустая реализация метода `log`.

```
trait Logger {
     def log(msg: String) // Это - абстрактный метод
}
```

Теперь унаследуем его в трейте из предыдущего раздела, реализующем вывод текущего времени. После этого трейт `TimestampLogger` перестанет компилироваться.

```
trait TimestampLogger extends Logger {
     override def log(msg: String) { // Переопределяет абстрактный метод
          super.log(new java.util.Date() + " " + msg) // super.log определен?
     }
}
```

Компилятор определит вызов `super.log` как ошибку.

Согласно обычным правилам наследования, этот вызов никогда не будет считаться корректным – метод `Logger.log` не имеет реализации. Но в действительности, как было показано в предыдущем разделе, заранее неизвестно, какой метод `log` на самом деле будет вызван, – это зависит от порядка смешивания трейтов.

Компилятор Scala считает `TimestampLogger.log` все еще абстрактным, а для смешивания необходима конкретная реализация метода `log`. Поэтому метод следует пометить *двумя* ключевыми словами, `abstract` и `override`, как показано ниже:

```
abstract override def log(msg: String) {
     super.log(new java.util.Date() + " " + msg)
}
```

10.7. Трейты с богатыми интерфейсами

Трейт может иметь массу вспомогательных методов, опирающихся в своей работе на абстрактные методы. Одним из примеров является трейт `Iterator`, определяющий десятки методов в терминах абстрактных методов `next` и `hasNext`.

Попробуем обогатить наш достаточно худосочный API журналирования. Обычно прикладные интерфейсы журналирования позволяют определять уровень каждого сообщения, чтобы отличать обычные информационные сообщения от предупреждений и сообщений об ошибках. Такую возможность легко можно добавить, не принуждая использовать какую-либо политику для вывода сообщений.

```
trait Logger {
     def log(msg: String)
     def info(msg: String) { log("INFO: " + msg) }
     def warn(msg: String) { log("WARN: " + msg) }
     def severe(msg: String) { log("SEVERE: " + msg) }
}
```

Обратите внимание, что теперь трейт представляет собой комбинацию абстрактных и конкретных методов.

Теперь класс, использующий трейт `Logger`, сможет вызывать любые из этих методов, например:

```
class SavingsAccount extends Account with Logger {
     def withdraw(amount: Double) {
          if (amount > balance) severe("Insufficient funds")
          else ...
     }
     ...
     override def log(msg:String) { println(msg); }
}
```

Такое использование конкретных и абстрактных методов широко используется в языке Scala. В Java потребовалось бы объявить интерфейс и отдельный класс, реализующий его (например, `Collection`/`AbstractCollection` или `MouseListener`/`MouseAdapter`).

10.8. Конкретные поля в трейтах

Поле в трейте может быть конкретным или абстрактным. Если в определении поля указывается начальное значение, это поле становится конкретным.

```
trait ShortLogger extends Logged {
     val maxLength = 15 // Конкретное поле
     ...
}
```

Класс, в который будет подмешан этот трейт, получит поле `maxLength`. В общем случае в классе будет создано отдельное поле для каждого конкретного поля, имеющегося в используемых им трейтах. Эти поля не наследуются – они просто добавляются в подкласс. Данное отличие может показаться несущественным, но в действительности оно играет важную роль. Рассмотрим эту особенность поближе.

```
class SavingsAccount extends Account with ConsoleLogger with ShortLogger {
     var interest = 0.0
     def withdraw(amount: Double) {
          if (amount > balance) log("Insufficient funds")
          else ...
     }
```

Обратите внимание, что наш подкласс имеет поле `interest`. Это самое обычное поле класса.

Допустим, что класс `Account` имеет поле.

```
class Account {
     var balance = 0.0
}
```

Класс `SavingsAccount` *унаследует* это поле, как обычно. Объект класса `SavingsAccount` будет иметь все поля, определяемые его суперклассами, наряду с полями, определяемыми в самом подклассе. Объект `SavingsAcccount` можно представить как «начинающийся» с объекта суперкласса (рис. 10.2).

Рис. 10.2. Поля трейта включаются в подкласс

В JVM класс может наследовать только один суперкласс, поэтому поля трейта не могут быть унаследованы тем же способом. Вместо этого поле `maxLength` добавляется в класс `SavingsAccount` вслед за полем `interest`.

Конкретные поля трейта можно считать «инструкциями по сборке» для класса, использующего этот трейт. Любые такие поля становятся полями класса.

10.9. Абстрактные поля в трейтах

Неинициализированное поле в трейте считается абстрактным и должно переопределяться в конкретном подклассе.

Например, следующее поле maxLength является абстрактным:

```
trait ShortLogger extends Logged {
     val maxLength: Int // Абстрактное поле
     override def log(msg: String) {
          super.log(
               if (msg.length <= maxLength) msg
               else msg.substring(0, maxLength - 3) + "...")
                    // Поле maxLength используется в реализации
     }
     ...
}
```

При использовании этого трейта в конкретном классе необходимо явно определить поле maxLength:

```
class SavingsAccount extends Account with ConsoleLogger with ShortLogger {
     val maxLength = 20 // Ключевое слово override не требуется
     ...
}
```

Теперь все выводимые сообщения будут усекаться после 20 символа.

Такой способ параметризации трейтов особенно удобен, когда приходится конструировать объекты «на лету». Вернемся к нашему оригинальному классу SavingsAccount:

```
class SavingsAccount extends Account with Logged { ... }
```

Теперь мы можем организовать усечение сообщений в экземпляре, как показано ниже:

```
val acct = new SavingsAccount with ConsoleLogger with ShortLogger {
     val maxLength = 20
}
```

10.10. Порядок конструирования трейтов

Как и классы, трейты могут иметь в своем теле конструкторы, выполняющие инициализацию полей и другие операции. Например:

```
trait FileLogger extends Logger {
     val out = new PrintWriter("app.log")     // Часть конструктора трейта
     out.println("# " + new Date().toString)  // Так же часть конструктора

     def log(msg: String) { out.println(msg); out.flush() }
}
```

Эти инструкции выполняются на этапе конструирования объекта, подмешивающего трейт.

Конструкторы выполняются в следующем порядке:

- первым вызывается конструктор суперкласса;
- конструкторы трейта выполняются после конструктора суперкласса, но перед конструктором класса;
- конструкторы нескольких трейтов вызываются в порядке слева направо;
- внутри каждого трейта родительские трейты конструируются первыми;
- если несколько трейтов имеют общего родителя, его конструирование второй раз не выполняется;
- после конструкторов всех трейтов вызывается конструктор подкласса.

Рассмотрим, например, следующий класс:

```
class SavingsAccount extends Account with FileLogger with ShortLogger
```

В данном случае конструкторы будут выполнены в следующем порядке:

1. `Account` (суперкласс).
2. `Logger` (родитель первого трейта).
3. `FileLogger` (первый трейт).
4. `ShortLogger` (второй трейт). Обратите внимание, что повторное конструирование родителя `Logger` не выполняется.
5. `SavingsAccount` (класс).

Примечание. Конструкторы вызываются в порядке, обратном «линеаризации» класса. Линеаризация – это список всех супертипов данного типа, составляемый в соответствии с правилом:

```
If C extends C1 with C2 with . . . with Cn, then lin(C) = C » lin(Cn)
» . . . » lin(C2) » lin(C1)
```

Знак » здесь означает «конкатенация и удаление дубликатов слева, в пользу тех, что находятся правее». Например:

```
lin(SavingsAccount)
= SavingsAccount » lin(ShortLogger) » lin(FileLogger) » lin(Account)
= SavingsAccount » (ShortLogger » Logger) » (FileLogger » Logger) »
lin(Account)
= SavingsAccount » ShortLogger » FileLogger » Logger » Account.
```

(Для простоты я опустил типы `ScalaObject`, `AnyRef` и `Any`, находящиеся в конце любой линеаризованной последовательности.)
Линеаризация определяет порядок разрешения для ключевого слова `super` в трейтах. Например, вызов `super` в `ShortLogger` вызовет метод в трейте `FileLogger`, а вызов `super` в `FileLogger` вызовет метод в трейте `Logger`.

10.11. Инициализация полей трейтов

Трейты не могут иметь конструкторов с параметрами. Каждый трейт имеет единственный *конструктор без параметров*.

Примечание. Самое интересное, что отсутствие конструкторов с параметрами – единственное отличие трейтов от классов, с технической точки зрения. Во всем остальном трейты обладают всеми чертами классов, такими как конкретные и абстрактные поля и суперклассы.

Данное ограничение может превратиться в проблему для трейтов, чье поведение желательно было бы настраивать. Рассмотрим в качестве примера трейт `FileLogger`. Было бы здорово иметь возможность указывать имя файла журнала, но трейты не позволяют определять конструкторы с параметрами:

```
val acct = new SavingsAccount with FileLogger("myapp.log")
    // Ошибка: Трейты не могут иметь конструкторов с параметрами
```

Одно из возможных решений было показано в предыдущем разделе. Трейт `FileLogger` может иметь абстрактное поле для хранения имени файла.

```
trait FileLogger extends Logger {
    val filename: String
    val out = new PrintStream(filename)
    def log(msg: String) { out.println(msg); out.flush() }
}
```

Класс, использующий такой трейт, может переопределить поле `filename`. К сожалению, тут кроется ловушка. Прямолинейное решение оказывается неработоспособным:

```
val acct = new SavingsAccount with FileLogger {
     val filename = "myapp.log" // Не работает
}
```

Проблема кроется в порядке конструирования. Конструктор `FileLogger` выполняется *перед* конструктором подкласса. Здесь сложно заметить подкласс. Инструкция `new` создает экземпляр анонимного класса, наследующего `SavingsAccount` (суперкласс) с трейтом `FileLogger`. Инициализация поля `filename` происходит только в анонимном подклассе. Фактически она вообще не происходит – перед вызовом конструктора подкласса будет возбуждено исключение, вызванное обращением по пустой ссылке в конструкторе `FileLogger`.

Решить эту проблему можно с помощью малопонятной особенности, рассматривавшейся в главе 8: с помощью *опережающего определения*. Ниже демонстрируется правильная версия:

```
val acct = new { // Блок опережающего определения после new
     val filename = "myapp.log"
} with SavingsAccount with FileLogger
```

Не очень красиво, но проблема решена. Опережающее определение перед началом обычной последовательности конструирования. Когда вызывается конструктор `FileLogger`, поле `filename` уже оказывается инициализированным.

Если то же самое потребуется сделать в классе, синтаксис этой конструкции будет выглядеть так:

```
class SavingsAccount extends { // Опережающее определение после extends
     val filename = "savings.log"
} with Account with FileLogger {
     ... // Реализация SavingsAccount
}
```

Другой способ – использовать в конструкторе `FileLogger` ленивое значение, как показано ниже:

```
trait FileLogger extends Logger {
     val filename: String
     lazy val out = new PrintStream(filename)
     def log(msg: String) { out.println(msg) } // override не требуется
}
```

Поле `out` будет инициализировано при первом обращении. К этому моменту поле `filename` уже будет инициализировано. Однако ленивые значения весьма неэффективны, поскольку перед каждым их использованием выполняется проверка инициализации.

10.12. Трейты, наследующие классы

Как было показано выше, трейты могут наследовать другие трейты, а конструирование иерархий трейтов – вполне обычное дело. Однако трейты могут также наследовать классы. Такие классы становятся суперклассами для любых классов, куда подмешиваются подобные трейты.

Ниже приводится пример, где трейт `LoggedException` наследует класс `Exception`:

```
trait LoggedException extends Exception with Logged {
    def log() { log(getMessage()) }
}
```

Трейт `LoggedException` имеет метод `log` для вывода сообщения из исключения. Обратите внимание, что метод `log` вызывает метод `getMessage`, унаследованный от суперкласса `Exception`.

Теперь попробуем сформировать класс, подмешав в него этот трейт:

```
class UnhappyException extends LoggedException { // Наследует трейт
    override def getMessage() = "arggh!"
}
```

Суперкласс трейта становится суперклассом нашего класса (как показано на рис. 10.3).

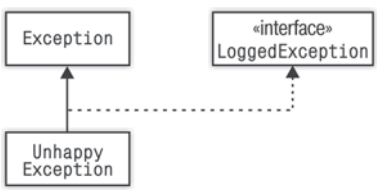


Рис. 10.3. Суперкласс трейта становится суперклассом любого класса, куда подмешивается трейт

А что, если класс уже наследует другой класс? В этом нет ничего особенного, если это подкласс суперкласса трейта. Например:

```
class UnhappyException extends IOException with LoggedException
```

Здесь класс `UnhappyException` наследует класс `IOException`, который уже наследует класс `Exception`. При подмешивании трейта этот суперкласс уже представлен, поэтому нет необходимости добавлять его.

Однако, если класс наследует совершенно посторонний класс, в него нельзя будет подмешать трейт. Например, сформировать следующий класс не удастся:

```
class UnhappyFrame extends JFrame with LoggedException
    // Ошибка: Неродственные суперклассы
```

Невозможно добавить сразу два суперкласса, `JFrame and Exception`.

10.13. Собственные типы L2

Когда трейт наследует класс, гарантируется, что этот класс будет суперклассом для любого класса, подмешивающего трейт. Однако в Scala имеется альтернативный механизм, позволяющий обеспечить такую гарантию: *определение собственного типа* (self types).

Когда трейт начинается с объявления

```
this: Type =>
```

он сможет подмешиваться только в подклассы указанного типа `Type`.

Воспользуемся этой возможностью в нашем трейте `LoggedException`:

```
trait LoggedException extends Logged {
    this: Exception =>
        def log() { log(getMessage()) }
}
```

Обратите внимание, что трейт *не* наследует класс `Exception`. Он просто объявил собственный тип `Exception`. Это означает, что данный трейт может подмешиваться только в подклассы класса `Exception`.

В методах трейта можно вызывать любые методы собственного типа. Например, вызов `getMessage()` в методе `log` вполне допустим, поскольку известно, что `this` имеет тип `Exception`.

Если попытаться подмешать трейт в класс, не совместимый с собственным типом, возникнет ошибка.

```
val f = new JFrame with LoggedException
     // Ошибка: JFrame не является подтипом Exception
     // собственного типа LoggedException
```

Трейт с собственным типом напоминает трейт, наследующий суперкласс. В обоих случаях гарантируется, что в классе, подмешивающем трейт, будут присутствовать поля и методы указанного типа.

Существует несколько ситуаций, когда объявление собственного типа обеспечивает большую гибкость, чем наследование суперкласса. Собственные типы можно использовать для обработки циклических зависимостей между трейтами. Такое может случиться, когда имеются два трейта, нуждающихся друг в друге.

Собственные типы могут также использоваться для определения циклических *структурных типов* – типов, просто определяющих методы, которые должен иметь класс, без указания имени класса. Ниже приводится версия `LoggedException`, использующая структурный тип:

```
trait LoggedException extends Logged {
     this: { def getMessage() : String } =>
          def log() { log(getMessage()) }
}
```

Этот трейт можно подмешивать в любые классы, имеющие метод `getMessage`.

Собственные и структурные типы более подробно будут рассматриваться в главе 18.

10.14. За кулисами

Компилятору Scala необходимо преобразовать трейты в классы и интерфейсы JVM. От вас не требуется знать, как это происходит, но эти знания могут пригодиться в будущем.

Трейт, содержащий только абстрактные методы, просто преобразуется в Java-интерфейс. Например,

```
trait Logger {
     def log(msg: String)
}
```

превратится в

```
public interface Logger { // Сгенерированный Java-интерфейс
     void log(String msg);
}
```

Если трейт имеет конкретные методы, то будет создан объект-компаньон, статические методы которого будут хранить реализацию этих методов трейта. Например,

```
trait ConsoleLogger extends Logger {
     def log(msg: String) { println(msg) }
}
```

превратится в

```
public interface ConsoleLogger extends Logger{ // Java-интерфейс
     void log(String msg);
}

public class ConsoleLogger$class { // Класс-компаньон на Java
     public static void log(ConsoleLogger self, String msg) {
          println(msg);
     }
```

Эти классы-компаньоны не имеют полей. Поля в трейтах превращаются в абстрактные методы доступа в интерфейсе. Когда класс реализует трейт, поля добавляются в этот класс.

Например,

```
trait ShortLogger extends Logger {
     val maxLength = 15 // Конкретное поле
     ...
}
```

транслируется в

```
public interface ShortLogger extends Logger{
     public abstract int maxLength();
     public abstract void weird_prefix$maxLength_$eq(int);
     ...
}
```

Странный метод записи `weird` необходим для инициализации поля. Инициализация поля происходит в методе инициализации класса-компаньона:

```
public class ShortLogger$class {
     public void $init$(ShortLogger self) {
          self.weird_prefix$maxLength_$eq(15)
     }
```

Когда трейт подмешивается в класс, класс получит поле `maxLength` с методами доступа. Конструктор этого класса вызовет метод инициализации.

Если трейт наследует суперкласс, класс-компаньон не наследует этот суперкласс. Вместо этого суперкласс будет наследовать любой класс, реализующий трейт.

Упражнения

1. Класс `java.awt.Rectangle` имеет очень удобные методы `translate` и `grow`, которые, к сожалению, отсутствуют в таких классах, как `java.awt.geom.Ellipse2D`. В Scala эту проблему легко исправить. Определите трейт `RectangleLike` с конкретными методами `translate` и `grow`. Добавьте любые абстрактные методы, которые потребуются для реализации, чтобы трейт можно было подмешивать, как показано ниже:

```
val egg = new java.awt.geom.Ellipse2D.Double(5,10,20,30) with
RectangleLike
egg.translate(10, -10)
egg.grow(10, 20)
```

2. Определите класс `OrderedPoint`, подмешав `scala.math.Ordered[Point]` в `java.awt.Point`. Используйте лексикографическое упорядочение, то есть $(x, y) < (x', y')$, если $x < x'$ или $x = x'$ и $y < y'$.
3. Исследуйте класс `BitSet` и постройте диаграмму всех его суперклассов и трейтов. Игнорируйте типы параметров (все, что внутри `[...]`). Затем постройте линеаризацию трейтов.
4. Реализуйте класс `CryptoLogger`, выполняющий шифрование сообщений с помощью алгоритма Caesar. По умолчанию должен использоваться ключ 3, но должна быть предусмотрена возможность изменить его. Напишите примеры использования с ключом по умолчанию и с ключом –3.

5. Спецификация JavaBeans включает понятие *обработчика события изменения свойства* (property change listener) – стандартный способ организации взаимодействий между компонентами посредством изменения их свойств. Класс `PropertyChangeSupport` является суперклассом для любых компонентов, желающих обеспечить поддержку обработчиков событий изменений свойств. К сожалению, класс, который уже наследует другой суперкласс, такой как `JComponent`, должен повторно реализовать методы. Определите свою реализацию `PropertyChangeSupport` в виде трейта и подмешайте его в класс `java.awt.Point`.
6. В библиотеке Java AWT имеется класс `Container`, наследующий класс `Component`, являющийся коллекцией нескольких компонентов. Например, `Button` – это `Component`, но `Panel` – это `Container`. Это пример составных компонентов в действии. Библиотека Swing имеет классы `JComponent` и `JContainer`, но, если взглянуть внимательнее, можно заметить одну странность. `JComponent` наследует `Container`, даже при том, что нет смысла добавлять другие компоненты, например в `JButton`. В идеале проектировщики предпочли бы организовать иерархию классов, изображенную на рис. 10.4.

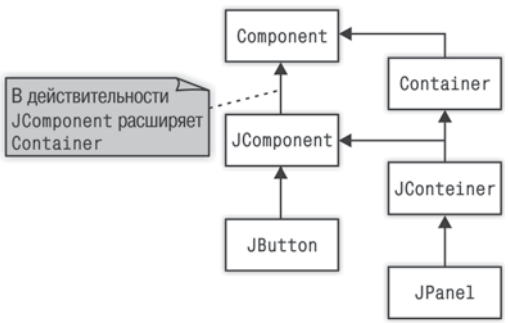


Рис. 10.4. Более удачная иерархия контейнеров Swing

Но в Java это невозможно. Объясните, почему. Как можно было бы организовать классы с применением трейтов в Scala?

7. Существуют десятки руководств, описывающих особенности использования трейтов в языке Scala на простеньких примерах с лающими собаками или философствующими лягушками.

Знакомство с иерархиями, высосанными из пальца, – весьма утомительное и малополезное занятие, а вот создание собственной иерархии действует весьма просветляюще. Создайте собственный простой пример иерархии трейтов, демонстрирующий многоуровневые трейты, конкретные и абстрактные методы, а также конкретные и абстрактные поля.

8. В библиотеке `java.io` имеется возможность добавить буферизацию в поток ввода с помощью декоратора `BufferedInputStream`. Реализуйте буферизацию как трейт. Для простоты переопределите метод `read`.
9. Используя трейты журналирования из этой главы, добавьте поддержку журналирования в предыдущее упражнение, чтобы продемонстрировать процесс буферизации.
10. Реализуйте класс `IterableInputStream`, наследующий `java.io.InputStream` и подмешивающий трейт `Iterable[Byte]`.

Глава 11. Операторы

Темы, рассматриваемые в этой главе L1
- 11.1. Идентификаторы.
- 11.2. Инфиксные операторы.
- 11.3. Унарные операторы.
- 11.4. Операторы присвоения.
- 11.5. Приоритет.
- 11.6. Ассоциативность.
- 11.7. Методы apply и update.
- 11.8. Экстракторы L2.
- 11.9. Экстракторы с одним аргументом или без аргументов L2.
- 11.10. Метод unapplySeq L2.
- Упражнения.

В этой главе рассматриваются особенности реализации собственных *операторов* – методов, синтаксис которых совпадает с обычными математическими операторами. Операторы часто используются для определения *предметно-ориентированных языков* (Domain Specific Language, DSL) – мини-языков, встраиваемых в язык Scala. *Неявные преобразования* (функции преобразования типа, которые вызываются автоматически) – еще один инструмент, упрощающий создание предметно-ориентированных языков. В этой главе также рассматриваются специальные методы `apply`, `update` и `unapply`.

Основные темы данной главы:
- идентификаторы могут содержать любые алфавитно-цифровые символы или символы операторов;
- унарные и двухместные операторы являются вызовами методов;
- приоритет оператора зависит от первого символа, а ассоциативность – от последнего;
- методы `apply` и `update` вызываются при вычислении выражения `expr(args)`;
- экстракторы извлекают кортежи или последовательности значений из входных аргументов L2.

11.1. Идентификаторы

Имена переменных, функций, классов и других элементов называются *идентификаторами*. В Scala у вас имеется более богатый выбор при формировании идентификаторов, чем в Java. Разумеется, вы можете продолжать следовать шаблону, проверенному временем: определять идентификаторы из последовательности алфавитно-цифровых символов, начиная их с буквы или подчеркивания, например: `fred12` или `_Wilma`.

Как и в Java, в идентификаторах допускается использовать символы Юникода. Например, идентификаторы `José` или `σοφός` являются допустимыми.

Кроме того, можно использовать любые последовательности символов операторов:

- ASCII-символы, отличные от букв, цифр, символа подчеркивания, скобок `()[]{}` или разделителей `.,;'"`, то есть любые символы из числа `!#$%&*+-/:<=>?@\^|~`;
- математические символы Юникода и другие символы Юникода из категорий Sm и So.

Например, в идентификаторах можно использовать символы `**` и √. Создав определение

```
val √ = scala.math.sqrt _
```

вы сможете использовать выражение `√(2)` для вычисления квадратного корня. Это может оказаться отличной идеей, если среда программирования позволяет легко вводить подобные символы.

Наконец, при использовании обратных апострофов допускается использовать последовательности любых символов. Например:

```
val `val` = 42
```

Это не самый показательный пример, но обратные апострофы могут иногда оказаться «спасательным кругом». Например, в языке Scala `yield` является зарезервированным словом, тем не менее, используя обратные апострофы, вы получаете возможность обращаться к методу Java с тем же именем: `Thread.`yield`()`.

11.2. Инфиксные операторы

Можно написать:

```
a identifier b
```

где `identifier` обозначает метод с двумя параметрами (один неявный и другой – явный)[1]. Например, выражение

```
1 to 10
```

в действительности является вызовом метода

```
1.to(10)
```

Такое выражение называется *инфиксным*, потому что оператор располагается между аргументами. Оператор может содержать буквы, как оператор `to`, или символы операторов, например

```
1 -> 10
```

– это вызов метода

```
1 .->(10)
```

Чтобы определить оператор для собственного класса, просто определите метод с именем желаемого оператора. Например, ниже приводится объявление класса `Fraction`, реализующего умножение двух рациональных чисел, согласно правилу

$$(n_1 / d_1) \times (n_2 / d_2) = (n_1 n_2 / d_1 d_2)$$

```
class Fraction(n: Int, d: Int) {
     private int num = ...
     private int den = ...
     ...
     def *(other: Fraction) = new Fraction(num * other.num, den * other.den)
}
```

[1] В данном случае метод с одним параметром используется как функция с двумя параметрами, первый из которых неявный (`this`), а второй – явный параметр метода. – *Прим. ред.*

11.3. Унарные операторы

Инфиксные операторы – это двухместные операторы, они имеют два параметра. Оператор с одним параметром называется *одноместным*, или *унарным*. Если такой оператор следует за аргументом, он называется *постфиксным*. Выражение

```
a identifier
```

– это вызов метода `a.identifier()`. Например, выражение

```
1 toString
```

– это фактически вызов метода

```
1.toString()
```

В качестве *префиксных*, то есть перед аргументом, можно использовать четыре оператора `+`, `-`, `!`, `~`. Они преобразуются в вызовы метода с именем `unary_operator`. Например,

```
-a
```

соответствует вызову `a.unary_-`.

11.4. Операторы присвоения

Оператор присвоения имеет вид `operator=`, а выражение

```
a operator= b
```

означает вызов

```
a = a operator b
```

Например, инструкция `a += b` эквивалентна инструкции `a = a + b`. Ниже перечислены некоторые технические особенности.

- Операторы `<=`, `>=` и `!=` не являются операторами присвоения.
- Операторы, начинающиеся с символа `=`, никогда не рассматриваются как операторы присвоения (`==`, `===`, `=/=` и т. д.).
- Если имеется метод с именем `operator=`, вызывается непосредственно этот метод.

11.5. Приоритет

Когда в одном выражении без скобок имеется два или более операторов, первым будет выполнен имеющий наивысший *приоритет*. Например, в выражении

```
1 + 2 * 3
```

оператор * будет выполнен первым. В языках программирования, таких как Java и C++, определено фиксированное количество операторов, и они стандартным образом определяют приоритеты операторов. В языке Scala количество операторов не ограничивается, поэтому в нем используется схема определения приоритетов, распространяющаяся на все операторы и соответствующая общепринятым правилам для стандартных операторов.

Исключение составляют операторы присвоения, приоритет которых определяется *первым символом* оператора.

Высший приоритет: символ оператора отличается от тех, что ниже
* / %
+ -
:
< >
! =
&
^
|
Символы, не являющиеся символами операторов
Низший приоритет: операторы присвоения

Символы, указанные в одной строке, в таблице выше, имеют одинаковый приоритет. Например, операторы + и -> имеют одинаковый приоритет.

Постфиксные операторы имеют более низкий приоритет в сравнении с инфиксными операторами. Например, выражение:

```
a infixOp b postfixOp
```

равносильно выражению

```
(a infixOp b) postfixOp
```

11.6. Ассоциативность

Когда имеется последовательность операторов с одинаковым приоритетом, порядок их выполнения (слева направо или справа налево) определяется их *ассоциативностью*. Например, выражение `17 - 2 - 9` будет вычислено как `(17 - 2) - 9`. Оператор - является левоассоциативным.

В Scala все операторы являются левоассоциативными, кроме:

- операторов, заканчивающихся двоеточием (`:`);
- операторов присвоения.

В частности, оператор `::` конструирования списков является правоассоциативным. Например,

```
1 :: 2 :: Nil
```

означает

```
1 :: (2 :: Nil)
```

Именно так и должно быть – сначала необходимо создать список, содержащий 2, после чего этот список становится концом списка, начинающегося с 1.

Правоассоциативный двухместный оператор – это метод второго аргумента. Например,

```
2 :: Nil
```

означает

```
Nil.::(2)
```

11.7. Методы apply и update

Scala позволяет распространять синтаксис вызова функций

```
f(arg1, arg2, ...)
```

на значения, не являющиеся функциями. Если `f` не является функцией или методом, тогда это выражение будет эквивалентно вызову

```
f.apply(arg1, arg2, ...)
```

если оно не находится слева от оператора присвоения. Выражение

```
f(arg1, arg2, ...) = value
```

соответствует вызову

```
f.update(arg1, arg2, ..., value)
```

Этот механизм используется в обычных и ассоциативных массивах. Например:

```
val scores = new scala.collection.mutable.HashMap[String, Int]
scores("Bob") = 100            // Вызовет scores.update("Bob", 100)
val bobsScore = scores("Bob")  // Вызовет scores.apply("Bob")
```

Метод `apply` также часто используется в объектах-компаньонах для конструирования объектов без использования ключевого слова `new`. Например, рассмотрим класс `Fraction`.

```
class Fraction(n: Int, d: Int) {
    ...
}

object Fraction {
    def apply(n: Int, d: Int) = new Fraction(n, d)
}
```

Благодаря методу `apply` мы можем конструировать объекты рациональных чисел как `Fraction(3, 4)` вместо `new Fraction(3, 4)`. На первый взгляд, это не кажется большим преимуществом, но, когда имеется множество значений `Fraction`, такое усовершенствование становится особенно удобным:

```
val result = Fraction(3, 4) * Fraction(2, 5)
```

11.8. Экстракторы L2

Экстрактор – это объект с методом `unapply`. Метод `unapply` можно считать противоположностью методу `apply` объекта-компаньона. Метод `apply` принимает параметры конструирования и превращает их в объект. Метод `unapply` принимает объект и извлекает из него значения, обычно на основе которых был сконструирован объект.

Рассмотрим класс Fraction из предыдущего раздела. Метод apply создает объект рационального числа из числителя и знаменателя. Метод unapply возвращает числитель и знаменатель. Его можно использовать в определении переменной

```
var Fraction(a, b) = Fraction(3, 4) * Fraction(2, 5)
    // a, b инициализируются значениями числителя и знаменателя результата
```

или в сопоставлении с образцом

```
case Fraction(a, b) => ... // a, b связываются с числителем и знаменателем
```

(Подробнее о сопоставлении с образцами рассказывается в главе 14.)

Операция сопоставления с образцом вообще может терпеть неудачу. Таким образом, метод unapply возвращает объект Option. Он содержит кортеж с одним значением для каждой соответствующей переменной. В данном случае мы получим Option[(Int, Int)].

```
object Fraction {
    def unapply(input: Fraction) =
        if (input.den == 0) None else Some((input.num, input.den))
}
```

Только чтобы показать, что такое возможно, этот метод возвращает None, если знаменатель равен нулю, свидетельствуя об отсутствии совпадения.

В предыдущем примере методы apply и unapply выполняют противоположные операции по отношению друг к другу. Однако это совсем необязательно. Экстракторы можно использовать для извлечения информации из объектов любого типа.

Например, допустим, что необходимо извлечь имя и фамилию из строки:

```
val author = "Cay Horstmann"
val Name(first, last) = author // Вызовет Name.unapply(author)
```

Реализуем объект Name с методом unapply, возвращающим Option[(String, String)]. Если сопоставление увенчается успехом, будет возвращена пара с именем и фамилией. Компоненты пары будут присвоены переменным в образце. В противном случае будет возвращено значение None.

```
object Name {
     def unapply(input: String) = {
          val pos = input.indexOf(" ")
          if (pos == -1) None
          else Some((input.substring(0, pos), input.substring(pos + 1)))
     }
}
```

> **Примечание.** В этом примере не определяется класс `Name`. Объект `Name` – это экстрактор для объектов `String`.

Все case-классы автоматически получают методы `apply` и `unapply`. (Case-классы обсуждаются в главе 14.) Например, взгляните на следующее определение:

```
case class Currency(value: Double, unit: String)
```

Создать экземпляр `Currency` можно таким образом:

```
Currency(29.95, "EUR") // Вызовет Currency.apply
```

А извлечь значения из объекта `Currency`:

```
case Currency(amount, "USD") => println("$" + amount)
     // Вызовет Currency.unapply
```

11.9. Экстракторы с одним аргументом или без аргументов L2

В Scala не существует кортежей с одним элементом. Если метод `unapply` извлечет единственное значение, он просто должен вернуть объект `Option` целевого типа. Например:

```
object Number {
     def unapply(input: String): Option[Int] =
     try {
          Some(Integer.parseInt(input.trim))
     } catch {
          case ex: NumberFormatException => None
     }
}
```

С помощью этого экстрактора можно извлечь число из строки:

```
val Number(n) = "1729"
```

Экстрактор может просто проверять входное значение, ничего не извлекая. В этом случае метод `unapply` должен возвращать логическое значение. Например:

```
object IsCompound {
     def unapply(input: String) = input.contains(" ")
}
```

Экстрактор можно использовать для добавления проверки в образец, например:

```
author match {
     case Name(first, last @ IsCompound()) => ...
          // Совпадет, если автор Peter van der Linden
     case Name(first, last) => ...
}
```

11.10. Метод unapplySeq L2

Чтобы извлечь произвольную последовательность значений, необходимо вызвать метод `unapplySeq`. Он возвращает `Option[Seq[A]]`, где `A` – тип извлекаемых значений. Например, экстрактор `Name` может возвращать последовательность компонентов имен:

```
object Name {
     def unapplySeq(input: String): Option[Seq[String]] =
          if (input.trim == "") None else Some(input.trim.split("\\s+"))
}
```

Теперь в сопоставлении можно указать произвольное количество переменных:

```
author match {
     case Name(first, last) => ...
     case Name(first, middle, last) => ...
     case Name(first, "van", "der", last) => ...
     ...
}
```

Упражнения

1. Как будут вычисляться следующие выражения с учетом правил приоритетов: `3 + 4 -> 5` и `3 -> 4 + 5`?
2. Класс `BigInt` имеет метод `pow`, но не имеет соответствующего оператора. Почему создатели библиотеки Scala не использовали `**` (как в Fortran) или `^` (как в Pascal) в качестве оператора возведения в степень?
3. Реализуйте класс `Fraction` с операциями `+ - * /`. Реализуйте нормализацию рациональных чисел, например чтобы число 15/–6 превращалось в –5/3, а также деление на наибольший общий делитель, как показано ниже:

```
class Fraction(n: Int, d: Int) {
     private val num: Int = if (d == 0) 1 else n * sign(d) / gcd(n, d);
     private val den: Int = if (d == 0) 0 else d * sign(d) / gcd(n, d);
     override def toString = num + "/" + den
     def sign(a: Int) = if (a > 0) 1 else if (a < 0) -1 else 0
     def gcd(a: Int, b: Int): Int = if (b == 0) abs(a) else gcd(b, a % b)
     ...
}
```

4. Реализуйте класс `Money` с полями для выражения суммы в долларах и центах. Реализуйте операторы `+`, `-`, а также операторы сравнения `==` и `<`. Например, выражение `Money(1, 75) + + Money(0, 50) == Money(2, 25)` должно возвращать `true`. Следует ли также реализовать операторы `*` и `/`? Почему «да» или почему «нет»?
5. Реализуйте операторы конструирования HTML-таблицы. Например, выражение

```
Table() | "Java" | "Scala" || "Gosling" | "Odersky" || "JVM" | "JVM, .NET"
```

должно возвращать

```
<table><tr><td>Java</td><td>Scala</td></tr><tr><td>Gosling...
```

6. Реализуйте класс `ASCIIArt`, объекты которого содержат фигуры, такие как

```
 /\_/\
( ' ' )
(  -  )
 | | |
(__|__)
```

Добавьте операторы для объединения двух фигур `ASCIIArt` по горизонтали

```
 /\_/\      -----
( ' ' )   / Hello \
(  -  ) <  Scala |
 | | |    \ Coder /
(__|__)    -----
```

или по вертикали. Выберите операторы с соответствующими приоритетами.

7. Реализуйте класс `BitSequence` для хранения 64-битных последовательностей в виде значений типа `Long`. Реализуйте операторы `apply` и `update` для получения и установки отдельных битов.
8. Реализуйте класс `Matrix` — выберите сами, какую матрицу реализовать: 2×2, квадратную произвольного размера или матрицу $m \times n$. Реализуйте операции `+` и `*`. Последняя должна также позволять выполнять умножение на скаляр, например `mat * 2`. Единственный элемент матрицы должен быть доступен как `mat(row, col)`.
9. Определите операцию `unapply` для класса `RichFile`, извлекающую путь к файлу, имя и расширение. Например, файл `/home/cay/readme.txt` имеет путь `/home/cay`, имя `readme` и расширение `txt`.
10. Определите операцию `unapplySeq` для класса `RichFile`, извлекающую все сегменты пути. Например, для файла `/home/cay/readme.txt` должна быть возвращена последовательность из трех сегментов: `home`, `cay` и `readme.txt`.

Глава 12. Функции высшего порядка

Темы, рассматриваемые в этой главе L1

- 12.1. Функции как значения.
- 12.2. Анонимные функции.
- 12.3. Функции с функциональными параметрами.
- 12.4. Вывод типов.
- 12.5. Полезные функции высшего порядка.
- 12.6. Замыкания.
- 12.7. Преобразование функций в SAM.
- 12.8. Карринг.
- 12.9. Абстракция управляющих конструкций.
- 12.10. Выражение return.
- Упражнения.

Язык программирования Scala соединяет в себе черты объектно-ориентированного и функционального программирования. В функциональных языках программирования функции считаются обычными объектами, которые можно передавать как любые другие данные и манипулировать ими. Это очень удобно, когда требуется передать в алгоритм некоторую операцию. В *функциональном языке* достаточно завернуть операцию в функцию и передать ее как параметр. В этой главе вы увидите, как работать с функциями, принимающими или возвращающими другие функции.

Основные темы этой главы:

- *функции* в Scala – это «обычные объекты», как, например, числа;
- имеется возможность создавать анонимные функции, обычно для передачи другим функциям;
- аргумент-функция определяет действия, которые должны быть выполнены позднее;
- многие методы коллекций принимают функции в качестве параметров для применения их к значениям в коллекциях;

- существует сокращенный синтаксис определения параметров-функций, упрощающий и облегчающий чтение таких определений;
- имеется возможность создавать функции, действующие как блоки кода и выглядящие как встроенные инструкции.

12.1. Функции как значения

В Scala функции являются «обычными объектами», как, например, числа. Функции можно сохранять в переменных:

```
import scala.math._
val num = 3.14
val fun = ceil _
```

Этот фрагмент сохранит в переменной `num` число `3.14`, а в переменной `fun` – функцию `ceil`.

Символ _, следующий за именем функции `ceil`, указывает, что в действительности имеется в виду функция и нужно не забыть передать ей аргументы.

> **Примечание.** Технически символ _ превращает метод `ceil` в функцию. В Scala можно манипулировать только функциями.

Если попытаться выполнить этот код в REPL, переменная `num` получит, что неудивительно, тип `Double`. Тип переменной `fun` будет определен как `(Double) => Double` – то есть как функция, принимающая и возвращающая значение типа `Double`.

Что можно делать с функциями? Две операции:

- *вызывать* их;
- передавать в виде значений, сохраняя в переменных или передавая функциям в виде *параметров*.

Ниже показано, как вызвать функцию, хранящуюся в переменной `fun`:

```
fun(num) // 4.0
```

Как видите, допускается использовать привычный синтаксис вызова функции. Единственное отличие в том, что `fun` – это *переменная, содержащая функцию*, а не некоторая фиксированная функция.

Ниже показано, как передать `fun` другой функции:

```
Array(3.14, 1.42, 2.0).map(fun) // Array(4.0, 2.0, 2.0)
```

Метод `map` принимает функцию, применяет ее ко всем значениям в массиве и возвращает массив значений, которые вернула функция. В этой главе вы увидите множество других методов, принимающих функции в виде параметров.

12.2. Анонимные функции

В Scala необязательно давать имена всем функциям, как необязательно давать имена всем числам. Ниже демонстрируется *анонимная функция*:

```
(x: Double) => 3 * x
```

Эта функция умножает свой аргумент на 3.

Разумеется, эту функцию можно было бы сохранить в переменной:

```
val triple = (x: Double) => 3 * x
```

Это равносильно использованию определения `def`:

```
def triple(x: Double) = 3 * x
```

Но вы не обязаны давать имена функциям. Функции можно просто передавать другим функциям:

```
Array(3.14, 1.42, 2.0).map((x: Double) => 3 * x)
    // Array(9.42, 4.26, 6.0)
```

Здесь мы сообщаем методу `map`: «умножить каждый элемент на 3».

Примечание. При желании аргументы функции можно заключать в фигурные скобки, например вызов

```
Array(3.14, 1.42, 2.0).map{ (x: Double) => 3 * x }
```

можно записать в более распространенной инфиксной нотации (без точки)

```
Array(3.14, 1.42, 2.0) map { (x: Double) => 3 * x }
```

12.3. Функции с функциональными параметрами

В этом разделе будет показано, как реализовать функцию, принимающую другую функцию в качестве параметра. Например:

```
def valueAtOneQuarter(f: (Double) => Double) = f(0.25)
```

Обратите внимание, что параметром может быть *любая* функция, принимающая и возвращающая значение типа `Double`. Функция `valueAtOneQuarter` передает этой функции `0.25` и возвращает полученное значение.

Например:

```
valueAtOneQuarter(ceil _) // 1.0
valueAtOneQuarter(sqrt _) // 0.5 (потому что 0.5 × 0.5 = 0.25)
```

Какой тип имеет функция `valueAtOneQuarter`? Это функция с одним параметром, поэтому ее тип записывается так:

(*parameterType*) => *resultType*

Тип результата *resultType*, очевидно, имеет тип `Double`, а тип параметра *parameterType* уже определен в заголовке функции как `(Double) => Double`. То есть функция `valueAtOneQuarter` имеет тип

```
((Double) => Double) => Double
```

Поскольку `valueAtOneQuarter` – это функция, принимающая другую функцию, она называется *функцией высшего порядка* (hight order function).

Функции высшего порядка могут также *возвращать функции*. Ниже приводится простой пример такой функции:

```
def mulBy(factor : Double) = (x : Double) => factor * x
```

Вызов `mulBy(3)`, например, вернет функцию `(x : Double) => 3 * x`, которую вы уже видели в предыдущем разделе. Вся прелесть функции `mulBy` в том, что она может вернуть функцию, выполняющую умножение на любое число:

```
val quintuple = mulBy(5)
quintuple(20) // 100
```

Функция `mulBy` имеет параметр типа `Double` и возвращает функцию типа `(Double) => Double`. То есть она имеет тип

```
(Double) => ((Double) => Double)
```

12.4. Вывод типов

При передаче анонимной функции другой функции или методу Scala автоматически выводит типы, когда это возможно. Например, вам необязательно писать

```
valueAtOneQuarter((x: Double) => 3 * x) // 0.75
```

Поскольку метод `valueAtOneQuarter` знает, что ему будет передаваться функция типа `(Double) => Double`, можно просто записать

```
valueAtOneQuarter((x) => 3 * x)
```

Как дополнительное преимущество для функций, имеющих единственный параметр, скобки `()`, окружающие параметр, можно опустить:

```
valueAtOneQuarter(x => 3 * x)
```

Еще лучше. Если параметр появляется только однажды справа от `=>`, его можно заменить символом подчеркивания:

```
valueAtOneQuarter(3 * _)
```

Это очень удобно и так намного проще читается: функция, умножающая нечто на 3.

Имейте в виду, что эти сокращения могут применяться, только когда тип параметра известен.

```
val fun = 3 * _                     // Ошибка: невозможно вывести тип
val fun = 3 * (_: Double)           // OK
val fun: (Double) => Double = 3 * _ // OK, потому что указан тип fun
```

Конечно, последнее определение избыточно. Но оно демонстрирует, что происходит, когда функция передается в параметре (которая имеет только такой тип).

12.5. Полезные функции высшего порядка

Отличный способ освоиться в обращении с функциями высшего порядка – попрактиковаться в использовании некоторых часто используемых (и, очевидно, полезных) методов коллекций в библиотеке Scala, принимающих параметры-функции.

Выше уже был представлен метод `map`, применяющий функцию ко всем элементам коллекции и возвращающий коллекцию результатов. Ниже демонстрируется быстрый способ создания коллекции, содержащей `0.1`, `0.2`, ..., `0.9`:

```
(1 to 9).map(0.1 * _)
```

> **Примечание.** Существует один универсальный принцип. Если необходимо создать последовательность значений, посмотрите, возможно, ее можно создать на основе более простой последовательности.

Попробуем напечатать треугольник:

```
(1 to 9).map("*" * _).foreach(println _)
```

Результат:

```
*
**
***
****
*****
******
*******
********
*********
```

Здесь был использован метод `foreach`, который действует подобно методу `map`, за исключением того, что его функция не должна возвращать значение. Метод `foreach` просто применяет функцию к каждому аргументу.

Метод `filter` возвращает все элементы, соответствующие определенному условию. Например, ниже показано, как извлечь из последовательности только четные числа:

```
(1 to 9).filter(_ % 2 == 0) // 2, 4, 6, 8
```

Конечно, это не самый эффективный способ получить данный результат.

Метод `reduceLeft` принимает *двухместную функцию*, то есть функцию с двумя аргументами, и применяет ее ко всем элементам последовательности, двигаясь в направлении слева направо. Например,

```
(1 to 9).reduceLeft(_ * _)
```

это

```
1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9
```

или, строго говоря,

```
(...((1 * 2) * 3) * ... * 9)
```

Обратите внимание на компактную форму записи функции умножения: `_ * _`. Каждый символ подчеркивания обозначает отдельный параметр.

Вам также может пригодиться двухместная функция сортировки. Например,

```
"Mary had a little lamb".split(" ").sortWith(_.length < _.length)
```

вернет массив, отсортированный по возрастанию длин строк: `Array("a", "had", "Mary", "lamb", "little")`.

12.6. Замыкания

В Scala можно определять функции внутри любой области видимости: в пакете, в классе и даже внутри другой функции или метода. В теле функции имеется доступ ко всем переменным, присутствующим в охватывающей области видимости. На первый взгляд, в этом нет ничего примечательного, но обратите внимание, что функция может быть вызвана даже *после выхода из области видимости*.

Например, рассмотрим функцию `mulBy` из раздела 12.3 «Функции с параметрами-функциями» выше.

```
def mulBy(factor : Double) = (x : Double) => factor * x
```

Обратите внимание на следующие вызовы:

```
val triple = mulBy(3)
val half = mulBy(0.5)
println(triple(14) + " " + half(14)) // Выведет 42 7
```

Теперь прокрутим этот фильм медленнее.

- Первый вызов `mulBy` установит переменную `factor` в значение `3`. Эта переменная используется в теле функции `(x : Double) => factor * x`, хранящейся в переменной `triple`. Затем переменная `factor` выталкивается из стека времени выполнения.
- Далее производится повторный вызов `mulBy`, теперь переменная `factor` получит значение `0.5`. Эта переменная используется в теле функции `(x : Double) => factor * x`, хранящейся в переменной `half`.

Каждая переменная возвращает функцию, имеющую собственное значение для переменной `factor`.

Такие функции называют *замыканиями*. Замыкание состоит из программного кода и определений всех нелокальных переменных, используемых в этом программном коде.

Фактически эти функции реализуются как объекты класса, с переменной экземпляра `factor` и методом `apply`, содержащим тело функции.

В действительности совершенно неважно, как реализуется замыкание. Это работа компилятора Scala – обеспечить доступ к нелокальным переменным из тела функции.

> **Примечание.** Замыкания не являются чем-то сложным или удивительным, будучи естественной частью языка. Многие современные языки, такие как JavaScript, Ruby и Python, поддерживают замыкания. Однако Java 7 не поддерживает замыкания, хотя ограниченная их поддержка предполагается в Java 8.

12.7. Преобразование функций в SAM

В Scala, когда необходимо сообщить функции, какие действия следует выполнить, ей передается другая функция в качестве параметра. В Java отсутствуют функции высшего порядка (пока), и программистам на Java приходится прилагать немалые усилия, чтобы добиться того же эффекта. Обычно они помещают операции в класс, реализующий интерфейс, и затем передают методу экземпляр этого класса.

Во многих случаях эти интерфейсы имеют единственный абстрактный метод. Такие типы в Java называют SAM (**S**ingle **A**bstract **M**ethod – единственный абстрактный метод).

Например, допустим, что требуется увеличивать счетчик, когда выполняется щелчок на кнопке.

```
var counter = 0

val button = new JButton("Increment")
button.addActionListener(new ActionListener {
     override def actionPerformed(event: ActionEvent) {
          counter += 1
     }
})
```

Какой объем шаблонного кода! Все выглядело бы гораздо проще, если бы имелась возможность просто передать функцию в метод `addActionListener`, как показано ниже:

```
button.addActionListener((event: ActionEvent) => counter += 1)
```

Для поддержки такого синтаксиса необходимо предоставить неявное преобразование. Эти преобразования подробно будут обсуждаться в главе 21, но вы легко можете добавить собственное, даже не зная всех деталей. Следующее преобразование превращает функцию в экземпляр `ActionListener`:

```
implicit def makeAction(action: (ActionEvent) => Unit) =
     new ActionListener {
          override def actionPerformed(event: ActionEvent) { action(event) }
     }
```

Просто поместите эту функцию в программный код с реализацией пользовательского интерфейса, и вы сможете передавать любые функции `(ActionEvent) => Unit` везде, где ожидается объект `ActionListener`.

12.8. Карринг

Карринг (в честь логика Хаскелла Брукса Карри (Haskell Brooks Curry)) – это процесс превращения функции с двумя аргументами в функцию с одним аргументом. Такая функция возвращает функцию, которая использует второй аргумент.

Каково? Посмотрим, что это значит, на примере. Следующая функция принимает два аргумента:

```
def mul(x: Int, y: Int) = x * y
```

А эта функция принимает один аргумент и возвращает функцию, которая также принимает один аргумент:

```
def mulOneAtATime(x: Int) = (y: Int) => x * y
```

Чтобы умножить два числа, можно вызвать

```
mulOneAtATime(6)(7)
```

Строго говоря, результатом `mulOneAtATime(6)` является функция `(y: Int) => 6 * y`. Этой функции передается число `7`, и она возвращает `42`.

Ниже демонстрируется сокращенная форма определения таких каррированных функций в Scala:

```
def mulOneAtATime(x: Int)(y: Int) = x * y
```

Как видите, множество параметров – это всего лишь украшательство, а не базовая особенность языка программирования. Это забавная теоретическая возможность проникновения в суть вещей, но она имеет практическое применение в Scala. Иногда бывает желательно каррировать параметр функции, чтобы механизм определения типа мог получить более подробную информацию.

Ниже приводится типичный пример. Метод `corresponds` сравнивает две последовательности в соответствии с некоторым критерием. Например:

```
val a = Array("Hello", "World")
val b = Array("hello", "world")
a.corresponds(b)(_.equalsIgnoreCase(_))
```

Обратите внимание, что функция `_.equalsIgnoreCase(_)` передается как каррированый параметр, в отдельной паре скобок `(...)`. Если заглянуть в Scaladoc, можно увидеть, что метод `corresponds` объявлен как

```
def corresponds[B](that: Seq[B])(p: (A, B) => Boolean): Boolean
```

Последовательность `that` и функция-предикат `p` – это отдельные, каррированные параметры. Механизм определения типа сможет за-

метить, что B имеет тип that, и затем использовать эту информацию при анализе функции, которая передается в параметре p.

В нашем примере that – это последовательность строк String. Таким образом, функция-предикат, как ожидается, должна иметь тип (String, String) => Boolean. Имея эту информацию, компилятор сможет принять объявление _.equalsIgnoreCase(_) как сокращенную форму для (a: String, b: String) => a.equalsIgnoreCase(b).

12.9. Абстракция управляющих конструкций

В Scala имеется возможность моделировать последовательность инструкций в виде *функции без параметров* и без возвращаемого значения. Например, ниже представлена функция, выполняющая некоторый программный код в отдельном потоке выполнения:

```
def runInThread(block: () => Unit) {
     new Thread {
          override def run() { block() }
     }.start()
}
```

Программный код передается ей в виде функции типа () => Unit. Однако вызов этой функции необходимо снабжать неприглядной конструкцией () =>:

```
runInThread { () => println("Hi"); Thread.sleep(10000); println("Bye") }
```

Чтобы убрать () => из вызова, используйте нотацию *вызова по имени*: опустите скобки (), но оставьте => в объявлении параметра и затем в вызове передайте функцию-параметр:

```
def runInThread(block: => Unit) {
     new Thread {
          override def run() { block }
     }.start()
}
```

Теперь вызов можно записать, как показано ниже:

```
runInThread { println("Hi"); Thread.sleep(10000); println("Bye") }
```

Так выглядит гораздо лучше. Программисты на Scala имеют возможность создавать свои *абстракции управляющих конструкций*: функции могут выглядеть подобно ключевым словам языка. Например, можно реализовать функцию, которую можно использовать *точно* как инструкцию `while`. Или можно проявить изобретательность и определить инструкцию `until`, действующую подобно инструкции `while`, но условие продолжения цикла интерпретируется как условие выхода из цикла:

```
def until(condition: => Boolean)(block: => Unit) {
     if (!condition) {
          block
          until(condition)(block)
     }
}
```

А вот как можно использовать функцию `until`:

```
var x = 10
until (x == 0) {
     x -= 1
     println(x)
}
```

На техническом языке такая функция-параметр называется параметром, *вызываемым по имени* (call-by-name). В отличие от обычного (или вызываемого по значению) параметра, параметр-выражение *не* вычисляется в момент вызова функции. В конце концов, нам не требуется, чтобы выражение `x == 0` давало в результате `false` в вызове `until`. Вместо этого выражение становится телом функции без аргументов. И уже эта функция передается как параметр.

Взгляните на определение функции `until`. Обратите внимание, что она каррирована: функция сначала принимает условие `condition`, а затем, во втором параметре, блок кода `block`. Без каррирования вызов выглядел бы так:

```
until(x == 0, { ... })
```

Что смотрится не так привлекательно.

12.10. Выражение return

В Scala нет необходимости использовать инструкцию `return`, чтобы вернуть значение из функции. Возвращаемое значение функции – это простое значение тела функции.

Однако return можно использовать, чтобы вернуть значение из анонимной функции во вмещающую именованную функцию. Эту особенность удобно использовать в абстракциях управления. Например, рассмотрим следующую функцию:

```
def indexOf(str: String, ch: Char): Int = {
     var i = 0
     until (i == str.length) {
          if (str(i) == ch) return i
               i += 1
     }
     return -1
}
```

Здесь функции until передается анонимная функция { if (str(i) == ch) return i; i += 1 }. Когда выполняется выражение return, вмещающая функция с именем indexOf завершается и возвращает указанное значение.

Если выражение return используется внутри именованной функции, необходимо указать возвращаемый ею тип. В примере выше компилятор не способен определить, что функция indexOf возвращает значение Int.

Управление потоком выполнения достигается за счет использования специального исключения, возбуждаемого выражением return в анонимной функции, переданной функции until, и перехватываемого функцией indexOf.

> **Предупреждение.** Если исключение будет перехвачено в блоке try, перед тем как оно попадет в именованную функцию, тогда значение не будет возвращено.

Упражнения

1. Напишите функцию values(fun: (Int) => Int, low: Int, high: Int), возвращающую коллекцию из значений в указанном диапазоне. Например, вызов values(x => x * x, -5, 5) должен вернуть коллекцию пар (-5, 25), (-4, 16), (-3, 9), ..., (5, 25).
2. Как получить наибольший элемент массива с помощью метода reduceLeft?
3. Реализуйте функцию вычисления факториала с помощью методов to и reduceLeft без применения цикла или рекурсии.

4. Предыдущая реализация должна предусматривать специальный случай, когда `n < 1`. Покажите, как избежать этого с помощью `foldLeft`. (Ознакомьтесь с описанием `foldLeft` в Scaladoc. Этот метод напоминает `reduceLeft`, за исключением того, что первое значение в цепочке поставляется в вызове.)
5. Напишите функцию `largest(fun: (Int) => Int, inputs: Seq[Int])`, возвращающую наибольшее значение функции внутри заданной последовательности. Например, вызов `largest(x => 10 * x - x * x, 1 to 10)` должен вернуть `25`. Не используйте цикл или рекурсию.
6. Измените предыдущую функцию так, чтобы она возвращала *входное значение*, соответствующее наибольшему выходному значению. Например, вызов `largestAt(fun: (Int) => Int, inputs: Seq[Int])` должен вернуть `5`. Не используйте цикл или рекурсию.
7. Получить последовательность пар очень просто, например

```
val pairs = (1 to 10) zip (11 to 20)
```

   Теперь представьте, что необходимо что-то сделать с такой последовательностью, например вычислить суммы значений элементов пар. Для этой цели нельзя использовать

```
pairs.map(_ + _)
```

   Функция `_ + _` принимает два параметра типа `Int`, а не пару `(Int, Int)`. Напишите функцию `adjustToPair`, принимающую функцию типа `(Int, Int) => Int` и возвращающую эквивалентную функцию, оперирующую парой. Например, вызов `adjustToPair(_ * _)((6, 7))` должен вернуть `42`.

   Затем воспользуйтесь этой функцией в комбинации с `map` для вычисления сумм элементов в парах.
8. В разделе 12.8 «Карринг» был представлен метод `corresponds`, использующий два массива строк. Напишите вызов `corresponds`, который проверял бы соответствие длин строк в одном массиве целочисленным значениям в другом.
9. Реализуйте метод `corresponds` без карринга. Затем попробуйте вызвать его из предыдущего упражнения. С какими проблемами вы столкнулись?
10. Реализуйте абстракцию управления потоком выполнения `unless`, действующую подобно `if`, но с инвертированным толкованием логического условия. Требуется ли оформить первый параметр как параметр, вызываемый по имени? Необходим ли здесь карринг?

Глава 13. Коллекции

Темы, рассматриваемые в этой главе A2

- 13.1. Основные трейты коллекций.
- 13.2. Изменяемые и неизменяемые коллекции.
- 13.3. Последовательности.
- 13.4. Списки.
- 13.5. Изменяемые списки.
- 13.6. Множества.
- 13.7. Операторы добавления и удаления элементов.
- 13.8. Общие методы.
- 13.9. Функции map и flatMap.
- 13.10. Функции reduce, fold и scan A3.
- 13.11. Функция zip.
- 13.12. Итераторы.
- 13.13. Потоки A3.
- 13.14. Ленивые представления.
- 13.15. Взаимодействие с коллекциями Java.
- 13.16. Потокобезопасные коллекции.
- 13.17. Параллельные коллекции.
- Упражнения.

В этой главе вы познакомитесь со стандартной библиотекой коллекций в языке Scala с точки зрения ее пользователя. Помимо простых и ассоциативных массивов, с которыми вы уже сталкивались, вы увидите другие полезные типы коллекций. Существует множество методов, которые можно применять к коллекциям, и эта глава представит их вам по порядку.

Основные темы данной главы:

- все коллекции наследуют трейт `Iterable`;
- коллекции делятся на три основные категории: последовательности, множества и ассоциативные массивы;
- в Scala имеются изменяемые и неизменяемые версии большинства коллекций;

- списки в Scala – это либо пустой список, либо конструкция из «головы» и «хвоста», где «хвост», в свою очередь, является списком;
- множества – это неупорядоченные коллекции;
- для получения данных в порядке вставки используется тип `LinkedHashSet`, а для итераций в порядке сортировки – тип `SortedSet`;
- `+` добавляет элемент в неупорядоченную коллекцию; `+:` и `:+` добавляет элемент в начало или в конец последовательности; `++` объединяет две коллекции; `-` и `--` удаляют элементы;
- трейты `Iterable` и `Seq` имеют десятки удобных методов для выполнения типовых операций – подумайте об их использовании, прежде чем пытаться писать циклы;
- отображение, свертка и упаковка – распространенные приемы применения функций или операций к элементам коллекций.

13.1. Основные трейты коллекций

На рис. 13.1 представлены наиболее важные *трейты*, образующие *иерархию коллекций* в Scala.

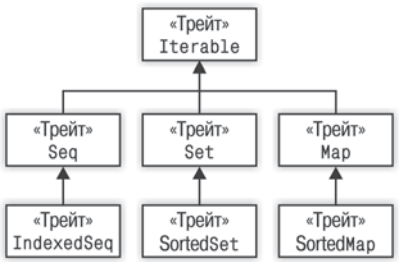


Рис. 13.1. Основные трейты, образующие иерархию коллекций в Scala

`Iterable` – это любая коллекция, способная возвращать итератор `Iterator`, обеспечивающий доступ ко всем элементам в коллекции:

```
val coll = ... // некоторая коллекция типа Iterable
val iter = coll.iterator
while (iter.hasNext)
    выполнить какие-либо операции над iter.next()
```

Это наиболее базовый способ обхода элементов коллекции. Однако, как будет показано в этой главе, для данной цели обычно используются более удобные средства.

`Seq` – это *упорядоченная коллекция* значений, такая как массив или список. `IndexedSeq` обеспечивает быстрый произвольный доступ с использованием целочисленных индексов. Например, `ArrayBuffer` обеспечивает доступ к элементам по индексам, а связанный список – нет.

`Set` – *неупорядоченная коллекция* значений. Обход элементов в `SortedSet` всегда выполняется в порядке сортировки.

`Map` – множество пар (*ключ*, *значение*). `SortedMap` обеспечивает обход элементов в порядке сортировки ключей. Дополнительная информация приводится в главе 4.

Эта иерархия напоминает аналогичную в языке Java, с парой улучшений:

1. Ассоциативные массивы не образуют отдельную иерархию, а являются частью общей иерархии.
2. `IndexedSeq` является супертипом массивов, но не списков, что свидетельствует об их непохожести.

> **Примечание.** В Java оба класса, `ArrayList` и `LinkedList`, реализуют общий интерфейс `List`, что усложняет разработку эффективного кода, когда требуется произвольный доступ, например для поиска в отсортированной последовательности. Это было ошибочным решением при разработке фреймворка коллекций в Java. В последней версии был добавлен маркер-интерфейс `RandomAccess`, решающий эту проблему.

Каждый трейт или класс коллекции в Scala имеет объект-компаньон с методом `apply`, создающим экземпляры коллекций. Например:

```
Iterable(0xFF, 0xFF00, 0xFF0000)
Set(Color.RED, Color.GREEN, Color.BLUE)
Map(Color.RED -> 0xFF0000, Color.GREEN -> 0xFF00, Color.BLUE -> 0xFF)
SortedSet("Hello", "World")
```

Это называется «принципом единообразия создания».

13.2. Изменяемые и неизменяемые коллекции

Scala поддерживает *изменяемые и неизменяемые коллекции*. Неизменяемые коллекции никогда не изменяются, поэтому их без опасения можно передавать по ссылке и использовать даже в много-

поточной среде выполнения. Например, существуют типы `scala.collection.mutable.Map` и `scala.collection.immutable.Map`. Оба имеют общий супертип `scala.collection.Map` (который, разумеется, не имеет операций, изменяющих содержимое объекта).

> **Примечание.** Используя ссылку на `scala.collection.immutable.Map`, можно быть уверенным, что никто не сможет изменить ассоциативный массив. Если используется ссылка на коллекцию типа `scala.collection.Map`, вы не сможете изменить ее, но она может быть доступна для изменения другим[1].

В языке Scala предпочтение отдается неизменяемым коллекциям. Объекты-компаньоны в пакете `scala.collection` производят неизменяемые коллекции. Например, `scala.collection.Map(«Hello» -> 42)` — это неизменяемая коллекция.

Кроме того, в пакете `scala` и объекте `Predef`, которые импортируются по умолчанию, определяются *псевдонимы* типов `List`, `Set` и `Map`, ссылающиеся на неизменяемые трейты. Например, `Predef.Map` — то же самое, что и `scala.collection.immutable.Map`.

> **Совет.** Добавив инструкцию
>
> `import scala.collection.mutable`
>
> вы сможете создавать неизменяемые ассоциативные массивы, используя псевдоним `Map`, а изменяемые – `mutable.Map`.

Если прежде вы не имели опыта работы с неизменяемыми коллекциями, у вас может появиться сомнение в возможности организации работы с ними. Ключом здесь является возможность создания новых коллекций на основе старых. Например, если предположить, что `numbers` — это неизменяемое множество, тогда `numbers + 9` — это новое множество, содержащее те же числа, что и `numbers`, плюс число `9`. Если `9` уже имеется в множестве, вы просто получите ссылку на старое множество. Такой порядок является особенно естественным в *рекурсивных вычислениях*. Например, ниже создается множество всех цифр, составляющих указанное целое число:

---

[1] Так как `collection.Map` является супертипом для `collection.mutable.Map`, то в одном потоке выполнения можно создать объект `collection.mutable.Map` и передать его в другой поток, но уже как объект `collection.Map`. В этом случае поток-создатель ассоциативного массива может изменять его содержимое, в то время как другой поток – нет (так как ссылка на тип не содержит таких методов). – *Прим. ред.*

```
def digits(n: Int): Set[Int] =
    if (n < 0) digits(-n)
    else if (n < 10) Set(n)
    else digits(n / 10) + (n % 10)
```

Метод начинается с создания множества, содержащего единственную цифру. На каждом шаге добавляется очередная цифра. Однако добавление цифры не изменяет множества. Вместо этого на каждом шаге создается новое множество.

13.3. Последовательности

На рис. 13.2 изображены наиболее важные неизменяемые *последовательности*.

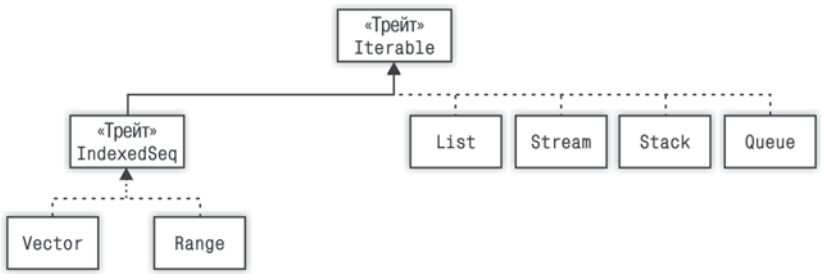


Рис. 13.2. Неизменяемые последовательности

Тип `Vector` – неизменяемый аналог `ArrayBuffer`: индексируемая последовательность с *быстрым произвольным доступом* к элементам. Векторы реализованы как деревья, где каждый узел может иметь до 32 потомков. Вектор с миллионом элементов имеет четыре слоя узлов. (Поскольку $10^3 \approx 2^{10}$, $10^6 \approx 32^4$.) Доступ к элементам в таком списке выполняется в четыре шага, тогда как в связанном списке такого же размера в среднем придется выполнить 500 000 шагов.

Класс `Range` представляет *последовательность целых чисел*, такую как `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` или `10`, `20`, `30`. Разумеется, объект `Range` хранит не всю последовательность значений, а только начальное, конечное и шаг. Объекты `Range` конструируются с помощью методов `to` и `until`, как описывается в главе 3.

Списки будут обсуждаться в следующем разделе, а потоки данных – в разделе 13.13 «Потоки».

На рис. 13.3 представлены наиболее интересные изменяемые последовательности.

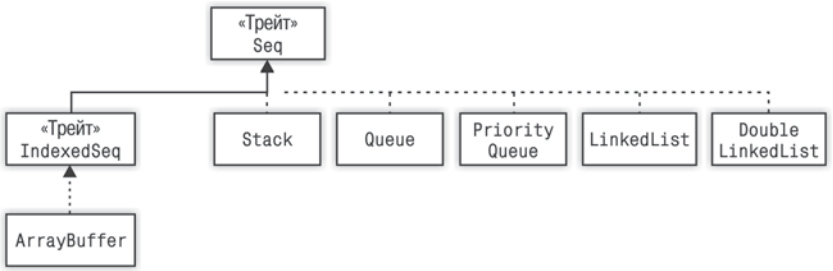


Рис. 13.3. Изменяемые последовательности

Буферы уже обсуждались в главе 3. Стеки, очереди и очереди с приоритетами – это стандартные структуры данных, которые могут пригодиться для реализации некоторых алгоритмов. Если вы уже знакомы с этими структурами, реализация их не станет для вас новостью.

Классы связанных списков в Scala, напротив, несколько отличаются от связанных списков, с которыми вы могли сталкиваться в Java, C++ или в ходе изучения структур данных. Мы рассмотрим их в следующем разделе.

13.4. Списки

В Scala *список* может быть либо значением `Nil` (то есть быть пустым), либо объектом с элементом `head` и элементом `tail`, который сам является списком. Например, рассмотрим список

```
val digits = List(4, 2)
```

Элемент `digits.head` имеет значение `4`, а элемент `digits.tail` – значение `List(2)`. Более того, элемент `digits.tail.head` имеет значение `2`, а элемент `digits.tail.tail` – значение `Nil`.

Оператор `::` создает новый список из заданных значений головы и хвоста списка. Например,

```
9 :: List(4, 2)
```

даст в результате список `List(9, 4, 2)`. Этот список можно также записать как

```
9 :: 4 :: 2 :: Nil
```

Обратите внимание, что оператор :: является правоассоциативным. При использовании оператора :: конструирование списка происходит с конца.

```
9 :: (4 :: (2 :: Nil))
```

В языках Java и C++ для обхода элементов связанного списка используется итератор. В Scala также можно использовать этот прием, но зачастую более естественными оказываются рекурсивные алгоритмы. Например, следующая функция вычисляет сумму всех целочисленных элементов связанного списка:

```
def sum(lst: List[Int]): Int =
    if (lst == Nil) 0 else lst.head + sum(lst.tail)
```

Или, если вы предпочитаете использовать *сопоставление с образцом*:

```
def sum(lst: List[Int]): Int = lst match {
    case Nil => 0
    case h :: t => h + sum(t) // h - это lst.head, t - это lst.tail
}
```

Обратите внимание на оператор :: во втором образце. Он «разбирает» список на голову и хвост.

> **Примечание.** Рекурсивный алгоритм выглядит так естественно, потому что хвост списка сам является списком.

Однако, прежде чем восхищаться элегантностью рекурсивного решения, загляните в библиотеку Scala. В ней уже имеется метод sum:

```
List(9, 4, 2).sum // Вернет 15
```

13.5. Изменяемые списки

Изменяемый тип LinkedList изменяемых списков действует подобно типу List, за исключением того, что позволяет изменять голову присваиванием по ссылке elem и хвост присваиванием по ссылке next.

Внимание. Имейте в виду, что элементы `head` и `tail` доступны только для чтения.

Например, следующий цикл заменит все отрицательные значения нулевыми:

```
val lst = scala.collection.mutable.LinkedList(1, -2, 7, -9)
var cur = lst
while (cur != Nil) {
     if (cur.elem < 0) cur.elem = 0
     cur = cur.next
}
```

А следующий цикл удалит каждый второй элемент из списка:

```
var cur = lst
while (cur != Nil && cur.next != Nil) {
     cur.next = cur.next.next
     cur = cur.next
}
```

Здесь переменная `cur` действует подобно итератору, но в действительности она имеет тип `LinkedList`.

Существует также тип `DoubleLinkedList` двухсвязных списков, с изменяемой ссылкой `prev`.

Внимание. Если необходимо сделать узел списка последним узлом в списке, вместо значения `Nil` ссылке `next` следует присвоить значение `LinkedList.empty`. Не присваивайте ей `null`, иначе получите ошибку обращения к пустому указателю при обходе списка.

13.6. Множества

Множество – это коллекция, в которой каждый элемент может присутствовать в единственном экземпляре. Попытка добавить существующий элемент не дает никакого эффекта. Например,

```
Set(2, 0, 1) + 1
```

то же самое, что и `Set(2, 0, 1)`.

В отличие от списков, множества не сохраняют порядок следования элементов, в каком они добавлялись в множество. По умол-

чанию множества реализуются как *хеш-множества*, в которых элементы организованы по значению, возвращаемому методом `hashCode`. (В Scala, как и в Java, каждый объект имеет метод `hashCode`.)

Например, если выполнить итерации через множество

```
Set(1, 2, 3, 4, 5, 6)
```

обход элементов будет выполнен в следующем порядке

```
5 1 6 2 3 4
```

Может показаться странным, почему множества не сохраняют порядок добавления элементов. Как оказывается, поиск элементов выполняется намного быстрее, если позволить множеству переупорядочить их. *Поиск элемента* в хеш-множестве выполняется *намного* быстрее, чем в массиве или в списке.

Связанное хеш-множество сохраняет порядок добавления элементов. Для этой цели в нем используется связанный список. Например:

```
val weekdays =
    scala.collection.mutable.LinkedHashSet("Mo", "Tu", "We", "Th", "Fr")
```

Если потребуется выполнить обход элементов в порядке сортировки, используйте *сортированное множество*:

```
scala.collection.immutable.SortedSet(1, 2, 3, 4, 5, 6)
```

Сортированные множества реализованы как *красно-черные деревья (red-black tree)*.

Внимание. В Scala 2.9 отсутствуют изменяемые сортированные множества. Если вам понадобятся структуры подобного типа, используйте класс `java.util.TreeSet`[1].

Примечание. Реализация структур данных, таких как хеш-таблицы и бинарные деревья поиска, а также эффективность операций с этими структурами входят в программу изучения информатики. Если у вас появится желание освежить эту информацию в памяти, поищите среди свобод-

[1] В Scala 2.10 появился класс `scala.collection.mutable.TreeSet`, основанный на AVL-деревьях (AVL-tree) – *Прим. перев.*

но распространяемых книг на сайте www.cs.williams.edu/javastructures/Welcome.html.

Битовое множество – это реализация множества неотрицательных целых чисел в виде последовательности бит, где *i*-й бит равен 1, если *i* присутствует в множестве. Эта реализация отличается высокой эффективностью, при условии, что максимальный элемент не слишком большой. В Scala имеются обе разновидности класса `BitSet`, изменяемая и неизменяемая.

Метод `contains` проверяет наличие в множестве указанного значения. Метод `subsetOf` проверяет, входят ли все элементы множества в другое множество.

```
val digits = Set(1, 7, 2, 9)
digits contains 0         // false
Set(1, 2) subsetOf digits // true
```

Методы `union`, `intersect` и `diff` реализуют наиболее типичные операции над множествами. Их можно также записывать как `|`, `&` и `&~`. Операцию объединения множеств можно также записать как `++`, а разность – как `--`. Например, если имеется множество

```
val primes = Set(2, 3, 5, 7)
```

тогда выражение `digits union primes` вернет `Set(1, 2, 3, 5, 7, 9)`, выражение `digits & primes` вернет `Set(2, 7)` и выражение `digits -- primes` вернет `Set(1, 9)`.

13.7. Операторы добавления и удаления элементов

В табл. 13.1 перечислены операторы, определяемые коллекциями разных типов для *добавления и удаления элементов*.

Вообще, `+` используется для добавления элемента в неупорядоченную коллекцию, а `+:` и `:+` – для добавления элемента в начало или в конец упорядоченной коллекции.

```
Vector(1, 2, 3) :+ 5 // Вернет Vector(1, 2, 3, 5)
1 +: Vector(1, 2, 3) // Вернет Vector(1, 1, 2, 3)
```

Таблица 13.1. Операторы добавления и удаления элементов

Оператор	Описание	Типы коллекций
`coll :+ elem` `elem +: coll`	Возвращает коллекцию того же типа, что и `coll`, с элементом `elem`, добавленным в конец или в начало	`Seq`
`coll + elem` `coll + (e1, e2, ...)`	Возвращает коллекцию того же типа, что и `coll`, с добавленными элементами	`Set, Map`
`coll - elem` `coll - (e1, e2, ...)`	Возвращает коллекцию того же типа, что и `coll`, из которой удалены указанные элементы	`Set, Map, ArrayBuffer`
`coll ++ coll2` `coll2 ++: coll`	Возвращает коллекцию того же типа, что и `coll`, содержащую элементы из обеих коллекций	`Iterable`
`coll -- coll2`	Возвращает коллекцию того же типа, что и `coll`, из которой удалены элементы, содержащиеся в коллекции `coll2`. (Для последовательностей используйте метод `diff`)	`Set, Map, ArrayBuffer`
`elem :: lst` `lst2 ::: lst`	Список с указанным элементом или списком, добавленным в начало. То же, что и `+:` и `++:`	`List`
`list ::: list2`	То же, что и `list ++: list2`	`List`

`set | set2`
`set & set2`
`set &~ set2`
	Объединение, пересечение и разность множеств. Оператор `|` действует так же, как `++`, а `&~` – как `--`	`Set`

`coll += elem`
`coll += (e1, e2, ...)`
`coll ++= coll2`
`coll -= elem`
`coll -= (e1, e2, ...)`
`coll --= coll2`
	Изменяют коллекцию `coll`, добавляя или удаляя указанные элементы	`Изменяемые коллекции`

`elem +=: coll` `coll2 ++=: coll`	Изменяют коллекцию `coll`, добавляя в начало указанный элемент или коллекцию	`ArrayBuffer`

Обратите внимание, что `+:`, подобно всем операторам, завершающимся двоеточием, является правоассоциативным и что он является методом правого операнда.

Эти операторы возвращают новые коллекции (того же типа, что и оригинальные), не изменяя оригинала. Изменяемые коллекции имеют оператор `+=`, изменяющий левый операнд. Например:

```
val numbers = ArrayBuffer(1, 2, 3)
numbers += 5 // Добавит 5 в numbers
```

Операторы += и :+= можно применять к неизменяемым коллекциям, только если слева от оператора находится var-переменная, как показано ниже:

```
var numbers = Set(1, 2, 3)
numbers += 5 // Присвоит переменной numbers новое неизменяемое множество
             // numbers + 5
var numberVector = Vector(1, 2, 3)
numberVector :+= 5 // += не работает, потому что векторы не поддерживают +
```

Удалить элемент можно с помощью оператора -:

```
Set(1, 2, 3) - 2 // Вернет Set(1, 3)
```

Добавить сразу несколько элементов можно с помощью оператора ++:

```
coll ++ coll2
```

вернет коллекцию того же типа, что и coll, содержащую элементы обеих коллекций, coll и coll2. Аналогично оператор -- удаляет сразу несколько элементов.

Совет. Как видите, в Scala имеется много разных операторов для добавления и удаления элементов. Ниже приводится краткая сводка по ним:

1. Добавление в конец (:+) или в начало (+:) последовательности.
2. Добавление (+) в неупорядоченную коллекцию.
3. Удаление с помощью -.
4. Используйте ++ и -- для добавления и удаления групп элементов.
5. При работе со списками предпочтительнее использовать :: и :::.
6. Изменяющие операторы += ++= -= --=.
7. При работе со множествами я предпочитаю ++ и --.
8. Я стараюсь избегать ++: +=: ++=:.

Примечание. При работе со списками можно использовать +: вместо ::, с одним исключением: сопоставление с образцом (case h::t) не работает с оператором +:.

13.8. Общие методы

В табл. 13.2 приводится краткий обзор наиболее важных методов трейта Iterable, расположенные в порядке функциональности.

Таблица 13.2. Важные методы трейта Iterable

Методы	Описание
`head`, `last`, `headOption`, `lastOption`	Возвращают первый или последний элемент; первый или последний элемент, такой как `Option`
`tail`, `init`	Возвращают все, кроме первого или последнего элемента
`length`, `isEmpty`	Возвращают длину или `true`, если длина равна нулю
`map(f)`, `foreach(f)`, `flatMap(f)`, `collect(pf)`	Применяют функцию ко всем элементам (см. раздел 13.9)
`reduceLeft(op)`, `reduceRight(op)`, `foldLeft(init)(op)`, `foldRight(init)(op)`	Применяют двухместную операцию ко всем элементам в указанном порядке (см. раздел 13.10)
`reduce(op)`, `fold(init)(op)`, `aggregate(init)(op, combineOp)`	Применяют двухместную операцию ко всем элементам в произвольном порядке (см. раздел 13.17)
`sum`, `product`, `max`, `min`	Возвращают сумму или произведение (элементы могут неявно преобразовываться в трейт `Numeric`), или максимальное или минимальное значение (элементы могут неявно преобразовываться в трейте `Ordered`)
`count(pred)`, `forall(pred)`, `exists(pred)`	Возвращают количество элементов, удовлетворяющих предикату: `true`, если все элементы или хотя бы один соответствует условию
`filter(pred)`, `filterNot(pred)`, `partition(pred)`	Возвращают все элементы, соответствующие условию, все элементы, не соответствующие условию; и те, и другие
`takeWhile(pred)`, `dropWhile(pred)`, `span(pred)`	Возвращают первые элементы, соответствующие условию; все, кроме первых соответствующих условию; и те, и другие
`take(n)`, `drop(n)`, `splitAt(n)`	Возвращают первые `n` элементов; все, кроме первых `n` элементов; и те, и другие
`takeRight(n)`, `dropRight(n)`	Возвращают последние `n` элементов; все, кроме последних `n` элементов
`slice(from, to)`	Возвращает элементы в диапазоне от `from` до `to`
`zip(coll2)`, `zipAll(coll2, fill, fill2)`, `zipWithIndex`	Возвращают пары элементов из указанной пары коллекций (см. раздел 13.11)
`grouped(n)`, `sliding(n)`	Возвращают итераторы фрагментов коллекций длиной `n`; `grouped` возвращает элементы с индексами `0 until n`, затем элементы с индексами `n until 2*n` и т. д.; `sliding` возвращает элементы с индексами `0 until n`, затем с индексами `1 until n+1` и т. д.

Таблица 13.2. Важные методы трейта Iterable (окончание)

Методы	Описание
`mkString(before, between, after)`, `addString(sb, before, between, after)`	Создают строку из всех элементов, добавляя указанные строки перед (`before`) первым, между (`between`) каждым и после (`after`) последнего элемента. Второй метод добавляет полученную строку в конец буфера построителя строк
`toIterable`, `toSeq`, `toIndexedSeq`, `toArray`, `toList`, `toStream`, `toSet`, `toMap`	Преобразуют коллекцию в коллекцию указанного типа
`copyToArray(arr)`, `copyToArray(arr, start, length)`, `copyToBuffer(buf)`	Копируют элементы в массив или в буфер

Трейт `Seq` добавляет несколько методов к методам трейта `Iterable`. Наиболее важные из них перечислены в табл. 13.3.

Таблица 13.3. Важные методы трейта Seq

Методы	Описание
`contains(elem)`, `containsSlice(seq)`, `startsWith(seq)`, `endsWith(seq)`	Возвращают `true`, если данная последовательность содержит указанный элемент или последовательность; если начинается или заканчивается с указанной последовательностью
`indexOf(elem)`, `lastIndexOf(elem)`, `indexOfSlice(seq)`, `lastIndexOfSlice(seq)`	Возвращают индекс первого или последнего вхождения указанного элемента или последовательности элементов
`indexWhere(pred)`	Возвращает индекс первого элемента, соответствующего условию `pred`
`prefixLength(pred)`, `segmentLength(pred, n)`	Возвращают длину наибольшей последовательности элементов, соответствующих условию `pred`, начиная с `0` или `n`
`padTo(n, fill)`	Возвращает копию данной последовательности, дополняя ее значением `fill` до достижения длины `n`
`intersect(seq)`, `diff(seq)`	Возвращают пересечение или разность последовательностей. Например, если `a` содержит пять единиц (1), а `b` – две, тогда `a intersect b` вернет две (наименьшее количество), а `a diff b` вернет три (разность)
`reverse`	Располагает элементы последовательности в обратном порядке

Таблица 13.3. Важные методы трейта Seq (окончание)

Методы	Описание
`sorted`, `sortWith(less)`, `sortBy(f)`	Сортируют последовательность путем сравнения элементов с использованием двухместной функции `less` или функции `f`, отображающей каждый элемент в порядковый тип
`permutations`, `combinations(n)`	Возвращают итератор обхода всех перестановок или комбинаций (подпоследовательностей с длиной `n`)

Примечание. Обратите внимание, что эти методы никогда не изменяют коллекции. Они возвращают коллекцию того же типа, что и исходная. Иногда это называют принципом «единообразия возвращаемого типа».

13.9. Функции map и flatMap

Иногда может потребоваться трансформировать все элементы коллекции. Метод `map` применяет функцию к коллекции и возвращает коллекцию результатов. Например, пусть имеется список строк

```
val names = List("Peter", "Paul", "Mary")
```

Привести все символы в строках к верхнему регистру можно так:

```
names.map(_.toUpperCase) // List("PETER", "PAUL", "MARY")
```

Это то же самое, что и

```
for (n <- names) yield n.toUpperCase
```

Если вместо единственного значения функция возвращает коллекцию, может потребоваться объединить все результаты в одну строку. В этом случае можно воспользоваться методом `flatMap`. Например, пусть имеется функция

```
def ulcase(s: String) = Vector(s.toUpperCase(), s.toLowerCase())
```

Тогда вызов `names.map(ulcase)` вернет

```
List(Vector("PETER","peter"), Vector("PAUL","paul"), Vector("MARY","mary"))
```

а вызов `names.flatMap(ulcase)`

```
List("PETER", "peter", "PAUL", "paul", "MARY", "mary")
```

> **Совет.** Если метод `flatMap` используется с функцией, возвращающей значение типа `Option`, возвращаемая коллекция будет содержать все значения `v`, для которых функция вернет `Some(v)`.

Метод `collect` работает с *частично определенными функциями* (partial functions) – функциями, которые могут быть определены не для всех входных значений. Он возвращает коллекцию всех значений аргументов функции, для которых она определена. Например:

```
"-3+4".collect { case '+' => 1 ; case '-' => -1 } // Vector(-1, 1)
```

Наконец, если необходимо применить функцию только ради ее побочного эффекта, не заботясь о возвращаемых ею значениях, можно воспользоваться методом `foreach`:

```
names.foreach(println)
```

13.10. Функции reduce, fold и scan A3

Метод `map` применяет одноместную функцию ко всем элементам коллекции. Методы, обсуждаемые в этом разделе, объединяют элементы с помощью *двухместной* функции. Вызов `c.reduceLeft(op)` применяет `op` к последующим элементам, как показано ниже:

```
              .
             .
            .
          op
         / \
       op coll(3)
      / \
    op coll(2)
   / \
coll(0) coll(1)
```

Например,

```
List(1, 7, 2, 9).reduceLeft(_ - _)
```

выполнит последовательность операций

```
          -
        /   \
       -     9
     /   \
    -     2
  /   \
1     7
```

или

```
((1 - 7) - 2) - 9 = 1 - 7 - 2 - 9 = -17
```

Метод `reduceRight` действует точно так же, но начинает обработку последовательности с конца. Например,

```
List(1, 7, 2, 9).reduceRight(_ - _)
```

выполнит последовательность операций

```
1 - (7 - (2 - 9)) = 1 - 7 + 2 - 9 = -13
```

Часто бывает желательно начинать вычисления с некоторого произвольного элемента коллекции, а не с ее начала. Вызов `coll.foldLeft(init)(op)` вычислит

```
             .
            .
           .
         op
        / \
      op coll(2)
     / \
   op coll(1)
  / \
init coll(0)
```

Например,

```
List(1, 7, 2, 9).foldLeft(0)(_ - _)
```

выполнит последовательность операций

```
0 - 1 - 7 - 2 - 9 = -19
```

Примечание. Начальное значение и оператор – это отдельные «каррированные» параметры, что позволяет Scala использовать тип начального значения для определения типа оператора. Например, в `List(1, 7, 2, 9).foldLeft("")(_ + _)` начальное значение является строкой, поэтому оператор должен быть функцией `(String, Int) => String`.

Операцию `foldLeft` можно также записать с помощью оператора `/:`, например:

```
(0 /: List(1, 7, 2, 9))(_ - _)
```

Внешним видом оператор `/:` должен напоминать вам дерево.

Примечание. В форме с оператором `/:` начальным значением является первый операнд. Имейте в виду, что поскольку оператор заканчивается двоеточием, он является методом второго операнда.

Ниже показано, как действует метод `foldRight` или оператор `:\`

```
       .
        .
         .
         op
        /  \
coll(n-3) op
          /  \
  coll(n-2) op
            /  \
    coll(n-1) init
```

Все эти примеры не выглядят особенно практичными. Да, вызовы `coll.reduceLeft(_ + _)` и `coll.foldLeft(0)(_ + _)` вычисляют сумму, но ее можно получить более простым способом, вызвав `coll.sum`.

Операция свертки иногда может служить неплохой заменой циклу. Допустим, например, что требуется определить частоту встречаемости букв в строке. Для этого можно было бы обойти все буквы, обновляя счетчики в изменяемом ассоциативном массиве.

```
val freq = scala.collection.mutable.Map[Char, Int]()
for (c <- "Mississippi") freq(c) = freq.getOrElse(c, 0) + 1
    // Теперь freq - это Map('i' -> 4, 'M' -> 1, 's' -> 4, 'p' -> 2)
```

Ниже изображено другое представление этого процесса. На каждом шаге объединяются ассоциативный массив с частотами и вновь встреченная буква, и возвращается новый ассоциативный массив. То есть свертка:

```
                .
               .
              .
            op
           / \
         op 's'
        / \
      op 'i'
     / \
empty map 'M'
```

Что делает `op`? Левый операнд – частично заполненный ассоциативный массив, а правый операнд – новая буква. Результатом является дополненный ассоциативный массив. Он подается на вход следующей операции `op`, и так до конца, в результате получается ассоциативный массив со всеми счетчиками. Решение в программном коде представлено ниже:

```
(Map[Char, Int]() /: "Mississippi") {
    (m, c) => m + (c -> (m.getOrElse(c, 0) + 1))
}
```

Обратите внимание, что здесь используется неизменяемый ассоциативный массив. На каждом шаге создается новый ассоциативный массив.

> **Примечание.** Операцией свертки можно заменить любой цикл `while`. Сконструируйте структуру данных, объединяющую все переменные, которые изменяются в цикле, и определите операцию, реализующую один шаг. Я не утверждаю, что это всегда оправдано, но вы можете найти интересной возможность отказаться от циклов и изменения данных.

Наконец, методы `scanLeft` и `scanRight` объединяют свертку и ассоциативный массив. В результате вы получаете коллекцию промежуточных результатов. Например,

```
(1 to 10).scanLeft(0)(_ + _)
```

вернет все промежуточные суммы:

```
Vector(0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55)
```

13.11. Функция zip

Методы, представленные в предыдущем разделе, применяют операции к смежным элементам в той же самой коллекции. Иногда имеются две исходные коллекции, соответствующие элементы которых нужно объединить попарно. Например, допустим, что имеется список цен на товары и список с количеством товаров:

```
val prices = List(5.0, 20.0, 9.95)
val quantities = List(10, 2, 1)
```

С помощью метода `zip` их можно объединить в список пар. Например,

```
prices zip quantities
```

даст в результате `List[(Double, Int)]`:

```
List[(Double, Int)] = List((5.0, 10), (20.0, 2), (9.95, 1))
```

Метод называется «zip»[1], потому что он объединяет две коллекции подобно собачке на застежке-молнии.

Теперь легко можно применить функцию к каждой паре.

```
(prices zip quantities) map { p => p._1 * p._2 }
```

Результатом будет список стоимостей объемов каждого товара:

```
List(50.0, 40.0, 9.95)
```

Затем можно получить общую стоимость всех товаров:

```
((prices zip quantities) map { p => p._1 * p._2 }) sum
```

---

[1] Упаковать, застегнуть. – *Прим. перев.*

Если одна коллекция окажется короче другой, результат будет содержать количество пар, соответствующее меньшей коллекции. Например,

```
List(5.0, 20.0, 9.95) zip List(10, 2)
```

даст в результате

```
List((5.0, 10), (20.0, 2))
```

Метод `zipAll` позволяет указать значения по умолчанию для более короткого списка:

```
List(5.0, 20.0, 9.95).zipAll(List(10, 2), 0.0, 1)
```

вернет

```
List((5.0, 10), (20.0, 2), (9.95, 1))
```

Метод `zipWithIndex` возвращает список пар, где вторым компонентом является индекс каждого элемента. Например,

```
"Scala".zipWithIndex
```

вернет

```
Vector(('S', 0), ('c', 1), ('a', 2), ('l', 3), ('a', 4))
```

Это может пригодиться, когда потребуется вычислить индекс элемента с определенным свойством. Например,

```
"Scala".zipWithIndex.max
```

вернет `('l', 3)`. Индекс символа с наибольшим кодом определяется как

```
"Scala".zipWithIndex.max._2
```

13.12. Итераторы

Получить итератор из коллекции можно с помощью метода `iterator`. Итераторы в Scala используются реже, чем в Java или C++,

потому что обычно желаемого можно добиться проще с применением методов коллекций, описанных в предыдущих разделах.

Однако итераторы могут пригодиться для работы с коллекциями, полное воссоздание которых является дорогостоящей операцией. Например, метод `Source.fromFile` возвращает итератор, потому что операция чтения файла в память целиком может оказаться весьма неэффективной операцией. Трейт `Iterable` определяет несколько методов, возвращающих итераторы, такие как `grouped` и `sliding`.

Обладая итератором, можно выполнять итерации по элементам коллекции с помощью методов `next` и `hasNext`.

```
while (iter.hasNext)
    Выполнить операции с элементом iter.next()
```

При желании можно использовать цикл `for`:

```
for (elem <- iter)
    Выполнить операции с элементом elem
```

Оба цикла перемещают итератор в конец коллекции, после чего итератор становится бесполезным.

Класс `Iterator` определяет ряд методов, действующих идентично методам коллекций. В частности, доступны все методы трейта `Iterable`, перечисленные в разделе 13.8 «Общие методы», за исключением `head`, `headOption`, `last`, `lastOption`, `tail`, `init`, `takeRight` и `dropRight`. После вызова такого метода, как `map`, `filter`, `count`, `sum` и даже `length`, итератор перемещается в конец коллекции и не может больше использоваться. При использовании других методов, таких как `find` и `take`, итератор останавливается за найденным или извлеченным элементом.

Если работа с итераторами покажется вам слишком утомительной, для копирования элементов в коллекцию всегда можно использовать методы, такие как `toArray`, `toIterable`, `toSeq`, `toSet` и `toMap`.

13.13. Потоки A3

В предыдущих разделах вы видели, что итератор является «ленивой» альтернативой коллекции. Вы получаете элементы по мере их необходимости. Если элементы не требуются, вам не приходится платить вычислительными мощностями.

Однако итераторы – вещь достаточно хрупкая. Каждый вызов `next` изменяет итератор. *Потоки* предлагают неизменяемую альтернативу.

Поток – это неизменяемый список, хвост которого вычисляется по требованию, то есть только когда вы попросите об этом.

Ниже приводится типичный пример:

```
def numsFrom(n: BigInt): Stream[BigInt] = n #:: numsFrom(n + 1)
```

Оператор `#::` похож на оператор `::` для списков, но создает потоки.

Когда производится вызов

```
val tenOrMore = numsFrom(10)
```

возвращается объект потока, который отображается как

```
Stream(10, ?)
```

Хвост потока не вычисляется. Если вызвать метод

```
tenOrMore.tail.tail.tail
```

возвращается

```
Stream(13, ?)
```

Методы потоков вычисляются «лениво». Например,

```
val squares = numsFrom(1).map(x => x * x)
```

вернет

```
Stream(1, ?)
```

Чтобы получить следующий элемент, вам придется вызвать метод `squares.tail`.

Если потребуется получить более одного элемента, можно воспользоваться связкой методов `take` и `force`, которая обеспечит принудительное получение всех значений. Например,

```
squares.take(5).force
```

вернет `Stream(1, 4, 9, 16, 25)`.

Конечно, не следует пользоваться связкой

```
squares.force // Нет!
```

В этом случае будет предпринята попытка вычислить все элементы бесконечного потока, что в конечном итоге вызовет исключение `OutOfMemoryError`.

Поток можно сконструировать на основе итератора. Например, метод `Source.getLines` возвращает `Iterator[String]`. С помощью итератора можно обрабатывать строки только по одной за раз. Поток кеширует полученные ранее строки, благодаря чему к ним можно обращаться снова:

```
val words = Source.fromFile("/usr/share/dict/words").getLines.toStream
words    // Stream(A, ?)
words(5) // Aachen
words    // Stream(A, A's, AOL, AOL's, Aachen, ?)
```

13.14. Ленивые представления

В предыдущем разделе вы видели, что методы потоков реализуют отложенные вычисления, возвращая результаты, только когда они необходимы. Аналогичный эффект можно получить и для других коллекций, применив метод `view`. Этот метод возвращает коллекцию, методы которой реализуют отложенные вычисления. Например,

```
val powers = (0 until 1000).view.map(pow(10, _))
```

вернет коллекцию, не имеющую элементов. (В отличие от потока, она не имеет даже первого элемента.) Если вызвать

```
powers(100)
```

будет произведен единственный вызов `pow(10, 100)`. В отличие от потоков, представления не кешируют полученные значения. Если вызвать `powers(100)` еще раз, повторно будет выполнен вызов `pow(10, 100)`.

Как и потоки, для принудительного получения нескольких элементов из представления необходимо использовать метод `force`. Он вернет коллекцию того же типа, что и исходная коллекция.

Ленивые представления особенно выгодно использовать, когда необходимо преобразовать большую коллекцию несколькими способами, потому что не требуется создавать большие промежуточные коллекции. Например, сравните

```
(0 to 1000).map(pow(10, _)).map(1 / _)
```

с

```
(0 to 1000).view.map(pow(10, _)).map(1 / _).force
```

В первом случае вычисляется коллекция степеней десятки, и затем к каждому элементу применяется операция получения обратных значений. Во втором случае создается представление, реализующее обе операции отображения. При принудительном вычислении обе операции применяются к каждому элементу, но без создания промежуточной коллекции.

> **Примечание.** Разумеется, в данном случае можно было бы просто вызвать `(0 to 1000).map(x => pow(10, -x))`. Однако, если коллекция обрабатывается в разных частях программы, удобнее передавать представление, накапливающее модификации.

13.15. Взаимодействие с коллекциями Java

Иногда возникает необходимость использовать коллекции Java. Можно применять их непосредственно и лишиться богатого набора методов, которыми обладают коллекции в языке Scala. Однако есть другой путь: создавать коллекции Scala и затем передавать их в Java-код. Объект `JavaConversions` предоставляет множество преобразований коллекций между Scala и Java.

Укажите тип целевого значения явно, чтобы инициировать преобразование. Например,

```
import scala.collection.JavaConversions._
val props: scala.collection.mutable.Map[String, String] =
    System.getProperties()
```

Если вас волнует проблема нежелательных неявных преобразований, просто импортируйте только необходимые. Например,

```
import scala.collection.JavaConversions.propertiesAsScalaMap
```

В табл. 13.4 перечислены преобразования из коллекций Scala в коллекции Java.

А в табл. 13.5 перечислены обратные преобразования, из коллекций Java в коллекции Scala.

Таблица 13.4 Преобразования из коллекций Scala в коллекции Java

Неявная функция	Исходный тип в scala.collection	Целевой тип в java.util
asJavaCollection	Iterable	Collection
asJavaIterable	Iterable	Iterable
asJavaIterator	Iterator	Iterator
asJavaEnumeration	Iterator	Enumeration
seqAsJavaList	Seq	List
mutableSeqAsJavaList	mutable.Seq	List
bufferAsJavaList	mutable.Buffer	List
setAsJavaSet	Set	Set
mutableSetAsJavaSet	mutable.Set	Set
mapAsJavaMap	Map	Map
mutableMapAsJavaMap	mutable.Map	Map
asJavaDictionary	Map	Dictionary
asJavaConcurrentMap	mutable.ConcurrentMap	concurrent.ConcurrentMap

Таблица 13.5 Преобразования из коллекций Java в коллекции Scala

Неявная функция	Исходный тип в java.util	Целевой тип в scala.collection
collectionAsScalaIterable	Collection	Iterable
iterableAsScalaIterable	Iterable	Iterable
asScalaIterator	Iterator	Iterator
enumerationAsScalaIterator	Enumeration	Iterator
asScalaBuffer	List	mutable.Buffer
asScalaSet	Set	mutable.Set
mapAsScalaMap	Map	mutable.Map
dictionaryAsScalaMap	Dictionary	mutable.Map
propertiesAsScalaMap	Properties	mutable.Map
asScalaConcurrentMap	concurrent.ConcurrentMap	mutable.ConcurrentMap

Имейте в виду, что преобразования возвращают обертки, позволяющие использовать целевой интерфейс для доступа к оригинальному типу. Например, использовать

```
val props: scala.collection.mutable.Map[String, String] =
    System.getProperties()
```

`props` будет представлять обертку, методы которой вызывают Java-методы, лежащие в основе. Если вызвать

```
props("com.horstmann.scala") = "impatient"
```

обертка вызовет метод `put("com.horstmann.scala", "impatient")` объекта `Properties`.

13.16. Потокобезопасные коллекции

При работе с изменяемыми коллекциями в многопоточной среде выполнения необходимо гарантировать, что один поток не попытается изменить ее, пока к ней обращается другой поток. В библиотеке Scala имеются шесть трейтов, которые можно подмешивать в коллекции для организации синхронизации операций с ними:

```
SynchronizedBuffer
SynchronizedMap
SynchronizedPriorityQueue
SynchronizedQueue
SynchronizedSet
SynchronizedStack
```

Например, ниже показано, как сконструировать ассоциативный массив с поддержкой синхронизации операций:

```
val scores = new scala.collection.mutable.HashMap[String, Int] with
    scala.collection.mutable.SynchronizedMap[String, Int]
```

> **Внимание.** Прежде чем подмешивать эти трейты, убедитесь, что понимаете, что они делают, а чего не делают. В предыдущем примере можно быть уверенным, что ассоциативный массив `scores` не будет поврежден – любая из операций с ним будет выполнена до конца, прежде чем другой поток сможет выполнить другую операцию. Однако одновременные из-

менения или итерации небезопасны и наверняка приведут к ошибкам во время выполнения.

В общем случае предпочтительнее использовать один из классов, имеющихся в пакете `java.util.concurrent`. Например, если необходимо обеспечить доступ к общему ассоциативному массиву из нескольких потоков выполнения, используйте `ConcurrentHashMap` или `ConcurrentSkipListMap`. Эти коллекции более эффективны, чем ассоциативный массив, который просто синхронизирует вызовы всех своих методов. Разные потоки могут одновременно обращаться к несвязанным частям структуры данных. (Не пытайтесь проделать это дома!) Кроме того, итераторы оказываются в многопоточной среде «непредставительными», так как они отражают представление, бывшее верным на момент получения итератора.

Адаптировать коллекции из `java.util.concurrent` для использования в программном коде на языке Scala можно, как описано в предыдущем разделе.

13.17. Параллельные коллекции

Непросто без ошибок написать многопоточную программу, и все же многопоточность в наше время пользуется большим спросом, так как позволяет максимально задействовать вычислительные мощности многопроцессорных систем. Scala предлагает особенно привлекательное решение для использования в задачах, связанных с манипулированием большими коллекциями. Часто такие задачи решаются с использованием параллельных алгоритмов. Например, чтобы вычислить сумму всех элементов, можно одновременно запустить несколько потоков, вычисляющих суммы различных фрагментов коллекции; в конце полученные результаты складываются. Конечно, распределять работу между несколькими потоками довольно хлопотно, но в Scala эти хлопоты можно забыть. Если предположить, что `coll` – это большая коллекция, тогда

```
coll.par.sum
```

вычислит сумму в нескольких потоках. Метод `par` воспроизводит параллельную реализацию коллекции. Эта реализация распределяет выполнение методов коллекции по нескольким потокам, если это возможно. Например,

```
coll.par.count(_ % 2 == 0)
```

подсчитает количество четных чисел в `coll`, применив предикат к сегментам коллекции в нескольких параллельных потоках и объединив результаты.

Для массивов, буферов, хеш-таблиц и сбалансированных деревьев параллельные реализации повторно используют реализации, лежащие в основе коллекции, которые весьма эффективны.

Можно распределить выполнение цикла `for` между несколькими потоками, применив `.par` к коллекции, по элементам которой выполняются итерации:

```
for (i <- (0 until 100).par) print(i + " ")
```

Попробуйте выполнить эту инструкцию – числа будут выводиться в порядке их обработки потоками.

В цикле `for/yield` результаты упорядочиваются. Попробуйте следующее:

```
for (i <- (0 until 100).par) yield i + " "
```

> **Внимание.** Если в процессе параллельных вычислений изменяются общие переменные, результат может оказаться непредсказуемым. Например, не следует изменять общий счетчик:
>
> ```
> var count = 0
> for (c <- coll.par) { if (c % 2 == 0) count += 1 } // Ошибка!
> ```

Параллельные коллекции, возвращаемые методом `par`, принадлежат к типам, наследующим трейты `ParSeq`, `ParSet` и `ParMap`, каждый из которых является подтипом `ParIterable`. Они *не* являются подтипами `Iterable`, поэтому параллельную коллекцию нельзя передать методу, ожидающему получить `Iterable`, `Seq`, `Set` или `Map`. Параллельную коллекцию можно преобразовать обратно в последовательную, применив метод `ser`. Или можно реализовать методы так, что они будут принимать параметры обобщенных типов `GenIterable`, `GenSeq`, `GenSet` или `GenMap`.

> **Примечание.** Не все методы могут распределить свою работу между несколькими потоками. Например, `reduceLeft` и `reduceRight` требуют последовательного выполнения операторов. Существует альтернативный

метод `reduce`, выполняющий операции над сегментами коллекции и затем объединяющий результаты. Однако для этого операция должна быть ассоциативной – она должна отвечать требованию `(a op b) op c = a op (b op c)`. Например, сложение – ассоциативная операция, а вычитание – нет: `(a - b) - c ≠ a - (b - c)`.

Существует также похожий метод `fold`, оперирующий сегментами коллекции. К сожалению, он не такой гибкий, как `foldLeft` или `foldRight` – оба аргумента оператора должны быть элементами. То есть можно выполнить свертку `coll.par.fold(0)(_ + _)`, но нельзя реализовать свертку, как было показано в конце раздела 13.10 «Сокращение размерности, свертка и сканирование».

Для решения этой проблемы можно использовать еще более обобщенный метод `aggregate`, который применяет оператор к сегментам коллекции, а затем с помощью другого оператора объединяет результаты. Например, `str.par.aggregate(Set[Char]())(_ + _, _ ++ _)` является эквивалентом `str.foldLeft(Set[Char]())(_ + _)`, формирующим множество уникальных символов в строке `str`.

Упражнения

1. Напишите функцию, возвращающую для указанной строки отображение индексов всех символов. Например, вызов `indexes("Mississippi")` должен вернуть ассоциативный массив, связывающий `'M'` с множеством `{0}`, `'i'` – с множеством `{1, 4, 7, 10}` и т. д. Используйте изменяемый ассоциативный массив, отображающий символы в изменяемые множества. Как обеспечить сортировку индексов в пределах множеств?
2. Реализуйте предыдущее упражнение с использованием неизменяемого ассоциативного массива символов в списки.
3. Напишите функцию, удаляющую нулевые элементы из связанного списка целых чисел.
4. Напишите функцию, принимающую коллекцию строк и ассоциативный массив, отображающий строки в целые числа. Она должна возвращать коллекцию целых чисел, значения которых соответствуют строкам в ассоциативном массиве, повторяющимся в исходной коллекции. Например, для `Array("Tom", "Fred", "Harry")` и `Map("Tom" -> 3, "Dick" -> 4, "Harry" -> 5)` функция должна вернуть `Array(3, 5)`. Подсказка: используйте `flatMap` для объединения значений типа `Option`, возвращаемых методом `get`.
5. Реализуйте функцию, действующую подобно `mkString`, используя `reduceLeft`.

6. Пусть имеется список `lst` целых чисел, что означает выражение `(lst :\ List[Int]())(_ :: _)? (List[Int]() /: lst)(_ :+ _)`? Что можно изменить здесь, чтобы перевернуть список?
7. В разделе 13.11 «Функция zip» выражение `(prices zip quantities) map { p => p._1 * p._2 }` выглядит несколько грубовато. Мы не можем написать `(prices zip quantities) map { _ * _ }`, потому что `_ * _` – это функция с двумя аргументами, а нам нужна функция, принимающая один аргумент – кортеж. Метод `tupled` класса `Function2` изменяет функцию с двумя аргументами, превращая ее в функцию, принимающую кортеж. Примените метод `tupled` к функции умножения так, чтобы можно было применить `map` к списку пар.
8. Напишите функцию, превращающую массив значений `Double` в двумерный массив. Число колонок должно передаваться в виде параметра. Например, для `Array(1, 2, 3, 4, 5, 6)` и трех колонок функция должна вернуть `Array(Array(1, 2, 3), Array(4, 5, 6))`. Используйте метод `grouped`.
9. Гарри Хакер (Harry Hacker) написал программу, принимающую последовательность имен файлов из командной строки. Для каждого файла программа запускает отдельный поток, который читает содержимое файла и обновляет ассоциативный массив с частотами встречаемости символов, объявленное как

```
val frequencies = new scala.collection.mutable.HashMap[Char, Int] with
     scala.collection.mutable.SynchronizedMap[Char, Int]
```

   При чтении символа `c` он вызывает

```
frequencies(c) = frequencies.getOrElse(c, 0) + 1
```

   Почему такой способ не работает? Будет ли работать следующий способ

```
import scala.collection.JavaConversions.asScalaConcurrentMap
val frequencies: scala.collection.mutable.ConcurrentMap[Char, Int] =
     new java.util.concurrent.ConcurrentHashMap[Char, Int]
```

10. Гарри Хакер читает файл в строку и хотел бы использовать параллельную коллекцию для обновления частот встречаемости символов в строке. Он использует следующий код:

```
val frequencies = new scala.collection.mutable.HashMap[Char, Int]
for (c <- str.par) frequencies(c) = frequencies.getOrElse(c, 0) + 1
```

    Почему этот прием не даст желаемых результатов? Как можно было бы распараллелить вычисления? (Подсказка: используйте `aggregate`.)

Глава 14. Сопоставление с образцом и case-классы

Темы, рассматриваемые в этой главе A2

- 14.1. Лучше чем switch.
- 14.2. Ограничители.
- 14.3. Переменные в образцах.
- 14.4. Сопоставление с типами.
- 14.5. Сопоставление с массивами, списками и кортежами.
- 14.6. Экстракторы.
- 14.7. Образцы в объявлениях переменных.
- 14.8. Образцы в выражениях for.
- 14.9. Case-классы.
- 14.10. Метод copy и именованные параметры.
- 14.11. Инфиксная нотация в предложениях case.
- 14.12. Сопоставление с вложенными структурами.
- 14.13. Так ли необходимы case-классы?
- 14.14. Запечатанные классы.
- 14.15. Имитация перечислений.
- 14.16. Тип Option.
- 14.17. Частично определенные функции L2.
- Упражнения.

Scala обладает мощным механизмом *сопоставления с образцами*, имеющим весьма широкую область применений: инструкции-переключатели, определение типа и «деструктуризация» (destructuring – извлечение частей сложных выражений). Кроме того, в Scala имеются case-классы, оптимизированные для работы с механизмом сопоставления с образцами.

Основные темы этой главы:

- выражение `match` лучше `switch` и не страдает эффектом «проваливания»;
- при отсутствии совпадения с образцом возбуждается исключение `MatchError`; используйте образец `case _`, чтобы избежать этого;

- образец может включать произвольное условие, называемое ограничителем (guard);
- имеется возможность выполнять сопоставление с типом выражения, что предпочтительнее, чем `isInstanceOf`/`asInstanceOf`;
- имеется возможность сопоставлять с образцами массивов, кортежей и case-классов и связывать отдельные части образцов с переменными;
- в выражении `for` несовпадающие фрагменты просто пропускаются;
- case-классы – это классы, для которых компилятор автоматически создает методы, необходимые для выполнения сопоставления с образцом;
- общий суперкласс в иерархии case-класса должен быть «запечатан»;
- используйте тип `Option` для значений, которые могут отсутствовать, – это безопаснее, чем использовать `null`.

14.1. Лучше, чем switch

Ниже приводится эквивалент C-подобной инструкции `switch` в языке Scala:

```
var sign = ...
val ch: Char = ...

ch match {
     case '+' => sign = 1
     case '-' => sign = -1
     case _   => sign = 0
}
```

Эквивалентом предложения `default` является универсальный образец `case _`. Желательно всегда включать такой универсальный образец, так как в случае отсутствия совпадения возбуждается исключение `MatchError`.

В отличие от инструкции `switch`, выражение сопоставления с образцом в Scala не страдает эффектом «проваливания». (В языке C и его производных для выхода из инструкции `switch` в конце каждой ветви приходится явно использовать ключевое слово `break`, иначе будет продолжено выполнение следующей ветви. Это иногда раздражает и способствует появлению ошибок.)

Примечание. В своей интереснейшей книге «Deep C Secrets» Питер ван дер Линден (Peter van der Linden) сообщает об исследовании значительного объема программного кода на C, которое показало, что в 97% случаев эффект «проваливания» не использовался.

Подобно `if`, `match` – это выражение, а не инструкция. Предыдущий пример можно упростить до:

```
sign = ch match {
     case '+' => 1
     case '-' => -1
     case _   => 0
}
```

Выражение `match` можно использовать не только с числами, но и с любыми другими типами. Например:

```
color match {
     case Color.RED   => ...
     case Color.BLACK => ...
     ...
}
```

14.2. Ограничители

Допустим, что необходимо добавить в наш пример сопоставление с остальными цифрами. В C-подобной инструкции `switch` можно было бы просто указать несколько меток `case`, например `case '0': case '1': ... case '9':`. (За исключением, конечно, троеточия `...`, вместо которого необходимо было бы явно указать все десять вариантов.) В Scala можно добавить в образец *ограничитель* (guard clause), например:

```
ch match {
     case '+' => sign = 1
     case '-' => sign = -1
     case _ if Character.isDigit(ch) => digit = Character.digit(ch, 10)
     case _ => sign = 0
}
```

Предложение-ограничитель (guard clause) может быть любым логическим условием.

Обратите внимание, что образцы всегда сопоставляются сверху вниз. Если образец с предложением-ограничителем (guard clause) не найдет совпадения, будет выполнена попытка сопоставления с универсальным образцом.

14.3. Переменные в образцах

Если за ключевым словом `case` следует имя переменной, результат выражения сопоставления будет присвоен этой переменной. Например:

```
str(i) match {
     case '+' => sign = 1
     case '-' => sign = -1
     case ch  => digit = Character.digit(ch, 10)
}
```

Универсальный образец `case _` можно рассматривать как частный случай этой особенности, где роль имени переменной играет _.

Имя переменной можно использовать в ограничителе (guard):

```
str(i) match {
     case ch if Character.isDigit(ch) => digit = Character.digit(ch, 10)
     ...
}
```

Внимание. К сожалению, переменные в образцах могут конфликтовать с выражениями-константами, например:

```
import scala.math._
x match {
     case Pi => ...
     ...
}
```

Откуда Scala знает, что `Pi` – это константа, а не переменная? В соответствии с правилами переменная должна начинаться с символа нижнего регистра.

Если имя константы начинается с символа нижнего регистра, заключите его в обратные апострофы:

```
import java.io.File._
str match {
     case `pathSeparator` => ...
     ...
}
```

14.4. Сопоставление с типами

При необходимости можно выполнить сопоставление с типом выражения, например:

```
obj match {
     case x: Int      => x
     case s: String   => Integer.parseInt(s)
     case _: BigInt   => Int.MaxValue
     case _           => 0
}
```

В Scala эта форма является более предпочтительной, чем использование оператора `isInstanceOf`.

Обратите внимание на имена переменных в образцах. В первом образце сопоставление будет выполнено с переменной `x` *как с типом* `Int`, а во втором образце сопоставление будет выполнено с переменной `s` как с типом `String`. Никаких приведений типов с помощью `asInstanceOf` не требуется!

Внимание. При сопоставлении с типами имя переменной должно быть указано обязательно. Иначе будет выполнено сопоставление с объектом:

```
obj match {
    case _: BigInt  => Int.MaxValue // Соответствует любому объекту BigInt
    case BigInt     => -1 // Соответствует объекту BigInt типа Class
}
```

Внимание. Сопоставление выполняется во время выполнения, когда все обобщенные (generic) типы стираются в виртуальной машине Java. Поэтому нельзя выполнить сопоставление с определенным типом ассоциативного массива Map.

```
case m: Map[String, Int] => ... // Нельзя
```

Но можно выполнить сопоставление с обобщенным (generic) типом:

```
case m: Map[_, _] => ... // OK
```

Однако типы массивов не стираются. Поэтому сопоставление с типом `Array[Int]` допустимо.

14.5. Сопоставление с массивами, списками и кортежами

Чтобы выполнить сопоставление с содержимым массива, используйте выражения `Array` в образцах, как показано ниже:

```
arr match {
     case Array(0)      => "0"
     case Array(x, y)   => x + " " + y
     case Array(0, _*)  => "0 ..."
     case _             => "something else"
}
```

Первому образцу соответствует массив, содержащий `0`. Второму – любой массив с двумя элементами, значения которых будут присвоены переменным `x` и `y`. Третьему образцу соответствует любой массив, начинающийся с нуля.

Аналогично можно выполнять сопоставление с выражениями `List`. Допускается также использовать оператор `::`:

```
lst match {
     case 0 :: Nil        => "0"
     case x :: y :: Nil   => x + " " + y
     case 0 :: tail       => "0 ..."
     case _               => "something else"
}
```

Для сопоставления с кортежами в образцах используется форма записи кортежей:

```
pair match {
     case (0, _) => "0 ..."
     case (y, 0) => y + " 0"
     case _      => "neither is 0"
}
```

Еще раз обратите внимание, как переменным в образцах присваивается фрагмент списка или кортежа. Эти присвоения дают простой доступ к частям сложных структур, поэтому данная операция называется *деструктуризацией* (destructuring).

14.6. Экстракторы

В предыдущем разделе было показано, как выполнять сопоставление с массивами, списками и кортежами. Такая возможность обеспечивается *экстракторами* – объектами с методом `unapply` или `unapplySeq`, извлекающим значения из объекта. Реализация этих ме-

тодов описывается в главе 11. Метод `unapply` используется для извлечения фиксированного количества объектов, а метод `unapplySeq` – последовательности объектов, длина которой может изменяться.

Например, взгляните на следующее выражение:

```
arr match {
     case Array(0, x) => ...
     ...
}
```

Объект-компаньон `Array` – это экстрактор, он определяет метод `unapplySeq`. Этому методу передается *сопоставляемое выражение* (в данном случае `arr`), а не то, что является параметром образца. Вызов `Array.unapplySeq(arr)` вернет последовательность значений, то есть значений элементов массива. Затем будет выполнено сравнение первого значения с нулем, а второе значение будет присвоено переменной `x`.

Еще одной областью применения экстракторов являются регулярные выражения. Когда регулярное выражение имеет группы, с помощью экстрактора образца можно выполнить сопоставление для каждой группы. Например:

```
val pattern = "([0-9]+) ([a-z]+)".r
"99 bottles" match {
     case pattern(num, item) => ...
          // num получит значение "99", а item - "bottles"
}
```

Вызов `pattern.unapplySeq("99 bottles")` вернет последовательность строк, совпавших с группами, которые будут присвоены переменным `num` и `item`.

Обратите внимание, что данный экстрактор является не объектом-компаньоном, а объектом регулярного выражения.

14.7. Образцы в объявлениях переменных

В предыдущих разделах вы видели, что образцы могут содержать переменные. Подобные образцы можно использовать в объявлениях переменных. Например,

```
val (x, y) = (1, 2)
```

одновременно определит переменную x со значением 1 и y со значением 2. Такой прием удобно использовать с функциями, возвращающими пары значений, например:

```
val (q, r) = BigInt(10) /% 3
```

Метод /% возвращает пару значений, частное и остаток, которые затем сохраняются в переменных q и r.

Тот же синтаксис может применяться в любых образцах с именами переменных. Например,

```
val Array(first, second, _*) = arr
```

присвоит первый и второй элементы массива arr переменным first и second.

14.8. Образцы в выражениях for

Образцы с переменными можно использовать в for-генераторах (for comprehensions). При обходе значений каждое из них поочередно присваивается переменным. Это позволяет реализовать обход элементов ассоциативного массива:

```
import scala.collection.JavaConversions.propertiesAsScalaMap
// Преобразование Java Properties
// в ассоциативный массив Scala - только ради примера
for ((k, v) <- System.getProperties())
     println(k + " -> " + v)
```

При переходе к очередной паре (*ключ*, *значение*) в ассоциативном массиве, переменной k присваивается ключ, а переменной v – значение.

Отсутствие совпадения с образцом в for-генераторе (for comprehension) просто игнорируется. Например, следующий цикл выведет ключи с пустыми значениями и пропустит все остальные:

```
for ((k, "") <- System.getProperties())
     println(k)
```

Можно также использовать ограничитель (guard). Обратите внимание, что оператор `if` следует после стрелки `<-`:

```
for ((k, v) <- System.getProperties() if v == "")
     println(k)
```

14.9. Case-классы

Case-классы – это классы особого рода, оптимизированные для использования в операциях сопоставления с образцом. В следующем примере определяются два case-класса, наследующих обычный класс (не являющийся case-классом):

```
abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
```

Можно также определять case-объекты:

```
case object Nothing extends Amount
```

При наличии объекта `Amount` можно воспользоваться операцией сопоставления с образцом, чтобы определить тип валюты и присвоить значения свойств переменным:

```
amt match {
     case Dollar(v)        => "$" + v
     case Currency(_, u)   => "Oh noes, I got " + u
     case Nothing          => ""
}
```

> **Примечание.** Скобки `()` используются только с экземплярами case-классов, но нее с case-объектами.

При объявлении case-класса автоматически выполняется следующее.

- Каждый параметр конструктора становится значением `val`, если явно не объявлен как `var` (что, впрочем, не рекомендуется).
- Создается объект-компаньон с методом `apply`, позволяющим конструировать объекты без оператора `new`, например: `Dollar(29.95)` или `Currency(29.95, "EUR")`.

- Предоставляется метод `unapply`, необходимый для операции сопоставления с образцом, – подробности см. в главе 11. (В действительности, чтобы использовать case-классы в операциях сопоставления с образцом, знать все эти подробности необязательно.)
- Генерируются методы `toString`, `equals`, `hashCode` и `copy`, если они не были реализованы явно.

В остальном case-классы ничем не отличаются от других классов. В них можно добавлять свои поля и методы, наследовать их в других классах и т. д.

14.10. Метод copy и именованные параметры

Метод `copy` case-класса создает новый объект с теми же значениями полей, что и в существующем. Например:

```
val amt = Currency(29.95, "EUR")
val price = amt.copy()
```

Сама по себе эта операция не несет большой пользы – в конце концов, объект `Currency` является неизменяемым, и его безопасно можно совместно использовать в разных частях программы. Однако, используя именованные параметры, можно изменять некоторые его свойства:

```
val price = amt.copy(value = 19.95) // Currency(19.95, "EUR")
```

или

```
val price = amt.copy(unit = "CHF") // Currency(29.95, "CHF")
```

14.11. Инфиксная нотация в предложениях case

В случаях, когда метод `unapply` возвращает пару значений, в предложениях `case` можно использовать инфиксную форму записи. В частности, можно использовать инфиксную форму записи имени класса с двумя параметрами. Например:

```
amt match { case a Currency u => ... } // То же, что и case Currency(a, u)
```

Конечно, это не очень наглядный пример. Данная возможность предназначена в первую очередь для сопоставления с последовательностями. Например, любой объект `List` может быть либо значением `Nil`, либо объектом case-класса `::`, объявленного как

```
case class ::[E](head: B, tail: List[E]) extends List[E]
```

То есть можно записать

```
lst match { case h :: t => ... }
    // То же, что и case ::(h, t), который вызовет ::.unapply(result)
```

В главе 19 вы встретитесь с case-классом `~`, используемым для пар значений результатов парсинга. Он также может использоваться в инфиксных выражениях в предложениях `case`:

```
result match { case p ~ q => ... } // То же, что и case ~(p, q)
```

Эти инфиксные выражения упрощают чтение программ, когда их более одного. Например,

```
result match { case p ~ q ~ r => ... }
```

читается проще, чем `~(~(p, q), r)`.

Если оператор заканчивается двоеточием, он является правоассоциативным. Например,

```
case first :: second :: rest
```

означает

```
case ::(first, ::(second, rest))
```

Примечание. Инфиксная форма записи может использоваться при работе с любым методом `unapply`, возвращающим пару значений. Например:

```
case object +: {
    def unapply[T](input: List[T]) =
        if (input.isEmpty) None else Some((input.head, input.tail))
}
```

Теперь можно разложить список с помощью оператора `+:`.

```
1 +: 7 +: 2 +: 9 +: Nil match {
    case first +: second +: rest => first + second + rest.length
}
```

14.12. Сопоставление с вложенными структурами

Case-классы часто используются для сопоставления с вложенными структурами. Например, представьте товары, продаваемые в магазине. Иногда они объединяются для предоставления скидки.

```
abstract class Item
case class Article(description: String, price: Double) extends Item
case class Bundle(description: String, discount: Double, items: Item*) extends Item
```

Отсутствие необходимости использовать ключевое слово `new` упрощает создание вложенных объектов:

```
Bundle("Father's day special", 20.0,
     Article("Scala for the Impatient", 39.95),
     Bundle("Anchor Distillery Sampler", 10.0,
          Article("Old Potrero Straight Rye Whiskey", 79.95),
          Article("Junípero Gin", 32.95)))
```

В образцах можно организовать сопоставление с конкретным вложением, например

```
case Bundle(_, _, Article(descr, _), _*) => ...
```

присвоит переменной `descr` описание первого товара в пакете.

При желании присвоить вложенное значение переменной можно с помощью аннотации `@`:

```
case Bundle(_, _, art @ Article(_, _), rest @ _*) => ...
```

Теперь переменной `art` будет присвоен первый товар в пакете, а переменной `rest` – последовательность остальных товаров.

Обратите внимание, что конструкция `_*` является обязательной в этом примере. Образец

```
case Bundle(_, _, art @ Article(_, _), rest) => ...
```

соответствует пакету с основным товаром и единственным дополнительным товаром, который будет присвоен переменной `rest`.

Как приложение ниже приводится функция, вычисляющая стоимость товара:

```
def price(it: Item): Double = it match {
     case Article(_, p) => p
     case Bundle(_, disc, its @ _*) => its.map(price _).sum - disc
}
```

14.13. Так ли необходимы case-классы?

Пример в предыдущем разделе может вызвать гнев у адептов объектно-ориентированного программирования. Разве элемент `price` не должен быть методом суперкласса? Разве подкласс не должен переопределять его? Разве полиморфизм хуже, чем сопоставление с типами?

Во многих ситуациях они будут правы. Если кто-то решит создать другую разновидность `Item`, ему придется пересмотреть все эти выражения `match`. В подобных ситуациях применение case-классов – не самое лучшее решение.

Case-классы отлично подходят для работы со структурами, определение которых не изменяется. Например, класс `List` в Scala реализован с применением case-классов. В упрощенном представлении список – это:

```
abstract class List
case object Nil extends List
case class ::(head: Any, tail: List) extends List
```

Список может быть либо пустым, либо будет иметь голову и хвост (последний из которых сам может быть пустым списком). Никому и в голову не взбредет добавить третий вариант. (В следующем разделе будет показано, как можно предотвратить подобные попытки.)

Однако в подходящих ситуациях case-классы оказываются весьма удобным инструментом по следующим причинам:

- сопоставление с образцом часто позволяет сократить объем программного кода в сравнении с наследованием;
- операции конструирования объектов без ключевого слова `new` читаются намного проще;
- вы бесплатно получаете методы `toString`, `equals`, `hashCode` и `copy`.

Эти методы, генерируемые автоматически, делают именно то, что они должны делать, – выводят, сравнивают, вычисляют значение хеш-функции и копируют каждое поле. Дополнительная информация о методе `copy` приводится в разделе 14.10 «Метод copy и именованные параметры» выше.

Для определенных разновидностей классов case-классы обеспечивают более точную семантику. Некоторые называют их *классами-значениями* («value classes»). Например, взгляните на класс `Currency`:

```
case class Currency(value: Double, unit: String)
```

Объект `Currency(10, "EUR")` считается эквивалентным любому другому объекту `Currency(10, "EUR")` – именно так реализованы методы `equals` и `hashCode`. Обычно такие классы являются неизменяемыми.

Case-классы с полями-переменными выглядят несколько сомнительно, по крайней мере это относится к хеш-коду. При работе с изменяемыми классами хеш-код всегда должен вычисляться только на основе неизменяемых полей, таких как числовые идентификаторы (ID).

> **Внимание.** Методы `toString`, `equals`, `hashCode` и `copy` не генерируются для case-классов, наследующих другие case-классы. Если один case-класс будет наследовать другой case-класс, компилятор выведет предупреждение. В будущих версиях Scala такое наследование вообще может быть сделано недопустимым[1]. Если вам потребуется организовать несколько уровней в иерархии наследования, чтобы вынести общие черты поведения, создавайте обычные классы, а case-классы делайте только листьями такого дерева наследования.

14.14. Запечатанные классы

При использовании case-классов в операциях сопоставления с образцом было бы желательно, чтобы компилятор проверил наличие в конструкции `match` всех возможных альтернатив. Добиться этого можно, объявив общий суперкласс *«запечатанным»* (sealed):

```
sealed abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
```

Все подклассы запечатанного класса должны объявляться в том же файле, что и сам класс. Например, если кто-то захочет добавить другой класс для евро

```
case class Euro(value: Double) extends Amount
```

ему придется добавить его в файл, где объявлен класс `Amount`.

---

[1] Начиная с версии Scala 2.10 такое наследование более недопустимо. – *Прим. ред.*

Когда класс объявляется запечатанным, все его подклассы будут доступны на этапе компиляции, что позволит компилятору проверить сопоставление с образцом на полноту. Решение, когда все case-классы наследуют запечатанный класс или трейт, считается более предпочтительным.

14.15. Имитация перечислений

Case-классы позволяют имитировать перечислимые типы в Scala.

```
sealed abstract class TrafficLightColor
case object Red extends TrafficLightColor
case object Yellow extends TrafficLightColor
case object Green extends TrafficLightColor

color match {
     case Red    => "stop"
     case Yellow => "hurry up"
     case Green  => "go"
}
```

Обратите внимание, что суперкласс объявлен как запечатанный (`sealed`). Это позволяет компилятору убедиться в полноте выражения `match`.

Кому такое решение покажется несколько тяжеловесным, может воспользоваться вспомогательным классом `Enumeration`, описанным в главе 6.

14.16. Тип Option

Для представления значений, которые могут отсутствовать, используется тип `Option`, имеющийся в стандартной библиотеке, также основанный на case-классах. Case-класс `Some` обертывает значение, например `Some("Fred")`. Case-объект `None` указывает на отсутствие значения.

Это более однозначно, чем использование пустой строки, и более безопасно, чем использование `null` для обозначения отсутствующего значения.

`Option` – это обобщенный тип. Например, `Some("Fred")` имеет тип `Option[String]`.

Метод `get` класса `Map` возвращает `Option`. Если для указанного ключа отсутствует значение, метод `get` вернет `None`. В противном случае он завернет возвращаемое значение в `Some`.

Для анализа таких значений можно использовать операцию сопоставления с образцом:

```
scores.get("Alice") match {
     case Some(score) => println(score)
     case None        => println("No score")
}
```

Честно говоря, это несколько утомительно. Как вариант можно использовать `isEmpty` и `get`:

```
val alicesScore = scores.get("Alice")
if (alicesScore.isEmpty) println("No score")
else println(alicesScore.get)
```

Однако это тоже утомительно. Более удачное решение дает метод `getOrElse`:

```
println(alicesScore.getOrElse("No score"))
```

Если `alicesScore` имеет значение `None`, тогда `getOrElse` вернет `"No Score"`. Эта ситуация настолько часто встречается в практике, что в класс `Map` был включен метод `getOrElse`:

```
println(scores.getOrElse("Alice", "No score"))
```

Если потребуется пропустить значение `None`, используйте `for`-генератор (`for` comprehension):

```
for (score <- scores.get("Alice")) println(score)
```

Если метод `get` вернет `None`, ничего не произойдет. Если он вернет `Some`, переменной `score` будет присвоено полученное от него значение.

Тип `Option` можно рассматривать как коллекцию, которая может быть пустой или содержать единственный элемент, и использовать методы, такие как `map`, `foreach` или `filter`. Например,

```
scores.get("Alice").foreach(println _)
```

выведет очки или не выведет ничего, если `get` вернет `None`.

14.17. Частично определенные функции L2

Множество предложений `case`, заключенных в фигурные скобки, образуют *частично определенную функцию* (partial function) – функцию, которая может быть определена не для всех входных значений. Такие функции являются экземплярами класса `PartialFunction[A, B]`. (`A` – тип параметра, `B` – тип возвращаемого значения.) Этот класс имеет два метода: `apply`, вычисляющий значение функции из сопоставления с образцом, и `isDefinedAt`, возвращающий `true`, если входное значение совпадает хотя бы с одним образцом.

Например,

```
val f: PartialFunction[Char, Int] = { case '+' => 1 ; case '-' => -1 }
f('-')              // Вызов f.apply('-') вернет -1
f.isDefinedAt('0')  // false
f('0')              // Возбудит исключение MatchError
```

Некоторые методы могут принимать `PartialFunction` в качестве параметра. Например, метод `collect` трейта `GenTraversable` применяет частично вычислимую функцию ко всем элементам, где она определена, и возвращает последовательность результатов.

```
"-3+4".collect { case '+' => 1 ; case '-' => -1 } // Vector(-1, 1)
```

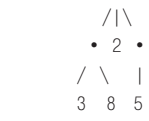
> **Примечание.** Выражение частично определенной функции должно находиться в контексте, где компилятор сможет определить тип возвращаемого значения. Это можно обеспечить, присвоив значение выражения типизированной переменной или передав ее в виде аргумента.

Упражнения

1. В состав дистрибутива Java Development Kit входит большая часть исходных текстов JDK в архиве src.zip. Распакуйте его и поищите метки `case` (регулярное выражение: `case [^:]+:`). Затем почитайте комментарии, начинающиеся с `//` и содержащие `alls? thr`, такие как `// Falls through` или `// just fall thru`. Учитывая, что соглашения по оформлению программного кода на Java требуют от JDK-программистов оставлять такие коммен-

тарии, посчитайте процент предложений `case`, допускающих «проваливание» в следующее предложение `case`.

2. Используя сопоставление с образцом, напишите функцию `swap`, которая принимает пару целых чисел и возвращает ту же пару, поменяв компоненты местами.
3. Используя сопоставление с образцом, напишите функцию `swap`, которая меняет местами первые два элемента массива, если он имеет длину не меньше двух.
4. Добавьте case-класс `Multiple`, наследующий класс `Item`. Например, `Multiple(10, Product("Blackwell Toaster", 29.95))` описывает десять тостеров. Разумеется, должна предусматриваться возможность обрабатывать любые элементы, такие как пакет или множитель, во втором аргументе. Расширьте функцию `price`, чтобы она могла обрабатывать новый вариант.
5. Для представления деревьев, хранящих значения только в листьях, можно использовать списки. Например, список `((3 8) 2 (5))` описывает дерево



В этом случае одни элементы списка будут числами, а другие – списками. Однако в Scala нельзя создавать разнородные списки, Поэтому придется использовать `List[Any]`. Напишите функцию `leafSum` для вычисления суммы всех значений листьев, используя сопоставление с образцом для отделения чисел от списков.

6. Такие деревья лучше всего моделировать с применением case-классов. Начните с бинарных деревьев.

```
sealed abstract class BinaryTree
case class Leaf(value: Int) extends BinaryTree
case class Node(left: BinaryTree, right: BinaryTree) extends BinaryTree
```

Напишите функцию, вычисляющую сумму всех значений листьев.

7. Расширьте дерево из предыдущего упражнения, чтобы каждый узел в нем мог иметь произвольное количество дочерних узлов, и перепишите функцию `leafSum`. Дерево в упражнении 5 должно выражаться как:

```
Node(Node(Leaf(3), Leaf(8)), Leaf(2), Node(Leaf(5)))
```

8. Расширьте дерево из предыдущего упражнения, чтобы каждый узел, не являющийся листом, вдобавок к дочерним узлам мог хранить оператор. Затем напишите функцию `eval`, вычисляющую значение. Например, дерево

```
   +
 / | \
 * 2 -
/ \   |
3  8  5
```

имеет значение $(3 \times 8) + 2 + (-5) = 21$.

9. Напишите функцию, вычисляющую сумму всех непустых значений в `List[Option[Int]]`. Не используйте выражение `match`.
10. Напишите функцию, получающую две функции типа `Double => Option[Double]` и конструирующую на их основе третью функцию того же типа. Новая функция должна возвращать `None`, если любая из двух исходных функций вернет это значение. Например:

```
def f(x: Double) = if (x >= 0) Some(sqrt(Double)) else None
def g(x: Double) = if (x != 1) Some(1 / (x - 1)) else None
val h = compose(f, g)
```

Вызов `h(2)` должен вернуть `Some(1)`, а вызовы `h(1)` и `h(0)` должны вернуть `None`.

Глава 15. Аннотации

Темы, рассматриваемые в этой главе A2

- 15.1. Что такое аннотации?
- 15.2. Что можно аннотировать?
- 15.3. Аргументы аннотаций.
- 15.4. Реализация аннотаций.
- 15.5. Аннотации для Java Features
  - Модификаторы Java.
  - Интерфейсы-маркеры.
  - Контролируемые исключения.
  - Списки аргументов переменной длины.
  - Компоненты JavaBeans.
- 15.6. Аннотации для оптимизации.
  - Хвостовая рекурсия.
  - Создание таблиц переходов и встраивание.
  - Игнорируемые методы.
  - Специализация простых типов.
- 15.7. Аннотации ошибок и предупреждений.
- Упражнения.

Аннотации позволяют добавлять дополнительную информацию в элементы программы. Эта информация может обрабатываться компилятором или внешними инструментами. В данной главе вы узнаете, как задействовать аннотации языка Java и как пользоваться аннотациями языка Scala.

Основные темы этой главы:

- аннотировать можно классы, методы, поля, локальные переменные, параметры, выражения, типы параметров и типы;
- в случае с выражениями и типами аннотация следует за аннотируемым элементом;
- аннотации имеют вид `@Annotation`, `@Annotation(value)` или `@Annotation(name1 = value1, ...)`;
- аннотации `@volatile`, `@transient`, `@strictfp` и `@native` генерируют эквивалентные Java-модификаторы;

- для создания определений `throws`, совместимых с Java, используйте аннотацию `@throws`;
- аннотация `@tailrec` позволяет убедиться, что к рекурсивной функции применяется оптимизация хвостового вызова;
- функция `assert` использует аннотацию `@elidable`; вы можете выборочно удалять проверки из своих программ на языке Scala;
- используйте аннотацию `@deprecated`, чтобы пометить особенности, не рекомендуемые к использованию.

15.1. Что такое аннотации?

Аннотации – это теги, добавляемые в исходный программный код с целью обработки некоторыми дополнительными инструментами. Эти инструменты могут действовать на уровне исходного программного кода или обрабатывать файлы классов, куда компилятор поместил аннотации.

Аннотации широко используются в Java, например для тестирования инструментами, такими как JUnit 4, или использования технологиями, такими как JavaEE.

Синтаксис аннотаций такой же, как в Java. Например:

```
@Test(timeout = 100) def testSomeFeature() { ... }

@Entity class Credentials {
     @Id @BeanProperty var username : String = _
     @BeanProperty var password : String = _
}
```

Аннотации Java можно использовать с классами Scala. Аннотации в предыдущих примерах принадлежат JUnit и JPA, двум фреймворкам Java, ничего не знающим о существовании Scala.

Можно также использовать аннотации языка Scala. Они являются характерными для Scala и обычно обрабатываются компилятором Scala или *расширениями компилятора*.

> **Примечание.** Реализация расширений для компилятора – весьма нетривиальная задача и не рассматривается в этой книге. Введение в создание расширений можно найти по адресу www.scala-lang.org/node/140.

Аннотации Java не влияют на порядок компиляции исходного кода в байт-код – они просто добавляют данные в байт-код, которые

затем могут быть обработаны внешними инструментами. Аннотации языка Scala воздействуют на сам процесс компиляции. Например, аннотация `@BeanProperty`, с которой вы познакомились в главе 5, вызывает автоматическое создание методов чтения и записи.

15.2. Что можно аннотировать?

В Scala можно аннотировать классы, методы, поля, локальные переменные и параметры, точно так же, как в Java.

```
@Entity class Credentials
@Test def testSomeFeature() {}
@BeanProperty var username = _
def doSomething(@NotNull message: String) {}
```

Можно одновременно указать несколько аннотаций. Порядок их следования не имеет значения.

```
@BeanProperty @Id var username = _
```

При аннотировании главного конструктора аннотация должна помещаться перед ним, и не забывайте добавлять пару пустых скобок, если аннотация не имеет аргументов.

```
class Credentials @Inject() (var username: String, var password: String)
```

Можно аннотировать выражения, добавляя двоеточие и аннотацию после выражения, например:

```
(myMap.get(key): @unchecked) match { ... }
     // Выражение myMap.get(key) аннотировано
```

Можно аннотировать типы параметров:

```
class MyContainer[@specialized T]
```

При аннотировании фактических типов аннотация должна помещаться *после* имени типа:

```
String @cps[Unit] // @cps имеет параметр типа
```

Здесь аннотирован тип `String`. (Подробнее об аннотации `@cps` рассказывается в главе 22.)

15.3. Аргументы аннотаций

Аннотации Java могут иметь именованные аргументы, такие как:

```
@Test(timeout = 100, expected = classOf[IOException])
```

Однако, если аргумент имеет имя `value`, его можно опустить. Например:

```
@Named("creds") var credentials: Credentials = _
    // Аргумент value имеет значение "creds"
```

Если аннотация не имеет аргументов, скобки можно опустить:

```
@Entity class Credentials
```

Большинство аргументов аннотаций имеют значения по умолчанию. Например, аргумент `timeout` аннотации `@Test` из фреймворка JUnit по умолчанию имеет значение `0`, что говорит об отсутствии тайм-аута. Значением по умолчанию аргумента `expected` является фиктивный (dummy) класс, что говорит о том, что по умолчанию никаких исключений не ожидается. Таким образом, аннотация

```
@Test def testSomeFeature() { ... }
```

эквивалентна аннотации

```
@Test(timeout = 0, expected = classOf[org.junit.Test.None])
def testSomeFeature() { ... }
```

Аргументы аннотаций Java могут быть лишь нескольких типов:

- числовые литералы;
- строки;
- литералы классов;
- перечисления Java;
- другие аннотации;
- массивы элементов вышеперечисленных типов (но не массивы массивов).

Аргументы аннотаций Scala могут быть произвольных типов, но лишь несколько аннотаций Scala пользуются этим дополнительным преимуществом. Например, аннотация `@deprecatedName` имеет аргумент типа `Symbol`.

15.4. Реализация аннотаций

Я не жду, что многие читатели этой книги почувствуют потребность реализовать собственные аннотации для Scala. Главная цель этого раздела – помочь разобраться в реализации существующих классов аннотаций.

Аннотация должна наследовать трейт `Annotation`. Например, ниже показано определение аннотации `unchecked`:

```
class unchecked extends annotation.Annotation
```

Обратите внимание на значение по умолчанию атрибута `from`.

Класс аннотации может при необходимости наследовать трейт `StaticAnnotation` или `ClassfileAnnotation`. Аннотации, наследующие `StaticAnnotation`, видимы в разных единицах компиляции (compilation units) – они должны помещать в файлы классов метаданные, характерные для языка Scala. Аннотации, наследующие `ClassfileAnnotation`, как предполагается, должны генерировать в файлах классов метаданные аннотаций Java. Однако в Scala 2.9 эта возможность пока не поддерживается.

> **Внимание.** Если у вас появится желание реализовать новую аннотацию Java, вам придется определить класс аннотации на языке Java. Разумеется, эту аннотацию можно будет использовать для аннотирования классов на языке Scala.

Обычно аннотации применяются к выражениям, переменным, полям, методам, классам или типам. Например, аннотация

```
def check(@NotNull password: String)
```

применяется к параметру `password`.

Однако определения полей в Scala могут дать начало множеству программных элементов в Java, каждый из которых мог бы быть аннотирован. Например, в следующем объявлении

```
class Credentials(@NotNull @BeanProperty var username: String)
```

имеются шесть элементов, которые могли бы быть целью аннотаций:

- параметр конструктора;
- приватное поле экземпляра;

- метод чтения `username`;
- метод записи `username_=`;
- метод чтения свойства компонента JavaBean `getUsername`;
- метод записи свойства компонента JavaBean `setUsername`.

По умолчанию аннотации параметров конструктора применяются только к самому параметру, а аннотации полей применяются лишь к полям. Мета-аннотации `@param`, `@field`, `@getter`, `@setter`, `@beanGetter` и `@beanSetter` указываются рядом с одними элементами, а применяются к другим. Например, аннотация `@deprecated` определена как

```
@getter @setter @beanGetter @beanSetter
class deprecated(message: String = "", since: String = "")
     extends annotation.StaticAnnotation
```

Эти аннотации можно также применить к конкретным элементам:

```
@Entity class Credentials {
     @(Id @beanGetter) @BeanProperty var id = 0
     ...
}
```

В данном случае аннотация `@Id` применяется к Java-методу `getId`, который требуется фреймворку JPA для доступа к полю.

15.5. Аннотации для элементов Java

15.5.1. Модификаторы Java

Для некоторых, редко используемых элементов Java вместо ключевых слов модификаторов в Scala используются аннотации.

Аннотация `@volatile` помечает поле как `volatile`:

```
@volatile var done = false // В JVM превратится в volatile-поле
```

Такое поле можно изменять из разных потоков выполнения.

Аннотация `@transient` помечает поле как `transient`:

```
@transient var recentLookups = new HashMap[String, String]
     // В JVM становится transient-полем
```

Такие поля не сериализуются. Подобные поля имеет смысл использовать для данных, которые не должны сохраняться при кешировании, или данных, которые легко могут быть вычислены заново.

Аннотация `@strictfp` является аналогом модификатора `strictfp` в Java:

```
@strictfp def calculate(x: Double) = ...
```

Этот метод выполняет арифметические операции с вещественными числами `double` в формате IEEE, без использования 80-битного представления с увеличенной точностью (которое используется в процессорах Intel по умолчанию). Результат вычисляется медленнее и получается менее точным, зато более переносимым.

Аннотация `@native` помечает методы, реализованные на языке C или C++. Это аналог модификатора `native` в Java.

```
@native def win32RegKeys(root: Int, path: String): Array[String]
```

15.5.2. Интерфейсы-маркеры

Вместо интерфейсов-маркеров (marker interfaces) `Cloneable` и `java.rmi.Remote` для обозначения клонируемых и удаленных объектов в Scala используются аннотации `@cloneable` и `@remote`.

```
@cloneable class Employee
```

> **Внимание.** Не рекомендуется использовать аннотацию `@serializable`. Вместо этого следует наследовать трейт `scala.Serializable`.

При определении сериализуемых классов, указать номер версии сериализации можно с помощью аннотации `@SerialVersionUID`:

```
@SerialVersionUID(6157032470129070425L)
class Employee extends Person with Serializable
```

> **Примечание.** Дополнительную информацию о таких концепциях языка Java, как volatile-поля, клонирование или сериализация, можно найти в книге К. Хорстманна (C. Horstmann) и Г. Корнелла (G. Cornell) «Core Java™, Eighth Edition» (Sun Microsystems Press, 2008)[1].

[1] Хорстманн К. С., Корнелл Г. Java 2. Библиотека профессионала. Основы. – Т. 1. – Вильямс, 2008. – ISBN: 978-5-8459-1378-4. – *Прим. перев.*

15.5.3. Контролируемые исключения

В отличие от компилятора Scala, компилятор Java следит за контролируемыми исключениями (checked exceptions). Если метод Scala вызывается из программного кода на Java, его сигнатура должна включать перечень контролируемых исключений, которые может возбуждать метод. Чтобы сгенерировать корректную сигнатуру, следует использовать аннотацию `@throws`. Например, объявление

```
class Book {
     @throws(classOf[IOException]) def read(filename: String) { ... }
     ...
}
```

сгенерирует в Java сигнатуру:

```
void read(String filename) throws IOException
```

Без аннотации `@throws` программный код на Java не сможет перехватывать исключения.

```
try {      // Это - Java
     book.read("war-and-peace.txt");
} catch (IOException ex) {
     ...
}
```

Компилятор Java должен знать, что метод `read` может возбуждать исключение `IOException`, иначе он отвергнет попытку перехватить его.

15.5.4. Списки аргументов переменной длины

Аннотация `@varargs` позволяет вызывать методы Scala со списками аргументов переменной длины из Java по умолчанию, если метод определяется как:

```
def process(args: String*)
```

Компилятор Scala преобразует список аргументов переменной длины в последовательность

```
def process(args: Seq[String])
```

Что затрудняет вызов такого метода из Java. Если добавить `@varargs`,

```
@varargs def process(args: String*)
```

тогда будет сгенерирован Java-метод

```
void process(String... args) // Переходный Java-метод
```

преобразующий массив `args` в `Seq` и вызывающий метод Scala.

15.5.5. Компоненты JavaBeans

В главе 5 вы познакомились с аннотацией `@BeanProperty`. Когда поле помечается аннотацией `@scala.reflect.BeanProperty`, компилятор генерирует методы чтения/записи в стиле компонентов JavaBeans. Например, для поля

```
class Person {
    @BeanProperty var name : String = _
}
```

будут сгенерированы методы

```
getName() : String
setName(newValue : String) : Unit
```

помимо обычных для Scala методов доступа.

Аннотация `@BooleanBeanProperty` генерирует для логического поля метод чтения с префиксом `is`.

> **Примечание.** Аннотации `@BeanDescription`, `@BeanDisplayName`, `@BeanInfo`, `@BeanInfoSkip` позволяют управлять более таинственными возможностями, определяемыми спецификациями JavaBeans. Однако потребность в этом возникает лишь у немногих программистов. Если вы принадлежите к их числу, поищите описание этих аннотаций в Scaladoc.

15.6. Аннотации для оптимизации

Некоторые аннотации в библиотеке Scala позволяют управлять оптимизациями, применяемыми компилятором. Они обсуждаются в разделах, следующих ниже.

15.6.1. Хвостовая рекурсия

Рекурсивные вызовы иногда можно преобразовать в цикл, что позволяет сэкономить пространство на стеке. Это особенно важно в функциональном программировании, где рекурсия особенно часто используется для организации обхода элементов коллекций.

Рассмотрим рекурсивный метод, вычисляющий сумму целых чисел в последовательности:

```
object Util {
     def sum(xs: Seq[Int]): BigInt =
          if (xs.isEmpty) 0 else xs.head + sum(xs.tail)
     ...
}
```

Этот метод нельзя оптимизировать, потому что последним шагом в нем является операция сложения, а не рекурсивный вызов. Но если его немного изменить:

```
def sum2(xs: Seq[Int], partial: BigInt): BigInt =
     if (xs.isEmpty) partial else sum2(xs.tail, xs.head + partial)
```

оптимизация станет возможной.

Промежуточная сумма передается методу в виде параметра, поэтому начальный его вызов выглядит так: `sum2(xs, 0)`. Поскольку теперь *последним* шагом является рекурсивный вызов, такой метод можно преобразовать в цикл. Компилятор Scala автоматически применит оптимизацию «хвостовой рекурсии» (tail recursion) ко второму методу. Если попытаться вызвать

```
sum(1 to 1000000)
```

будет получена ошибка переполнения стека (по крайней мере, со значением размера стека по умолчанию в JVM), но вызов

```
sum2(1 to 1000000, 0)
```

вернет сумму `500000500000`.

Однако иногда попытка компилятора Scala выполнить оптимизацию хвостовой рекурсии блокируется по неочевидным причинам. Если вы полагаетесь на выполнение компилятором оптимизации хвостовой рекурсии, необходимо пометить метод аннотацией `@tail-`

rec. В этом случае, если компилятор не сможет применить оптимизацию, будет выведено сообщение об ошибке.

Например, допустим, что метод sum2 определен в классе, а не в объекте:

```
class Util {
     @tailrec def sum2(xs: Seq[Int], partial: BigInt): BigInt =
          if (xs.isEmpty) partial else sum2(xs.tail, xs.head + partial)
     ...
}
```

Теперь попытка скомпилировать программу будет завершаться ошибкой с сообщением: "could not optimize @tailrec annotated method sum2: it is neither private nor final so can be overridden" (невозможно выполнить оптимизацию метода sum2, помеченного аннотацией @tailrec: он не является ни приватным (private), ни финальным (final) и может быть переопределен). В этой ситуации можно переместить метод в объект или объявить его приватным (private) или финальным (final).

> **Примечание.** Существует более универсальный механизм устранения рекурсии под названием «трамплининг» («trampolining»). Реализация этого механизма выполняет цикл, в котором вызывает функцию, возвращающую другую функцию для вызова в следующей итерации. Хвостовая рекурсия – особый случай, когда каждая функция возвращает саму себя. Более обобщенный механизм позволяет вызывать взаимно рекурсивные функции (mutual recursion) – см. пример ниже.

В Scala имеется вспомогательный объект TailCalls, который упрощает реализацию механизмов «трамплининга». Взаимно рекурсивные функции должны иметь возвращаемое значение типа TailRec[A] и возвращать либо done(result), либо tailcall(fun), где fun – функция для вызова в следующей итерации. Это должна быть функция без параметров с возвращаемым значением типа TailRec[A]. Ниже приводится простой пример:

```
import scala.util.control.TailCalls._
def evenLength(xs: Seq[Int]): TailRec[Boolean] =
     if (xs.isEmpty) done(true) else tailcall(oddLength(xs.tail))
def oddLength(xs: Seq[Int]): TailRec[Boolean] =
     if (xs.isEmpty) done(false) else tailcall(evenLength(xs.tail))
```

Извлечь окончательный результат из объекта `TailRec` можно с помощью метода `result`:

```
evenLength(1 to 1000000).result
```

15.6.2. Создание таблиц переходов и встраивание

В C++ и Java инструкция `switch` часто компилируется в таблицу переходов, обладающую большей эффективностью, чем последовательность выражений `if/else`. Scala также пытается генерировать таблицы переходов для выражений `match`. Аннотация `@switch` позволяет гарантировать, что предложения `match` в Scala действительно будут компилироваться в такие таблицы. Аннотация должна предшествовать выражению `match`:

```
(n: @switch) match {
     case 0 => "Zero"
     case 1 => "One"
     case _ => "?"
}
```

Еще одна типичная оптимизация – встраивание (inlining) методов, когда вызов метода заменяется его телом. Чтобы предложить компилятору встраивать некоторый метод, пометьте его аннотацией `@inline`. Чтобы исключить возможность встраивания, пометьте его аннотацией `@noinline`. Вообще говоря, встраивание выполняется в виртуальной машине JVM, динамический («just in time») компилятор которой прекрасно справляется с этим, не требуя явных аннотаций. Аннотации `@inline` и `@noinline` позволяют управлять компилятором Scala, когда в этом возникает явная необходимость.

15.6.3. Игнорируемые методы

Аннотация `@elidable` помечает методы, которые можно удалить из окончательной версии. Например, если такой код

```
@elidable(500) def dump(props: Map[String, String]) { ... }
```

скомпилировать с флагами

```
scalac -Xelide-below 800 myprog.scala
```

код метода не будет сгенерирован. Объект `elidable` определяет следующие числовые константы:

- `MAXIMUM` или `OFF = Int.MaxValue`
- `ASSERTION = 2000`
- `SEVERE = 1000`
- `WARNING = 900`
- `INFO = 800`
- `CONFIG = 700`
- `FINE = 500`
- `FINER = 400`
- `FINEST = 300`
- `MINIMUM` или `ALL = Int.MinValue`

Любую из этих констант можно использовать в аннотации:

```
import scala.annotation.elidable._
@elidable(FINE) def dump(props: Map[String, String]) { ... }
```

Имена констант можно также использовать в командной строке:

```
scalac -Xelide-below INFO myprog.scala
```

Если флаг `-Xelide-below` не указан, компилятор будет игнорировать методы, помеченные аннотациями со значениями ниже 1000, оставив методы `SEVERE` и `ASSERTION`, но удалив `WARNING`.

> **Примечание.** Уровни `ALL` и `OFF` часто путают. Аннотация `@elide(ALL)` означает: «игнорировать все методы». А аннотация `@elide(OFF)` означает: «не игнорировать ни один из методов». Но `-Xelide-below OFF` означает: «игнорировать все», а `-Xelide-below ALL` означает «не игнорировать ничего». Именно по этой причине были добавлены имена `MAXIMUM` и `MINIMUM`.

Объект `Predef` определяет игнорируемый метод `assert`. Например, если метод

```
def makeMap(keys: Seq[String], values: Seq[String]) = {
    assert(keys.length == values.length, "lengths don't match")
    ...
}
```

вызвать с несоответствующими друг другу аргументами, метод `assert` возбудит исключение `AssertionError` с сообщением `"assertion`

failed: lengths don't match" («ошибка условия: длины аргументов не совпадают»).

Чтобы запретить такие проверки, достаточно выполнить компиляцию с ключом -Xelide-below 2001 или -Xelide-below MAXIMUM. Имейте в виду, что по умолчанию проверки *не* запрещены. Это давно ожидаемое усовершенствование проверок в Java.

> **Внимание.** Вызовы игнорируемых методов заменяются объектами Unit. Если программа использует значение, возвращаемое игнорируемым методом, будет возбуждено исключение ClassCastException. Поэтому аннотацию @elidable лучше применять к методам, не возвращающим значений.

15.6.4. Специализация простых типов

Операция преобразования простых типов в объекты-обертки и обратно весьма неэффективна, но в обычном программном коде это случается достаточно часто. Например, взгляните на следующий метод:

```
def allDifferent[T](x: T, y: T, z: T) = x != y && x != z && y != z
```

Если выполнить вызов allDifferent(3, 4, 5), перед вызовом самого метода каждое целое число будет преобразовано в объект-обертку java.lang.Integer. Конечно, можно реализовать перегруженную версию:

```
def allDifferent(x: Int, y: Int, z: Int) = ...
```

а также еще семь версий для других простых типов.

Однако эти же методы можно сгенерировать автоматически, пометив типы параметров аннотацией @specialized:

```
def allDifferent[@specialized T](x: T, y: T, z: T) = ...
```

Специализацию можно ограничить подмножеством типов:

```
def allDifferent[@specialized(Long, Double) T](x: T, y: T, z: T) = ...
```

В аннотации можно указывать любые подмножества из типов Unit, Boolean, Byte, Short, Char, Int, Long, Float, Double.

15.7. Аннотации ошибок и предупреждений

Если пометить программный элемент аннотацией `@deprecated`, компилятор будет генерировать предупреждение при его использовании. Аннотация имеет два необязательных аргумента, `message` и `since`.

```
@deprecated(message = "Use factorial(n: BigInt) instead")
def factorial(n: Int): Int = ...
```

Аннотация `@deprecatedName` применяется к параметрам и определяет прежнее имя параметра.

```
def draw(@deprecatedName('sz) size: Int, style: Int = NORMAL)
```

Вы все еще можете оформлять вызовы как `draw(sz = 12)`, но это будет приводить к выводу предупреждений во время компиляции.

> **Примечание.** Аргумент аннотации – символ – имя, предваряемое апострофом. Все символы уникальны, даже с одинаковыми именами. Символы несколько эффективнее строк. Их метод `==` сравнивает ссылки, тогда как метод `==` строк сравнивает их содержимое. Но гораздо важнее семантическая разница между ними: символ определяет имя некоторого элемента программы.

Когда неявный параметр недоступен, аннотация `@implicitNotFound` генерирует осмысленное сообщение об ошибке. Подробности см. в главе 21.

Аннотация `@unchecked` подавляет вывод предупреждений, когда в программе встречается неполное выражение `match`. Например, допустим, нам известно, что передаваемый список никогда не будет пустым:

```
(lst: @unchecked) match {
    case head :: tail => ...
}
```

Компилятор не будет предупреждать об отсутствии варианта сопоставления с `Nil`. Конечно, если `lst` окажется пустым списком `Nil`, во время выполнения будет возбуждено исключение.

Аннотация `@uncheckedVariance` подавляет вывод сообщений о несоответствии. Например, эту аннотацию имеет смысл применять к реализациям интерфейса `java.util.Comparator`, чтобы обеспечить их

контравариантность (contravariant). Например, если `Student` – подтип `Person`, везде, где требуется `Comparable[Student]`, можно было бы использовать `Comparator[Person]`. Однако обобщенные типы (generics) в Java не поддерживают такую возможность. Исправить этот недостаток можно с помощью аннотации `@uncheckedVariance`:

```
trait Comparable[-T] extends
    java.lang.Comparable[T @uncheckedVariance]
```

Упражнения

1. Напишите тесты, использующие аннотацию `@Test` из фреймворка JUnit со всеми аргументами и без них. Выполните тестирование под управлением JUnit.
2. Напишите класс, демонстрирующий все возможные способы размещения аннотаций. В качестве образцовой используйте аннотацию `@deprecated`.
3. Какие аннотации в библиотеке Scala используют какую-либо из мета-аннотаций: `@param`, `@field`, `@getter`, `@setter`, `@beanGetter` и `@beanSetter`?
4. Напишите метод `sum` с переменным числом целочисленных аргументов, возвращающий сумму своих аргументов. вызовите его из Java.
5. Напишите метод, возвращающий строковое значение с содержимым текстового файла. Вызовите его из Java.
6. Реализуйте объект с volatile-полем типа `Boolean`. Приостановите выполнение одного потока на некоторое время, затем присвойте этому полю значение `true` в этом же потоке, выведите сообщение и завершите работу потока. Другой поток, выполняющийся параллельно, должен проверять значение этого поля, и если оно имеет значение `true` – выводить сообщение и завершаться. В противном случае он должен приостанавливаться на короткое время и повторять попытку. Что случится, если поле не будет объявлено как `volatile`?
7. Приведите пример, демонстрирующий, почему оптимизация хвостовой рекурсии не может быть произведена, если метод допускает возможность переопределения.
8. Добавьте метод `allDifferent` в объект, скомпилируйте и загляните в байт-код. Какие методы будут сгенерированы после применения аннотации `@specialized`?

9. Метод `Range.foreach` помечен аннотацией `@specialized(Unit)`. Почему? Загляните в байт-код, производимый командой

```
javap -classpath /path/to/scala/lib/scala-library.jar
scala.collection.immutable.Range
```

и обратите внимание на аннотации `@specialized` в `Function1`. Щелкните на ссылке `Function1.scala` в Scaladoc, чтобы увидеть их.

10. Добавьте вызов `assert(n >= 0)` в метод `factorial`. Скомпилируйте, разрешив выполнение проверок, и убедитесь, что вызов `factorial(-1)` генерирует исключение. Скомпилируйте, запретив выполнение проверок. Что изменилось? Используйте `javap` для выявления изменений в точках вызовов проверок.

Глава 16. Обработка XML

Темы, рассматриваемые в этой главе A2

- 16.1. Литералы XML.
- 16.2. Узлы XML.
- 16.3. Атрибуты элементов.
- 16.4. Встроенные выражения.
- 16.5. Выражения в атрибутах.
- 16.6. Необычные типы узлов.
- 16.7. XPath-подобные выражения.
- 16.8. Сопоставление с образцом.
- 16.9. Модификация элементов и атрибутов.
- 16.10. Трансформация XML.
- 16.11. Загрузка и сохранение.
- 16.12. Пространства имен.
- Упражнения.

В Scala имеется встроенная поддержка для работы с литерами XML, упрощающая создание фрагментов разметки XML в программах. Библиотека Scala включает функции для решения типичных задач обработки XML. В данной главе вы узнаете, как пользоваться этими функциями для чтения, анализа, создания и записи разметки XML.

Основные темы этой главы:

- литералы XML `<как>этот</как>` типа `NodeSeq`;
- возможность встраивания программного кода на Scala в литералы XML;
- свойство `child` класса `Node` возвращает дочерние узлы;
- свойство `attributes` класса `Node` возвращает объект `MetaData` с атрибутами узла;
- операторы \ и \\ реализуют возможность поиска в стиле XPath;
- имеется возможность выполнять сопоставление литералов XML с образцами узлов в предложениях `case`;
- использование экземпляров `RuleTransformer` и `RewriteRule` для трансформации вложенных узлов;
- загрузка и сохранение разметки XML выполняется объектом `XML` посредством взаимодействия с Java-методами обработки XML;

- `ConstructingParser` — альтернативный парсер, сохраняющий комментарии и разделы `CDATA`.

16.1. Литералы XML

Scala имеет встроенную поддержку XML. Вы можете определять литералы XML, просто используя код разметки XML:

```
val doc =
    <html><head><title>Fred's Memoirs</title></head><body>...</body></html>
```

В данном случае `doc` получит значение типа `scala.xml.Elem`, представляющее элемент XML.

Литерал XML также может быть последовательностью узлов. Например,

```
val items = <li>Fred</li><li>Wilma</li>
```

сохранит в `items` значение типа `scala.xml.NodeSeq`. Классы `Elem` и `NodeSeq` будут обсуждаться в следующем разделе.

> **Внимание.** Иногда компилятор по ошибке распознает литералы XML там, где их нет. Например:
>
> ```
> val (x, y) = (1, 2)
> x < y  // OK
> x <y   // Ошибка - не закрытый литерал XML
> ```
>
> В данном случае необходимо добавить пробел после <.

16.2. Узлы XML

Класс `Node` является родительским классом для всех типов узлов XML. Двумя наиболее важными его подклассами являются `Text` и `Elem`. Полная его иерархия представлена ниже, в этом же разделе.

Класс `Elem` описывает элементы XML, такие как:

```
val elem = <a href="http://scala-lang.org">The <em>Scala</em> language</a>
```

Свойство `label` возвращает имя тега (здесь, `"a"`), а свойство `child` — дочерний узел последовательности (в данном примере: два узла `Text` и узел `Elem`).

Последовательности узлов представлены типом `NodeSeq`, являющимся подтипом `Seq[Node]`, который добавляет поддержку XPath-подобных операторов (см. раздел 16.7 «XPath-подобные выражения»). К последовательностям XML допускается применять любые операции, поддерживаемые `Seq`, которые описаны в главе 13. Для обхода элементов последовательности можно использовать простой цикл `for`, например:

```
for (n <- elem.child) обработать n
```

> **Примечание.** Класс `Node` наследует `NodeSeq`. Единственный узел – это последовательность с длиной, равной 1. Это, как предполагается, должно было упростить работу с функциями, которые могут возвращать единственный узел или последовательность узлов. (В действительности же это обстоятельство не столько решает проблемы, сколько порождает их, поэтому я не рекомендую использовать его в своих программах.)

Существуют также классы узлов XML для комментариев (`<!-- ... -->`), мнемоник (`&...;`) и инструкций обработки (`<? ... ?>`). На рис. 16.1 изображены все типы узлов XML.

Если потребуется создать последовательность узлов программно, можно воспользоваться классом `NodeBuffer`, являющимся подклассом `ArrayBuffer[Node]`.

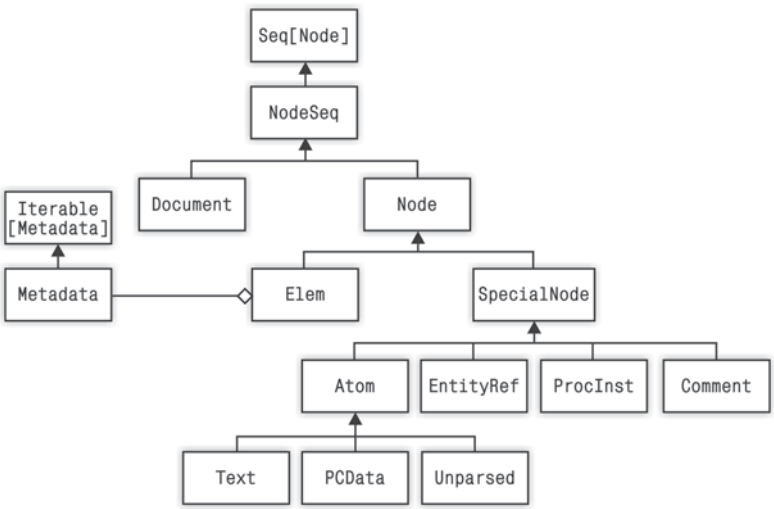


Рис. 16.1. Типы узлов XML

```
val items = new NodeBuffer
items += <li>Fred</li>
items += <li>Wilma</li>
val nodes: NodeSeq = items
```

Внимание. Значение типа `NodeBuffer` – это `Seq[Node]`. Оно может неявно преобразовываться в тип `NodeSeq`. После такого преобразования необходимо проявлять осторожность, чтобы не изменить буфер узлов, поскольку последовательность узлов XML считается неизменяемой коллекцией.

16.3. Атрибуты элементов

Для обработки ключей и значений атрибутов элемента используется свойство `attributes`. Оно возвращает объект типа `MetaData`, который почти, но не совсем, является ассоциативным массивом (`Map`) ключей атрибутов в их значения. Для доступа к значениям по ключам можно использовать оператор `()`:

```
val elem = <a href="http://scala-lang.org">The Scala language</a>
val url = elem.attributes("href")
```

К сожалению, он возвращает последовательность узлов, а не строку, потому что атрибут может содержать мнемоники. Например, взгляните на следующий пример:

```
val image = <img alt="San José State University Logo"
     src="http://www.sjsu.edu/publicaffairs/pics/sjsu_logo_color_web.jpg"/>
val alt = image.attributes("alt")
```

Здесь значением ключа `"alt"` будет последовательность, содержащая текстовый узел со строкой `"San Jos"`, значение типа `EntityRef` для `&eacute;` и еще один текстовый узел со строкой `" State University Logo"`.

Почему не выполняется разрешение мнемоник? Потому что нет никакого способа определить, что означает `&eacute;`. В XHTML эта мнемоника означает `&#233;` (код символа é), но в документах других типов она может иметь иное значение.

Совет. Если вы посчитаете такой способ работы с мнемониками в литералах XML неудобным, используйте соответствующие обозначения символов: `<img alt="San Jos&#233; State University Logo" .../>`.

Если вы уверены в отсутствии неразрешенных мнемоник в атрибутах, можно просто вызвать метод `text`, чтобы преобразовать последовательность узлов в строку:

```
val url = elem.attributes("href").text
```

Если искомый атрибут отсутствует, оператор `()` вернет `null`. Если вы предпочитаете не работать со значением `null`, используйте метод `get`, который вернет `Option[Seq[Node]]`.

К сожалению, класс `MetaData` не имеет метода `getOrElse`, но можно воспользоваться методом `getOrElse` объекта `Option`, возвращаемого методом `get`:

```
val url = elem.attributes.get("href").getOrElse(Text(""))
```

Чтобы выполнить обход всех атрибутов, используйте следующий прием:

```
for (attr <- elem.attributes)
     обработать attr.key и attr.value.text
```

С другой стороны, можно воспользоваться методом `asAttrMap`:

```
val image = <img alt="TODO" src="hamster.jpg"/>
val map = image.attributes.asAttrMap
     // Map("alt" -> "TODO", «src» -> "hamster.jpg")
```

16.4. Встроенные выражения

В литералы XML допускается включать блоки программного кода на языке Scala для динамического создания элементов. Например:

```
<ul><li>{items(0)}</li><li>{items(1)}</li></ul>
```

Здесь каждый блок будет вычислен и результат встроен в дерево XML.

Если блок возвращает последовательность узлов, узлы просто добавляются в XML. Все остальное преобразуется в `Atom[T]`, контейнер для типа `T`. Благодаря такому подходу в дереве XML можно сохранять любые значения. Извлечь значение из узла `Atom` можно с помощью свойства `data`.

Во многих случаях нет необходимости беспокоиться об извлечении элементов из атомов. При сохранении документа XML каждый атом будет преобразован в строку вызовом метода `toString` свойства `data`.

> **Внимание.** Возможно, кого-то удивит, но встроенные строки преобразуются не в узлы типа `Text`, а в узлы типа `Atom[String]`. Это не одно и то же – класс `Text` является подклассом `Atom[String]`. При сохранении документа это не имеет значения. Но если позднее вы решите выполнить сопоставление по типу в поисках узлов типа `Text`, попытка окончится неудачей. Чтобы избежать этого, вместо строк необходимо явно вставлять узлы типа `Text`:
>
> ```
> <li>{Text("Another item")}</li>
> ```

В литералы XML можно встраивать не только программный код на языке Scala, но и встраиваемый код Scala может содержать литералы XML. Например, при наличии списка элементов может появиться желание поместить каждый элемент в теги `<li>`:

```
<ul>{for (i <- items) yield <li>{i}</li>}</ul>
```

Здесь, внутри элемента `ul`, имеется блок Scala `<ul>{...}</ul>`. Этот блок возвращает последовательность выражений XML.

```
for (i <- items) yield литерал XML
```

Этот литерал XML `<li>...</li>` содержит другой блок с кодом Scala!

```
<li>{i}</li>
```

Получается код Scala внутри XML внутри кода Scala внутри XML. Мысли начинают путаться, стоит только подумать об этом. Но если взглянуть с другой стороны, получается весьма естественная конструкция: создать элемент `ul`, содержащий `li` для каждого элемента из `items`.

> **Примечание.** Чтобы поместить символ открывающей или закрывающей угловой скобки в литерал XML, используйте две скобки подряд. Следующее определение:
>
> ```
> <h1>The Natural Numbers {{1, 2, 3, ...}}</h1>
> ```
>
> воспроизведет
>
> ```
> <h1>The Natural Numbers {1, 2, 3, ...}</h1>
> ```

16.5. Выражения в атрибутах

Значения атрибутов можно вычислять с помощью выражений Scala, например:

```
<img src={makeURL(fileName)}/>
```

Здесь функция `makeURL` возвращает строку, которая становится значением атрибута.

> **Внимание.** Фигурные скобки внутри строк в кавычках не вычисляются. Например,
>
> ```
> <img src="{makeURL(fileName)}"/>
> ```
>
> определит значение атрибута `src` как строку `"{makeURL(fileName)}"`, что, вероятно, не совсем то, что предполагалось.

Встраиваемые блоки могут также порождать последовательности узлов. Это может пригодиться, если потребуется включить в атрибут мнемоники или атомы:

```
<a id={new Atom(1)} ... />
```

Если встроенный блок вернет `null` или `None`, атрибут не будет установлен. Например:

```
<img alt={if (description == "TODO") null else description} ... />
```

Если переменная `description` будет имеет значение `"TODO"` или `null`, элемент не получит атрибута `alt`.

Того же эффекта можно добиться с помощью `Option[Seq[Node]]`. Например:

```
<img alt={if (description=="TODO") None else Some(Text(description))}... />
```

> **Внимание.** Считается синтаксической ошибкой, если блок возвращает нечто иное, кроме `String`, `Seq[Node]` или `Option[Seq[Node]]`. Здесь наблюдается непоследовательность, в сравнении с блоками внутри элементов, где результат преобразуется в объект типа `Atom`. Если вам потребуется определить значение атрибута как атом, его придется создать вручную.

16.6. Необычные типы узлов

Иногда бывает необходимо включить в документ XML текст, не являющийся разметкой XML. Типичным примером может служить программный код на JavaScript в странице XHTML. Для этого можно воспользоваться разметкой CDATA:

```
val js = <script><![CDATA[if (temp < 0) alert("Cold!")]]></script>
```

Однако синтаксический анализатор не замечает, что текст помещен в раздел CDATA. В результате получается узел с дочерним узлом Text. Если на выходе должен получиться раздел CDATA, включите узел PCData, как показано ниже:

```
val code = """if (temp < 0) alert("Cold!")"""
val js = <script>{PCData(code)}</script>
```

В узел Unparsed можно включить любой текст. Он будет сохраняться в исходном состоянии. Такие узлы можно создавать программно или определять в виде литералов:

```
val n1 = <xml:unparsed><&></xml:unparsed>
val n2 = Unparsed("<&>")
```

Однако я не рекомендую пользоваться этой возможностью, так как в результате легко можно создать некорректную разметку XML.

Наконец, последовательность узлов можно сгруппировать в один «групповой» узел.

```
val g1 = <xml:group><li>Item 1</li><li>Item 2</li></xml:group>
val g2 = Group(Seq(<li>Item 1</li>, <li>Item 2</li>))
```

Групповые узлы «разгруппируются» при выполнении итераций по ним. Сравните:

```
val items = <li>Item 1</li><li>Item 2</li>
for (n <- <xml:group>{items}</xml:group>) yield n
     // Вернет два элемента li

for (n <- <ol>{items}</ol>) yield n
     // Вернет один элемент ol
```

16.7. XPath-подобные выражения

Класс `NodeSeq` предоставляет методы, напоминающие операторы / и // в XPath (XML Path Language – язык определения путей в XML, www.w3.org/TR/xpath). Поскольку комбинация // обозначает комментарий и потому не может играть роль оператора, в Scala используются операторы \ и \\.

Оператор \ отыскивает непосредственных потомков узла или последовательности узлов. Например, выражение

```
val list =
    <dl><dt>Java</dt><dd>Gosling</dd><dt>Scala</dt><dd>Odersky</dd></dl>
val languages = list \ "dt"
```

присвоит переменной `languages` последовательность узлов, содержащую `<dt>Java</dt>` и `<dt>Scala</dt>`.

Подстановочный символ соответствует любому элементу. Например, выражение

```
doc \ "body" \ "_" \ "li"
```

отыщет все элементы `li`, содержащиеся в `ul`, в `ol` или в любом другом элементе, содержащемся в элементе `body`.

Оператор \\ позволяет находить потомков на более глубоких уровнях вложенности. Например,

```
doc \\ "img"
```

отыщет все элементы `img` в любом месте внутри `doc`.

Строка, начинающаяся с `@`, означает поиск атрибута. Например,

```
img \ "@alt"
```

вернет значение атрибута `alt` указанного узла, а

```
doc \\ "@alt"
```

отыщет все атрибуты `alt` во всех элементах внутри `doc`.

> **Примечание.** В поиске атрибутов нельзя использовать подстановочный символ; выражение `img \ "@_"` не вернет всех атрибутов.

> **Внимание.** В отличие от XPath, в Scala нельзя использовать одиночный обратный слеш \ для извлечения атрибутов из нескольких узлов. Например, выражение `doc \\ "img" \ "@src"` не будет работать, если документ содержит более одного элемента `img`. Используйте `doc \\ "img" \\ "@src"`.

Результатом операторов \ и \\ является последовательность узлов. Она может содержать единственный узел, но если точно не известно, лучше выполнить обход элементов последовательности. Например:

```
for (n <- doc \\ "img") обработать n
```

Если вызвать метод `text` результата, возвращаемого операторами \ и \\, содержимое всех текстовых узлов будет объединено в одну строку. Например,

```
(<img src="hamster.jpg"/><img src="frog.jpg"/> \\ "@src").text
```

вернет строку `"hamster.jpgfrog.jpg"`.

16.8. Сопоставление с образцом

Литералы XML можно использовать в выражениях сопоставления с образцом. Например:

```
node match {
    case <img/> => ...
    ...
}
```

Первое сопоставление будет успешным, если `node` будет представлять элемент `img` с любыми атрибутами и без вложенных элементов.

Обработка дочерних элементов требует некоторых ухищрений. Совпадение с единственным дочерним элементом можно реализовать так:

```
case <li>{_}</li> => ...
```

Однако, если дочерних элементов окажется больше, например `<li>An <em>important</em> item</li>`, сопоставление с этим образцом потерпит неудачу. Чтобы обеспечить совпадение с любым количеством дочерних элементов, используйте:

```
case <li>{_*}</li> => ...
```

Обратите внимание на фигурные скобки – они напоминают форму записи внедрения программного кода в литералы XML. Однако внутри образцов XML фигурные скобки определяют образцы кода, а не выполняемый код.

Вместо подстановочных символов можно использовать имена переменных. В этом случае совпадение будет присвоено переменной.

```
case <li>{child}</li> => child.text
```

Для сопоставления с текстовым узлом используйте сопоставление с case-классом, как показано ниже:

```
case <li>{Text(item)}</li> => item
```

Присвоить последовательность узлов переменной можно следующим образом:

```
case <li>{children @ _*}</li> => for (c <- children) yield c
```

> **Внимание.** В таких операциях сопоставления `children` имеет тип `Seq[Node]`, а не `NodeSeq`.

В предложении `case` можно использовать только один узел. Например, следующий код недопустим:

```
case <p>{_*}</p><br/> => ... // Недопустимо
```

Образцы XML не могут включать атрибуты.

```
case <img alt="TODO"/> => ... // Недопустимо
```

Чтобы реализовать сопоставление с атрибутами, используйте ограничитель:

```
case n @ <img/> if (n.attributes("alt").text == "TODO") => ...
```

16.9. Модификация элементов и атрибутов

Узлы и последовательности узлов в Scala являются неизменяемыми. Если потребуется изменить узел, необходимо создать его копию, произведя все необходимые изменения в копии.

Для копирования узлов `Elem` используйте метод `copy`. Он имеет пять именованных параметров: уже знакомые вам `label`, `attributes` и `child`, а также `prefix` и `scope`, которые используются для определения пространства имен (см. раздел 16.12 «Пространства имен»). Любые неуказанные параметры копируются из исходного элемента. Например,

```
val list = <ul><li>Fred</li><li>Wilma</li></ul>
val list2 = list.copy(label = "ol")
```

создаст копию списка, заменив имя `ul` на `ol`. Дочерние узлы будут общими, но в этом нет ничего особенного, потому что последовательности узлов являются неизменяемыми.

Чтобы добавить дочерний узел, вызовите метод `copy`, как показано ниже:

```
list.copy(child = list.child ++ <li>Another item</li>)
```

Добавить или изменить атрибут можно с помощью оператора `%`:

```
val image = <img src="hamster.jpg"/>
val image2 = image % Attribute(null, "alt", "An image of a hamster", Null)
```

Первый аргумент – пространство имен. Последний – список дополнительных метаданных. Так же как `Node` наследует `NodeSeq`, трейт `Attribute` наследует `MetaData`. Чтобы добавить несколько атрибутов, можно сконструировать такую цепочку:

```
val image3 = image % Attribute(null, "alt", "An image of a frog",
     Attribute(null, "src", "frog.jpg", Null))
```

> **Внимание.** Здесь `scala.xml.Null` – это пустой список атрибутов, а не тип `scala.Null`.

Добавление атрибута с тем же ключом заменит существующий атрибут. Элемент `image3` имеет единственный атрибут с ключом `"src"`; его значением является строка `"frog.jpg"`.

16.10. Трансформация XML

Иногда бывает необходимо изменить все вложенные элементы, соответствующие определенному образцу. Библиотека XML предоставляет класс `RuleTransformer`, применяющий один или более экземпляров правил `RewriteRule` к узлу и к вложенным в него элементам.

Например, представьте, что потребовалось заменить все узлы `ul` в документе на `ol`. Ниже приводится объявление экземпляра правила `RewriteRule`, переопределяющее метод `transform`:

```
val rule1 = new RewriteRule {
     override def transform(n: Node) = n match {
          case e @ <ul>{_*}</ul> => e.asInstanceOf[Elem].copy(label = "ol")
          case _ => n
     }
}
```

Теперь это правило можно использовать для преобразования дерева узлов командой:

```
val transformed = new RuleTransformer(rule1).transform(root)
```

Конструктору `RuleTransformer` можно передать любое количество правил:

```
val transformer = new RuleTransformer(rule1, rule2, rule3);
```

Метод `transform` выполняет обход вложенных узлов, применяет все правила и возвращает преобразованное дерево.

16.11. Загрузка и сохранение

Загрузить документ XML из файла можно вызовом метода `loadFile` объекта `XML`:

```
import scala.xml.XML
val root = XML.loadFile("myfile.xml")
```

Загрузить можно также из `java.io.InputStream`, `java.io.Reader` или из URL:

```
val root2 = XML.load(new FileInputStream("myfile.xml"))
val root3 = XML.load(new InputStreamReader(
     new FileInputStream("myfile.xml"), "UTF-8"))
val root4 = XML.load(new URL("http://horstmann.com/index.html"))
```

Загрузка документа выполняется с применением парсера SAX из библиотеки Java. К сожалению, определение типа документа (Document Type Definition, DTD) остается недоступным.

> **Внимание.** Этот парсер страдает от проблемы, унаследованной из библиотеки Java. Он не читает определения DTD из локального каталога. В частности, извлечение файла XHTML может занять длительное время или вообще потерпеть неудачу, при попытке парсера получить DTD с сайта w3c.org.
>
> Чтобы задействовать локальный каталог, необходимо использовать класс `CatalogResolver` из пакета `com.sun.org.apache.xml.internal.resolver.tools` в JDK, или, если вы предпочитаете не пользоваться классами, не входящими в официальный API, из проекта Apache Commons Resolver (http://xml.apache.org/commons/components/resolver/resolver-article.html).
>
> К сожалению, объект `XML` не имеет API для установки механизма разрешения мнемоник. Ниже показано, как можно сделать это, зайдя с черного хода:
>
> ```
> val res = new CatalogResolver
> val doc = new factory.XMLLoader[Elem] {
>      override def adapter = new parsing.NoBindingFactoryAdapter() {
>           override def resolveEntity(publicId: String, systemId: String) = {
>                res.resolveEntity(publicId, systemId)
>           }
>      }
> }.load(new URL("http://horstmann.com/index.html"))
> ```

Существует еще один парсер, сохраняющий комментарии, разделы `CDATA` и, при необходимости, пробельные символы:

```
import scala.xml.parsing.ConstructingParser
import java.io.File
val parser =
     ConstructingParser.fromFile(new File("myfile.xml"), preserveWS = true)
val doc = parser.document
val root = doc.docElem
```

Обратите внимание, что `ConstructingParser` возвращает узел типа `Document`. Его метод `docElem` возвращает корень документа.

Если документ имеет определение DTD и оно может вам понадобиться (например, при сохранении документа), вы сможете получить его, обратившись к `doc.dtd`.

Внимание. По умолчанию `ConstructingParser` не производит разрешение мнемоник, но преобразует их в комментарии, такие как:

```
<!-- unknown entity nbsp; -->
```

Для чтения файла XHTML можно воспользоваться подклассом `XhtmlParser`:

```
val parser = new XhtmlParser(scala.io.Source.fromFile("myfile.html"))
val doc = parser.initialize.document
```

Иначе необходимо добавить мнемоники в карту мнемоник парсера. Например:

```
parser.ent ++= List(
     "nbsp" -> ParsedEntityDecl("nbsp", IntDef("\u00A0")),
     "eacute" -> ParsedEntityDecl("eacute", IntDef("\u00E9")))
```

Сохранить документ XML в файл можно с помощью метода:

```
XML.save("myfile.xml", root)
```

Этот метод принимает три необязательных параметра:

- `enc` определяет кодировку символов (по умолчанию `"ISO-8859-1"`);
- `xmlDecl` определяет необходимость вывода объявления XML (`<?xml...?>`) в начале (по умолчанию `false`);
- `doctype` — объект case-класса `scala.xml.dtd.DocType` (по умолчанию `null`).

Например, записать файл XHTML можно следующим образом:

```
XML.save("myfile.xhtml", root,
enc = "UTF-8",
xmlDecl = true,
doctype = DocType("html",
     PublicID("-//W3C//DTD XHTML 1.0 Strict//EN",
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"),
     Nil))
```

В последнем параметре конструктора `DocType` можно указать внутренние объявления DTD – малопонятная особенность XML, которую я не буду обсуждать здесь.

Сохранение можно выполнить также в `java.io.Writer`, но тогда придется указать все параметры.

```
XML.save(writer, root, "UTF-8", false, null)
```

Примечание. При сохранении XML-файла элементы без содержимого не записываются как самозакрывающиеся теги. Например:

```
<img src="hamster.jpg"></img>
```

Если вы предпочитаете:

```
<img src="hamster.jpg"/>
```

используйте

```
val str = xml.Utility.toXML(node, minimizeTags = true)
```

Совет. Если потребуется добавить отступы в XML-код, используйте класс `PrettyPrinter`:

```
val printer = new PrettyPrinter(width = 100, step = 4)
val str = printer.formatNodes(nodeSeq)
```

16.12. Пространства имен

Пространства имен в XML используются, чтобы избежать конфликтов имен, подобно пакетам в Java или Scala. Однако пространством имен в XML являются URI (и обычно URL), такие как

```
http://www.w3.org/1999/xhtml
```

Пространство имен объявляется в атрибуте `xmlns`, например:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>...</head>
    <body>...</body>
</html>
```

Элемент `html` и его дочерние элементы (`head`, `body` и другие) помещаются в это пространство имен.

Дочерний элемент может определять собственное пространство имен, например:

```
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="100">
     <rect x="25" y="25" width="50" height="50" fill="#ff0000"/>
</svg>
```

В Scala каждый элемент имеет свойство `scope` типа `NamespaceBinding`. Свойство `uri` этого класса возвращает URI пространства имен.

При необходимости смешивать элементы из нескольких пространств имен может оказаться утомительным использовать адреса URL этих пространств имен. Чтобы избежать этого, можно использовать *префиксы пространств имен*. Например, тег

```
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:svg="http://www.w3.org/2000/svg">
```

объявляет префикс `svg` для пространства имен `http://www.w3.org/2000/svg`. Все элементы с префиксом `svg:` принадлежат этому пространству имен. Например:

```
<svg:svg width="100" height="100">
     <svg:rect x="25" y="25" width="50" height="50" fill="#ff0000"/>
</svg:svg>
```

В разделе 16.9 «Модификация элементов и атрибутов» говорилось, что каждый объект `Elem` имеет значения `prefix` и `scope`. Парсер автоматически вычисляет эти значения. Определить пространство имен элемента можно с помощью значения `scope.uri`. Однако, когда элементы XML создаются программно, вам потребуется устанавливать значения `prefix` и `scope` вручную. Например:

```
val scope = new NamespaceBinding("svg", "http://www.w3.org/2000/svg",
                                 TopScope)
val attrs = Attribute(null, "width", "100",
     Attribute(null, "height", "100", Null))
val elem = Elem(null, "body", Null, TopScope,
     Elem("svg", "svg", attrs, scope))
```

Упражнения

1. Что означает `<fred/>(0)`? `<fred/>(0)(0)`? Почему?
2. Каков будет результат следующего выражения:

```
<ul>
     <li>Opening bracket: [</li>
     <li>Closing bracket: ]</li>
     <li>Opening brace: {</li>
     <li>Closing brace: }</li>
</ul>
```

Как его исправить?

3. Сравните:

```
<li>Fred</li> match { case <li>{Text(t)}</li> => t }
```

и

```
<li>{"Fred"}</li> match { case <li>{Text(t)}</li> => t }
```

Почему они действуют по-разному?

4. Прочитайте файл XHTML и выведите все элементы `img`, не имеющие атрибута `alt`.
5. Выведите имена всех изображений в файле XHTML. То есть выведите значения атрибутов `src` всех элементов `img`.
6. Прочитайте файл XHTML и выведите таблицу всех гиперссылок в файле вместе с их адресами URL. То есть выведите содержимое дочернего текстового узла и значение атрибута `href` каждого элемента `a`.
7. Напишите функцию, принимающую параметр типа `Map[String, String]` и возвращающую элемент `dl` с элементом `dt` – для каждого ключа и `dd` – для каждого значения. Например, для

```
Map("A" -> "1", "B" -> "2")
```

функция должна вернуть

```
<dl><dt>A</dt><dd>1</dd><dt>B</dt><dd>2</dd></dl>
```

8. Напишите функцию, принимающую элемент `dl` и превращающую его в `Map[String, String]`. Эта функция является полной противоположностью функции из предыдущего упражнения, при условии, что все дочерние элементы `dt` уникальны.
9. Трансформируйте документ XHTML, добавив атрибут `alt="TODO"` во все элементы `img` без атрибута `alt`, сохранив имеющиеся.
10. Напишите функцию, которая будет читать документы XHTML, применять трансформацию, реализованную в предыдущем упражнении, и сохранять результат. Особое внимание уделите сохранности DTD и всех разделов `CDATA`.

Глава 17. Параметризованные типы

Темы, рассматриваемые в этой главе L2

- 17.1. Обобщенные классы.
- 17.2. Обобщенные функции.
- 17.3. Границы изменения типов.
- 17.4. Границы представления.
- 17.5. Границы контекста.
- 17.6. Границы контекста Manifest.
- 17.7. Множественные границы.
- 17.8. Ограничение типов L3.
- 17.9. Вариантность.
- 17.10. Ко- и контравариантные позиции.
- 17.11. Объекты не могут быть обобщенными.
- 17.12. Подстановочный символ.
- Упражнения.

В Scala поддерживается возможность использовать параметры типов для реализации классов и функций, работающих со значениями разных типов. Например, `Array[T]` может хранить элементы произвольного типа `T`. Сама идея очень проста, но понимание деталей ее реализации может вызывать сложности. Иногда бывает необходимо наложить ограничения на тип. Например, чтобы иметь возможность сортировать элементы, тип `T` должен поддерживать такое понятие, как порядок следования. Кроме того, что происходит с параметризованным типом при изменении типа параметра? Например, можно ли передать `Array[String]` в функцию, ожидающую получить `Array[Any]`? В Scala можно определить, как должны изменяться типы в зависимости от своих параметров.

Основные темы этой главы:

- классы, трейты, методы и функции могут иметь параметры типов;
- параметр типа помещается после имени и заключается в квадратные скобки;

- границы типов определяются в виде `T <: UpperBound`, `T >: LowerBound`, `T <% ViewBound`, `T : ContextBound`;
- имеется возможность ограничивать методы с помощью ограничителя типа, такого как `(implicit ev: T <:< UpperBound)`;
- используйте `+T` (ковариантность), чтобы указать, что отношение родства подтипа обобщенного типа следует в одном направлении с `T`, или `-T` (контравариантность), чтобы указать, что отношение родства следует в обратном направлении;
- ковариантность применяется к параметрам, обозначающим выходные значения, такие как элементы в неизменяемых коллекциях;
- контравариантность применяется к параметрам, обозначающим входные значения, такие как аргументы функций.

17.1. Обобщенные классы

Так же как в Java или C++, классы и трейты могут иметь параметризованные типы. Для определения параметра типа в Scala используются квадратные скобки, например:

```
class Pair[T, S](val first: T, val second: S)
```

Эта строка определяет класс с двумя параметрами типов, `T` и `S`. Параметры типов используются в определении класса для обозначения типов переменных, параметров методов и возвращаемых значений.

Класс с одним или более параметрами типов называется *обобщенным* (generic). Если на место параметров типов подставить фактические типы, вы получите обычный класс, такой как `Pair[Int, String]`.

Самое приятное, что компилятор Scala пытается вывести фактические типы из параметров на этапе конструирования объекта:

```
val p = new Pair(42, "String") // Это - Pair[Int, String]
```

Однако типы можно указать явно:

```
val p2 = new Pair[Any, Any](42, "String")
```

17.2. Обобщенные функции

Функции и методы также могут принимать параметры типов. Например:

```
def getMiddle[T](a: Array[T]) = a(a.length / 2)
```

Как и в случае с обобщенными классами, параметр типа помещается после имени.

Компилятор Scala выводит фактические типы из аргументов в вызове.

```
getMiddle(Array("Mary", "had", "a", "little", "lamb"))
    // Вызовет getMiddle[String]
```

При необходимости тип можно указать явно:

```
val f = getMiddle[String] _ // Функция будет сохранена в f
```

17.3. Границы изменения типов

Иногда бывает необходимо ограничить диапазон изменения типов. Допустим, имеется тип `Pair`, где оба компонента имеют один и тот же тип, как показано ниже:

```
class Pair[T](val first: T, val second: T)
```

Теперь необходимо добавить метод, выбирающий наименьшее значение:

```
class Pair[T](val first: T, val second: T) {
    def smaller = if (first.compareTo(second) < 0) first else second
        // Ошибка
}
```

Это неправильно – нам неизвестно, имеется ли метод `compareTo` у аргумента `first`. Чтобы решить эту проблему, можно добавить в определении класса *верхнюю границу* (upper bound) для T: `T <: Comparable[T]`.

```
class Pair[T <: Comparable[T]](val first: T, val second: T) {
    def smaller = if (first.compareTo(second) < 0) first else second
}
```

Это означает, что тип `T` должен быть подтипом `Comparable[T]`.

Теперь можно создать экземпляр `Pair[java.lang.String]`, но нельзя создать экземпляр `Pair[java.io.File]`, потому что `String` являет-

ся подтипом `Comparable[String]`, а тип `File` не реализует интерфейс `Comparable[File]`. Например:

```
val p = new Pair("Fred", "Brooks")
println(p.smaller) // Выведет Brooks
```

> **Внимание.** Этот пример несколько упрощен. Если попытаться создать экземпляр `new Pair(4, 2)`, компилятор сообщит, что для `T = Int` не выполняется условие `T <: Comparable[T]`. Решение этой проблемы приводится в разделе 17.4 «Границы представлений».

Существует также возможность определить нижнюю границу типа. Например, допустим, что необходимо определить метод, замещающий первый компонент пары другим значением. Наши пары – неизменяемые объекты, поэтому метод должен возвращать новую пару. Ниже демонстрируется первая попытка:

```
class Pair[T](val first: T, val second: T) {
     def replaceFirst(newFirst: T) = new Pair[T](newFirst, second)
}
```

Однако есть лучшее решение, чем это. Допустим, что имеется экземпляр `Pair[Student]`. Теоретически должно быть возможно заменить первый компонент экземпляром `Person`. Разумеется, в результате должен получиться экземпляр `Pair[Person]`. Вообще, замещающий тип должен быть супертипом для типа компонента пары.

```
def replaceFirst[R >: T](newFirst: R) = new Pair[R](newFirst, second)
```

Здесь я включил в возвращаемую пару параметр типа для большей ясности. Это определение можно было бы записать так:

```
def replaceFirst[R >: T](newFirst: R) = new Pair(newFirst, second)
```

В этом случае тип возвращаемого значения будет определен как `Pair[R]`.

> **Внимание.** Если опустить верхнюю границу
>
> `def replaceFirst[R](newFirst: R) = new Pair(newFirst, second)`
>
> метод будет компилироваться, но он будет возвращать значение типа `Pair[Any]`.

17.4. Границы представления

В предыдущем разделе был представлен пример верхней границы:

```
class Pair[T <: Comparable[T]]
```

К сожалению, если попытаться выполнить `new Pair(4, 2)`, компилятор сообщит, что `Int` не является подтипом `Comparable[Int]`. В отличие от типа-обертки `java.lang.Integer`, тип `Int` в языке Scala не реализует интерфейс `Comparable`. Однако тип `RichInt` реализует `Comparable[Int]`, что приводит к *неявному преобразованию* значения типа `Int` в значение типа `RichInt`. (Подробнее о неявных преобразованиях рассказывается в главе 21.)

Решить проблему можно с помощью «границ представления» (view bound), например:

```
class Pair[T <% Comparable[T]]
```

Отношение `<%` означает, что тип `T` может быть преобразован в тип `Comparable[T]` посредством неявного преобразования.

> **Примечание.** Правильнее использовать трейт `Ordered`, добавляющий операторы отношений к интерфейсу `Comparable`:
>
> ```
> class Pair[T <% Ordered[T]](val first: T, val second: T) {
>     def smaller = if (first < second) first else second
> }
> ```
>
> Я не использовал эту возможность в предыдущем разделе, потому что `java.lang.String` реализует интерфейс `Comparable[String]`, но не реализует `Ordered[String]`. При использовании границ представления это не проблема, потому что строки неявно будут преобразовываться в значения типа `RichString`, который является подтипом `Ordered[String]`.

17.5. Границы контекста

Чтобы иметь возможность определять границы представления `T <% V`, должно существовать неявное преобразование из `T` в `V`. *Границы контекста* (context bounds) определяются как `T : M`, где `M` – это другой обобщенный тип. Для этого требуется наличие «неявного значения» (implicit value) типа `T[M]`. Подробнее о неявных значениях мы поговорим в главе 21. Например,

```
class Pair[T : Ordering]
```

требует существования неявного значения типа `Ordering[T]`. Это неявное значение затем сможет использоваться в методах класса. Объявляя метод, использующий неявное значение, необходимо добавить «неявный параметр» (implicit parameter). Например:

```
class Pair[T : Ordering](val first: T, val second: T) {
     def smaller(implicit ord: Ordering[T]) =
          if (ord.compare(first, second) < 0) first else second
}
```

Как будет показано в главе 21, неявные значения обладают большей гибкостью, чем неявные преобразования.

17.6. Границы контекста Manifest

Чтобы создать экземпляр обобщенного класса `Array[T]`, необходим объект `Manifest[T]`[1]. Это требуется для корректной работы массивов простых типов. Например, если `T` – это тип `Int`, в виртуальной машине должен быть создан массив `int[]`. Класс `Array` в Scala – это библиотечный класс, не имеющий специального значения для компилятора. Если вы пишете обобщенную функцию, создающую обобщенный массив, вам нужно помочь ей и передать этот объект манифеста. Поскольку это неявный параметр конструктора, можно использовать границы контекста, как показано ниже:

```
def makePair[T : Manifest](first: T, second: T) = {
     val r = new Array[T](2); r(0) = first; r(1) = second; r
}
```

Если в программе вызвать `makePair(4, 9)`, компилятор отыщет неявный объект `Manifest[Int]` и фактически вызовет `makePair(4, 9)(intManifest)`. Затем метод вызовет конструктор `new Array(2)(intManifest)`, который вернет простой массив `int[2]`.

К чему все эти сложности? В виртуальной машине обобщенные типы исчезают. Остается только единственный метод `makePair`, который должен работать со всеми типами `T`.

17.7. Множественные границы

Переменная типа может иметь обе границы, верхнюю и нижнюю. Определение обеих границ имеет следующий синтаксис:

[1] В Scala 2.10 Manifest помечен как устаревший, вместо него рекомендуется использовать класс TypeTag предоставляющий более богатый API. – *Прим. ред.*

```
T <: Upper >: Lower
```

Нельзя определить несколько верхних или нижних границ. Однако можно потребовать, чтобы тип реализовал множество трейтов, например:

```
T <: Comparable[T] with Serializable with Cloneable
```

Допускается указывать несколько границ представления:

```
T <% Comparable[T] <% String
```

Также можно указать несколько границ контекста:

```
T : Ordering : Manifest
```

17.8. Ограничение типов L3

Механизм ограничения типов (type constraints) является еще одним способом сузить круг допустимых типов. Ниже представлены три оператора отношений, используемых для этой цели:

```
T =:= U
T <:< U
T <%< U
```

Эти операторы проверяют, является ли тип `T` равным типу `U`, подтипом `U` или может неявно преобразовываться в тип `U`. Чтобы иметь возможность пользоваться такими ограничениями, следует добавить «неявный параметр подтвержения» (implicit evidence parameter), как показано ниже:

```
class Pair[T](val first: T, val second: T)(implicit ev: T <:< Comparable[T])
```

> **Примечание.** Данные операторы не являются встроенными конструкциями языка. Они реализованы в библиотеке Scala. Необычный синтаксис и особенности работы механизма ограничения типа разъясняются в главе 21.

В примере выше применение механизма ограничения типа не дает никаких преимуществ перед использованием границы класса `Pair[T`

<: Comparable[T]]. Однако ограничение типа может пригодиться в некоторых особых случаях. В этом разделе будут показаны два примера использования механизма ограничения типа.

Ограничение типа дает возможность определить в обобщенном классе метод, который может использоваться только при определенных условиях. Например, такое определение класса:

```
class Pair[T](val first: T, val second: T) {
     def smaller(implicit ev: T <:< Ordered[T]) =
          if (first < second) first else second
}
```

позволяет создать экземпляр типа Pair[File], даже при том, что объекты класса File не являются порядковыми значениями. Ошибка компиляции будет генерироваться только при попытке вызвать метод smaller.

Еще одним примером может служить метод orNull класса Option:

```
val friends = Map("Fred" -> "Barney", ...)
val friendOpt = friends.get("Wilma") // Экземпляр Option[String]
val friendOrNull = friendOpt.orNull  // Строка String или null
```

Метод orNull удобно использовать при работе с программным кодом Java, где принято определять отсутствующие значения как null. Но он не может применяться к значениям типов, таких как Int, для которых null не является допустимым значением. Так как метод orNull реализован с использованием ограничения Null <:< A, вы можете создавать экземпляры Option[Int], при условии, что будете держаться подальше от метода orNull этих экземпляров.

Ограничение типа может также применяться с целью помочь механизму определения типа. Взгляните:

```
def firstLast[A, C <: Iterable[A]](it: C) = (it.head, it.last)
```

При вызове

```
firstLast(List(1, 2, 3))
```

вы получите сообщение, что выведенные типы аргументов [Nothing, List[Int]] не соответствуют сигнатуре [A, C <: Iterable[A]]. Откуда взялся тип Nothing? Механизм определения типа не смог сообразить, что тип A выводится из List(1, 2, 3), потому что он выполняет со-

поставление типов A и C в единственном шаге. Чтобы помочь ему, следует обеспечить сначала сопоставление типа C, а затем A:

```
def firstLast[A, C](it: C)(implicit ev: C <:< Iterable[A]) =
    (it.head, it.last)
```

> **Примечание.** Аналогичный прием был продемонстрирован в главе 12, где метод corresponds проверял, содержат ли две последовательности соответствующие элементы:
>
> ```
> def corresponds[B](that: Seq[B])(match: (A, B) => Boolean): Boolean
> ```
>
> Предикат match – это каррированый параметр, благодаря чему механизм определения типа сначала определяет тип аргумента B, а затем использует эту информацию для анализа аргумента match. В вызове
>
> ```
> Array("Hello", "Fred").corresponds(Array(5, 4))(_.length == _)
> ```
>
> компилятор с успехом определяет тип аргумента B как Int. После этого он оказывается в состоянии понять смысл _.length == _.

17.9. Вариантность

Представьте, что имеется функция, выполняющая некоторую операцию с экземпляром Pair[Person]:

```
def makeFriends(p: Pair[Person])
```

Если класс Student является подклассом Person, можно ли вызвать makeFriend с аргументом типа Pair[Student]? По умолчанию это считается ошибкой. Даже при том, что тип Student является подтипом Person, типы Pair[Student] и Pair[Person] *не* связаны отношениями родства.

Если необходимо, чтобы такая связь существовала, ее нужно определить в классе Pair явно:

```
class Pair[+T](val first: T, val second: T)
```

Знак + означает, что тип является *ковариантным* (covariant) к T, то есть изменяется в том же направлении. Поскольку тип Student является подтипом Person, тип Pair[Student] теперь также считается подтипом Pair[Person].

Существует возможность определить вариантность и в другом направлении. Рассмотрим обобщенный тип Friend[T], обозначающий субъекта, готового подружиться с любым субъектом типа T.

```
trait Friend[-T] {
     def befriend(someone: T)
}
```

Теперь допустим, что имеется функция

```
def makeFriendWith(s: Student, f: Friend[Student]) { f.befriend(s) }
```

Можно ли вызвать ее с аргументом типа `Friend[Person]`? То есть если имеется такой код:

```
class Person extends Friend[Person]
class Student extends Person
val susan = new Student
val fred = new Person
```

завершится ли успехом вызов `makeFriendWith(susan, fred)`? Похоже, что да. Если Фред (Fred) может подружиться с любым человеком, он, скорее всего, сможет подружиться и со Сьюзен (Susan).

Обратите внимание, что изменение типа происходит в направлении, обратном отношению с подтипом. `Student` – это подтип `Person`, а `Friend[Student]` – супертип для `Friend[Person]`. В подобных случаях параметр типа объявляется как *контравариантный* (contravariant):

```
trait Friend[-T] {
     def befriend(someone: T)
}
```

Допускается определять обе вариантности для единственного обобщенного типа. Например, функции с единственным аргументом имеют тип `Function1[-A, +R]`. Чтобы убедиться, что они обладают соответствующими вариантностями, рассмотрим функцию:

```
def friends(students: Array[Student], find: Function1[Student, Person]) =
     // Второй параметр можно записать как find: Person => Person
     for (s <- students) yield find(s)
```

Допустим, что имеется функция

```
def findStudent(p: Person) : Student
```

Можно ли передать ее в вызов функции `friends`? Конечно. Она примет любого субъекта, сделает из него студента (`Student`) и вернет результат типа `Student`, который затем можно поместить в массив `Array[Person]`.

17.10. Ко- и контравариантные позиции

В предыдущем разделе было показано, что функции являются контравариантными (contravariant) к своим аргументам и ковариантными (covariant) к результатам. В общем случае контравариантность имеет смысл использовать для входных значений, получаемых объектами, и ковариантность – для выходных значений.

Если объект допускает и то, и другое, тип следует оставить *инвариантным* (invariant). В общем случае инвариантность свойственна изменяемым структурам данных. Например, массивы в Scala являются инвариантными. Нельзя преобразовать тип `Array[Student]` в `Array[Person]` или обратно. Это может быть небезопасно. Например:

```
val students = new Array[Student](length)
val people: Array[Person] = students // Нельзя, но допустим, что можно...
people(0) = new Person("Fred") // Вот как! Теперь students(0) уже не Student
```

Попробуем наоборот:

```
val people = new Array[Person](length)
val students: Array[Student] = people // Нельзя, но допустим, что можно...
people(0) = new Person("Fred") // Опять students(0) перестал быть Student
```

> **Примечание.** В языке Java можно преобразовать массив `Student[]` в массив `Person[]`, но если попытаться добавить в такой массив не студента, будет возбуждено исключение `ArrayStoreException`. Компилятор Scala отказывается компилировать программы, которые могут вызывать ошибки при обращении с типами.

Допустим, что мы хотим попробовать объявить ковариантную *изменяемую* пару. У нас ничего не получится. Такая пара напоминает массив с двумя элементами, и при компиляции ее будет обнаружена та же ошибка, которую мы только что увидели.

Действительно, если попробовать скомпилировать:

```
class Pair[+T](var first: T, var second: T) // Ошибка
```

будет получена ошибка, сообщающая, что обнаружен ковариантный тип `T` в *контравариантной позиции* (contravariant position) в методе записи

```
first_=(value: T)
```

Параметры методов являются контравариантными позициями, а возвращаемые типы – ковариантными.

Однако внутри функции вариантность параметров меняется – они становятся ковариантными. Например, взгляните на метод `foldLeft` типа `Iterable[+A]`:

```
foldLeft[B](z: B)(op: (A, B) => B): B
               -       +  +     -   +
```

Обратите внимание, что `A` теперь находится в ковариантной позиции.

Правила определения позиций просты и надежны, но иногда они мешают выполнять вполне безопасные действия. Вернемся к методу `replaceFirst` класса неизменяемой пары из раздела 17.3 «Границы изменения типов»:

```
class Pair[+T](val first: T, val second: T) {
    def replaceFirst(newFirst: T) = new Pair[T](newFirst, second) // Ошибка
}
```

Компилятор отвергнет такое объявление, потому что тип `T` параметра находится в контравариантной позиции. Но этот метод не может повредить существующую пару, потому что возвращает новую пару.

Решение проблемы заключается в добавлении второго параметра типа для метода, как показано ниже:

```
def replaceFirst[R >: T](newFirst: R) = new Pair[R](newFirst, second)
```

Теперь метод стал обобщенным методом с другим параметром типа `R`. Но тип `R` является *инвариантным*, поэтому он может находиться в контравариантной позиции.

17.11. Объекты не могут быть обобщенными

Объекты не поддерживают параметризацию типов. Рассмотрим в качестве примера неизменяемые списки. Список элементов типа `T` может быть либо пустым, либо узлом с головой типа `T` и хвостом типа `List[T]`:

```
abstract class List[+T] {
     def isEmpty: Boolean
     def head: T
     def tail: List[T]
}

class Node[T](val head: T, val tail: List[T]) extends List[T] {
     def isEmpty = false
}

class Empty[T] extends List[T] {
     def isEmpty = true
     def head = throw new UnsupportedOperationException
     def tail = throw new UnsupportedOperationException
}
```

> **Примечание.** Здесь я объявил классы `Node` и `Empty`, чтобы Java-программистам было проще следить за дискуссией. Если вы обладаете значительным опытом работы с языком Scala, просто мысленно подставляйте `::` и `Nil`.

На первый взгляд, кажется странным определять `Empty` как класс. Он не имеет полей. Но его нельзя просто превратить в объект:

```
object Empty[T] extends List[T] // Ошибка
```

В объекты нельзя добавлять параметризованные типы. В данном случае проблему можно решить, унаследовав `List[Nothing]`:

```
object Empty extends List[Nothing]
```

В главе 8 говорилось, что тип `Nothing` является подтипом всех типов. То есть при создании списка из одного элемента

```
val lst = new Node(42, Empty)
```

проверка типа дает положительный результат. Благодаря ковариантности тип `List[Nothing]` можно преобразовать в тип `List[Int]` и вызвать конструктор `Node[Int]`.

17.12. Подстановочный символ

В языке Java все обобщенные типы инвариантны. Однако вы можете изменять типы, где они используются, с помощью подстановочного символа. Например, следующий метод:

```
void makeFriends(Pair<? extends Person> people) // Это - Java
```

можно вызвать с аргументом типа `List<Student>`.

В языке Scala тоже можно использовать подстановочный символ. Выглядит это так:

```
def process(people: java.util.List[_ <: Person]) // Это - Scala
```

В Scala нет необходимости использовать подстановочный символ для ковариантного класса `Pair`. Но если предположить, что класс `Pair` объявлен как инвариантный:

```
class Pair[T](var first: T, var second: T)
```

Тогда можно определить:

```
def makeFriends(p: Pair[_ <: Person])
    // Можно вызвать с аргументом Pair[Student]
```

Подстановочный символ можно также использовать в контравариантных объявлениях:

```
import java.util.Comparator
def min[T](p: Pair[T])(comp: Comparator[_ >: T])
```

Подстановочный символ – это «синтаксический сахар» для *экзистенциальных типов* (existential types), которые мы подробно обсудим в главе 18.

Внимание. Подстановочный символ может помочь в некоторых сложных ситуациях. Например, следующее объявление считается недопустимым в версии Scala 2.9:

```
def min[T <: Comparable[_ >: T]](p: Pair[T]) = ...
```

Решить эту проблему можно следующим образом:

```
type SuperComparable[T] = Comparable[_ >: T]
def min[T <: SuperComparable[T]](p: Pair[T]) = ...
```

Упражнения

1. Определите неизменяемый класс `Pair[T, S]` с методом `swap`, возвращающим новую пару, где компоненты поменяны местами.
2. Определите изменяемый класс `Pair[T]` с методом `swap`, который меняет компоненты пары местами.
3. Для класса `Pair[T, S]` напишите обобщенный метод `swap`, который принимает пару в виде аргумента и возвращает новую пару с компонентами, поменянными местами.
4. Почему не требуется объявлять верхнюю границу в методе `replaceFirst` в разделе 17.3 «Границы изменения типов», когда предполагается заменить первый компонент в экземпляре `Pair[Person]` экземпляром `Student`?
5. Почему `RichInt` реализует `Comparable[Int]`, а не `Comparable[RichInt]`?
6. Напишите обобщенный метод `middle`, возвращающий средний элемент из любого экземпляра `Iterable[T]`. Например, вызов `middle("World")` должен вернуть `'r'`.
7. Посмотрите список методов трейта `Iterable[+A]`. Какие из них используют параметр типа `A`? Почему в этих методах он находится в ковариантной позиции?
8. В разделе 17.10 «Ко- и контравариантные позиции» в методе `replaceFirst` определена граница типа. Почему нельзя определить эквивалентный метод для изменяемого класса `Pair[T]`?

```
def replaceFirst[R >: T](newFirst: R) { first = newFirst } // Ошибка
```

9. На первый взгляд, кажется странной необходимость ограничивать параметры метода неизменяемого класса `Pair[+T]`. Но представьте, что в `Pair[T]` можно написать такое определение метода:

```
def replaceFirst(newFirst: T)
```

Проблема в том, что подобный метод можно переопределить не совсем правильным способом. Придумайте пример проблемы. Напишите подкласс `NastyDoublePair` класса `Pair[Double]`, переопределяющий метод `replaceFirst`, который создает пару,

где первый элемент является результатом извлечения квадратного корня из аргумента `newFirst`. Затем сконструируйте вызов метода `replaceFirst(«Hello»)` типа `Pair[Any]`, который в действительности является типом `NastyDoublePair`.

10. Для изменяемого класса `Pair[S, T]` используйте механизм ограничения типа, чтобы определить метод swap, который можно вызывать с параметрами одного типа.

Глава 18. Дополнительные типы

Темы, рассматриваемые в этой главе L2
- 18.1. Типы-одиночки.
- 18.2. Проекции типов.
- 18.3. Цепочки.
- 18.4. Псевдонимы типов.
- 18.5. Структурные типы.
- 18.6. Составные типы.
- 18.7. Инфиксные типы.
- 18.8. Экзистенциальные типы.
- 18.9. Система типов языка Scala.
- 18.10. Собственные типы.
- 18.11. Внедрение зависимостей.
- 18.12. Абстрактные типы L3.
- 18.13. Родовой полиморфизм L3.
- 18.14. Типы высшего порядка L3.
- Упражнения.

В этой главе вы познакомитесь со всеми типами, которые может предложить язык Scala, включая некоторые специализированные. В заключение обсуждения мы рассмотрим собственные (self) типы и механизм внедрения зависимостей.

Основные темы этой главы:
- типы-одиночки (singleton types) удобно использовать для составления цепочек из вызовов методов и определения методов, принимающих объекты в виде параметров;
- проекция типа включает экземпляры внутреннего класса во все объекты внешнего класса;
- поддержка псевдонимов позволяет определять более короткие имена для типов;
- структурные типы являются эквивалентом «утиной типизации»;
- экзистенциальные типы позволяют формализовать групповые символы в параметрах обобщенных типов;

- объявление собственного (self) типа позволяет указать, что трейт требует другого типа;
- шаблон проектирования «слоеный пирог» (cake pattern) использует собственные (self) типы для реализации механизма внедрения зависимостей;
- в подклассах абстрактный тип должен преобразовываться в конкретный;
- тип высшего порядка имеет параметр типа, который сам является параметризованным типом.

18.1. Типы-одиночки

Для любого значения `v` можно сформировать тип `v.type`, имеющий два значения: `v` и `null`. На первый взгляд, такой тип кажется не более чем курьезом, однако у него есть пара полезных применений.

Для начала рассмотрим методы, возвращающие `this`, благодаря чему появляется возможность составлять цепочки из вызовов методов:

```
class Document {
     def setTitle(title: String) = { ...; this }
     def setAuthor(author: String) = { ...; this }
     ...
}
```

Теперь можно составить такую цепочку вызовов:

```
article.setTitle("Whatever Floats Your Boat").setAuthor("Cay Horstmann")
```

Однако в подклассе такой прием становится невозможным:

```
class Book extends Document {
     def addChapter(chapter: String) = { ...; this }
     ...
}

val book = new Book()
book.setTitle("Scala for the Impatient").addChapter(chapter1) // Ошибка
```

Компилятор Scala определит тип ссылки `this`, возвращаемой методом `setTitle`, как `Document`. Но класс `Document` не имеет метода `addChapter`.

Чтобы решить эту проблему, тип значения, возвращаемого методом `setTitle`, следует объявить как `this.type`:

```
def setTitle(title: String): this.type = { ...; this }
```

Теперь значение, возвращаемое вызовом `book.setTitle("...")`, будет иметь тип `book.type`, а поскольку значение `book` имеет метод `addChapter`, цепочка вызовов будет работать.

Кроме того, если потребуется объявить метод, принимающий экземпляр `Object` в виде параметра, можно воспользоваться типом-одиночкой (singleton). Появляется законный вопрос: зачем это нужно? В конце концов, если экземпляр имеется в единственном числе, метод может просто использовать его, вместо того чтобы заставлять вызывающий программный код передавать его.

Однако некоторым нравится конструировать «свободные интерфейсы», которые читаются как фразы на английском языке, например следующая конструкция:

```
book set Title to "Scala for the Impatient"
```

будет скомпилирована как

```
book.set(Title).to("Scala for the Impatient")
```

Здесь `set` – это метод, аргументом которого является объект-одиночка (singleton) `Title`:

```
object Title

class Document {
     private var useNextArgAs: Any = null
     def set(obj: Title.type): this.type = { useNextArgAs = obj; this }
     def to(arg: String) = if (useNextArgAs == Title) title = arg; else ...
     ...
}
```

Обратите внимание на параметр `Title.type`. Здесь нельзя объявить

```
def set(obj: Title) ... // Ошибка
```

потому что `Title` является *объектом*-одиночкой (singleton), а не типом.

18.2. Проекции типов

В главе 5 было показано, что вложенные классы принадлежат *объекту*, в который они вкладываются. Например:

```
import scala.collection.mutable.ArrayBuffer
class Network {
     class Member(val name: String) {
          val contacts = new ArrayBuffer[Member]
     }

     private val members = new ArrayBuffer[Member]

     def join(name: String) = {
          val m = new Member(name)
          members += m
          m
     }
}
```

Каждый экземпляр класса `Network` будет иметь собственный класс `Member`. Например, ниже создаются два экземпляра:

```
val chatter = new Network
val myFace  = new Network
```

Теперь `chatter.Member` и `myFace.Member` — это *разные классы*.

Вы не сможете добавлять членов одного сообщества (network) в другое:

```
val fred = chatter.join("Fred")    // Имеет тип chatter.Member
val barney = myFace.join("Barney") // Имеет тип myFace.Member
fred.contacts += barney            // Ошибка
```

Если такое ограничение вас не устраивает, нужно просто вынести определение класса `Member` за пределы класса `Network`. Для этого прекрасно подойдет объект-компаньон класса `Network`.

Если же вы желаете сохранить подобную организацию классов, но обеспечить более широкое толкование, используйте *проекцию типа* (type projection) `Network#Member`, которая означает: «Member of any Network» (член любого сообщества).

```
class Network {
     class Member(val name: String) {
          val contacts = new ArrayBuffer[Network#Member]
     }
     ...
}
```

Такой подход может пригодиться, когда желательно сохранить «уникальность внутреннего класса для каждого объекта» в отдельных участках программы, но не везде.

> **Внимание.** Проекция типа, такая как `Network#Member`, не считается «цепочкой» (path), и ее нельзя импортировать. Цепочки рассматриваются в следующем разделе.

18.3. Цепочки

Взгляните на следующий тип:

```
com.horstmann.impatient.chatter.Member
```

или если вложить класс `Member` в объект-компаньон:

```
com.horstmann.impatient.Network.Member
```

Такое выражение называется *цепочкой* (path).

Каждый компонент цепочки, кроме последнего, должен быть «стабильным», то есть обозначать единственную, определенную область видимости. Все следующие компоненты отвечают этому условию:

- пакет (package);
- объект (object);
- значение `val`;
- `this`, `super`, `super[S]`, `C.this`, `C.super` или `C.super[S]`.

Компонент цепочки не может быть классом, потому что, как было показано, вложенный класс не является единственным типом – он образует отдельный тип для каждого экземпляра.

Кроме того, компонент цепочки не может быть переменной `var`. Например:

```
var chatter = new Network
...
val fred = new chatter.Member // Ошибка – chatter не стабильный компонент
```

Поскольку переменной `chatter` может быть присвоено другое значение, компилятор не сможет определить точный смысл типа `chatter.Member`.

> **Примечание.** Внутри компилятор транслирует все вложенные выражения типов, такие как `a.b.c.T`, в проекции типов `a.b.c.type#T`. Например, `chatter.Member` превратится в `chatter.type#Member` – любой экземпляр типа `Member` внутри экземпляра-одиночки `chatter.type`. Вообще, это не является поводом для беспокойства. Однако порой вы будете сталкиваться с сообщениями об ошибках, где упоминаются типы в форме `a.b.c.type#T`. Просто мысленно преобразуйте их обратно в форму `a.b.c.T`.

18.4. Псевдонимы типов

С помощью ключевого слова `type` можно создавать простые *псевдонимы* для сложных имен типов, как показано ниже:

```
class Book {
     import scala.collection.mutable._
     type Index = HashMap[String, (Int, Int)]
     ...
}
```

Теперь для ссылки на громоздкое имя типа `scala.collection.mutable.HashMap[String, (Int, Int)]` можно использовать `Book.Index`.

Определение псевдонима типа (type alias) должно вкладываться в класс или объект. Оно не может появляться на верхнем уровне в файле с исходным кодом Scala. Однако в REPL допускается объявлять псевдонимы на верхнем уровне, поскольку все объявления, выполняемые в REPL, неявно заключены в объект верхнего уровня.

> **Примечание.** Ключевое слово `type` также используется для объявления абстрактных типов (abstract types), которые должны преобразовываться в конкретные типы внутри подклассов, например:
>
> ```
> abstract class Reader {
>     type Contents
>     def read(fileName: String): Contents
> }
> ```
>
> Подробнее абстрактные типы будут рассматриваться в разделе 18.12 «Абстрактные типы».

18.5. Структурные типы

«Структурный тип» (structural type) – это описание абстрактных методов, полей и типов, которыми должен обладать соответствующий тип. Например, следующий метод имеет параметр структурного типа:

```
def appendLines(target: { def append(str: String): Any },
                        lines: Iterable[String]) {
    for (l <- lines) { target.append(l); target.append("\n") }
}
```

Методу `appendLines` можно передать экземпляр *любого* типа, обладающего методом `append`. Такой подход дает больше гибкости, чем определение трейта `Appendable`, потому что не всегда есть возможность добавить требуемый трейт в используемые классы.

Внутри компилятор Scala вызывает `target.append(...)` с использованием механизма рефлексии. Структурные типы обеспечивают простой и надежный способ выполнения таких вызовов.

Однако вызовы с использованием механизма рефлексии *намного* дороже обычных вызовов методов. По этой причине прибегать к помощи структурных типов следует только для моделирования общего поведения классов, которые не могут совместно использовать трейт.

> **Примечание.** Структурные типы напоминают механизм «утиной типизации» (duck tiping) в таких языках, как JavaScript или Ruby. В них переменные не имеют типа. Когда вы пишете в программе вызов `obj.quack()`, наличие метода `quack` у объекта, на который ссылается переменная `obj`, выясняется только во время выполнения. Иными словами, вам не требуется объявлять `obj` как переменную типа `Duck` (утка), главное, чтобы она ходила и крякала как утка.

18.6. Составные типы

Составной тип (compound type) имеет вид:

```
T1 with T2 with T3 ...
```

где T_1, T_2, T_3 и т. д. – это типы. Чтобы принадлежать составному типу, значение должно принадлежать каждому отдельно взятому типу. То есть такой тип можно назвать типом пересечения.

Составной тип можно использовать для манипулирования значениями, включающими несколько трейтов. Например:

```
val image = new ArrayBuffer[java.awt.Shape with java.io.Serializable]
```

Объект `image` можно нарисовать инструкцией `for (s <- image) graphics.draw(s)`. Его можно сериализовать, потому что известно, что все его элементы сериализуемы.

Конечно, добавлять в эту коллекцию можно только экземпляры, одновременно являющиеся фигурами и поддерживающие сериализацию:

```
val rect = new Rectangle(5, 10, 20, 30)
image += rect           // OK - Rectangle реализует интерфейс Serializable
image += new Area(rect) // Ошибка: Area реализует Shape но не Serializable
```

> **Примечание.** Если имеется объявление
>
> `trait ImageShape extends Shape with Serializable`
>
> это означает, что `ImageShape` наследует пересечение типов `Shape` и `Serializable`.

Имеется возможность добавлять объявления структурных типов в простые и составные типы. Например:

```
Shape with Serializable { def contains(p: Point): Boolean }
```

Экземпляр этого типа должен быть подтипом `Shape` и `Serializable` и иметь метод `contains` с параметром типа `Point`.

Технически структурный тип

```
{ def append(str: String): Any }
```

является более краткой формой записи

```
AnyRef { def append(str: String): Any }
```

а объявление составного типа

```
Shape with Serializable
```

более краткой формой записи

```
Shape with Serializable {}
```

18.7. Инфиксные типы

Инфиксный тип – это тип с двумя параметрами типов, определение которого записывается в «инфиксной» нотации, когда имя определяемого типа помещается между параметрами типов. Например, вместо

```
Map[String, Int]
```

можно записать

```
String Map Int
```

Инфиксная нотация часто используется в математике. Например, $A \times B = \{(a, b) \mid a \in A, b \in B\}$ – это множество пар с компонентами типов A и B. На языке Scala этот тип записывается как `(A, B)`. Если вы предпочитаете математическую нотацию, можно добавить такое определение:

```
type ×[A, B] = (A, B)
```

а затем использовать форму записи `String × Int` вместо `(String, Int)`.

Все инфиксные операторы типа имеют одинаковый приоритет. Как и обычные операторы, они являются левоассоциативными, если их имена не заканчиваются двоеточием (`:`). Например, определение

```
String × Int × Int
```

означает `((String, Int), Int)`. Этот тип похож на тип `(String, Int, Int)`, но не является им, потому что последний нельзя записать в инфиксной нотации.

> **Примечание.** Имя инфиксного типа может быть последовательностью любых символов операторов, кроме единственного символа `*`. Это правило помогает избежать путаницы с объявлениями переменного числа аргументов `T*`.

18.8. Экзистенциальные типы

Экзистенциальные типы были добавлены в язык Scala для совместимости с групповыми символами в языке Java. *Экзистенциальный тип* – это выражение типа, за которым следует конструкция `forSome`

{ ... }, где фигурные скобки окружают объявления type и val. Например,

```
Array[T] forSome { type T <: JComponent }
```

– это то же самое, что подстановочные типы (wildcard type)

```
Array[_ <: JComponent]
```

которые мы видели в главе 17.

Подстановочный типы в Scala – это синтаксический сахар для экзистенциальных типов. Например,

```
Array[_]
```

суть то же самое, что и

```
Array[T] forSome { type T }
```

а

```
Map[_, _]
```

то же самое, что и

```
Map[T, U] forSome { type T; type U }
```

Конструкция forSome позволяет определять более сложные отношения, чем можно выразить с применением подстановочного символа, например:

```
Map[T, U] forSome { type T; type U <: T }
```

В блоке forSome можно использовать объявления val, потому что val может иметь собственные подтипы (см. раздел 18.1 «Типы-одиночки»). Например:

```
n.Member forSome { val n: Network }
```

Само по себе это определение не представляет большого интереса – вместо него можно было бы использовать простую проекцию типа (type projection) Network#Member. Но взгляните на следующий пример:

```
def process[M <: n.Member forSome {val n: Network}](m1: M, m2: M) = (m1, m2)
```

Этот метод будет принимать членов одного сообщества и отвергать членов разных сообществ:

```
val chatter = new Network
val myFace = new Network
val fred = chatter.join("Fred")
val wilma = chatter.join("Wilma")
val barney = myFace.join("Barney")
process(fred, wilma)   // OK
process(fred, barney) // Ошибка
```

18.9. Система типов языка Scala

Документация по языку Scala содержит исчерпывающий список всех типов данных, который воспроизводится в табл. 18.1 с краткими пояснениями к каждому типу.

Таблица 18.1. Типы в языке Scala

Тип	Синтаксис	Примечания
Класс или трейт	`class C ..., trait C ...`	См. главу 5 и главу 10
Кортеж (tuple)	`(T1, ..., Tn)`	Раздел 4.7
Функция	`(T1, ..., Tn) => T`	
Аннотированный тип	`T @A`	См. главу 15
Параметризованный тип	`A[T1, ..., Tn]`	См. главу 17
Тип-одиночка (singleton)	`value.type`	См. раздел 18.1
Проекция типа	`O#I`	См. раздел 18.2
Составной тип	`T1 with T2 with ... with Tn { объявления }`	См. раздел 18.6
Инфиксный тип	`T1 A T2`	См. раздел 18.7
	`T forSome { объявления type и val }`	См. раздел 18.8

Примечание. В табл. 18.1 перечислены типы, которые вы как программист можете объявлять. Существует еще ряд типов, используемых компилятором Scala для своих целей. Например, иногда вам придется сталкиваться с типом метода, который обозначается как (T1, ..., Tn)T без `=>`. Например, если ввести

```
def square(x: Int) = x * x
```

в интерактивной оболочке Scala REPL, она ответит:

```
square (x: Int)Int
```

тогда как в ответ на ввод

```
val triple = (x: Int) => 3 * x
```

она вернет

```
triple: Int => Int
```

Метод можно преобразовать в функцию, введя его имя и добавив в конце символ подчеркивания (_). Введя `square _`, вы получите ответ: `Int => Int`.

18.10. Собственные типы

В главе 10 было показано, как трейт может требовать, чтобы класс, в который он подмешивается, наследовал другой тип. Для этого трейт определяется с объявлением собственного типа (self type):

```
this: Type =>
```

Такой трейт сможет подмешиваться только в подклассы указанного типа. В следующем примере представлен трейт `LoggedException`, который можно подмешивать лишь в классы, наследующие класс `Exception`:

```
trait Logged {
     def log(msg: String)
}

trait LoggedException extends Logged {
     this: Exception =>
          def log() { log(getMessage()) }
               // Можно вызвать getMessage, потому что это Exception
}
```

Если попытаться подмешать трейт в класс, не соответствующий объявлению собственного типа (self type), возникнет ошибка:

```
val f = new JFrame with LoggedException
     // Ошибка: JFrame не является подтипом Exception
     // собственного типа LoggedException
```

Если понадобится потребовать соответствие нескольким типам, используйте составной тип:

```
this: T with U with ... =>
```

Примечание. Синтаксис определения собственного типа можно совместить с синтаксисом определения «псевдонима ссылки `this` на вмещающий класс», коротко представленным в главе 5. Если указать имя, отличное от имени `this`, его можно будет использовать внутри подтипов. Например:

```
trait Group {
     outer: Network =>
          class Member {
          ...
     }
}
```

Трейт `Group` сможет подмешиваться только в подтипы `Network`, а внутри `Member` можно будет ссылаться на `Group.this` по имени `outer`.
Похоже, что этот синтаксис органично развивался в течение долгого времени; к сожалению, он несет большое количество неразберихи, давая взамен лишь небольшое расширение функциональности.

Внимание. Собственные типы не наследуются автоматически. Если попытаться объявить:

```
trait ManagedException extends LoggedException { ... }
```

вы получите ошибку с сообщением, что собственный тип трейта `ManagedException` не соответствует собственному типу `Exception` трейта `LoggedException`. В этой ситуации необходимо повторить объявление собственного типа:

```
trait ManagedException extends LoggedException {
     this: Exception =>
     ...
}
```

18.11. Внедрение зависимостей

При построении крупных систем из компонентов с различными реализациями необходимо предусмотреть возможность сборки системы с компонентами по выбору. Например, в вашем распоряжении могут иметься компоненты, реализующие фиктивную и действительную базу данных, или компоненты вывода информации в консоль и в файл. Конкретная реализация может требовать поддержки действительной базы данных и вывода информации в консоль для проведения экспериментов или фиктивной базы данных и вывода информации в файл для запуска сценария автоматического тестирования.

Обычно между компонентами имеются некоторые зависимости. Например, компонент доступа к базе данных может требовать поддержки журналирования.

В Java имеется множество инструментов, позволяющих программисту описывать зависимости, такие как фреймворк Spring или система модулей OSGi. Для каждого компонента описывается, от каких других компонентов он зависит. Ссылки на фактические реализации компонентов «внедряются» в процессе сборки приложения.

Простейшая форма внедрения зависимостей (dependency injection) в языке Scala реализуется на основе трейтов и собственных типов.

Допустим, имеется трейт, описывающий журналирование:

```
trait Logger { def log(msg: String) }
```

с реализациями в виде классов `ConsoleLogger` и `FileLogger`.

Трейт аутентификации пользователя требует поддержки журналирования для регистрации ошибок аутентификации:

```
trait Auth {
     this: Logger =>
          def login(id: String, password: String): Boolean
}
```

Логика приложения требует поддержки и аутентификации, и журналирования:

```
trait App {
     this: Logger with Auth =>
          ...
}
```

Теперь приложение можно собрать, как показано ниже:

```
object MyApp extends App
     with FileLogger("test.log") with MockAuth("users.txt")
```

Такое оформление зависимостей путем объединения трейтов выглядит неуклюжим. В конце концов, приложение не является механизмом аутентификации и журналирования в файл. Оно включает

эти компоненты, и было бы более естественным использовать переменные экземпляра для хранения данных компонентов, чем объединять их в один огромный тип. Более удачное решение позволяет получить шаблон проектирования «Слоеный пирог» (cake pattern). В этом шаблоне определяются трейты для каждой из требуемых служб, содержащие:

- определения собственных типов для всех требуемых компонентов;
- трейт, описывающий интерфейс службы;
- абстрактное значение `val` для экземпляра службы;
- возможно, реализацию интерфейса службы.

```
abstract trait LoggerComponent {
     trait Logger { ... }
     val logger: Logger
     class FileLogger(file: String) extends Logger { ... }
     ...
}

abstract trait AuthComponent {
     this: LoggerComponent => // Доступ к поддержке журналирования
     trait Auth { ... }
     val auth: Auth
     class MockAuth(file: String) extends Auth { ... }
     ...
}
```

Обратите внимание: объявление собственного типа в этом примере свидетельствует, что компонент аутентификации зависит от компонента журналирования.

Теперь настройку конфигурации компонентов можно определить в одном месте:

```
object AppComponents extends LoggerComponent with AuthComponent {
     val logger = new FileLogger("test.log")
     val auth = new MockAuth("users.txt")
}
```

Любой подход из представленных лучше, чем описание связей компонентов в XML-файле, потому что компилятор может проверить, удовлетворяются ли зависимости модулей.

18.12. Абстрактные типы L3

Классы и трейты могут определять *абстрактные типы*, которые должны преобразовываться в конкретные типы внутри подклассов. Например:

```
trait Reader {
     type Contents
     def read(fileName: String): Contents
}
```

Здесь тип `Contents` – абстрактный. Конкретный подкласс должен определить этот тип:

```
class StringReader extends Reader {
     type Contents = String
     def read(fileName: String) = Source.fromFile(fileName, "UTF-8").mkString
}

class ImageReader extends Reader {
     type Contents = BufferedImage
     def read(fileName: String) = ImageIO.read(new File(fileName))
}
```

Того же эффекта можно добиться с помощью параметризованных типов:

```
trait Reader[C] {
     def read(fileName: String): C
}

class StringReader extends Reader[String] {
     def read(fileName: String) = Source.fromFile(fileName, "UTF-8").mkString
}

class ImageReader extends Reader[BufferedImage] {
     def read(fileName: String) = ImageIO.read(new File(fileName))
}
```

Какой подход лучше? В Scala используются следующие правила:

- параметризованные типы используются, когда типы определяются в момент создания экземпляра класса, например при создании экземпляра `HashMap[String, Int]`;

- абстрактные типы используются, когда конкретные типы определяются в подклассах, как в примере с подклассами трейта `Reader`.

Ничего страшного не произойдет, если при определении подкласса будут использоваться параметризованные типы. Но при наличии большого количества зависимостей от типов лучше использовать абстрактные типы – это избавит от длинных списков параметров типов. Например:

```
trait Reader {
     type In
     type Contents
     def read(in: In)
}

class ImageReader extends Reader {
     type In = File
     type Contents = BufferedImage
     def read(file: In) = ImageIO.read(file)
}
```

При использовании параметризованных типов класс `ImageReader` должен был бы наследовать `Reader[File, BufferedImage]`. В этом нет ничего особенного, но подобный прием плохо масштабируется в более сложных случаях.

Кроме того, абстрактные типы можно использовать для выражения тонких взаимозависимостей между типами. Пример такой ситуации приводится в следующем разделе.

Абстрактные типы могут иметь границы типов, подобно параметрам типов. Например:

```
trait Listener {
     type Event <: java.util.EventObject
     ...
}
```

Подкласс этого класса должен определить совместимый тип, например:

```
trait ActionListener {
     type Event = java.awt.event.ActionEvent // OK, это - подтип
}
```

18.13. Родовой полиморфизм L3

Довольно сложно смоделировать семейство типов, изменяющихся совместно, использующих общий программный код, и сохранить при этом безопасность типов. Рассмотрим в качестве примера реализацию обработки событий на языке Java. События могут быть самых разных типов (такие как `ActionEvent`, `ChangeEvent` и т. д.). Каждый тип имеет отдельный интерфейс обработчиков (listener interface) событий (`ActionListener`, `ChangeListener` и т. д.). Это пример родового полиморфизма (family polymorphism).

Попробуем спроектировать универсальный механизм управления обработкой событий. Сначала рассмотрим реализацию на основе обобщенных типов, а затем перейдем к абстрактным типам.

В Java каждый интерфейс обработчика (listener interface) событий определяет свое имя для метода, вызываемого при возникновении события: `actionPerformed`, `stateChanged`, `itemStateChanged` и т. д. Мы поступим иначе и унифицируем имя метода:

```
trait Listener[E] {
     def occurred(e: E): Unit
}
```

Источник события должен поддерживать коллекцию обработчиков и метод для вызова их всех:

```
trait Source[E, L <: Listener[E]] {
     private val listeners = new ArrayBuffer[L]
     def add(l: L) { listeners += l }
     def remove(l: L) { listeners -= l }
     def fire(e: E) {
          for (l <- listeners) l.occurred(e)
     }
}
```

Теперь рассмотрим кнопку, нажатие которой вызывает событие. Определим тип обработчика событий:

```
trait ActionListener extends Listener[ActionEvent]
```

Класс `Button`, в который можно подмешать трейт `Source`:

```
class Button extends Source[ActionEvent, ActionListener] {
     def click() {
```

```
        fire(new ActionEvent(this, ActionEvent.ACTION_PERFORMED, "click"))
    }
}
```

Цель достигнута: в классе `Button` не пришлось копировать программный код управления обработкой событиями, и типы обработчиков получились безопасными. Вы не сможете добавить в экземпляр кнопки обработчик типа `ChangeListener`.

При создании экземпляра класса `ActionEvent` ему передается ссылка `this` на источник события, но тип источника события определяется как `Object`. Мы можем увеличить безопасность типов, определив собственный тип (self type):

```
trait Event[S] {
    var source: S = _
}

trait Listener[S, E <: Event[S]] {
    def occurred(e: E): Unit
}

trait Source[S, E <: Event[S], L <: Listener[S, E]] {
    this: S =>
    private val listeners = new ArrayBuffer[L]
    def add(l: L) { listeners += l }
    def remove(l: L) { listeners -= l }
    def fire(e: E) {
        e.source = this // здесь используется собственный тип
        for (l <- listeners) l.occurred(e)
    }
}
```

Обратите внимание на объявление собственного типа `this: S =>`, необходимое для настройки источника в `this`. Иначе значением `this` мог бы быть любой экземпляр `Source`, не обязательно требуемый трейтом `Event[S]`.

Ниже показано определение кнопки:

```
class ButtonEvent extends Event[Button]

trait ButtonListener extends Listener[Button, ButtonEvent]

class Button extends Source[Button, ButtonEvent, ButtonListener] {
    def click() { fire(new ButtonEvent) }
}
```

В примере выше можно видеть, насколько быстро разрастается список параметров типов. Использование абстрактных типов несколько улучшает ситуацию.

```
trait ListenerSupport {
     type S <: Source
     type E <: Event
     type L <: Listener

     trait Event {
          var source: S = _
     }

     trait Listener {
          def occurred(e: E): Unit
     }

     trait Source {
          this: S =>
          private val listeners = new ArrayBuffer[L]
          def add(l: L) { listeners += l }
          def remove(l: L) { listeners -= l }
          def fire(e: E) {
               e.source = this
               for (l <- listeners) l.occurred(e)
          }
     }
}
```

Но это решение имеет свою цену. Псевдонимы типов нельзя объявлять на верхнем уровне. Именно поэтому пришлось всю конструкцию заключить в трейт `ListenerSupport`.

Теперь, когда потребуется определить кнопку с событием и обработчиком, достаточно заключить определение в модуль, наследующий этот трейт:

```
object ButtonModule extends ListenerSupport {
     type S = Button
     type E = ButtonEvent
     type L = ButtonListener

     class ButtonEvent extends Event
     trait ButtonListener extends Listener
     class Button extends Source {
```

```
        def click() { fire(new ButtonEvent) }
    }
}
```

А когда потребуется использовать кнопку, достаточно импортировать модуль:

```
object Main {
    import ButtonModule._

    def main(args: Array[String]) {
        val b = new Button
        b.add(new ButtonListener {
            override def occurred(e: ButtonEvent) { println(e) }
        })
        b.click()
    }
}
```

Примечание. В этом примере я использовал однобуквенные имена для абстрактных типов, чтобы проще было провести аналогию с версией, где используются параметризованные типы. Однако в языке Scala принято использовать более описательные имена типов, что улучшает самодокументирование кода:

```
object ButtonModule extends ListenerSupport
{
    type SourceType = Button
    type EventType = ButtonEvent
    type ListenerType = ButtonListener
    ...
}
```

18.14. Типы высшего порядка L3

Обобщенный тип `List` зависит от типа `T` и производит требуемый тип. Например, указав тип `Int`, вы получите тип `List[Int]`. По этой причине обобщенные типы, такие как `List`, иногда называют *конструкторами типов*. В языке Scala можно подняться уровнем выше и определить тип, зависящий от типа, который, в свою очередь, также зависит от типа.

Чтобы понять, где это может пригодиться, взгляните на упрощенный пример трейта `Iterable`:

```
trait Iterable[E] {
     def iterator(): Iterator[E]
     def map[F](f: (E) => F): Iterable[F]
}
```

А теперь на класс, реализующий этот трейт:

```
class Buffer[E] extends Iterable[E]
     def iterator(): Iterator[E] = ...
     def map[F](f: (E) => F): Buffer[F] = ...
}
```

Ожидается, что в случае с буфером метод `map` будет возвращать экземпляр класса `Buffer`, а не просто `Iterable`. То есть мы не можем реализовать метод `map` в трейте `Iterable`. Решение заключается в том, чтобы параметризовать `Iterable` конструктором типа, как показано ниже:

```
trait Iterable[E, C[_]] {
     def iterator(): Iterator[E]
     def build[F](): C[F]
     def map[F](f : (E) => F) : C[F]
}
```

Теперь трейт `Iterable` зависит от конструктора типа, обозначенного как `C[_]`. Это делает трейт `Iterable` типом высшего порядка (higher-kinded type).

Тип значения, возвращаемого методом `map`, может и не совпадать с типом трейта `Iterable`, для которого был вызван метод `map`. Например, если вызвать метод `map` экземпляра типа `Range`, результатом необязательно должен быть диапазон, то есть метод `map` должен сконструировать другой тип, такой как `Buffer[F]`. Подобный тип `Range` объявляется как

```
class Range extends Iterable[Int, Buffer]
```

Обратите внимание, что второй параметр – это конструктор типа `Buffer`.

Для реализации метода `map` в `Iterable` необходима дополнительная поддержка. Трейт `Iterable` должен иметь возможность производить

контейнер, хранящий значения любого типа F. Определим трейт Container как некоторый тип контейнера, куда можно добавлять значения:

```
trait Container[E] {
     def +=(e: E): Unit
}
```

Для получения такого объекта необходим метод build:

```
trait Iterable[E, C[X] <: Container[X]] {
     def build[F](): C[F]
     ...
}
```

Теперь конструктор типа C ограничен типом Container, благодаря чему нам известно, что в объект, возвращаемый методом build, можно добавлять элементы. Больше не нужно использовать подстановочный символ для параметра C, так как мы должны указать, что C[X] является контейнером элементов того же типа X.

> **Примечание.** Трейт Container – это упрощенная версия механизма построителей (builder mechanism), используемого в библиотеке коллекций языка Scala.

Теперь метод map можно реализовать в трейте Iterable:

```
def map[F](f : (E) => F) : C[F] = {
     val res = build[F]()
     val iter = iterator()
     while (iter.hasNext) res += f(iter.next())
     res
}
```

Классы итерируемых объектов более не должны поставлять собственную реализацию метода map. Ниже приводится пример класса Range:

```
class Range(val low: Int, val high: Int) extends Iterable[Int, Buffer] {
     def iterator() = new Iterator[Int] {
          private var i = low
          def hasNext = i <= high
```

```
        def next() = { i += 1; i - 1 }
    }

    def build[F]() = new Buffer[F]
}
```

Обратите внимание, что класс `Range` наследует трейт `Iterable`: он позволяет выполнять итерации через его содержимое. Но он не является наследником трейта `Container`: в него нельзя добавлять значения.

Класс `Buffer`, напротив, наследует оба трейта:

```
class Buffer[E : Manifest] extends Iterable[E, Buffer] with Container[E] {
    private var capacity = 10
    private var length = 0
    private var elems = new Array[E](capacity) // См. примечание

    def iterator() = new Iterator[E] {
        private var i = 0
        def hasNext = i < length
        def next() = { i += 1; elems(i - 1) }
    }

    def build[F : Manifest]() = new Buffer[F]

    def +=(e: E) {
        if (length == capacity) {
            capacity = 2 * capacity
            val nelems = new Array[E](capacity) // См. примечание
            for (i <- 0 until length) nelems(i) = elems(i)
            elems = nelems
        }
        elems(length) = e
        length += 1
    }
}
```

> **Примечание.** В этом примере есть одна дополнительная сложность, не имеющая отношения к типам высшего порядка (higher-kinded types). Чтобы создать обобщенный массив `Array[E]`, тип `E` должен соответствовать границам контекста `Manifest`, о котором рассказывалось в главе 17.

Этот пример демонстрирует применение типов высшего порядка. Трейт `Iterator` зависит от трейта `Container`, но трейт `Container` не является типом – это механизм создания типов.

Трейт `Iterable` в библиотеке коллекций языка Scala не имеет явного параметра для создания коллекций. Вместо этого для создания целевой коллекции используется *неявный параметр*, появляющийся как по волшебству. Дополнительную информацию см. в главе 21.

Упражнения

1. Реализуйте класс `Bug`, моделирующий жука, перемещающегося по горизонтальной линии. Метод `move` перемещает жука в текущем направлении, метод `turn` изменяет направление на противоположное, а метод `show` выводит текущую позицию. Обеспечьте возможность составления цепочек из вызовов этих методов. Например, цепочка:

```
bugsy.move(4).show().move(6).show().turn().move(5).show()
```

   должна вывести 4 10 5.
2. Реализуйте «свободный» интерфейс для класса `Bug` из предыдущего упражнения, чтобы можно было записать:

```
bugsy move 4 and show and then move 6 and show turn around move 5 and show
```

3. Дополните свободный интерфейс, представленный в разделе 18.1 «Типы-одиночки», так, чтобы можно было записать вызов:

```
book set Title to "Scala for the Impatient" set Author to "Cay
Horstmann"
```

4. Реализуйте метод `equals` в классе `Member`, вложенном в класс `Network`, в разделе 18.2 «Проекции типов». Два члена сообщества могут быть признаны равными, если только они принадлежат одному сообществу.
5. Взгляните на следующий псевдоним типа:

```
type NetworkMember = n.Member forSome { val n: Network }
```

   и на функцию:

```
def process(m1: NetworkMember, m2: NetworkMember) = (m1, m2)
```

6. Назовите отличия от функции `process` из раздела 18.8 «Экзистенциальные типы».
   В библиотеке Scala имеется тип `Either`, который можно использовать в реализациях алгоритмов, возвращающих либо результат, либо некоторую информацию об ошибке. Напишите функцию, принимающую два параметра: отсортированный массив

целых чисел и целочисленное значение. Функция должна возвращать индекс значения в массиве или индекс ближайшего по значению элемента. Для возвращаемого значения используйте инфиксный тип.

7. Реализуйте метод, принимающий экземпляр любого класса, который имеет метод

```
def close(): Unit
```

вместе с функцией обработки этого объекта. Функция должна вызывать метод `close` по завершении обработки или в случае какого-либо исключения.

8. Напишите функцию `printValues` с тремя параметрами `f`, `from` и `to`, выводящую все значения `f`, для входных значений в заданном диапазоне от `from` до `to`. Здесь `f` должен быть любым объектом с методом `apply`, получающим и возвращающим значение типа `Int`. Например:

```
printValues((x: Int) => x * x, 3, 6)              // Выведет 9 16 25 36
printValues(Array(1, 1, 2, 3, 5, 8, 13, 21, 34, 55),3,6) // Выведет 3 5 8 13
```

9. Взгляните на следующий класс, моделирующий некоторое физическое измерение:

```
abstract class Dim[T](val value: Double, val name: String) {
     protected def create(v: Double): T
     def +(other: Dim[T]) = create(value + other.value)
     override def toString() = value + « « + name
}
```

Ниже демонстрируется конкретный подкласс:

```
class Seconds(v: Double) extends Dim[Seconds](v, “s”) {
     override def create(v: Double) = new Seconds(v)
}
```

Однако теперь какой-нибудь неумеха сможет определить:

```
class Meters(v: Double) extends Dim[Seconds](v, «m») {
     override def create(v: Double) = new Seconds(v)
}
```

позволив складывать метры с секундами. Попробуйте предотвратить это, определив собственный тип.

10. Обычно вместо собственных типов можно использовать трейты, наследующие классы, но в некоторых ситуациях применение собственных типов изменяет порядок инициализации и переопределения. Смоделируйте пример такой ситуации.

Глава 19. Парсинг

Темы, рассматриваемые в этой главе A3

- 19.1. Грамматики.
- 19.2. Комбинирование операций парсера.
- 19.3. Преобразование результатов парсинга.
- 19.4. Отбрасывание лексем.
- 19.5. Создание деревьев синтаксического анализа.
- 19.6. Уход от левой рекурсии.
- 19.7. Дополнительные комбинаторы.
- 19.8. Уход от возвратов.
- 19.9. Packrat-парсеры.
- 19.10. Что такое парсеры?
- 19.11. Парсеры на основе регулярных выражений.
- 19.12. Парсеры на основе лексем.
- 19.13. Обработка ошибок.
- Упражнения.

В этой главе вы увидите, как использовать библиотеку «парсер-комбинаторов» (combinator parser) для анализа данных с фиксированной структурой. Примерами таких данных могут служить программы на языке программирования или данные в таких форматах, как HTML или JSON. Далеко не всем придется писать *парсеры* (синтаксические анализаторы) для этих языков, поэтому данная глава может не иметь для вас практической пользы. Даже если вы знакомы с основными понятиями грамматик и парсеров, все равно хотя бы бегло просмотрите эту главу, потому что библиотека парсеров в Scala является отличным примером сложного *предметно-ориентированного языка* (Domain-Specific Language, DSL), встроенного в Scala.

Основные темы этой главы:

- выбор из альтернатив, конкатенация, опции и повторения в грамматике превращаются в `|`, `~`, `opt` и `rep` в парсер-комбинаторах языка Scala;
- в парсерах RegexParsers литералы строк и регулярные выражения соответствуют лексемам;

- используйте `^^` для обработки результатов парсинга;
- используйте сопоставление с образцами в функциях, передаваемых `^^`, для разбора `~` результатов;
- используйте `~>` и `<~` для удаления лексем, ставших ненужными после сопоставления;
- комбинатор `repsep` обрабатывает общий случай повторения элементов с разделителем;
- парсеры на основе лексем удобно использовать для парсинга языков с зарезервированными словами и операторами; будьте готовы определить собственный механизм лексического анализа;
- парсеры – это функции, получающие инструмент чтения и возвращающие результаты парсинга: успех, отказ или ошибка;
- результат `Failure` содержит подробную информацию об ошибке;
- чтобы повысить подробность сообщений об ошибках, может потребоваться добавить комбинаторы `failure` в грамматику;
- благодаря символам операторов, неявным преобразованиям и механизму сопоставления с образцом библиотека парсер-комбинаторов упрощает создание парсеров для всех, кто понимает контекстно-свободные грамматики (context-free grammars); даже если вам нет нужды писать собственные парсеры, вы можете рассматривать эту библиотеку как интереснейший пример реализации эффективного предметно-ориентированного языка.

19.1. Грамматики

Чтобы разбираться в библиотеке парсинга языка Scala, необходимо знать некоторые понятия из теории формальных языков. *Грамматика* – это набор правил составления строк, соответствующих определенному формату. Например, можно сказать, что арифметические выражения составляются в соответствии со следующими правилами:

- каждое число целиком является арифметическим выражением;
- `+ - *` являются операторами;
- если *left* и *right* – арифметические выражения, а *op* – оператор, тогда *left op right* является арифметическим выражением;
- если *expr* – арифметическое выражение, тогда *(expr)* является арифметическим выражением.

Согласно этим правилам, `3+4` и `(3+4)*5` являются арифметическими выражениями, а `3+)`, `3^4` или `3+x` таковыми не являются.

Грамматика обычно записывается в нотации, называемой *формой Бэкуса-Наура* (Backus-Naur Form, BNF). Ниже приводится определение грамматики для нашего языка выражений в форме BNF:

```
op ::= "+" | "-" | "*"
expr ::= number | expr op expr | "(" expr ")"
```

В этом примере элемент number не определен. Его можно определить как

```
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
number ::= digit | digit number
```

Но на практике гораздо эффективнее отбирать числа перед началом парсинга, на отдельном этапе, называемом *лексическим анализом* (lexical analysis). *Механизм лексического анализа* (lexer) удаляет пробельные символы и комментарии и формирует *лексемы* – идентификаторы, числа или символы. В нашем языке выражений лексемами являются числа и символы `+ - * ( )`.

Обратите внимание, что `op` и `expr` не являются лексемами. Это структурные элементы, введенные автором грамматики с целью обеспечить правильный порядок следования лексем. Такие символы называются *нетерминальными* (nonterminal). Один из нетерминальных символов является корнем иерархии. В нашем случае это `expr`. Он называется *начальным символом* (start symbol). Чтобы получить правильно отформатированную строку, вы начинаете с начального символа и применяете правила грамматики, пока все нетерминальные символы не заменены и не останутся одни лексемы. Например, следующая последовательность анализа:

```
expr -> expr op expr -> number op expr ->
  -> number "+" expr -> number "+" number
```

показывает, что `3+4` является допустимым выражением.

На практике чаще используется «расширенная форма Бэкуса-Наура» (Extended Backus-Naur form, EBNF) , позволяющая определять необязательные элементы и повторения. Воспользуемся уже знакомыми операторами регулярных выражений, `? * +` , для обозначения повторений 0 или 1 раз, 0 или более раз, 1 или более раз соответственно. Например, список чисел, разделенных запятыми, в грамматике можно описать так:

```
numberList ::= number ( "," numberList )?
```

или

```
numberList ::= number ( "," number )*
```

В качестве еще одного примера EBNF определим улучшенную грамматику арифметических выражений, поддерживающую приоритет операторов, как показано ниже:

```
expr ::= term ( ( "+" | "-" ) expr )?
term ::= factor ( "*" factor )*
factor ::= number | "(" expr ")"
```

19.2. Комбинирование операций парсера

Чтобы задействовать библиотеку парсинга в языке Scala, необходимо реализовать класс, наследующий трейт `Parsers` и определяющий операции парсинга, состоящие из комбинаций элементарных операций, таких как:

- сопоставление с лексемами;
- выбор между двумя операциями (`|`);
- последовательное выполнение двух операций (`~`);
- повторение операций (`rep`);
- выполнение необязательной операции (`opt`).

Следующий класс парсера реализует анализ арифметических выражений. Он наследует трейт `RegexParsers`, в свою очередь, наследующий трейт `Parsers`, способный определять лексемы с помощью регулярных выражений. Здесь мы определили значение `number` как регулярное выражение `"[0-9]+".r`:

```
class ExprParser extends RegexParsers {
    val number = "[0-9]+".r

    def expr: Parser[Any] = term ~ opt(("+" | "-") ~ expr)
    def term: Parser[Any] = factor ~ rep("*" ~ factor)
    def factor: Parser[Any] = number | "(" ~ expr ~ ")"
}
```

Обратите внимание, что этот парсер представляет собой прямолинейную реализацию определения EBNF из предыдущего раздела.

Для объединения частей выражения используется оператор ~, а вместо ? и * используются opt и rep.

В нашем примере каждая функция возвращает значение типа Parser[Any]. Этот тип не имеет большой практической пользы, и мы улучшим его в следующем разделе.

Чтобы запустить парсер, необходимо вызвать унаследованный метод parse, например:

```
val parser = new ExprParser
val result = parser.parseAll(parser.expr, "3-4*5")
if (result.successful) println(result.get)
```

Метод parseAll принимает метод для вызова, то есть метод, ассоциированный с начальным символом грамматики, и строку для анализа.

> **Примечание.** Существует также метод parse, выполняющий парсинг начала строки и останавливающийся при невозможности обнаружения другого совпадения. Этот метод не имеет большого практического значения; например вызов parser.parse(parser.expr, "3-4/5") разберет часть выражения 3-4, а затем остановится, столкнувшись с символом /, который не сможет обработать.

Этот фрагмент программы выведет:

```
((3~List())~Some((-~((4~List((*~5)))~None))))
```

Чтобы интерпретировать этот вывод, необходимо знать следующее:

- литералы строк и регулярные выражения возвращают значения типа String;
- p ~ q возвращает экземпляр case-класса ~, близко напоминающий пару;
- opt(p) возвращает экземпляр Option:, либо Some(...), либо None;
- rep(p) возвращает экземпляр List.

Вызов expr возвращает результат вызова term, связанный с необязательной частью Some(...), которую я не буду анализировать.

Поскольку метод term определен как

```
def term = factor ~ rep(("*" | "/" ) ~ factor)
```

он возвращает результат вызова `factor`, связанный с экземпляром `List`. То есть с пустым списком, потому что слева от оператора «минус» (`-`) отсутствует оператор `*` или `/`.

Конечно, анализировать такой результат достаточно утомительно. В следующем разделе будет показано, как преобразовать его в более простой вид.

19.3. Преобразование результатов парсинга

Вместо того чтобы пытаться создать парсер, анализирующий сложную структуру из `~`, опций и списков, можно просто преобразовать промежуточные результаты в более приемлемый вид. Продолжим рассмотрение примера парсера арифметических выражений. Если целью является вычисление выражения, тогда каждая из функций — `expr`, `term` и `factor` — должна возвращать значение проанализированного подвыражения. Начнем с

```
def factor: Parser[Any] = number | "(" ~ expr ~ ")"
```

Нам нужно, чтобы она возвращала значение типа `Int`:

```
def factor: Parser[Int] = ...
```

Если встречено число, оно должно быть превращено в целочисленное значение:

```
def factor: Parser[Int] = number ^^ { _.toInt } | ...
```

Здесь оператор `^^` применяет функцию `{ _.toInt }` к полученному числу.

> **Примечание.** Символ `^^` был выбран не потому, что он имеет какое-то особое значение. Просто он имеет более низкий приоритет, чем `~`, и более высокий, чем `|`.

В предположении, что `expr` изменится и будет возвращать `Parser[Int]`, мы сможем вычислять `"(" ~ expr ~ ")"`, просто возвращая `expr`, который дает значение типа `Int`. Ниже приводится одна из возможных реализаций; еще более простая реализация будет представлена в следующем разделе:

```
def factor: Parser[Int] = ... | "(" ~ expr ~ ")" ^^ {
     case _ ~ e ~ _ => e
}
```

В данном случае аргументом оператора ^^ является частично определенная функция (partial function) { case _ ~ e ~ _ => e }.

> **Примечание.** Чтобы упростить сопоставление, комбинатор ~ возвращает экземпляр case-класса ~, а не пару значений. Если бы ~ возвращал пару, тогда вместо case _ ~ e ~ _ пришлось бы записать case ((_, e), _).

Аналогичный прием сопоставления с образцом используется для вычисления суммы или разности. Обратите внимание, что opt возвращает значение типа Option: либо None, либо Some(...).

```
def expr: Parser[Int] = term ~ opt(("+" | "-") ~ expr) ^^ {
     case t ~ None => t
     case t ~ Some("+" ~ e) => t + e
     case t ~ Some("-" ~ e) => t - e
}
```

Наконец, что касается умножения, обратите внимание, что rep("*" ~ factor) возвращает список List элементов в форме "*" ~ f, где значение f имеет тип Int. Мы извлекаем второй компонент каждой пары ~ и вычисляем произведение:

```
def term: Parser[Int] = factor ~ rep("*" ~ factor) ^^ {
     case f ~ r => f * r.map(_._2).product
}
```

В этом примере мы просто вычисляем значение выражения. При построении компилятора или интерпретатора обычно целью является построение *дерева синтаксического анализа* – древовидной структуры, описывающей результат парсинга; см. раздел 19.5 «Создание деревьев синтаксического анализа».

> **Внимание.** opt(p) можно записать как p?, а rep(p) как p*, например:
>
> ```
> def expr: Parser[Any] = term ~ (("+" | "-") ~ expr)?
> def term: Parser[Any] = factor ~ ("*" ~ factor)*
> ```
>
> Использование знакомых операторов кажется более удобным, но они конфликтуют с оператором ^^. Чтобы устранить конфликт, необходимо добавить еще одну пару круглых скобок, например:
>
> ```
> def term: Parser[Any] = factor ~ (("*" ~ factor)*) ^^ { ... }
> ```
>
> По этой причине я предпочитаю opt и rep.

19.4. Отбрасывание лексем

Как было показано в предыдущем разделе, при анализе соответствий обработка лексем может оказаться утомительным занятием. Лексемы необходимы для парсинга, но после сопоставления их часто можно отбросить. Для сопоставления и удаления лексем используются операторы ~> и <~. Например, результатом выражения `"*" ~> factor` будет результирующий `factor`, а не значение в форме `"*" ~ f`. Благодаря такой форме записи можно упростить функцию `term`, как показано ниже:

```
def term = factor ~ rep("*" ~> factor) ^^ {
     case f ~ r => f * r.product
}
```

Аналогично можно отбросить скобки, окружающие выражение:

```
def factor = number ^^ { _.toInt } | "(" ~> expr <~ ")"
```

Дальнейшее преобразование `"(" ~> expr <~ ")"` не требуется, потому что его значением будет простое значение `e`, которое уже возвращает `Int`.

Обратите внимание, что операторы «подсказывающих стрелок», ~> и <~, указывают на часть, которая остается.

> **Внимание.** Необходимо проявлять особое внимание при использовании множества операторов ~, ~> и <~ в одном выражении. Оператор <~ имеет более низкий приоритет, чем ~ и ~>. Так, в следующем примере:
>
> `"if" ~> "(" ~> expr <~ ")" ~ expr`
>
> будет отброшена не скобка `")"`, а подвыражение `")" ~ expr`. Чтобы решить эту проблему, необходимо добавить дополнительные скобки:
>
> `"if" ~> "(" ~> (expr <~ ")") ~ expr.`

19.5. Создание деревьев синтаксического анализа

Парсеры в предыдущих примерах просто вычисляют числовые результаты. При разработке интерпретатора или компилятора вместо вычисления выражения требуется сконструировать дерево синтаксического анализа. Делается это обычно с помощью case-классов.

Например, следующие классы можно использовать для представления арифметических выражений:

```
class Expr
case class Number(value: Int) extends Expr
case class Operator(op: String, left: Expr, right: Expr) extends Expr
```

Задача парсера состоит в том, чтобы преобразовать исходное выражение, такое как `3+4*5`, в значение

```
Operator("+", Number(3), Operator("*", Number(4), Number(5))).
```

В интерпретаторе подобное выражение можно сразу вычислить. В компиляторе его можно превратить в программный код.

Для создания дерева синтаксического анализа используется оператор `^^` с функциями, возвращающими узлы дерева. Например:

```
class ExprParser extends RegexParsers {
    ...
    def term: Parser[Expr] = (factor ~ opt("*" ~> term)) ^^ {
        case a ~ None => a
        case a ~ Some(b) => Operator("*", a, b)
    }
    def factor: Parser[Expr] = wholeNumber ^^ (n => Number(n.toInt)) |
        "(" ~> expr <~ ")"
}
```

19.6. Уход от левой рекурсии

Если функция парсера, вызывающая сама себя, не поглощает некоторую часть входного текста, есть риск, что рекурсия никогда не прекратится. Рассмотрим функцию, которая, как предполагается, поглощает произвольную последовательность единиц:

```
def ones: Parser[Any] = "1" ~ rep("1")
```

Такие функции называются *леворекурсивными* (left-recursive). Чтобы избежать рекурсии, можно переформулировать грамматику. Ниже показаны две альтернативы:

```
def ones: Parser[Any] = "1" ~ ones | "1"
```

или

```
def ones: Parser[Any] = rep1("1")
```

Данная проблема часто возникает на практике. Например, взгляните на наш парсер арифметических выражений:

```
def expr: Parser[Any] = term ~ opt(("+" | "-") ~ expr)
```

Правило для `expr` имеет неприятный эффект, связанный с вычитанием. Выражения группируются в неправильном порядке. Если на вход подать выражение `3-4-5`, оно будет разобрано, как показано на рис. 19.1.

То есть `3` воспринимается как `term`, а `-4-5` как `"-" ~ expr`. В результате получается ошибочный ответ: `4` вместо `-6`.

А что, если изменить порядок определений в грамматике?

```
def expr: Parser[Any] = expr ~ opt(("+" | "-") ~ term)
```

В этом случае могло бы получиться правильное дерево синтаксического анализа. Но этот способ не работает, потому что функция `expr` является леворекурсивной.

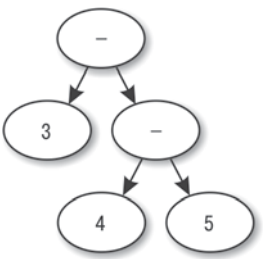


Рис. 19.1. Неправильный порядок группировки выражений

Оригинальная версия устраняет левую рекурсию, но ценой усложнения вычисления результата: необходимо собрать промежуточные результаты и выполнить оставшиеся операции в правильном порядке.

Сбор промежуточных результатов легче организовать, если воспользоваться повторениями, возвращающими списки `List` собранных значений. Например, `expr` – это последовательность значений `term`, объединяемых операторами «плюс» (`+`) и «минус» (`-`):

```
def expr: Parser[Any] = term ~ rep(("+" | "-") ~ term)
```

Чтобы вычислить выражение, замените каждую комбинацию `s ~ t` в повторении на `t` или `-t` в зависимости от операции «`+`» или «`-`». Затем вычислите сумму элементов в списке.

```
def expr: Parser[Int] = term ~ rep(
     ("+" | "-") ~ term ^^ {
          case "+" ~ t => t
          case "-" ~ t => -t
     }) ^^ { case t ~ r => t + r.sum }
```

Если переписать грамматику окажется слишком сложно, см. раздел 19.9 «Packrat-парсеры», где приводится еще одно решение.

19.7. Дополнительные комбинаторы

Метод `rep` обеспечивает ноль или более совпадений. В табл. 19.1 перечислено несколько разновидностей этого комбинатора, из которых наиболее часто используется `repsep`. Например, список чисел, разделенных запятыми, можно определить так:

```
def numberList = number ~ rep("," ~> number)
```

или более кратко:

```
def numberList = repsep(number, ",")
```

В табл. 19.2 перечислены дополнительные комбинаторы, которые также могут иногда пригодиться. Комбинатор `into` можно использовать для сохранения в переменной информации, полученной от предыдущего комбинатора, для последующего использования. Например, в правиле грамматики

```
def term: Parser[Any] = factor ~ rep("*" ~> factor)
```

первый сомножитель (`factor`) можно сохранить в переменной, как показано ниже:

```
def term: Parser[Int] = factor into { first =>
     rep("*" ~> factor) ^^ { first * _.product }
}
```

Комбинатор `log` можно задействовать для отладки грамматики. Замените парсер `p` на `log(p)(str)`, и при каждом вызове `p` будет выводиться соответствующая информация. Например,

```
def factor: Parser[Int] = log(number)("number") ^^ { _.toInt } | ...
```

выведет:

```
trying number at scala.util.parsing.input.CharSequenceReader@76f7c5
number --> [1.2] parsed: 3
```

Таблица 19.1. Комбинаторы повторений

Комбинатор	Описание	Примечания
`rep(p)`	0 или более совпадений с `p`	
`rep1(p)`	1 или более совпадений с `p`	`rep1("[" ~> expr <~ "]")` вернет список выражений, заключенных в квадратные скобки, например определяющих границы многомерного массива
`rep1(p, q)`	1 совпадение с `p`, за которым следует 0 или более совпадений с `q`	
`repN(n, p)`	`n` совпадений с `p`	`repN(4, number)` соответствует последовательности из четырех чисел, например определяющих прямоугольник

`repsep(p, s)`
`rep1sep(p, s)`
где `p` – это `Parser[P]`
	(0 или более)/(1 или более) совпадений с `p`, разделенных совпадениями с `s`; результатом является список `List[P]`, из которого исключены совпадения с `s`	`repsep(expr, ",")` вернет список выражений, которые были разделены запятыми; удобно использовать для парсинга аргументов в вызовах функций

`chainl1(p, s)`	Действует подобно `rep1sep`, но после совпадения с каждым разделителем аргумент `s` должен производить функцию с двумя параметрами, используемую для объединения соседних значений. Если `p` производит значения *v0*, *v1*, *v2*, . . ., а `s` производит функции *f1*, *f2*, . . . , тогда результатом будет *(v0 f1 v1) f2 v2* . . .	`chainl1(number ^^ { _.toInt }, "*" ^^^ { _ * _})` вычислит произведение последовательности целых чисел, разделенных оператором умножения `(*)`

Таблица 19.2. Дополнительные комбинаторы

Комбинатор	Описание	Примечания
`p ^^^ v`	Действует подобно `^^`, но возвращает постоянный результат	Удобно использовать для анализа литералов: `"true" ^^^ true`
`p into f` или `p >> f`	`f` – функция, аргументом которой является результат вызова `p`; удобно использовать для присвоения результата переменной	`(number ^^ { _.toInt }) >> { n => repN(n, number) }` выполнит разбор последовательности чисел, где первое число определяет количество чисел в последовательности
`p ^? f` `p ^? (f, error)`	Действует подобно `^^`, но принимает частично вычислимую функцию `f`. Завершается с ошибкой, если `f` нельзя применить к результату `p`. Во второй версии `error` – функция, которая в зависимости от типа результата `p` возвращает строку с сообщением об ошибке	`ident ^? (symbols, «undefined symbol « + _)` отыщет `ident` в ассоциативном массиве `symbols` и сообщит об ошибке, если искомый ключ отсутствует в ассоциативном массиве. Обратите внимание, что ассоциативный массив можно преобразовать в частично вычислимую функцию
`log(p)(str)`	Выполнит `p` и выведет сообщение	`log(number)(«number») ^^ { _.toInt }` будет выводить сообщение всякий раз, когда будет проанализировано очередное число
`guard(p)`	Вызовет `p` и, независимо от успеха или неудачи, восстановит входной поток данных, как если бы `p` не вызывался	Удобно для выполнения опережающих проверок. Например, чтобы отличить обращение к переменной от вызова функции, можно использовать `guard(ident ~ "(")`
`not(p)`	Вызовет `p`, и результат будет считаться успешным, если вызов `p` завершится неудачей, и наоборот	
`p ~! q`	Действует подобно `~`, но если сопоставление терпит неудачу, неудача превращается в ошибку, подавляющую возврат назад в пределах объемлющего `|`	См. раздел 19.8

Таблица 19.2. Дополнительные комбинаторы (окончание)

Комбинатор	Описание	Примечания
`accept(descr, f)`	Принимает элемент, принятый частично вычисленным функцией `f`, и возвращает результат функции. Строка `descr` используется для описания ожидаемого элемента в тексте сообщения об ошибке	`accept("string literal", { case t: lexical.StringLit => t.chars })`
`success(v)`	Всегда возвращает успех для значения `v`	Можно использовать для добавления элемента `v` в результат
`failure(msg)` `err(msg)`	Терпит неудачу с указанным сообщением об ошибке	См. раздел 19.13, где описывается, как улучшить вывод сообщений об ошибках
`phrase(p)`	Завершается успехом, если вызов `p` преуспевает и на входе ничего не остается	Удобно использовать для определения метода `parseAll`; см. пример в разделе 19.12
`positioned(p)`	Добавляет позицию в результат `p` (должен наследовать `Positional`)	Удобно использовать для вывода собщений об ошибках после завершения парсинга

19.8. Уход от возвратов

Всякий раз, когда выполняется выбор из альтернатив `p | q` и `p` терпит неудачу, делается попытка применить парсер `q` к тому же фрагменту исходных данных. Такие возвраты могут снижать эффективность парсинга. Например, рассмотрим парсер арифметических выражений со следующими правилами:

```
def expr: Parser[Any] = term ~ ("+" | "-") ~ expr | term
def term: Parser[Any] = factor ~ "*" ~ term | factor
def factor: Parser[Any] = "(" ~ expr ~ ")" | number
```

При парсинге выражения `(3+4)*5` оно целиком совпадет с `term`. Тогда попытка сопоставления с `+` или `-` потерпит неудачу и компилятор вернется, чтобы попробовать вторую альтернативу, вызвав `term` еще раз.

Часто, чтобы избежать возвратов, бывает достаточно просто переупорядочить правила грамматики. Например:

```
def expr: Parser[Any] = term ~ opt(("+" | "-") ~ expr)
def term: Parser[Any] = factor ~ rep("*" ~ factor)
```

Теперь вместо ~ можно использовать оператор ~!, чтобы указать на отсутствие необходимости выполнять возврат:

```
def expr: Parser[Any] = term ~! opt(("+" | "-") ~! expr)
def term: Parser[Any] = factor ~! rep("*" ~! factor)
def factor: Parser[Any] = "(" ~! expr ~! ")" | number
```

Когда при вычислении p ~! q сопоставление q терпит неудачу, другие альтернативы в охватывающем методе | исследоваться не будут. Например, если factor обнаружит "(" и затем expr не сможет найти совпадение, парсер даже не будет пытаться выполнить сопоставление с number.

19.9. Packrat-парсеры

Packrat-парсер использует эффективный алгоритм парсинга, кеширующий промежуточные результаты. В этом есть два преимущества:

- время парсинга прямо пропорционально длине исходного текста;
- парсер способен принимать леворекурсивные грамматики.

Чтобы задействовать packrat-парсинг в Scala, выполните следующие шаги:

1. Подмешайте трейт PackratParsers в свой парсер.
2. Определите все функции парсера не через def, а через val или lazy val. Это важно, потому что парсер кеширует эти значения и опирается на их идентичность. (Определение def каждый раз возвращает новое значение.)
3. Каждая функция парсера должна возвращать PackratParser[T] вместо Parser[T].
4. Используйте PackratReader и реализуйте метод parseAll (который по досадной случайности отсутствует в трейте PackratParsers).

Например:

```
class OnesPackratParser extends RegexParsers with PackratParsers {
    lazy val ones: PackratParser[Any] = ones ~ "1" | "1"

    def parseAll[T](p: Parser[T], input: String) =
```

```
        phrase(p)(new PackratReader(new CharSequenceReader(input)))
}
```

19.10. Что такое парсеры?

Строго говоря, `Parser[T]` – это функция с одним аргументом типа `Reader[Elem]`, возвращающая значение типа `ParseResult[T]`. В этом разделе мы поближе познакомимся с подобными типами.

Тип `Elem` – это абстрактный тип трейта `Parsers`. (Дополнительная информация об абстрактных типах приводится в разделе 18.12 «Абстрактные типы».) Трейт `RegexParsers` определяет тип `Elem` как `Char`, а трейт `StdTokenParsers` – как `Token`. (Парсеры на основе лексем будут рассматриваться в разделе 19.12 «Парсеры на основе лексем».)

Экземпляр `Reader[Elem]` читает последовательность значений типа `Elem` (то есть символов или лексем) из некоторого источника и запоминает их позиции на случай вывода сообщений об ошибках.

Когда функция `Parser[T]` вызывает механизм чтения, он возвращает объект одного из трех подклассов класса `ParseResult[T]`: `Success[T]`, `Failure` или `Error`.

Значение типа `Error` завершает работу парсера и все, что его вызывало. Это значение может быть возвращено в одном из трех случаев:

- парсер `p ~! q` завершился неудачей при сопоставлении с `q`;
- метод `commit(p)` завершился неудачей;
- встретился комбинатор `err(msg)`.

Значение `Failure` возникает в результате неудачи при сопоставлении; обычно это вызывает переход к другой альтернативе в охватывающем методе `|`.

Экземпляр `Success[T]` имеет свойство `result` типа `T`. Он также имеет свойство типа `Reader[Elem]` с именем `next`, содержащее фрагмент исходных данных, расположенный далее, за совпадением, которое осталось поглотить.

Взгляните на следующую часть реализации парсера арифметических выражений:

```
val number = "[0-9]+".r
def expr = number | "(" ~ expr ~ ")"
```

Наш парсер наследует `RegexParsers`, для которого существует неявное преобразование из `Regex` в `Parser[String]`. Регулярное выражение `number` преобразуется в такой парсер – функцию, поглощающую `Reader[Char]`.

Если начальные символы, возвращаемые механизмом чтения, соответствуют регулярному выражению, функция вернет `Success[String]`. Свойство `result` возвращаемого объекта содержит фрагмент исходного текста, совпавший с регулярным выражением, а свойство `next` – фрагмент исходного текста, оставшийся после сопоставления.

Если начальные символы, возвращаемые механизмом чтения, не совпадут с регулярным выражением, функция парсера вернет объект `Failure`.

Метод `|` объединяет два парсера. То есть если `p` и `q` являются функциями, тогда `p | q` также является функцией. Объединенная функция поглощает символы, возвращаемые механизмом чтения, например `r`. Она вызовет `p(r)`. Если этот вызов вернет результат типа `Success` или `Error`, он станет возвращаемым значением `p | q`. В противном случае возвращаемым значением станет результат вызова `q(r)`.

19.11. Парсеры на основе регулярных выражений

Трейт `RegexParsers`, использовавшийся до сих пор во всех примерах, обеспечивает два неявных преобразования для определяемых парсеров:

- `literal` преобразует строковый литерал (такой как `"+"`) в `Parser[String]`;
- `regex` преобразует регулярное выражение (такое как `"[0-9]".r`) в `Parser[String]`.

По умолчанию парсеры на основе регулярных выражений пропускают пробельные символы. Если в вашем случае под пробельным символом понимается нечто иное, чем `"""\s+""".r` (например, комментарии), замените `whiteSpace` своим определением. Если пробельные символы не должны пропускаться, используйте:

```
override val whiteSpace = "".r
```

Трейт `JavaTokenParsers` наследует `RegexParsers` и определяет пять лексем, перечисленных в табл. 19.3. Ни одна из них не имеет точного соответствия со своими формами в языке Java, что существенно снижает практическую пользу этого трейта.

Таблица 19.3. Предопределенные лексемы в JavaTokenParsers

Лексема	Регулярное выражение
ident	[a-zA-Z_]\w*
wholeNumber	-?\d+
decimalNumber	(\d+(\.\d*)?|\d*\.\d+
stringLiteral	"([^"\p{Cntrl}\\]|\\[\\/bfnrt]|\u[a-fA-F0-9]{4})*"
floatingPointNumber	-?(\d+(\.\d*)?|\d*\.\d+)([eE][+-]?\d+)?[fFdD]?

19.12. Парсеры на основе лексем

Вместо `Reader[Char]` парсеры на основе лексем используют `Reader[Token]`. Тип `Token` определен в трейте `scala.util.parsing.combinator.token.Tokens`. Наследующий его трейт `StdTokens` определяет четыре типа лексем, наиболее часто используемые при парсинге исходных текстов на языках программирования:

- `Identifier`;
- `Keyword`;
- `NumericLit`;
- `StringLit`.

Класс `StandardTokenParsers` предоставляет парсер, возвращающий эти лексемы. Идентификаторы состоят из букв, цифр или символа подчеркивания (_), но не могут начинаться с цифры.

> **Внимание.** Правила для букв и цифр несколько отличаются от тех, что используются в Java или Scala. Поддерживаются цифры из любых алфавитов, но буквы в «дополнительном» диапазоне (выше U+FFFF) исключены из рассмотрения.

Числовые литералы – это последовательности цифр. Строковые литералы – это последовательности символов, заключенные в кавычки `"..."` или `'...'`, без экранированных последовательностей (escape-последовательностей). Комментарии – последовательности символов, заключенные в `/* ... */` или от пары символов // до конца строки – считаются пробельными символами.

При наследовании от этого парсера добавьте зарезервированные слова и специальные лексемы в множества `lexical.reserved` и `lexical.delimiters`:

```
class MyLanguageParser extends StandardTokenParser {
    lexical.reserved += ("auto", "break", "case", "char", "const", ...)
```

```
    lexical.delimiters += ("=", "<", "<=", ">", ">=", "==", "!=", ...)
    ...
}
```

При встрече зарезервированного слова оно преобразуется в значение типа `Keyword`, а не `Identifier`.

Парсер сортирует разделители согласно правилу «захватить как можно больше». Например, когда исходный текст содержит последовательность `<=`, она будет возвращена как единая лексема, а не как последовательность лексем `<` и `=`.

Функция `ident` анализирует идентификатор; `numericLit` и `stringLit` анализируют литералы.

Например, ниже представлена наша грамматика арифметических выражений, построенная с помощью `StandardTokenParsers`:

```
class ExprParser extends StandardTokenParsers {
    lexical.delimiters += ("+", "-", "*", "(", ")")

    def expr: Parser[Any] = term ~ rep(("+" | "-") ~ term)
    def term: Parser[Any] = factor ~ rep("*" ~> factor)
    def factor: Parser[Any] = numericLit | "(" ~> expr <~ ")"

    def parseAll[T](p: Parser[T], in: String): ParseResult[T] =
        phrase(p)(new lexical.Scanner(in))
}
```

Обратите внимание на необходимость определить собственную реализацию метода `parseAll`, который по досадной случайности отсутствует в классе `StandardTokenParsers`. В этом методе используется экземпляр `lexical.Scanner`, который является реализацией `Reader[Token]`, предоставляемой трейтом `StdLexical`.

> **Совет.** Если потребуется реализовать парсинг языка с другими лексемами, для этой цели легко можно приспособить парсер на основе лексем. Унаследуйте трейт `StdLexical` и переопределите метод `token`, чтобы он распознавал требуемые лексемы. В качестве руководства используйте исходный код `StdLexical` – он достаточно короткий. Затем унаследуйте `StdTokenParsers` и переопределите свойство `lexical`:
>
> ```
> class MyParser extends StdTokenParsers {
>     val lexical = new MyLexical
>     ...
> }
> ```

Совет. Метод `token` в `StdLexical` имеет довольно сложную реализацию. Для определения лексем лучше использовать регулярные выражения. Добавьте следующее определение в свою реализацию, наследующую `StdLexical`:

```
def regex(r: Regex): Parser[String] = new Parser[String] {
     def apply(in: Input) = r.findPrefixMatchOf(
          in.source.subSequence(in.offset, in.source.length)) match {
               case Some(matched) =>
                    Success(in.source.subSequence(in.offset,
                         in.offset + matched.end).toString,
                         in.drop(matched.end))
               case None => Failure("string matching regex `" + r +
                         "' expected but " + in.first + " found", in)
     }
}
```

После этого в своем методе `token` вы сможете использовать регулярные выражения, например:

```
override def token: Parser[Token] = {
     regex("[a-z][a-zA-Z0-9]*".r) ^^ { processIdent(_) } |
     regex("0|[1-9][0-9]*".r) ^^ { NumericLit(_) } |
     ...
}
```

19.13. Обработка ошибок

Когда парсер не может принять исходный текст, вам может потребоваться вывести сообщение, как можно более точно описывающее проблему.

Парсер генерирует сообщение об ошибке, описывающее позицию, в которой парсер не смог продолжить синтаксический анализ. Если было встречено несколько ошибок в разных местах, сообщено будет о самой последней.

Собственные сообщения об ошибках могут оказаться желательными при парсинге альтернатив. Например, если имеется правило

```
def value: Parser[Any] = numericLit | "true" | "false"
```

и парсер не смог найти соответствие ни с одной из альтернатив, сообщение, извещающее, что произошла ошибка сопоставления с `"false"`, только внесет сумятицу. В подобной ситуации можно добавить предложение `failure` с явным текстом сообщения об ошибке:

```
def value: Parser[Any] = numericLit | "true" | "false" |
     failure("Not a valid value")
```

Когда парсер потерпит неудачу, метод `parseAll` вернет результат типа `Failure`. Его свойство `msg` будет содержать текст сообщения об ошибке, которое можно показать пользователю. Свойство `next` типа `Reader` будет указывать на позицию в исходном тексте, соответствующую точке возникновения ошибки. Благодаря этому в сообщении об ошибке можно вывести номер строки и позицию в строке, хранящиеся в свойствах `next.pos.line` и `next.pos.column`.

Наконец, свойство `next.first` будет хранить лексический элемент, вызвавший ошибку. При использовании трейта `RegexParsers` этот элемент будет иметь тип `Char`, что не очень полезно для сообщений об ошибках. Но при использовании парсера на основе лексем свойство `next.first` будет содержать лексему, которую имеет смысл добавить в текст сообщения.

> **Совет.** При необходимости сообщить об ошибках, обнаруженных после успешного парсинга (например, об ошибках, связанных с несовместимостью типов в языках программирования), для добавления информации о позиции можно воспользоваться комбинатором `positioned`. Типом результата должен быть тип, наследующий трейт `Positional`. Например:
>
> ```
> def vardecl = "var" ~ positioned(ident ^^ { Ident(_) }) ~ "=" ~ value
> ```

Упражнения

1. Добавьте в парсер арифметических выражений операции `/` и `%`.
2. Добавьте в парсер арифметических выражений оператор `^`. Как и в математике, оператор `^` должен иметь более высокий приоритет, чем оператор умножения, и должен быть правоассоциативным. То есть выражение `4^2^3` должно интерпретироваться как `4^(2^3)` и давать в результате значение `65536`.
3. Напишите парсер, анализирующий список целых чисел (такой как `(1, 23, -79)`) и возвращающий `List[Int]`.
4. Напишите парсер, анализирующий дату и время в формате ISO 8601. Парсер должен возвращать объект `java.util.Date`.
5. Напишите парсер, анализирующий подмножество XML. Он должен обрабатывать теги вида `<ident> ... </ident>` или `<ident/>`. Теги могут быть вложенными. Реализуйте обработку атрибутов тегов. Значения атрибутов могут ограничиваться апострофами

или двойными кавычками. Символьные данные не должны обрабатываться (то есть текст внутри тегов или разделы `CDATA`). Парсер должен возвращать значение Scala XML типа `Elem`. Основная сложность заключается в генерировании ошибок при встрече непарных тегов. Подсказка: используйте `into`, `accept`.

6. Допустим, что парсер из раздела 19.5 «Создание деревьев синтаксического анализа» дополнен следующим определением:

```
class ExprParser extends RegexParsers {
     def expr: Parser[Expr] = (term ~ opt(("+" | "-") ~ expr)) ^^ {
          case a ~ None => a
          case a ~ Some(op ~ b) => Operator(op, a, b)
     }
     ...
}
```

   К сожалению, этот парсер создает неправильное дерево синтаксического анализа – операторы с одинаковым приоритетом вычисляются справа налево. Измените парсер так, чтобы он генерировал правильное дерево синтаксического анализа. Например, для выражения `3-4-5` он должен возвращать `Operator("-", Operator("-", 3, 4), 5)`.

7. В разделе 19.6 «Уход от левой рекурсии» мы сначала выполняли парсинг `expr` в список `~` с операциями и значениями:

```
def expr: Parser[Int] = term ~ rep(("+" | "-") ~ term) ^^ {...}
```

   Чтобы получить результат, мы должны вычислить ((t0 ± t1) ± ± t2) ± ... . Реализуйте это вычисление как свертку (см. главу 13).

8. Добавьте поддержку переменных и операции присвоения в программу калькулятора. Переменные должны создаваться при первом их использовании. Неинициализированные переменные должны иметь нулевое значение. Для вывода значения оно должно присваиваться специальной переменной `out`.
9. Расширьте предыдущий пример, реализовав на его основе парсер для языка программирования, поддерживающего операцию присвоения значений переменным, логические выражения и инструкции `if/else` и `while`.
10. Добавьте в язык программирования из предыдущего примера поддержку определения функций.

Глава 20. Акторы

Темы, рассматриваемые в этой главе A3

- 20.1. Создание и запуск акторов.
- 20.2. Отправка сообщений.
- 20.3. Прием сообщений.
- 20.4. Отправка сообщений другим акторам.
- 20.5. Каналы.
- 20.6. Синхронные сообщения и Futures.
- 20.7. Совместное использование потоков.
- 20.8. Жизненный цикл акторов.
- 20.9. Связывание акторов.
- 20.10. Проектирование приложений с применением акторов.
- Упражнения.

Акторы (actors)[1] представляют собой механизм реализации параллельных вычислений, альтернативный традиционным механизмам на основе блокировок. Отсутствие блокировок и совместно используемых данных в акторах упрощает разработку программ и позволяет не опасаться взаимоблокировок и состояний гонки за ресурсами (race conditions). Библиотека Scala предоставляет простейшую реализацию модели акторов, которую мы обсудим в этой главе. Более совершенные реализации акторов доступны на сторонних ресурсах, таких как проект Akka (http://akka.io).

Основные темы этой главы:

- наследование класса `Actor` и реализация метода `act` в каждом акторе;
- отправка сообщений акторам с использованием конструкции `actor ! message`;
- отправка сообщений выполняется асинхронно, по принципу «отправил и забыл»;

---

[1] В этой главе описывается прежняя реализация акторов, однако, начиная с версии 2.10, в Scala была включена поддержка акторов Akka Actors, а прежняя реализация объявлена устаревшей (дополнительную информацию см. на странице http://docs.scala-lang.org/actors-migration/). – *Прим. ред.*

- прием сообщений акторами выполняется вызовом метода `receive` или `react`, обычно в цикле;
- аргументом методов `receive`/`react` является блок альтернатив `case` (технически – частично определенная функция);
- акторы не должны совместно использовать какие-либо данные – данные всегда должны отправляться в виде сообщений;
- не вызывайте методы акторов, взаимодействуйте с ними посредством сообщений;
- избегайте синхронного обмена сообщениями – не связывайте воедино отправку сообщения и ожидание ответа на него;
- акторы могут совместно использовать потоки выполнения (threads), используя метод `react` вместо `receive`, если поток управления обработчика сообщений достаточно прост;
- не пугайтесь отказов в акторах, если имеются другие акторы, контролирующие их аварийное завершение; используйте прием связывания для настройки отношений контроля между ними.

20.1. Создание и запуск акторов

Актор (actor) – это класс, наследующий трейт `Actor`. Этот трейт имеет один абстрактный метод `act`. Реализуя этот метод, вы определяете поведение актора.

Обычно метод `act` содержит цикл приема сообщений.

```
import scala.actors.Actor

class HiActor extends Actor {
     def act() {
          while (true) {
               receive {
                    case «Hi» => println(«Hello»)
               }
          }
     }
}
```

Метод `act` похож на метод `run` интерфейса `Runnable` в Java. Как и методы `run`, выполняющиеся в различных потоках выполнения, методы `act` разных акторов выполняются параллельно друг другу. Однако оптимизированы для обработки сообщений, тогда как по-

токи выполнения могут выполнять произвольные операции. (См. раздел 20.3 «Прием сообщений», где описывается метод `receive`.)

Чтобы запустить актор, создайте экземпляр и вызовите метод `start`:

```
val actor1 = new HiActor
actor1.start()
```

После этого метод `act` актора `actor1` будет выполняться параллельно, и вы можете начать посылать ему сообщения. Поток выполнения, созданный в результате вызова метода `start`, будет продолжать работу.

Иногда бывает полезно создавать акторы «на лету», не определяя отдельного класса. Создать новый актор и запустить его можно вызовом метода `actor` объекта-компаньона `Actor`:

```
import scala.actors.Actor._

val actor2 = actor {
     while (true) {
          receive {
               case «Hi» => println(«Hello»)
          }
     }
}
```

> **Примечание.** Иногда возникает необходимость из анонимного актора отправить ссылку на него другому актору. Эта ссылка доступна в виде свойства `self`.

20.2. Отправка сообщений

Актор – это объект, обрабатывающий асинхронные сообщения. Вы посылаете сообщение актору, а он его обрабатывает и, возможно, посылает другому актору для дальнейшей обработки.

Сообщение может быть любым объектом. Например, акторы в предыдущем разделе выполняют некоторые операции при получении строки `"Hi"`.

Отправка сообщения выполняется с помощью оператора `!`, определенного для акторов:

```
actor1 ! "Hi"
```

После отправки сообщения актору поток выполнения продолжает свою работу. Это называется: «отправил и забыл». Кроме того, актор может приостановиться, чтобы дождаться ответа (хотя это нетипично для акторов), – см. раздел 20.6 «Синхронные сообщения и Futures».

На роль сообщений отлично подходят case-классы. Благодаря этому для обработки сообщения актор может использовать сопоставление с образцом.

Например, допустим, что имеется актор, проверяющий, не выполняется ли мошенническая операция с кредитной картой. Мы можем послать ему сообщение о том, что выполняется операция зачисления на счет. Ниже приводится соответствующий case-класс:

```
case class Charge(creditCardNumber: Long, merchant: String, amount: Double)
```

Отправка объекта case-класса актору выполняется следующим образом:

```
fraudControl ! Charge(4111111111111111L, "Fred's Bait and Tackle", 19.95)
```

Допустим, что метод `act` актора содержит инструкцию следующего вида:

```
receive {
     case Charge(ccnum, merchant, amt) => ...
}
```

Тогда значения, используемые в операции зачисления, будут доступны в переменных `ccnum`, `merchant` и `amt`.

20.3. Прием сообщений

Сообщения, посылаемые актору, попадают в его «почтовый ящик». При вызове метод `receive` извлекает из почтового ящика очередное сообщение и передает его своему аргументу – частично определенной функции. Например:

```
receive {
     case Deposit(amount) => ...
     case Withdraw(amount) => ...
}
```

Здесь аргументом метода `receive` является:

```
{
     case Deposit(amount) => ...
     case Withdraw(amount) => ...
}
```

Этот блок преобразуется в объект типа `PartialFunction[Any, T]`, где `T` – тип результата, вычисляемого правыми частями выражений `case`. Это – «частично определенная» функция, потому что она определена только для аргументов, соответствующих предложениям case.

Метод `receive` передает сообщения из почтового ящика указанной частично определенной функции.

> **Примечание.** Доставка сообщений выполняется асинхронно. Порядок, в каком они будут доставлены, заранее неизвестен, поэтому вы должны проектировать свои приложения так, чтобы они не зависели от какого-то определенного порядка доставки.

Если на момент вызова метода `receive` сообщение недоступно, он блокируется до момента получения сообщения.

Если в почтовом ящике не окажется сообщений, пригодных для обработки частично определенной функцией, вызов `receive` также будет заблокирован до появления соответствующего сообщения.

> **Внимание.** Вполне возможно, что почтовый ящик окажется доверху заполнен сообщениями, не совпадающими ни с одним из предложений `case`. На этот случай можно добавить предложение `case _` для обработки произвольных сообщений.

Сообщения хранятся в почтовом ящике в виде последовательности. Актор действует в единственном потоке. Он извлекает сначала первое сообщение, затем следующее и т. д. В теле актора не приходится беспокоиться о состоянии «гонки за ресурсами» (race conditions). Например, допустим, что имеется актор, обновляющий баланс:

```
class AccountActor extends Actor {
     private var balance = 0.0

     def act() {
          while (true) {
```

```
            receive {
                case Deposit(amount) => balance += amount
                case Withdraw(amount) => balance -= amount
                ...
            }
        }
    }
}
```

Нет никакой опасности смешивания операций прихода и расхода.

> **Внимание.** Актор может без опаски изменять собственные данные. Но если он изменяет данные, совместно используемые несколькими акторами, тогда может возникнуть состояние гонки за ресурсами.

Не используйте один и тот же объект в нескольких акторах, если не уверены в безопасности выполнения параллельных операций над ним. В идеале акторы никогда не должны обращаться к данным или изменять их, кроме своих собственных. Однако Scala не принуждает к соблюдению этого правила.

20.4. Отправка сообщений другим акторам

Когда вычисления распределяются между множеством акторов, каждый из которых решает свою часть общей задачи, в конце необходимо собрать результаты воедино. Акторы могли бы сохранять свои результаты в общей структуре, поддерживающей возможность использования в многопоточной среде, такой как параллельный хеш, но модель акторов не приветствует использование разделяемых данных. Альтернативное решение заключается в том, чтобы акторы, вычисляющие результаты, посылали сообщения другому актору, выполняющему сборку результатов.

Как актор узнает, куда посылать результаты? На выбор есть несколько шаблонов проектирования.

1. В программе может иметься ряд глобальных акторов. Однако такое решение не масштабируемо для случая с большим количеством акторов.
2. Ссылка на один или более акторов, выполняющих сборку результатов, может быть передана актору при его создании.

3. Ссылку на другой актор можно передавать в виде сообщения. В практике часто используется прием передачи ссылки на актор в запросе, например:

```
actor ! Compute(data, continuation)
```

где `continuation` – другой актор, которому должны быть отправлены результаты вычислений.

4. Актор может возвращать сообщение отправителю. Метод `receive` сохраняет в поле `sender` ссылку на отправителя текущего сообщения.

Внимание. Когда актору передается ссылка на другой актор, эта ссылка должна использоваться только для отправки сообщений, но не для вызова методов актора. Мало того, что это нарушает основополагающие принципы модели акторов, но и может привести к состоянию гонки за ресурсами – проблеме, для решения которой и создавались акторы.

20.5. Каналы

Вместо ссылок на акторы в приложении можно использовать *каналы* (channels) к акторам. Этот подход имеет два преимущества.

1. Каналы обеспечивают безопасность на уровне типов – они могут отправлять или принимать сообщения только определенного типа.
2. Каналы исключают вероятность вызова методов акторов по неосторожности.

Канал может быть экземпляром трейта `OutputChannel` (с методом `!`) или `InputChannel` (с методом `receive` или `react`). Класс `Channel` наследует оба трейта, `OutputChannel` и `InputChannel`.

Чтобы создать канал, ему нужно передать актор:

```
val c = new Channel[Int](someActor)
```

При вызове конструктора без параметра канал будет подключен к текущему актору.

Акторам, которые должны возвращать результаты, обычно передается канал вывода:

```
case class Compute(input: Seq[Int], result: OutputChannel[Int])
class Computer extends Actor {
     while (true) {
```

```
          receive {
               case Compute(input, out) => {val answer = ...; out ! answer}
          }
     }
}

actor {
     val c = new Channel[Int]
     val computeActor: Computer = ...
     val input: Seq[Double] = ...
     computeActor ! Compute(input, channel)
     channel.receive {
          case x => ... // известно, что x имеет тип Int
     }
}
```

Внимание. Обратите внимание, что здесь вызывается метод `receive` канала, а не самого актора. Если необходимо принимать сообщения непосредственно от актора, добавьте сопоставление с экземпляром ! case-класса, как показано ниже:

```
receive {
     case !(c, x) => ...
}
```

20.6. Синхронные сообщения и Futures

С помощью оператора `?!` актор может отправить сообщение и ждать ответа. Например:

```
val reply = account !? Deposit(1000)
reply match {
     case Balance(bal) => println("Current Balance: " + bal)
}
```

Чтобы этот прием сработал, получатель должен вернуть сообщение отправителю:

```
receive {
     case Deposit(amount) => { balance += amount; sender ! Balance(balance) }
     ...
}
```

Отправитель блокируется, пока ответ не будет получен.

Примечание. Вместо `sender ! Balance(balance)` можно записать `reply(Balance(balance))`.

Внимание: Применение синхронных сообщений легко может привести к взаимоблокировкам. В общем случае лучше избегать блокирования вызовов внутри метода `act` актора.

Едва ли можно назвать надежной систему, которая может остановиться «навечно» в ожидании ответа. Чтобы этого не произошло, используйте метод `receiveWithin`, которому можно указать максимальное время ожидания в миллисекундах.

Если в течение указанного промежутка времени сообщение не появится, актор получит сообщение с объектом `Actor.TIMEOUT`.

```
actor {
     worker ! Task(data, self)
     receiveWithin(seconds * 1000) {
          case Result(data) => ...
          case TIMEOUT => log(...)
     }
}
```

Примечание. Существует также версия метода `react` с ограничением по времени, с именем `reactWithin`. (Описание метода `react` см. в следующем разделе.)

Вместо ожидания ответа можно организовать прием результата в будущем – объекта *future*, который вернет результат, когда он будет готов. Для этого можно использовать метод `!!`:

```
val replyFuture = account !! Deposit(1000)
```

Метод `isSet` позволяет проверить доступность результата. Для получения результата используется нотация вызова функции:

```
val reply = replyFuture()
```

Этот вызов может быть заблокирован до получения ответа.

Примечание. Если пытаться извлекать результат из объекта future немедленно, будут утрачены все преимущества перед синхронными вызовами методов. Однако актор может отложить обработку объекта future на более позднее время или передать его другому актору.

20.7. Совместное использование потоков выполнения

Представьте актор, отправляющий сообщение другому актору. Поток управления легче реализовать, если каждый актор действует в отдельном потоке выполнения. Актор-отправитель помещает сообщение в почтовый ящик, и его поток выполнения (thread) продолжает работу. Поток выполнения актора-получателя автоматически возобновляет работу всякий раз, когда в почтовом ящике появляется новое сообщение.

Некоторые программы содержат такое большое количество акторов, что было бы слишком накладно выделять отдельный поток выполнения для каждого из них. Можно ли организовать управление несколькими акторами в одном потоке выполнения? Если предположить, что большую часть времени актор проводит в ожидании сообщений, тогда вместо того, чтобы заключать каждый актор в отдельный поток выполнения, можно было бы создать единственный поток, выполняющий функции обработки сообщений нескольких акторов. Такое решение оправдано, если функция-обработчик основное время тратит на ожидание следующего сообщения.

Подобная организация больше похожа на архитектуру, управляемую событиями. Однако реализации механизмов обработки событий страдают от проблемы «инверсии управления» (inversion of control). Представьте, например, что актор ожидает получить сначала сообщение одного типа, а затем другого. Чтобы не писать линейный код, принимающий сначала одно событие, а затем другое, в обработчике события приходится следить за тем, какое событие уже было принято.

В Scala имеется более удачное решение. Метод `react` принимает частично определенную функцию, добавляет ее в почтовый ящик и выходит. Допустим, у нас имеются две вложенные инструкции `react`:

```
react {                              // Частично определенная функция f1
     case Withdraw(amount) =>
          react {                    // Частично определенная функция f2
               case Confirm() =>
                    println("Confirming " + amount)
          }
}
```

Здесь я дал имена частично определенным функциям, аргументам метода `react`, которые буду использовать для объяснения того, как действует поток управления.

Первый вызов react устанавливает связь между функцией f1 и почтовым ящиком актора и выходит. При появлении сообщения Withdraw вызывается функция f1, которая, в свою очередь, вызывает метод react. Этот вызов react устанавливает связь между функцией f2 и почтовым ящиком актора и выходит. При появлении сообщения Confirm вызывается функция f2.

> **Примечание.** Второй вызов react может быть выполнен в отдельной функции, потому что для выхода react возбуждает исключение.

Частично определенная функция, переданная методу react, не возвращает значение – она выполняет некоторые действия и наталкивается на следующий вызов react, который производит выход. Под выходом здесь подразумевается возврат в метод, координирующий работу актора. Типом возвращаемого значения такой функции является Nothing – тип, свидетельствующий о ненормальном завершении.

Поскольку метод react осуществляет принудительный выход, его нельзя просто поместить в цикл while, как показано ниже:

```
def act() {
     while (true) {
          react { // Частично определенная функция f1
               case Withdraw(amount) => println("Withdrawing " + amount)
          }
     }
}
```

Когда будет вызван метод act, он вызовет метод react, связывающий функцию f1 с почтовым ящиком, и выйдет. Когда позднее будет вызвана функция f1, она обработает сообщение, но не сможет вернуться в цикл – это всего лишь маленькая функция:

```
{ case Withdraw(amount) => println("Withdrawing " + amount) }
```

Исправить проблему можно повторным вызовом метода act в обработчике сообщений:

```
def act() {
     react { // Частично определенная функция f1
          case Withdraw(amount) => {
               println("Withdrawing " + amount)
```

```
                act()
            }
        }
}
```

Этот прием подменяет бесконечный цикл бесконечной рекурсией.

> **Примечание.** Этот вид рекурсии не влечет бесконтрольного потребления памяти для стека. Каждый вызов `react` завершается исключением, которое приводит к очистке стека.

Возлагать обязанность продолжать цикл на каждый обработчик сообщений не совсем правильно. Для этой цели существуют специальные «комбинаторы управления выполнением» (control flow combinators), которые автоматически образуют циклы.

Комбинатор `loop` образует бесконечный цикл:

```
def act() {
    loop {
        react {
            case Withdraw(amount) => process(amount)
        }
    }
}
```

Для создания циклов с условием используется комбинатор `loopWhile`:

```
loopWhile(count < max) {
    react {
        ...
    }
}
```

> **Примечание.** Метод `eventloop` является упрощенной версией бесконечного цикла для `react`, который выполняется, пока частично определенная функция не вызовет `react` еще раз.
>
> ```
> def act() {
>     eventloop {
>         case Withdraw(amount) => println("Withdrawing " + amount)
>     }
> }
> ```

20.8. Жизненный цикл акторов

Метод `act` актора запускается, когда вызывается его метод `start`. Обычно после этого актор входит в цикл, такой как

```
def act() {
     while (есть что делать) {
          receive {
               ...
          }
     }
}
```

Работа актора завершается в одном из трех случаев:

1. Когда метод `act` возвращает управление.
2. Когда работа метода `act` прерывается исключением.
3. Когда актор вызывает метод `exit`.

> **Примечание.** Метод `exit` – это защищенный (protected) метод. Он должен вызываться из подклассов класса `Actor`. Например:
>
> ```
> val actor1 = actor {
>      while (true) {
>           receive {
>                case "Hi" => println("Hello")
>                case "Bye" => exit()
>           }
>      }
> }
> ```

Другие методы не могут вызывать `exit()` для завершения работы актора.

Метод `exit` имеет необязательный аргумент, описывающий *причину выхода*. Вызов `exit` без аргумента эквивалентен вызову `exit('normal)`.

Когда актор завершает работу в результате исключения, роль описания причины выхода играет экземпляр case-класса `UncaughtException`. Этот класс имеет следующие свойства:

- `actor`: актор, возбудивший исключение;
- `message`: `Some(msg)`, где `msg` является последним сообщением, обработанным актором, или `None`, если актор завершился до того, как успел обработать хотя бы одно сообщение;
- `sender`: `Some(channel)`, где `channel` является каналом вывода, представляющим отправителя последнего сообщения, или `None`, ес-

ли актор завершился до того, как успел обработать хотя бы одно сообщение;

- `thread`: поток выполнения, в котором действовал завершившийся актор;
- `cause`: исключение.

Как извлекать причину выхода, будет показано в следующем разделе.

Примечание. По умолчанию любое необработанное исключение вызывает завершение работы актора с причиной `UnhandledException`. В большинстве случаев в этом есть определенный смысл. Однако у вас есть возможность изменить это поведение, переопределив метод `exceptionHandler`. Данный метод должен возвращать значение типа `PartialFunction[Exception, Unit]`. Если частично определенная функция применима к исключению, она будет вызвана, и актор завершится с причиной `'normal`. Например, если вы не считаете неконтролируемые исключения (unchecked exception) чем-то ненормальным, добавьте такой обработчик:

```
override def exceptionHandler = {
     case e: RuntimeException => log(e)
}
```

20.9. Связывание акторов

Если *связать* два актора, каждый из них получит извещение, когда другой завершит работу. Чтобы установить связь, достаточно вызвать метод `link` со ссылкой на другой актор.

```
def act() {
     link(master)
     ...
}
```

Он устанавливает двунаправленную связь. Например, если имеется главный актор, распределяющий работу среди нескольких подчиненных акторов, он должен знать, когда какой-то из подчиненных акторов завершается, чтобы его работу можно было передать другому актору. Напротив, если завершается главный актор, подчиненные акторы должны узнать об этом, чтобы прекратить вычисления.

Внимание. Даже при том, что метод `link` устанавливает двунаправленную связь, он не является симметричным. Нельзя заменить вызов `link(worker)` на `worker.link(self)`. Метод должен вызываться актором, запрашивающим установление связи.

По умолчанию актор завершается, если связанный актор завершается с причиной, отличной от `'normal`. В этом случае в качестве причины выхода принимается причина выхода связанного актора.

Изменить это поведение можно, установив свойство `trapExit` в значение `true`. После этого актор, принявший сообщение типа `Exit`, сможет получить завершившийся актор и причину выхода.

```
override def act() {
     trapExit = true
     link(worker)
     while (...) {
          receive {
               ...
               case Exit(linked, UncaughtException(_, _, _, _, cause)) => ...
               case Exit(linked, reason) => ...
          }
     }
}
```

При работе с акторами их неожиданное завершение является нормальным явлением. Просто свяжите все акторы с «главным» актором, который будет обслуживать завершение подчиненных акторов и перераспределять работу или перезапускать их.

В больших системах акторы можно объединять в группы, каждой из которых управляет свой главный актор.

> **Примечание.** После завершения актор сохраняет внутреннюю информацию и почтовый ящик. Если потребуется извлечь имеющиеся сообщения, можно перезапустить завершившийся актор. Но перед этим, возможно, придется восстановить его внутреннее состояние или установить флаг, указывающий, что теперь актор выполняется в спасательном режиме (salvage mode). Также придется переустановить все связи с этим актором, потому что по завершении все связи автоматически разрываются.

20.10. Проектирование приложений с применением акторов

Теперь вы знакомы с механизмом работы акторов, но знание правил ничего не говорит о том, как правильно организовать приложение, использующее акторы. Ниже приводится несколько советов:

1. Избегайте совместно используемых данных. Актор не должен обращаться ни к каким данным, кроме своих переменных

экземпляра. Все необходимые данные должны передаваться в виде сообщений.

Остерегайтесь использования изменяемых структур данных в сообщениях. Если актор отправляет ссылку на изменяемый объект другому актору, оба актора смогут пользоваться одним и тем же объектом. Если это действительно необходимо, тогда актор-отправитель должен немедленно очищать свою копию ссылки, тем самым целиком и полностью отдавая объект актору-получателю.

2. Не вызывайте методы акторов. Если один актор будет вызывать методы другого актора, вы получите те же самые проблемы синхронизации, которые требуют применения блокировок в традиционном параллельном программировании. Используйте каналы, чтобы избежать искушения.
3. Сохраняйте реализацию акторов как можно более простой. В программах на основе акторов проще разбираться, если каждый актор выполняет простые операции в цикле – прием задания, вычисление ответа и его отправка следующему актору.
4. Включайте в сообщения контекстные данные. Актор должен иметь возможность правильно интерпретировать сообщение, находясь в изоляции, не имея возможности отслеживать связанные сообщения. Например, разбивая крупную задачу на n фрагментов, совсем нелишним будет указать порядковый номер фрагмента i из n.
5. Минимизируйте ответы, возвращаемые отправителю. Акторы не предназначены играть роль удаленных процедур. Работа должна распределяться так, чтобы сеть акторов вычисляла промежуточные результаты и передавала их другим акторам, реализующим их объединение.
6. Минимизируйте количество блокируемых вызовов методов. Когда актор блокируется, он не может принимать сообщения, но акторы, отправляющие сообщения, не знают об этом. Как результат сообщения будут накапливаться в почтовом ящике заблокированного актора. При реализации ввода/вывода подумайте об использовании неблокирующих (асинхронных) операций ввода/вывода, таких как асинхронные каналы в Java. Избегайте синхронных вызовов. Они блокируются и могут привести к состоянию взаимоблокировки.
7. Используйте метод `react`, когда это возможно. Акторы, использующие метод `react`, могут совместно действовать в одном

потоке выполнения. Метод `react` можно использовать во всех случаях, когда обработчик сообщения выполняет некоторые действия и выходит.

8. Предусматривайте возможность аварийного завершения акторов. В аварийном завершении актора нет ничего необычного, необходимо лишь выделить специальный актор, который будет выполнять необходимые сопутствующие операции. Объединяйте взаимосвязанные акторы в группы и создавайте главный актор в каждой из них. Единственной обязанностью главного актора должно быть управление аварийно завершившимися акторами.

Упражнения

1. Напишите программу, генерирующую массив из n случайных чисел (где n – достаточно большое значение, например 1 000 000) и затем вычисляющую среднее значение, распределяя работу между несколькими акторами, каждый из которых вычисляет сумму для фрагмента массива и отправляет результат актору, объединяющему результаты.
   Какой прирост в скорости получится при выполнении программы на двух- или четырехъядерном процессоре в сравнении с однопоточной реализацией?
2. Напишите программу, которая читает большое изображение в экземпляр `BufferedImage` с помощью `javax.imageio.IOImage.read`. Реализуйте несколько акторов, каждый из которых инвертирует цвета в своей полосе изображения. После обработки всех полос сохраните результат.
3. Напишите программу, подсчитывающую количество совпадений с заданным регулярным выражением во всех файлах, во всех подкаталогах, начиная с указанного каталога. Создайте по одному актору для каждого файла, один актор для обхода дерева подкаталогов и один актор для сбора результатов.
4. Измените программу из предыдущего упражнения так, чтобы она выводила найденные совпадения.
5. Измените программу из предыдущего упражнения так, чтобы для каждого совпадения выводился список всех файлов, где это совпадение было найдено.
6. Напишите программу, создающую 100 акторов, использующих цикл `while(true)/receive` и вызывающих `println(Thread.`

`currentThread)` при получении сообщения `'Hello`, и 100 акторов, делающих то же самое с использованием конструкции `loop/react`. Запустите их все и отправьте всем сообщение. Сколько потоков выполнения было создано в первом случае и во втором?

7. Добавьте главный актор в программу из упражнения 3, который следил бы за акторами, выполняющими чтение файлов, и регистрировал бы случаи их завершения в результате исключения `IOException`. Попробуйте искусственно способствовать возбуждению исключения, удаляя файлы, запланированные для обработки.
8. Покажите, как программа на основе акторов может попасть в ситуацию взаимоблокировки, когда все акторы используют только синхронные сообщения.
9. Напишите ошибочную версию программы из упражнения 3, в которой акторы изменяют совместно используемый счетчик. Сможете ли вы показать, что программа действует неправильно?
10. Перепишите программу из упражнения 1, организовав взаимодействия посредством каналов.

Глава 21. Неявные параметры и преобразования

Темы, рассматриваемые в этой главе L3

- 21.1. Неявные преобразования.
- 21.2. Использование неявных преобразований для расширения существующих библиотек.
- 21.3. Импорт неявных преобразований.
- 21.4. Правила неявных преобразований.
- 21.5. Неявные параметры.
- 21.6. Неявные преобразования с неявными параметрами.
- 21.7. Границы контекста.
- 21.8. Неявный параметр подтверждения.
- 21.9. Аннотация `@implicitNotFound`.
- 21.10. Тайна `CanBuildFrom`.
- Упражнения.

Неявные преобразования и неявные параметры являются мощным инструментом языка Scala, выполняющим большой объем работ за кулисами. В этой главе вы узнаете, как использовать неявные преобразования для расширения возможностей существующих классов и как неявные объекты автоматически вызываются для выполнения преобразований и решения других задач. Применяя неявные преобразования, можно писать весьма элегантные библиотеки, скрывающие ненужные подробности от своих пользователей.

Основные темы этой главы:

- неявные преобразования используются для преобразования типов объектов;
- чтобы неявные преобразования были доступны в виде простых идентификаторов, их следует импортировать;
- неявный параметр требует наличия объекта данного типа; такой объект можно получить из неявного объекта, определяемого в области видимости в виде простого идентификатора, или из объекта-компаньона требуемого типа;

- если неявный параметр является функцией с единственным аргументом, он также используется как неявное преобразование;
- граница контекста параметра типа требует наличия неявного объекта данного типа;
- если есть возможность отыскать неявный объект, он может служить доказательством допустимости преобразования типа.

21.1. Неявные преобразования

Функция неявного преобразования (implicit conversion function) – это функция с единственным параметром, объявленным с ключевым словом `implicit`. Как можно предположить из названия, такая функция автоматически применяется для преобразования значений из одного типа в другой.

Ниже приводится простой пример – функция, преобразующая целое число n в рациональное $n/1$.

```
implicit def int2Fraction(n: Int) = Fraction(n, 1)
```

Теперь мы получаем возможность определять такие вычисления:

```
val result = 3 * Fraction(4, 5) // Вызовет int2Fraction(3)
```

Функция неявного преобразования превратит целое число 3 в объект `Fraction`, а затем будет выполнено умножение этого объекта на значение `Fraction(4, 5)`.

Функции преобразования можно дать любое имя. Поскольку эта функция не будет вызываться явно, может появиться соблазн дать ей короткое имя, такое как `i2f`. Но, как будет показано в разделе 21.3 «Импорт неявных преобразований», иногда бывает необходимо импортировать функции преобразований. Поэтому при выборе имен я предлагаю придерживаться соглашения *source2target*.

Scala – не первый язык, позволяющий программисту определять автоматические преобразования. Но Scala дает программисту более полный контроль над тем, когда применять эти преобразования. В следующих разделах будет говориться о том, когда именно производятся преобразования и как можно управлять этим процессом.

Примечание. В C++ неявные преобразования можно определять в виде конструкторов с единственным аргументом или функций-членов с именем `operator Type()`. Однако в C++ нельзя выборочно запрещать или разрешать использование этих функций, что часто приводит к нежелательным преобразованиям.

21.2. Использование неявных преобразований для расширения существующих библиотек

Приходилось ли вам когда-нибудь сожалеть об отсутствии в классе некоторого метода? Например, было бы весьма кстати, если бы в классе `java.io.File` имелся метод `read` для чтения файла:

```
val contents = new File("README").read
```

Как программисту на Java, вам остается только обратиться с просьбой в корпорацию Oracle Corporation, чтобы они добавили этот метод. Удачи!

В Scala имеется возможность определить расширенный тип, реализующий все, что вам необходимо:

```
class RichFile(val from: File) {
    def read = Source.fromFile(from.getPath).mkString
}
```

И объявить неявное преобразование в этот тип:

```
implicit def file2RichFile(from: File) = new RichFile(from)
```

После этого у вас появится возможность вызывать метод `read` для объектов типа `File`, которые будут неявно преобразовываться в объекты типа `RichFile`.

21.3. Импорт неявных преобразований

Компилятор Scala автоматически обнаруживает функции неявного преобразования:

1) в объекте-компаньоне исходного или целевого типа.
2) определенные в текущей области видимости (то есть видимые как простые идентификаторы).

Рассмотрим в качестве примера функцию `int2Fraction`. Ее можно поместить в объект-компаньон `Fraction`, и она будет доступна для преобразования целых чисел в рациональные.

С другой стороны, допустим, что мы поместили ее в объект `FractionConversions`, который определен в пакете `com.horstmann.impatient`.

Если в программе возникнет потребность в этом преобразовании, можно импортировать объект `FractionConversions`, как показано ниже:

```
import com.horstmann.impatient.FractionConversions._
```

Следующей инструкции импорта будет недостаточно:

```
import com.horstmann.impatient.FractionConversions
```

Она импортирует объект `FractionConversions`, и метод `int2Fraction` будет доступен под именем `FractionConversions.int2Fraction` любому, кто пожелает выполнить преобразование явно. Но если функция недоступна под простым, неквалифицированным идентификатором `int2Fraction`, компилятор не будет использовать ее.

> **Совет.** Введите команду `:implicits` в интерактивной оболочке REPL, чтобы получить список неявных преобразований, импортированных из источников, отличных от `Predef`, или команду `:implicits -v`, чтобы получить список всех неявных преобразований.

Для минимизации количества нежелательных преобразований импорт можно локализовать. Например:

```
object Main extends App {
     import com.horstmann.impatient.FractionConversions._
     val result = 3 * Fraction(4, 5)
          // Использует импортированное преобразование
     println(result)
}
```

Имеется даже возможность выборочно импортировать только необходимые преобразования. Допустим, что имеется еще одно преобразование:

```
object FractionConversions {
     ...
     implicit def fraction2Double(f: Fraction) = f.num * 1.0 / f.den
}
```

Если вы предпочтете это преобразование перед `int2Fraction`, его можно импортировать как:

```
import com.horstmann.impatient.FractionConversions.fraction2Double
val result = 3 * Fraction(4, 5) // result получит значение 2.4
```

Кроме того, имеется возможность исключать отдельные преобразования из импорта:

```
import com.horstmann.impatient.FractionConversions.{fraction2Double => _, _}
     // Импортирует все, кроме fraction2Double
```

> **Совет.** Если вы ищите причину, почему компилятор не использует неявное преобразование, которое, по вашему мнению, должен использовать, попробуйте выполнить преобразование явно, например вызвав `fraction2Double(3) * Fraction(4, 5)`. В результате вы можете получить сообщение об ошибке, проясняющее проблему.

21.4. Правила неявных преобразований

Неявные преобразования применяются в трех разных ситуациях:

- если тип выражения отличается от ожидаемого типа:

```
sqrt(Fraction(1, 4))
     // Вызовет fraction2Double, потому что sqrt ожидает получить Double
```

- если выполняется обращение к несуществующему члену объекта:

```
new File("README").read
     // Вызовет file2RichFile, потому что File не имеет метода read
```

- если вызывается метод, параметры которого не соответствуют указанным аргументам:

```
3 * Fraction(4, 5)
     // Вызовет int2Fraction, потому что метод * типа Int не принимает
     // аргумент типа  Fraction
```

С другой стороны, существуют три ситуации, когда неявное преобразование не выполняется:

- если программный код компилируется без неявного преобразования, например если выражение `a * b` может быть скомпилировано без неявного преобразования, компилятор не будет пытаться скомпилировать его как `a * convert(b)` или `convert(a) * b`;

- компилятор никогда не пытается применить сразу несколько преобразований, таких как `convert1(convert2(a)) * b`;
- неоднозначные преобразования считаются ошибкой, например если оба преобразования, `convert1(a) * b` и `convert2(a) * b`, допустимы, компилятор сообщит об ошибке.

Внимание. Правило, касающееся неоднозначности, применяется только к объектам, к которым применяется преобразование. Случай

```
Fraction(3, 4) * 5
```

не является неоднозначным, только потому что оба,

```
Fraction(3, 4) * int2Fraction(5)
```

и

```
fraction2Double(Fraction(3, 4)) * 5
```

преобразования допустимы. Предпочтение будет отдано первому преобразованию, потому что оно не требует преобразования объекта, к которому применяется метод *.

Совет. Если вам потребуется определить, какое именно неявное преобразование было применено компилятором, скомпилируйте программу командой:

```
scalac -Xprint:typer MyProg.scala
```

Вы увидите исходный программный код после добавления неявного преобразования.

21.5. Неявные параметры

Функция или метод может иметь список параметров, помеченный ключевым словом `implicit`. В этом случае компилятор будет подыскивать значения по умолчанию для передачи в вызов функции. Ниже приводится простой пример:

```
case class Delimiters(left: String, right: String)

def quote(what: String)(implicit delims: Delimiters) =
     delims.left + what + delims.right
```

Метод `quote` можно вызвать, явно передав ему объект типа `Delimiters`:

```
quote("Bonjour le monde")(Delimiters("«", "»")) // Вернет «Bonjour le monde»
```

Обратите внимание, что существуют два списка аргументов. Это – «каррированная» функция (см. главу 12).

Список неявных параметров можно опустить:

```
quote("Bonjour le monde")
```

В этом случае компилятор попробует отыскать неявное значение типа `Delimiters`. Это должно быть значение, объявленное как `implicit`. Поиск такого объекта будет выполнен компилятором в двух местах:

- среди всех объявлений `val` и `def` требуемого типа, находящихся в области видимости и имеющих простые идентификаторы;
- в объекте-компаньоне типа, связанного с требуемым типом; в число связанных типов входит сам искомый тип и, если это параметризованный тип, его параметры типа.

В нашем примере будет целесообразно создать объект, такой как:

```
object FrenchPunctuation {
    implicit val quoteDelimiters = Delimiters("«", "»")
    ...
}
```

и затем импортировать все значения из объекта:

```
import FrenchPunctuation._
```

или только требуемое значение:

```
import FrenchPunctuation.quoteDelimiters
```

после чего в функцию `quote` будут передаваться французские ограничители.

> **Примечание.** Существовать может только одно неявное значение `val` заданного типа. То есть использовать неявные параметры наиболее распространенных типов – не самая лучшая идея. Например, такое определение:
>
> ```
> def quote(what: String)(implicit left: String, right: String) // Нет!
> ```
>
> не будет работать – невозможно передать в вызов две разные строки.

21.6. Неявные преобразования с неявными параметрами

Неявные параметры функций также могут использоваться для неявного преобразования. Чтобы убедиться в этом, рассмотрим для начала следующую простую обобщенную функцию:

```
def smaller[T](a: T, b: T) = if (a < b) a else b // Ничего особенного
```

В действительности такое определение не будет компилироваться, потому что компилятор не уверен, принадлежат ли значения `a` и `b` типам, обладающим оператором `<`.

Для решения проблемы можно передать функцию преобразования:

```
def smaller[T](a: T, b: T)(implicit order: T => Ordered[T])
    = if (order(a) < b) a else b
```

Так как трейт `Ordered[T]` имеет оператор `<`, принимающий значение типа `T`, эта версия будет считаться корректной.

Подобные ситуации встречаются в практике настолько часто, что в объект `Predef` были включены определения неявных значений типа `T => Ordered[T]` для широкого круга типов, включая все типы, уже реализующие `Ordered[T]` или `Comparable[T]`. Благодаря этому можно вызвать:

```
smaller(40, 2)
```

и

```
smaller("Hello", "World")
```

Чтобы вызвать

```
smaller(Fraction(1, 7), Fraction(2, 9))
```

необходимо определить функцию `Fraction => Ordered[Fraction]` и либо передать ее в вызов, либо сделать доступной как неявное значение (implicit val). Оставляю это вам в качестве самостоятельного упражнения, потому что это уводит нас в сторону от темы данного раздела.

А теперь вернемся к сути. Взгляните еще раз на определение:

```
def smaller[T](a: T, b: T)(implicit order: T => Ordered[T])
```

Обратите внимание, что `order` – это функция с единственным параметром, она отмечена ключевым словом `implicit` и имеет простой идентификатор. Таким образом, это не только неявный параметр, *но и неявное преобразование*. То есть вызов `order` в теле нашей функции можно опустить:

```
def smaller[T](a: T, b: T)(implicit order: T => Ordered[T])
    = if (a < b) a else b
        // Вызовет order(a) < b, если a не имеет оператора <
```

21.7. Границы контекста

Параметр типа может иметь границу контекста в форме `T : M`, где `M` – другой обобщенный тип. Это требует наличия в области видимости неявного значения типа `T[M]`.

Например, объявление

```
class Pair[T : Ordering]
```

требует наличия неявного значения типа `Ordering[T]`. Это неявное значение может затем использоваться в методах класса. Например:

```
class Pair[T : Ordering](val first: T, val second: T) {
    def smaller(implicit ord: Ordering[T]) =
        if (ord.compare(first, second) < 0) first else second
}
```

Если попытаться создать new `Pair(40, 2)`, компилятор определит, что требуется создать `Pair[Int]`. Благодаря наличию неявного значения типа `Ordering[Int]` в области видимости `Predef` тип `Int` соответствует границе контекста. Это значение становится полем класса и будет передаваться методам, требующим его.

При желании неявное значение можно извлекать с помощью метода `implicity` в классе `Predef`:

```
class Pair[T : Ordering](val first: T, val second: T) {
    def smaller =
```

```
        if (implicitly[Ordering[T]].compare(first, second) < 0) first
        else second
}
```

Функция `implicitly` определена в `Predef.scala`, как показано ниже:

```
def implicitly[T](implicit e: T) = e
    // Для вызова неявных значений из преисподней
```

> **Примечание.** Как метко отмечает комментарий, неявные объекты «живут» в «преисподней» и незаметно добавляются в методы.

Также можно использовать тот факт, что трейт `Ordered` определяет неявное преобразование из типа `Ordering` в тип `Ordered`. После импорта этого преобразования появляется возможность использовать операторы отношений:

```
class Pair[T : Ordering](val first: T, val second: T) {
    def smaller = {
        import Ordered._;
        if (first < second) first else second
    }
}
```

Однако все это лишь незначительные вариации; самое важное заключается в том, что вы можете создавать экземпляры `Pair[T]` везде, где доступно неявное значение типа `Ordering[T]`. Например, если потребуется создать пару `Pair[Point]`, необходимо будет определить неявное значение типа `Ordering[Point]`:

```
implicit object PointOrdering extends Ordering[Point] {
    def compare(a: Point, b: Point) = ...
}
```

21.8. Неявный параметр подтверждения

В главе 17 вы познакомились с механизмом ограничения типов (type constraints):

```
T =:= U
T <:< U
T <%< U
```

Механизм ограничения типов проверяет, является ли тип T равным типу U, подтипом U или может неявно преобразовываться в тип U. Чтобы иметь возможность пользоваться такими ограничениями, следует добавить неявный параметр, такой как

```
def firstLast[A, C](it: C)(implicit ev: C <:< Iterable[A]) =
    (it.head, it.last)
```

Конструкции =:=, <:< и <%< в действительности являются классами с неявными значениями, объявленными в объекте Predef. Например, ниже приводится фактическое определение класса <:<

```
abstract class <:<[-From, +To] extends Function1[From, To]

object <:< {
    implicit def conforms[A] = new (A <:< A) { def apply(x: A) = x }
}
```

Допустим, что компилятору требуется обработать конструкцию implicit ev: String <:< AnyRef. Она напоминает объект-компаньон для неявного объекта типа String <:< AnyRef. Обратите внимание, что <:< является контравариантным в позиции From и ковариантным в позиции To. Поэтому объект

```
<:<.conforms[String]
```

может использоваться как экземпляр String <:< AnyRef. (Также мог бы использоваться экземпляр <:<.conforms[AnyRef], но он менее специализированный и потому не рассматривается.)

Мы называем ev «объектом подтверждения» (evidence object) – его существование подтверждает тот факт, что в данном случае тип String является подтипом AnyRef.

В нашем примере объект подтверждения является функцией идентичности (identity function). Чтобы понять, зачем нужна функция идентичности, взгляните внимательнее на следующее определение:

```
def firstLast[A, C](it: C)(implicit ev: C <:< Iterable[A]) =
    (it.head, it.last)
```

В действительности компилятор не знает, что C является типом Iterable[A] – не забывайте, что <:< не особенность языка, а всего

лишь класс. Поэтому вызовы `it.head` и `it.last` с его точки зрения недопустимы. Но `ev` является функцией с одним параметром и потому – неявным преобразованием из `C` в `Iterable[A]`, и компилятор применяет ее, вычисляя `ev(it).head` и `ev(it).last`.

> **Совет.** Чтобы проверить наличие обобщенного неявного объекта, можно вызвать функцию `implicitly` в интерактивной оболочке REPL. Например, введите `implicitly[String <:< AnyRef]`, и вы получите результат (который является функцией). Но команда `implicitly[AnyRef <:< String]` завершится с сообщением об ошибке.

21.9. Аннотация @implicitNotFound

Аннотация `@implicitNotFound` генерирует сообщение об ошибке, когда компилятор не может создать параметр аннотированного типа. Назначение аннотации – предоставить программисту полезное сообщение об ошибке. Например, класс `<:<` аннотирован как

```
@implicitNotFound(msg = "Cannot prove that ${From} <:< ${To}.")
abstract class <:<[-From, +To] extends Function1[From, To]
```

То есть если попытаться вызвать:

```
firstLast[String, List[Int]](List(1, 2, 3))
```

компилятор выведет сообщение:

```
Cannot prove that List[Int] <:< Iterable[String]
(Невозможно убедиться, что List[Int] <:< Iterable[String])
```

Что для программиста будет более очевидной подсказкой, чем сообщение по умолчанию:

```
Could not find implicit value for parameter ev: <:<[List[Int],Iterable[String]]
(Не найдено неявное значение для параметра ev: <:<[List[Int],Iterable[String]])
```

Обратите внимание, что в сообщении об ошибке вместо `${From}` и `${To}` были подставлены параметры типа `From` и `To` аннотированного класса.

21.10. Тайна CanBuildFrom

В главе 1 говорилось, что вы должны игнорировать неявный параметр `CanBuildFrom`. Теперь вы готовы узнать, как он действует.

Рассмотрим метод `map`. Несколько упрощая, можно сказать, что метод `map` является методом `Iterable[A, Repr]` со следующей реализацией:

```
def map[B,That](f:(A) => B)(implicit bf:CanBuildFrom[Repr,B,That]):That = {
     val builder = bf()
     val iter = iterator()
     while (iter.hasNext) builder += f(iter.next())
     builder.result
}
```

Здесь `Repr` – это «тип представления» (representation type). Данный параметр позволяет выбрать соответствующие фабрики построителей (builder factories) для необычных коллекций, таких как `Range` или `String`.

> **Примечание.** В библиотеке Scala метод `map` в действительности определен в трейте `TraversableLike[A, Repr]`. То есть в более часто используемом трейте `Iterable` нет необходимости определять параметр типа `Repr`.

Трейт `CanBuildFrom[From, E, To]` является подтверждением возможности создания коллекции типа `To`, хранящей значения типа `E`, то есть совместимой с типом `From`. Но, прежде чем обсуждать особенности создания этих объектов подтверждений (evidence objects), посмотрим, что они делают.

Трейт `CanBuildFrom` имеет метод `apply`, возвращающий объект типа `Builder[E, To]`. Тип `Builder` имеет методы `+=`, для добавления элементов во внутренний буфер и `result` для создания желаемой коллекции.

```
trait Builder[-E, +To] {
     def +=(e: E): Unit
     def result(): To
}

trait CanBuildFrom[-From, -E, +To] {
     def apply(): Builder[E, To]
}
```

То есть метод map просто конструирует построитель указанного типа, заполняет построитель значениями функции f и возвращает получившуюся коллекцию.

Каждая коллекция имеет неявный объект CanBuildFrom в своем объекте-компаньоне. Взгляните на упрощенную версию стандартного класса ArrayBuffer:

```
class Buffer[E : Manifest] extends Iterable[E, Buffer[E]]
          with Builder[E, Buffer[E]] {
     private var elems = new Array[E](10)
     ...
     def iterator() = ...
          private var i = 0
          def hasNext = i < length
          def next() = { i += 1; elems(i - 1) }
     }
     def +=(e: E) { ... }
     def result() = this
}

object Buffer {
     implicit def canBuildFrom[E : Manifest] =
               new CanBuildFrom[Buffer[_], E, Buffer[E]] {
          def apply() = new Buffer[E]
     }
}
```

Рассмотрим вызов buffer.map(f), где f — функция типа A => B. Неявный параметр bf получается вызовом метода canBuildFrom[B] объекта-компаньона Buffer. Его метод apply возвращает построитель, в данном случае Buffer[B].

Так случилось, что класс Buffer уже имеет метод +=, возвращающий сам экземпляр класса. То есть Buffer является построителем для самого себя.

Однако построитель для класса Range не возвращает значение типа Range, и совершенно очевидно, что это невозможно. Например, выражение (1 to 10).map(x => x * x) имеет значение, тип которого отличается от типа Range. В действительности в библиотеке Scala класс Range наследует IndexedSeq[Int], а объект-компаньон IndexedSeq определяет построитель, конструирующий экземпляр типа Vector.

Ниже приводится упрощенное определение класса Range, где в качестве построителя используется Buffer:

```
class Range(val low: Int, val high: Int) extends Iterable[Int, Range] {
     def iterator() = ...
}

object Range {
     implicit def canBuildFrom[E : Manifest] =
               new CanBuildFrom[Range, E, Buffer[E]] {
          def apply() = new Buffer[E]
     }
}
```

Теперь рассмотрим вызов `Range(1, 10).map(f)`. Этот метод имеет неявный параметр `implicit f: CanBuildFrom[Repr, B, That]`. Так как `Repr` — это тип `Range`, связанными типами являются `CanBuildFrom`, `Range`, `B` и неизвестный тип `That`. Объект `Range` возвращает соответствие вызовом его метода `canBuildFrom[B]`, который возвращает `CanBuildFrom[Range, B, Buffer[B]]`. Этот объект является аргументом `bf`, а его метод `apply` возвращает `Buffer[B]` для построения результата.

Как только что было показано, неявный параметр `CanBuildFrom[Repr, B, That]` отыскивает фабричный объект, который может создать построитель для коллекции указанного типа. Фабрика построителей (builder factory) определена как неявное значение в объекте-компаньоне `Repr`.

Упражнения

1. Как действует оператор `->`? То есть как заставить выражения `"Hello" -> 42` и `42 -> "Hello"` образовывать пары `("Hello", 42)` и `(42, "Hello")`? Подсказка: `Predef.any2ArrowAssoc`.
2. Определите оператор `+%`, добавляющий указанный процент к значению. Например, выражение `120 +% 10` должно вернуть `132`. Подсказка: так как операторы являются методами, а не функциями, вам также придется реализовать неявное преобразование.
3. Было бы неплохо иметь возможность определить оператор вычисления факториала, чтобы выражение `5!` возвращало `120`. Почему это невозможно? Попробуйте реализовать оператор факториала как `i`.
4. Некоторые программисты обожают «свободные API», которые читаются почти как обычный текст на английском языке.

Определите такой API для чтения в консоли целых и вещественных чисел, а также строк, чтобы с его помощью можно было записать, например, такую инструкцию: `Obtain aString askingFor «Your name» and anInt askingFor «Your age» and aDouble askingFor «Your weight»`.

5. Реализуйте все необходимое для вычисления выражения

```
smaller(Fraction(1, 7), Fraction(2, 9))
```

из раздела 21.6 «Неявные преобразования с неявными параметрами». Добавьте класс `RichFraction`, наследующий `Ordered[Fraction]`.

6. Реализуйте лексикографическое сравнение объектов класса `java.awt.Point`.
7. Продолжите предыдущее упражнение и реализуйте сравнение двух точек на координатной плоскости по их расстояниям от центра координат. Как можно обеспечить выбор между двумя сравнениями?
8. Воспользуйтесь командой `implicitly` в интерактивной оболочке REPL, чтобы получить список неявных объектов, описанных в разделе 21.5 «Неявные параметры» и в разделе 21.6 «Неявные преобразования с неявными параметрами». Какие объекты вы получили?
9. Отыщите объект `=:=` в `Predef.scala`. Объясните, как он действует.
10. Результатом выражения `«abc».map(_.toUpper)` является значение типа `String`, а результатом выражения `«abc».map(_.toInt)` — значение типа `Vector`. Объясните, почему.

Глава 22. Ограниченные продолжения

Темы, рассматриваемые в этой главе L3

- 22.1. Сохранение и вызов продолжений.
- 22.2. Вычисления с «дырками».
- 22.3. Управление выполнением в блоках `reset` и `shift`.
- 22.4. Значение выражения `reset`.
- 22.5. Типы выражений `reset` и `shift`.
- 22.6. CPS-аннотации.
- 22.7. Преобразование рекурсии в итерации.
- 22.8. Устранение инверсии управления.
- 22.9. CPS-трансформация.
- 22.10. Трансформирование вложенных контекстов управления.
- Упражнения.

Продолжения (continuations) – весьма мощная конструкция, позволяющая реализовать механизмы управления потоком выполнения, отличные от знакомых условных инструкций, циклов, вызовов функций и исключений. Например, с их помощью можно вернуться к предыдущим вычислениям или изменить порядок выполнения фрагментов программы. Кому-то эти возможности могут показаться слишком заумными – они не предназначены для прикладных программистов, но разработчики библиотек могут взять на вооружение всю мощь продолжений и сделать ее прозрачно доступной для пользователей библиотек.

Основные темы этой главы:

- продолжения позволяют вернуться назад, к некоторой точке в программе;
- продолжения захватываются в блоке `shift`;
- функция продолжения простирается до конца охватывающего блока `reset`;
- продолжение – это «все остальные вычисления» от выражения с блоком `shift` и до конца охватывающего блока `reset`, где блок `shift` замещается «дыркой»;

- при вызове продолжения с аргументом он замещает «дырку»;
- программный код, содержащий выражения `shift`, можно переписать в «стиле продолжений» (continuation-passing style), или CPS, вплоть до конца охватывающего выражения `reset`;
- метод, содержащий выражение `shift` без выражения `reset`, должен отмечаться CPS-аннотацией;
- продолжения могут использоваться для преобразования рекурсивного обхода древовидных структур в итерации;
- продолжения могут избавлять от проблемы «инверсии управления» (inversion of control) в веб-приложениях и в приложениях с графическим интерфейсом.

22.1. Сохранение и вызов продолжений

Продолжение (continuation) – это механизм, позволяющий вернуться назад к некоторой точке в программе. Где это может пригодиться? Рассмотрим операцию чтения файла:

```
contents = scala.io.Source.fromFile(filename, "UTF-8").mkString
```

Если файл не существует, будет возбуждено исключение. Вы можете перехватить это исключение и попросить пользователя указать другое имя файла. Но как тогда вернуться назад, к операции чтения файла? Разве не было бы проще просто перейти назад, к точке, где возникла ошибка, и повторить попытку? Продолжения позволяют сделать это.

Сначала необходимо *захватить* (capture) продолжение, задействовав конструкцию `shift`. Внутри этой конструкции необходимо точно описать, что вы собираетесь делать с продолжением, когда оно у вас появится. Чаще всего продолжения сохраняются для использования в дальнейшем, как показано ниже:

```
var cont: (Unit => Unit) = null
...
shift { k: (Unit => Unit) =>  // продолжение передается в shift
     cont = k                 // захватить для дальнейшего использования
}
```

В данном случае продолжение – функция без аргументов и без возвращаемого значения (точнее, с аргументом и возвращаемым

значением типа `Unit`). С продолжениями, имеющими аргумент и возвращаемое значение, мы познакомимся ниже.

Чтобы вернуться в точку `shift`, нужно просто вызвать продолжение, обратившись к методу `cont`.

В языке Scala используются *ограниченные продолжения* (delimited continuations) – простирающиеся до определенных границ. Границы продолжения определяются блоком `reset { ... }`:

```
reset {
     ...
     shift { k: (Unit => Unit) =>
          cont = k
     } // Выполнение метода cont начинается в этой точке...
     ...
}      // ...и заканчивается здесь
```

При вызове метода `cont` выполнение начинается с точки, следующей сразу за блоком `shift`, и продолжается до границы блока `reset`.

Ниже приводится полный пример. Здесь выполняется чтение из файла и сохраняется продолжение.

```
var cont: (Unit => Unit) = null
var filename = "myfile.txt"
var contents = ""

reset {
     while (contents == "") {
          try {
               contents = scala.io.Source.fromFile(filename, "UTF-8").mkString
          } catch { case _ => }
          shift { k: (Unit => Unit) =>
               cont = k
          }
     }
}
```

Чтобы вернуться назад, достаточно просто вызвать продолжение:

```
if (contents == "") {
     print("Try another filename: ");
     filename = readLine()
     cont() // Вернуться назад к shift
}
println(contents)
```

Примечание. В версии Scala 2.9, чтобы скомпилировать программу с продолжениями, необходимо подключить расширение компилятора, реализующее поддержку продолжений[1]:

```
scalac -P:continuations:enable MyProg.scala
```

22.2. Вычисления с «дырками»

Для более точного понимания, что сохраняется в продолжении, можно представлять блок `shift` как «дырку» (hole) в блоке `reset`. При вызове продолжения можно передать значение в эту «дырку», и вычисления будут выполняться, как если бы блок `shift` имел это значение. Рассмотрим следующий вымышленный пример:

```
var cont : (Int => Double) = null
reset {
     0.5 * shift { k : (Int => Double) => cont = k } + 1
}
```

Если заменить блок `shift` «дыркой», получится:

```
reset {
     0.5 *    + 1
}
```

Если вызвать `cont(3)`, дырка заполнится значением `3` и выражение в блоке `reset` вернет `2.5`. Иными словами, `cont` в данном случае – это простая функция от `x: Int => 0.5 * x + 1`.

Продолжение имеет тип `Int => Double` потому, что дырка заполняется значением типа `Int`, а выражение вычисляет значение типа `Double`.

Примечание. В предыдущем разделе было создано продолжение, не принимающее и не возвращающее никаких значений. Именно поэтому оно было объявлено с сигнатурой `Unit => Unit`.

22.3. Управление выполнением в блоках reset и shift

Порядок выполнения программного кода в блоках `reset`/`shift` может показаться немного странным, потому что они преследуют две цели сразу – определение функции продолжения и ее сохранение.

[1] В Scala 2.10 поддержка продолжений включена в компилятор по умолчанию. – *Прим. ред.*

Когда в программе встречается блок `reset`, выполняется его тело. Когда встречается блок `shift`, его тело выполняется с функцией продолжения в качестве аргумента. Когда блок `shift` заканчивается, управление немедленно передается в конец охватывающего блока `reset`. Рассмотрим следующий код:

```
var cont : (Unit => Unit) = null
reset {
     println("Before shift")
     shift {
          k: (Unit => Unit) => {
               cont = k
               println("Inside shift") // Переход в конец блока reset
          }
     }
     println("After shift")
}
println("After reset")
cont()
```

Во время выполнения блока `reset` этот фрагмент выведет:

```
Before shift
Inside shift
```

Затем управление будет передано в конец блока `reset`, и фрагмент выведет:

```
After reset
```

Наконец, когда произойдет вызов `cont`, управление будет передано обратно в блок `reset`, и фрагмент выведет:

```
After shift
```

Обратите внимание, что код перед `shift` не является частью продолжения. Продолжение начинается с выражения, содержащего блок `shift` (который превращается в «дырку»), и простирается до конца блока `reset`. В данном случае продолжение имеет вид:

```
 : Unit =>    ; println("After shift")
```

Функция `cont` имеет параметр типа `Unit`, и дырка просто замещается значением `()`.

> **Примечание.** Как видите, `shift` внутри блока `reset` производит немедленный выход за пределы блока `reset`. То же происходит, когда вызывается продолжение, которое переходит внутрь блока `reset` и встречает другой блок `shift` (или тот же самый блок `shift` в цикле). Вызванная функция немедленно завершается и возвращает значение блока `shift`.

22.4. Значение выражения reset

Если выполнение блока `reset` завершается в результате встречи с блоком `shift`, его значением становится значение блока `shift`:

```
val result = reset { shift { k: (String => String) => "Exit" }; "End" }
     // результат: "Exit"
```

Однако, если блок `reset` выполняется до конца, его значением становится значение последнего вычисленного в блоке выражения:

```
val result = reset{ if (false) shift{ k:(String=>String) => "Exit" }; "End" }
     // результат: "End"
```

На практике может происходить и то, и другое, если блок `reset` содержит условную инструкцию или цикл:

```
val result = reset {
     if (scala.util.Random.nextBoolean()) {
          shift {
               k: (String => String) => {
                    "Exit"
               }
          }
     }
     else "End"
}
```

В данном случае результат случайным образом будет получать значение `"Exit"` или `"End"`.

22.5. Типы выражений reset и shift

Оба выражения, `reset` и `shift`, в действительности являются методами с параметрами типов: `reset[B, C]` и `shift[A, B, C]`. Следующая «шпаргалка» демонстрирует, как выводятся типы.

```
reset {
     перед
     shift { k: (A => B) => // Типы A и B выводятся отсюда
          внутри            // Тип C
     }                      // "Дырка" имеет тип A
     после                  // Должно возвращаться значение типа B
}
```

Здесь тип `A` – это тип аргумента продолжения, то есть тип значения, «заполняющего дырку». Тип `B` – это тип значения, возвращаемого продолжением при его вызове. Тип `C` – это ожидаемый тип выражения `reset`, то есть тип значения, возвращаемого блоком `shift`.

> **Внимание.** Если блок `reset` может возвращать значение любого из типов, `B` или `C` (из-за наличия условных инструкций или циклов), тогда тип `B` должен быть подтипом `C`.

Эти типы требуют особого внимания. Если компилятор не сможет вывести их правильно, он выдаст весьма туманное сообщение об ошибке. Рассмотрим, например, кажущееся безвредным изменение в примере из предыдущего раздела:

```
val result = reset {
     if (util.Random.nextBoolean()) {
          shift {
               k: (Unit => Unit) => {  // A и B объявлены как Unit
                    "Exit"             // C - это String
               }                       // "дырка" shift имеет тип Unit
          }
     }
     else "End"                        // тип String не является типом Unit
}
```

Проблема в том, что блок `reset` может вернуть значение типа `String`, который не является типом `Unit`.

> **Совет.** Если типы выводятся не так, как хотелось бы, в блоки `reset` и `shift` можно добавить параметры типов. Например, пример выше можно сделать работоспособным, если изменить тип продолжения на `Unit => Any` и использовать `reset[Any, Any]`.

Из-за надоедливых сообщений об ошибках может появиться соблазн подбирать типы методом перебора, пока сообщения не ис-

чезнут. Не поддавайтесь искушению и внимательно изучайте, что должно происходить при вызове продолжения. В данном абстрактном случае это сложно сделать. Но при решении конкретной проблемы обычно бывает точно известно, какое значение будет передаваться в вызов продолжения и какое значение должно быть возвращено. Это позволит определить типы `A` и `B`. Как правило, в качестве типа `C` можно принять тип `B`. В результате блок `reset` всегда будет возвращать значение типа `B`, независимо от того, как произошел выход из него.

22.6. CPS-аннотации

В некоторых виртуальных машинах продолжения реализованы как срезы стека времени выполнения. Когда производится вызов продолжения, стек восстанавливается из среза. К сожалению, виртуальная машина Java не допускает подобных манипуляций со стеком. Для реализации продолжений в JVM компилятор Scala реорганизует код внутри блока `reset` в «стиле продолжений» (continuation-passing style), или CPS. Поближе с этой реорганизацией мы познакомимся в разделе 22.9 «CPS-трансформация».

Реорганизованный программный код существенно отличается от обычного кода на языке Scala, и их нельзя смешивать. В большей степени это отличие сказывается на методах. Если метод содержит блок `shift`, он не компилируется в обычный метод. Его необходимо пометить как «трансформированный».

В предыдущем разделе говорилось, что блок `shift` имеет три параметра типов – типы параметров и возвращаемого значения функции продолжения, тип блока. Чтобы иметь возможность вызывать метод, содержащий блок `shift[A, B, C]`, его необходимо аннотировать аннотацией `@cpsParam(B, C)`. В общем случае, когда типы `B` и `C` совпадают, используется аннотация `@cps[B]`.

> **Примечание.** Вместо `@cps[Unit]` можно использовать аннотацию `@suspendable`. Я не буду использовать ее здесь – она длиннее, и, на мой взгляд, ее использование не добавляет ясности.

Рассмотрим приложение из раздела 22.1 «Сохранение и вызов продолжений». Если поместить цикл чтения в метод, этот метод необходимо аннотировать:

```
def tryRead(): Unit @cps[Unit] = {
     while (contents == "") {
          try {
               contents = scala.io.Source.fromFile(filename, "UTF-8").mkString
          } catch { case _ => }
          shift { k : (Unit => Unit) =>
               cont = k
          }
     }
}
```

> **Внимание.** При аннотировании метода необходимо указать тип возвращаемого значения и использовать знак =, даже если метод возвращает значение Unit. Это ограничение накладывается синтаксисом аннотаций и никак не связано с продолжениями.

Когда метод, помеченный аннотацией @cps, вызывается из другого метода и вызов выполняется не из блока reset, вызывающий метод также должен быть аннотирован. Иными словами, аннотироваться должны любые методы между reset и shift.

Все это похоже на одно большое неудобство. Но имейте в виду, что аннотации не предназначены для использования прикладными программистами. Они должны использоваться разработчиками библиотек, создающими специализированные конструкции управления потоком выполнения. Особенности использования продолжений следует тщательно скрывать от пользователей библиотек.

22.7. Преобразование рекурсии в итерации

Теперь мы готовы к первому серьезному применению продолжений. Рассмотрим простой процесс рекурсивного обхода всех узлов древовидной структуры. При посещении узла выполняется его обработка и происходит обход всех его дочерних узлов. Процедура обхода начинается с корневого узла дерева.

Например, следующий метод выводит список файлов, хранящихся в указанном каталоге и в его подкаталогах:

```
def processDirectory(dir : File) {
     val files = dir.listFiles
     for (f <- files) {
```

```
        if (f.isDirectory)
            processDirectory(f)
        else
            println(f)
    }
}
```

Он отлично подходит для случая, когда требуется получить полный список всех файлов, но как быть, если потребуется вывести только первую сотню? Мы не можем остановить рекурсию на полпути. Однако с продолжениями в этом нет ничего сложного. Всякий раз, попадая в новый узел, мы будем выходить из рекурсии, а для получения большего количества результатов – возвращаться в нее.

Команды `reset` и `shift` особенно хорошо подходят для реализации такого шаблона управления выполнением. После выполнения блока `shift` программа будет покидать охватывающий блок `reset`. При вызове сохраненного продолжения она будет возвращаться в точку, следующую сразу за блоком `shift`.

Чтобы реализовать этот план, блок `shift` следует поместить в точку, где рекурсия должна прерываться.

```
if (f.isDirectory)
    processDirectory(f)
else {
    shift {
        k: (Unit => Unit) => {
            cont = k
        }
    }
    println(f)
}
```

Блок `shift` играет двоякую роль. Он обеспечивает выход за пределы блока `reset` и сохраняет продолжение, к которому мы можем вернуться.

Заключите точку запуска процесса обработки дерева в блок `reset` и затем вызывайте сохраненное продолжение столько раз, сколько потребуется:

```
reset {
    processDirectory(new File(rootDirName))
}
for (i <- 1 to 100) cont()
```

Безусловно, метод `processDirectory` необходимо снабдить CPS-аннотацией:

```
def processDirectory(dir : File) : Unit @cps[Unit]
```

К сожалению, необходимо также заменить цикл `for`. Дело в том, что цикл `for` транслируется в вызов метода `foreach`, а он *не* помечен CPS-аннотацией, из-за чего мы не можем вызывать его. Проблему можно решить с помощью простого цикла `while`:

```
var i = 0
while (i < files.length) {
     val f = files(i)
     i += 1
     ...
}
```

Ниже приводится полный исходный текст программы:

```
import scala.util.continuations._
import java.io._

object PrintFiles extends App {
     var cont : (Unit => Unit) = null

     def processDirectory(dir : File) : Unit @cps[Unit] = {
          val files = dir.listFiles
          var i = 0
          while (i < files.length) {
               val f = files(i)
               i += 1
               if (f.isDirectory)
                    processDirectory(f)
               else {
                    shift {
                         k: (Unit => Unit) => {
                              cont = k             // ❷
                         }
                    }                              // ❺
                    println(f)
               }
          }
     }

     reset {
          processDirectory(new File("/"))    // ❶
```

```
    }                                    // ❸
    for (i <- 1 to 100) cont()           // ❹
}
```

При входе в блок `reset` вызывается метод `processDirectory` ❶. Как только метод находит первый файл, не являющийся каталогом, он входит в блок `shift` ❷. Функция продолжения сохраняется в переменной `cont`, и программа переходит в конец блока `reset` ❸.

Затем вызывается функция `cont` ❹, и программа возвращается обратно в рекурсию ❺, в «дырку `shift`». Рекурсия продолжается, пока не будет найден следующий файл и не произойдет повторный вход в блок `shift`. В конце блока `shift` программа вновь переходит в конец блока `reset` и возвращает функцию `cont`.

> **Примечание.** Я постарался сохранить программу максимально простой и короткой, но прикладной программист совсем не так будет использовать продолжения. Блоки `reset` и `shift` будут спрятаны в недрах библиотеки и останутся невидимыми для пользователя библиотеки. В данном случае блок `reset` можно было бы спрятать в конструкторе итератора, а вызов `cont` – в методе `next` итератора.

22.8. Устранение инверсии управления

Одним из перспективных применений продолжений является устранение «инверсии управления» (inversion of control) в веб-приложениях и в приложениях с графическим интерфейсом. Рассмотрим типичное веб-приложение, запрашивающее некоторую информацию у пользователя на одной странице и дополнительную информацию – на следующей странице.

В обычной программе это могло бы выглядеть так:

```
val response1 = getResponse(page1)
val response2 = getResponse(page2)
process(response1, response2)
```

Но веб-приложения действуют иначе. Вы не можете управлять потоком выполнения приложения. Вместо этого после отправки первой страницы пользователю программа ждет получения ответа. Когда в конечном счете ответ поступит, он должен быть передан в другую часть программы, отправляющую вторую страницу. Когда пользователь ответит на вопросы во второй странице, их обработка будет выполнена в третьем месте программы.

Эта проблема с успехом может быть решена веб-фреймворком, построенным на основе продолжений. После отправки веб-страницы пользователю приложение сохранит продолжение и вызовет его, когда поступит ответ от пользователя. Все это выглядит более чем прозрачно для прикладного программиста.

Для простоты я продемонстрирую эту концепцию на примере приложения с графическим интерфейсом. Приложение имеет метку с текстом вопроса и текстовое поле ввода, куда пользователь сможет ввести ответ (рис. 22.1).

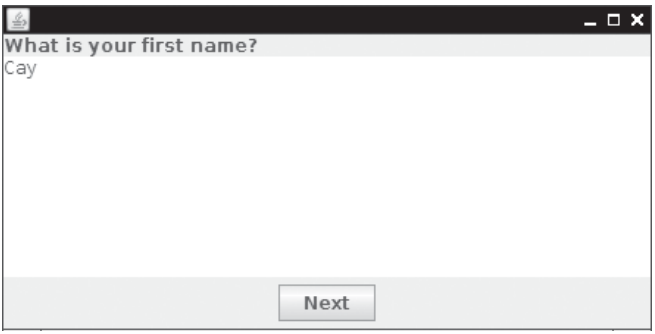


Рис. 22.1. Демонстрация устранения проблемы инверсии управления

После щелчка на кнопке **Next** (Далее) ответ пользователя возвращается в метод `getResponse`. Ниже представлен программный код, который мог бы написать прикладной программист:

```
def run() {
     reset {
          val response1 = getResponse("What is your first name?")   // ❶
          val response2 = getResponse("What is your last name?")
          process(response1, response2) //
     }                                                              // ❸
}

def process(s1: String, s2: String) {
     label.setText("Hello, " + s1 + " " + s2)
}
```

Метод `process` не требуется отмечать CPS-аннотацией, так как он не содержит блока `shift`.

Метод `getResponse` сохраняет продолжение самым обычным способом. Так как он содержит блок `shift`, он отмечен аннотацией `@cps`.

```
def getResponse(prompt: String): String @cps[Unit] = {
     label.setText(prompt)
     setListener(button) { cont() }
     shift { k: (Unit => Unit) =>
          cont = k  // ❷
     }              // ❹
     setListener(button) { }
     textField.getText
}
```

Метод `getResponse` возлагает на продолжение роль обработчика события щелчка на кнопке **Next** (Далее), используя вспомогательный метод, реализация которого представлена в полном листинге программы, в конце этого раздела.

Обратите внимание, что прикладной код в методе `run` заключен в блок `reset`. Когда метод `run` выполнит первый вызов метода `getResponse` ❶, он войдет в блок `shift` ❷, сохранит продолжение, выполнит переход в конец блока `reset` ❸ и вернет управление приложению.

Затем пользователь введет ответ и щелкнет на кнопке. Обработчик события щелчка вызовет продолжение, и выполнение будет передано в точку ❹. Ввод пользователя вернется в метод `run`.

Затем метод `run` выполнит второй вызов метода `getResponse` и опять вернет управление приложению, сохранив продолжение. Когда пользователь введет второй ответ и щелкнет на кнопке, результат снова попадет в метод `run` и будет передан методу `process` ❺.

Интересно отметить, что все операции производятся в потоке доставки событий, и при этом нет необходимости использовать блокировки. Чтобы продемонстрировать это, метод `run` вызывается из обработчика событий от кнопки.

Ниже приводится полный листинг программы:

```
import java.awt._
import java.awt.event._
import javax.swing._
import scala.util.continuations._

object Main extends App {
     val frame = new JFrame
     val button = new JButton("Next")
```

```
    setListener(button) { run() }
    val textField = new JTextArea(10, 40)
    textField.setEnabled(false)
    val label = new JLabel(«Welcome to the demo app»)
    frame.add(label, BorderLayout.NORTH)
    frame.add(textField)
    val panel = new JPanel
    panel.add(button)
    frame.add(panel, BorderLayout.SOUTH)
    frame.pack()
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
    frame.setVisible(true)

    def run() {
        reset {
            val response1 = getResponse("What is your first name?")
            val response2 = getResponse("What is your last name?")
            process(response1, response2) // ❺
        }
    }

    def process(s1: String, s2: String) {
        label.setText("Hello, " + s1 + " " + s2)
    }

    var cont: Unit => Unit = null

    def getResponse(prompt: String): String @cps[Unit] = {
        label.setText(prompt)
        setListener(button) { cont() }
        shift {
            k: (Unit => Unit) => {
                cont = k
            }
        }
        setListener(button) { }
        textField.getText
    }

    def setListener(button: JButton)(action: => Unit) {
        for (l <- button.getActionListeners) button.removeActionListener(l)
        button.addActionListener(new ActionListener {
            override def actionPerformed(event: ActionEvent) { action }
        })
    }
}
```

22.9. CPS-трансформация

Как уже упоминалось в разделе 22.6 «CPS-аннотации», компилятор Scala не может опираться на поддержку продолжений в виртуальной машине. Поэтому он должен транслировать программный код, находящийся внутри блоков `reset`, в специальное представление «в стиле продолжений» (continuation-passing style), или CPS. Чтобы пользоваться продолжениями, совсем необязательно понимать все тонкости этой трансформации, тем не менее понимание основ CPS может пригодиться при расшифровывании сообщений об ошибках.

В результате CPS-трансформации создается объект, в котором определена функция, содержащая «все остальные вычисления». Метод `shift`

```
shift { fun }
```

возвращает объект

```
ControlContext[A, B, C](fun)
```

Напомню, что телом блока `shift` является функция типа `(A => B) => C`. Она получает в качестве параметра функцию продолжения типа `A => B` и возвращает значение типа `C`, которое передается за пределы охватывающего блока `reset`.

> **Примечание.** Здесь я несколько упростил описание класса `ControlContext`. В действительности класс также реализует обработку исключений и оптимизирует константы.

Контекст управления определяет, что делать с функцией продолжения. Обычно он куда-то сохраняет эту функцию и иногда вычисляет ее значение.

Контекст управления не знает, как создать функцию продолжения. Он полагается на то, что кто-то другой сделает это, и просто ожидает получить ее.

Безусловно, продолжение должно быть вычислено. Допустим, что оставшаяся часть вычислений состоит из части `f`, известной нам, за которой следует что-то еще, нам не известное. Мы можем упростить задачу и задействовать часть `f`. Теперь предположим, что кто-то передает нам то, что следует за `f`. Допустим, что это `k1`. Тогда мы

сможем произвести вычисления целиком, сначала применив часть f, а затем k1.

Теперь переведем эти размышления в контекст управления. Блок shift преобразуется в контекст управления, который знает, что делать с программным кодом, следующим за блоком shift. Теперь часть f находится после shift. Мы можем сделать еще шаг вперед и сформировать новый контекст управления, занимающийся обработкой оставшейся части, следующей за f, как показано ниже:

```
new ControlContext(k1 => fun(a => k1(f(a))))
```

Здесь a => k1(f(a)) сначала выполняет f и затем завершает вычисления в части k1. А fun обрабатывает это своим обычным способом.

Такое «пошаговое движение вперед» является фундаментальной операцией контекстов управления. Она называется map. Ниже приводится определение метода map:

```
class ControlContext[+A, -B, +C](val fun: (A => B) => C) {
     def map[A1](f: A => A1) = new ControlContext[A1, B, C](
          (k1: (A1 => B)) => fun(x: A => k1(f(x))))
     ...
}
```

> **Примечание.** Имя метода map (а также метода flatMap, с которым вы познакомитесь в следующем разделе) кажется непонятным – он никак не связан с функцией отображения коллекции значений. Однако, как оказывается, упомянутые методы map и flatMap следуют своду правил, получившему название «законы монад» (monad laws), как и знакомые вам методы map и flatMap. Необязательно иметь полное представление о монадах, чтобы усвоить материал этого раздела. Я упомянул об этом, только чтобы пояснить происхождение этих имен.

После добавления всех параметров типов определение метода map выглядит слишком узкоспециальным, но сама идея проста. Интуитивно понятно, что cc.map(f) принимает контекст управления и превращает его в то, что будет обрабатывать оставшиеся вычисления, следующие за f.

Постепенно, шаг за шагом, мы подходим к моменту, когда все необходимое будет сделано. Это происходит при достижении границы блока reset. После этого можно просто передать fun ничего не делающий метод и получить результат блока shift.

Ниже приводится точное определение метода `reset`:

```
def reset[B, C](cc: ControlContext[B, B, C]) = cc.fun(x => x)
```

Рассмотрим для начала простой пример:

```
reset {
     0.5 * { shift { k: (Int => Double) => cont = k } } + 1
}
```

В данном случае компилятор может вычислить продолжение целиком за один шаг:

```
=> 0.5 *    + 1
```

То есть результатом будет:

```
reset {
  new ControlContext[Int, Double, Unit](k => cont = k).map( => 0.5 *    + 1)
}
```

То же самое, что и:

```
reset {
     new ControlContext[Double, Double, Unit](k1 =>
          cont = k1(x: Int => 0.5 * x + 1)
}
```

Теперь блок `reset` может быть вычислен. `k1` – это функция идентичности (identity function), а ее результатом является

```
cont = x: Int => 0.5 * x + 1
```

Это в точности соответствует поведению, описанному в разделе 22.2 «Вычисления с “дырками”». Вызов метода `reset` просто устанавливает значение `cont` и не имеет никакого другого эффекта. В следующем разделе вы увидите более сложный пример.

> **Совет.** Если выполнить компиляцию с флагом `-Xprint:selectivecps`, можно увидеть программный код, генерируемый CPS-трансформацией.

22.10. Трансформирование вложенных контекстов управления

CPS-трансформация существенно усложняется, когда дело доходит до вызовов трансформированных функций. Чтобы убедиться в этом, исследуем пример преобразования рекурсивного обхода древовидной структуры в итерации, подобный тому, что рассматривался в разделе 22.7 «Преобразование рекурсии в итерации». Но для простоты заменим дерево связанным списком:

```
def visit(a: List[String]): String @cps[String] = {
     if (a.isEmpty) "" else {
          shift {
               k: (Unit => String) => {
                    cont = k
                    a.head
               }
          }
          visit(a.tail)
     }
}
```

Как и в предыдущем примере, блок `shift` превращается в контекст управления:

```
new ControlContext[Unit, String, String](k => { cont = k; a.head })
```

На этот раз за блоком `shift` следует вызов метода `visit`, возвращающий другой экземпляр `ControlContext`.

> **Примечание.** В соответствии с объявлением метод `visit` возвращает экземпляр `String`, но в результате трансформации в действительности возвращается `ControlContext`. Выполнить эту трансформацию от компилятора требует аннотация `@cps`.

Если говорить точнее, блок `shift` замещается значением `()`, потому что функция продолжения имеет аргумент типа `Unit`. Таким образом, остальная часть вычислений имеет вид:

```
() => visit(a.tail)
```

Если следовать процедуре, описанной в первом примере, можно было бы вызвать метод `map` с этой функцией. Но так как она возвращает контекст управления, мы используем метод `flatMap`:

```
if (a.isEmpty) new ControlContext(k => k("")) else
  new ControlContext(k => { cont = k; a.head }).flatMap(() => visit(a.tail))
```

Ниже приводится определение метода `flatMap`:

```
class ControlContext[+A, -B, +C](val fun: (A => B) => C) {
    ...
    def flatMap[A1, B1, C1 <: B](f: A => Shift[A1, B1, C1]) =
        new ControlContext[A1, B1, C](
            (k1: (A1 => B1)) => fun(x: A => f(x).fun(k1)))
}
```

Интуитивно понятно, что это означает следующее. Если оставшаяся часть вычислений начинается с другого контекста управления, обрабатывающего оставшуюся часть оставшейся части, надо позволить ему сделать это. Так определяется обрабатываемое нами продолжение.

> **Примечание.** Обратите внимание на границу типа `C1 <: B`. Причина в том, что `f(x).fun(k1)` имеет тип `C1`, а функция `fun` ожидает получить функцию `A => B`.

Рассмотрим вызов:

```
val lst = List("Fred")
reset { visit(lst) }
```

Так как `lst` не является пустым списком:

```
reset {
    new ControlContext(k=>{ cont=k; lst.head }).flatMap(()=>visit(lst.tail))
}
```

Используя определение `flatMap`, получаем:

```
reset {
    new ControlContext(k1=>{ cont=() => visit(lst.tail).fun(k1); lst.head })
}
```

Метод `reset` сохраняет в `k1` функцию идентичности, и получается:

```
cont = () => visit(lst.tail).fun(x => x)
lst.head
```

Теперь рассмотрим вызов `cont`. Если бы использовался более длинный список, значение `lst.tail` было бы непустым списком, и мы снова получили бы тот же результат, но уже с вызовом `visit(lst.tail.tail)`. Но так как список исчерпан, `visit(lst.tail)` вернет

```
new ControlContext(k => k(""))
```

Применение функции идентичности даст в результате "".

> **Примечание.** Возврат пустой строки в данном случае выглядит немного странно, но дело в том, что здесь нельзя вернуть Unit, потому что cont ожидает получить значение типа `String`.

Как уже упоминалось, для использования продолжений в своих программах необязательно понимать все тонкости CPS-трансформаций. Но иногда полезно иметь некоторые базовые представления, особенно при отладке ошибок, связанных с типами данных.

Упражнения

1. В примере из раздела 22.1 «Сохранение и вызов продолжений» предполагается, что файл `myfile.txt` отсутствует. Теперь присвойте переменной `filename` имя другого несуществующего файла и вызовите `cont`. Что произойдет? Присвойте переменной `filename` имя существующего файла и снова вызовите `cont`. Что произойдет? Вызовите `cont` еще раз. Что произойдет? Сначала попробуйте представить весь процесс мысленно, а затем запустите программу для проверки своих предположений.
2. Улучшите пример из раздела 22.1 «Сохранение и вызов продолжений» так, чтобы имя следующего файла передавалось функции продолжения в качестве параметра.
3. Преобразуйте пример из раздела 22.7 «Преобразование рекурсии в итерации» в итератор. Конструктор итератора должен содержать блок `reset`, а метод `next` должен вызывать продолжение.

4. Пример в разделе 22.8 «Устранение инверсии управления» выглядит немного неприглядно – инструкция `reset` попадает в поле зрения прикладного программиста. Перенесите блок `reset` из метода `run` в обработчик событий от кнопки. Будет ли после этого прикладной программист пребывать в блаженном неведении о существовании продолжения?
5. Взгляните на следующий пример программы, использующей продолжения для превращения итераций в итератор:

```
object Main extends App {
     var cont: Unit => String = null
     val a = "Mary was a little lamb".split(" ")
     reset {
          var i = 0
          while (i < a.length) {
               shift {
                    k: (Unit => String) => {
                         cont = k
                         a(i)
                    }
               }
               i += 1
          }
          ""
     }
     println(cont())
     println(cont())
}
```

Скомпилируйте ее с флагом `-Xprint:selectivecps` и посмотрите на сгенерированный код. Как стала выглядеть инструкция `while` после CPS-трансформации?

Предметный указатель

Символы

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

А

Б

Л

M

H

О

П

P

C

Т

У

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по поч товому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Кей Хостманн

Scala для нетерпеливых

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 25.02.2013. Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 25,5. Тираж 300 экз.

Веб-сайт издательства: www.dmk-press.ru

Функциональное программирование

SCALA для нетерпеливых

Книга в сжатой форме описывает, что можно делать на языке Scala, и как это делать. Кей Хорстманн (Cay Horstmann), основной автор всемирного бестселлера «Core Java™», дает быстрое и практическое введение в язык программирования, основанное на примерах программного кода. Он знакомит читателя с концепциями языка Scala и приемами программирования небольшими «порциями», что позволяет быстро осваивать их и применять на практике. Практические примеры помогут вам пройти все стадии компетентности, от новичка до эксперта.

В книге рассматриваются:

- быстрое знакомство с интерпретатором Scala, синтаксисом, инструментами и уникальными идиомами языка;
- основные особенности языка: функции, массивы, ассоциативные массивы, кортежи, пакеты, импорт, обработка исключений и многое другое;
- знакомство с объектно-ориентированными особенностями языка Scala: классами, трейтами и наследованием;
- использование Scala для решения практических задач: обработка файлов, применение регулярных выражений и работа с XML-документами;
- использование функций высшего порядка и обширная библиотека коллекций;
- мощный механизм сопоставления с шаблонами и применение case-классов;
- создание многопоточных программ с использованием акторов;
- реализация предметно-ориентированных языков;
- исследование системы типов языка Scala;
- применение дополнительных мощных инструментов, таких как аннотации, неявные параметры и значения, и ограниченные продолжения.

Изучение языка Scala позволяет быстро достичь той грани, за которой начнут формироваться новые навыки программирования. Эта книга поможет программистам с опытом объектно-ориентированного программирования немедленно приступить к созданию практичных приложений и постепенно овладевать передовыми приемами программирования.

ISBN 978-5-94074-920-2

9 785940 749202 >

Internet-магазин:
www.dmk-press.ru
Книга – почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел./факс: (499) 725-5409
e-mail: books@alians-kniga.ru

ДМК Пресс
издательство
www.дмк.рф