3 Последующие шаги в Scala

В этой главе мы продолжим знакомство с языком Scala. Здесь будут рассмотрены более сложные функциональные возможности. Когда вы усвоите материал главы, у вас будет достаточно знаний для начала создания полезных сценариев на Scala. Мы вновь рекомендуем по мере чтения текста получать практические навыки с помощью приводимых примеров. Лучше всего осваивать Scala, начиная создавать код на этом языке.

Шаг 7. Параметризация массивов типами

В Scala создавать объекты или экземпляры класса можно с помощью ключевого слова `new`. При создании объекта в Scala вы можете выполнить его параметризацию с использованием значений и типов. Параметризация означает конфигурирование экземпляра при его создании. Параметризация экземпляра значениями производится путем передачи конструктору объектов в круглых скобках. Например, следующий код Scala:

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

создает новый объект `java.math.BigInteger`, выполняя его параметризацию значением `"12345"`.

Параметризация экземпляра типами производится указанием одного или нескольких типов в квадратных скобках. Пример показан в листинге 3.1. Здесь `greetStrings` является типом `Array[String]` («массив строк»), инициализируемым длиной `3` путем его параметризации значением `3` в первой строке кода. Если запустить код в листинге 3.1 в качестве сценария, вы увидите еще одно приветствие `Hello, world!`. Учтите, что при параметризации экземпляра как типом, так и значением тип стоит первым и указывается в квадратных скобках, а за ним следует значение в круглых скобках.

Листинг 3.1. Параметризация массива типом

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

greetStrings(0) = "Hello"
```

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

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

ПРИМЕЧАНИЕ

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

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

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

При условии применения имеющегося в Scala вывода типов эта строка кода семантически эквивалентна первой строке листинга 3.1. Но в этой форме показано, что, хотя та часть параметризации, которая относится к типу (название типа в квадратных скобках), формирует часть типа экземпляра, часть параметризации, которая относится к значению (значения в круглых скобках), в формировании не участвует. Типом `greetStrings` является `Array[String]`, а не `Array[String](3)`.

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

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

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

Эти три строки кода иллюстрируют важное понятие, помогающее осмыслить значение для Scala `val`-переменных. Когда переменная определяется с помощью `val`, повторно присвоить ей значение нельзя, но объект, на который она ссылается, потенциально может быть изменен. Следовательно, в данном случае задать `greetStrings` значение другого массива невозможно — переменная `greetStrings` всегда будет указывать на один и тот же экземпляр типа `Array[String]`, с которым она была инициализирована. Но впоследствии в элементы типа `Array[String]` можно вносить изменения, следовательно, сам массив является изменяемым.

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

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

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

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

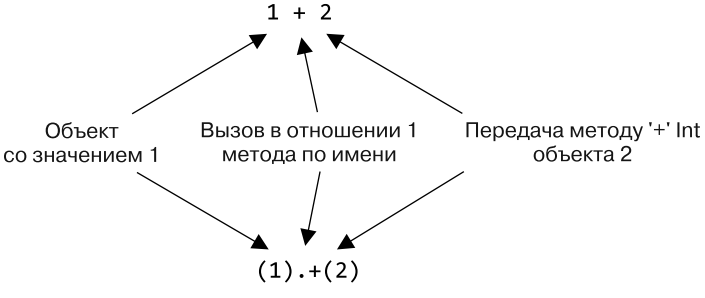

Рис. 3.1. Все операции в Scala являются вызовами методов

Еще одна весьма важная идея, проиллюстрированная в этом примере, поможет понять, почему доступ к элементам массивов Scala осуществляется с использованием круглых скобок. В Scala меньше особых случаев по сравнению с Java. Массивы в Scala, как и в случае с любыми другими классами, являются просто экземплярами классов. При использовании круглых скобок, окружающих одно или несколько значений переменной, Scala преобразует код в вызов метода по имени `apply` применительно к этой переменной. Следовательно, код `greetStrings(i)` преобразуется в код `greetStrings.apply(i)`. Получается, что элемент массива в Scala является просто вызовом обычного метода, ничем не отличающегося от любого своего собрата. Этот принцип не ограничивается массивами: любое применение объекта в отношении каких-либо аргументов в круглых скобках будет преобразовано в вызов метода `apply`. Разумеется, этот код будет откомпилирован, только если в этом типе объекта определяется метод `apply`. То есть это не особый случай, а общее правило.

---

[1] Этот метод to фактически возвращает не массив, а иную последовательность, содержащую значения 0, 1 и 2, перебор которых выполняется выражением for. Последовательности и другие коллекции будут рассматриваться в главе 17.

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

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

будет преобразован в код:

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

Таким образом, следующий код является семантическим эквивалентом коду листинга 3.1:

```
val greetStrings = new Array[String](3)
greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")
for (i <- 0.to(2))
  print(greetStrings.apply(i))
```

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

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

Листинг 3.2. Создание и инициализация массива

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

Фактически в листинге 3.2 вызывается фабричный метод по имени `apply`, создающий и возвращающий новый массив. Этот метод `apply` получает переменное количество аргументов[1] и определяется в объекте-спутнике `Array`. Подробнее объекты-спутники будут рассматриваться в разделе 4.3. Если вам приходилось программировать на Java, можете воспринимать это как вызов статического метода

[1] Списки аргументов переменной длины или повторяющиеся параметры рассматриваются в разделе 8.8.

по имени `apply` в отношении класса `Array`. Менее лаконичный способ вызова того же метода `apply` выглядит так:

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

Шаг 8. Использование списков

Одной из превосходных отличительных черт функционального стиля программирования является полное отсутствие у методов побочных эффектов. Единственным действием метода должно быть вычисление и возвращение значения. Получаемые от применения такого подхода преимущества заключаются в том, что методы становятся менее запутанными, что упрощает их чтение и повторное использование. Еще одно преимущество (в статически типизированных языках) заключается в том, что все, попадающее в метод и выходящее за его пределы, проходит проверку на принадлежность к определенному типу, поэтому логические ошибки, скорее всего, проявятся сами по себе в виде ошибок типов. Применение данной функциональной философии к миру объектов означает превращение этих объектов в неизменяемые.

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

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

Листинг 3.3. Создание и инициализация списка

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

Код в листинге 3.3 создает новую `val`-переменную по имени `oneTwoThree`, инициализируемую типом `newList[Int]` с целочисленными элементами `1`, `2` и `3`[1]. Из-за своей неизменяемости списки ведут себя подобно строкам в Java: при вызове метода в отношении списка из-за имени этого метода может казаться, что обрабатываемый список будет изменен, но вместо этого создается и возвращается новый список с новым значением. Например, в `List` для объединения списков имеется метод, обозначаемый как `:::`. Используется он следующим образом:

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

[1] Применять запись new List не нужно, поскольку List.apply() определен в объекте-спутнике scala.List как фабричный метод. Более подробно объекты-спутники рассматриваются в разделе 4.3.

```
val oneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " and " + threeFour + " were not mutated.")
println("Thus, " + oneTwoThreeFour + " is a new list.")
```

Запустив этот сценарий, вы увидите следующую картину:

```
List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new list.
```

Возможно, со списками чаще всего будет использоваться оператор `::`, который произносится `cons` («конс»). Он добавляет в начало существующего списка новый элемент и возвращает получившийся в результате этого список. Например, если запустить на выполнение следующий сценарий:

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

вы увидите:

```
List(1, 2, 3)
```

ПРИМЕЧАНИЕ

> В выражении 1 :: twoThree метод :: является методом его правого операнда — списка twoThree. Можно подумать, что с ассоциативностью метода :: что-то не то, но есть простое мнемоническое правило: если метод используется в виде оператора, например a * b, то он вызывается в отношении левого операнда, как в выражении a.*(b), если только имя метода не заканчивается двоеточием. А если оно заканчивается двоеточием, то метод вызывается в отношении правого операнда. Поэтому в выражении 1 :: twoThree метод :: вызывается в отношении twoThree с передачей ему 1, как здесь: twoThree.::(1). Ассоциативность операторов более подробно будет рассматриваться в разделе 5.9.

Исходя из того, что самым кратким вариантом указания пустого списка является `Nil`, одним из способов инициализации новых списков выступает связывание элементов с помощью `cons`-оператора с `Nil` в качестве последнего элемента[1]. Например, следующий сценарий создаст ту же картину на выходе, что и предыдущий `List(1, 2, 3)`:

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Имеющийся в Scala класс `List` укомплектован весьма полезными методами, многие из которых показаны в табл. 3.1. Вся эффективность списков будет раскрыта в главе 16.

---

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

Почему со списками не следует применять операцию добавления

В классе `List` предлагается операция добавления, которая записывается как `:+` (о ней говорится в главе 24), но очень редко используется, поскольку время, которое она тратит на добавление к списку, увеличивается линейно с размером списка, а на добавление в начало списка с помощью метода `::` всегда затрачивается одно и то же время. Если нужно добиться эффективности при создании списка путем дополнения элементами, можно добавлять их в начало, а завершив добавление, вызвать метод реверсирования `reverse`. Или же можно воспользоваться изменяемым списком `ListBuffer`, предлагающим операцию добавления, а затем, завершив добавление, вызвать метод `toList` и преобразовать его в обычный список. Список типа `ListBuffer` будет рассмотрен в разделе 22.2.

Таблица 3.1. Некоторые методы класса List и их использование

Что используется	Что этот метод делает
List() или Nil	Создает пустой список List
List("Cool", "tools", "rule")	Создает новый список типа List[String] с тремя значениями: "Cool", "tools" и "rule"
val thrill = "Will" :: "fill" :: "until" :: Nil	Создает новый список типа List[String] с тремя значениями: "Will", "fill" и "until"
List("a", "b") ::: List("c", "d")	Объединяет два списка (возвращает новый список типа List[String] со значениями "a", "b", "c" и "d")
thrill(2)	Возвращает элемент с индексом 2 (при начале отсчета с нуля) списка thrill (возвращает "until")
thrill.count(s => s.length == 4)	Подсчитывает количество строковых элементов в thrill, имеющих длину 4 (возвращает 2)
thrill.drop(2)	Возвращает список thrill без его первых двух элементов (возвращает List("until"))
thrill.dropRight(2)	Возвращает список thrill без самых правых двух элементов (возвращает List("Will"))
thrill.exists(s => s == "until")	Определяет наличие в списке thrill строкового элемента, имеющего значение "until" (возвращает true)
thrill.filter(s => s.length == 4)	Возвращает список всех элементов списка thrill, имеющих длину 4, соблюдая порядок их следования в списке (возвращает List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	Показывает, оканчиваются ли все элементы в списке thrill буквой "l" (возвращает true)
thrill.foreach(s => print(s))	Выполняет инструкцию print в отношении каждой строки в списке thrill (выводит "Willfilluntil")
thrill.foreach(print)	Делает то же самое, что и предыдущий код, но с использованием более лаконичной формы записи (также выводит "Willfilluntil")
thrill.head	Возвращает первый элемент в списке thrill (возвращает "Will")

Что используется	Что этот метод делает
thrill.init	Возвращает список всех элементов списка thrill, кроме последнего (возвращает List("Will", "fill"))
thrill.isEmpty	Показывает, не пуст ли список thrill (возвращает false)
thrill.last	Возвращает последний элемент в списке thrill (возвращает "until")
thrill.length	Возвращает количество элементов в списке thrill (возвращает 3)
thrill.map(s => s + "y")	Возвращает список, который получается в результате добавления "y" к каждому строковому элементу в списке thrill (возвращает List("Willy", "filly", "untily"))
thrill.mkString(", ")	Создает строку с элементами списка (возвращает "Will, fill, until")
thrill.filterNot(s => s.length == 4)	Возвращает список всех элементов в порядке их следования в списке thrill, за исключением имеющих длину 4 (возвращает List("until"))
thrill.reverse	Возвращает список, содержащий все элементы списка thrill, следующие в обратном порядке (возвращает List("until", "fill", "Will"))
thrill.sort((s, t) => s.charAt(0).toLower < t.charAt(0).toLower)	Возвращает список, содержащий все элементы списка thrill в алфавитном порядке с первым символом, преобразованным в символ нижнего регистра (возвращает List("fill", "until", "will"))
thrill.tail	Возвращает список thrill за минусом его первого элемента (возвращает List("fill", "until"))

Шаг 9. Применение кортежей

Еще одним полезным объектом-контейнером является *кортеж*. Кортежи, как и списки, нельзя изменять, но, в отличие от списков, в кортежах могут содержаться различные типы элементов. Список может быть типа `List[Int]` или `List[String]`, а кортеж способен содержать одновременно как целые числа, так и строки. Кортежи находят широкое применение, например при возвращении из метода сразу нескольких объектов. Там, где на Java для хранения нескольких возвращаемых значений зачастую приходится создавать JavaBean-подобный класс, в Scala можно просто вернуть кортеж. Все делается просто: для создания экземпляра нового кортежа, содержащего объекты, нужно лишь заключить объекты в круглые скобки, отделив их друг от друга запятыми. После создания экземпляра кортежа доступ к его элементам можно получить, используя точку, знак подчеркивания и индекс элемента, причем подсчет элементов начинается с единицы. Пример показан в листинге 3.4.

Листинг 3.4. Создание и использование кортежа

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

В первой строке листинга 3.4 создается новый кортеж, содержащий в качестве первого элемента целочисленное значение `99`, а в качестве второго — строку `"Luftballons"`. Scala выводит тип кортежа в виде `Tuple2[Int, String]`, а также присваивает этот тип паре переменных. Во второй строке выполняется доступ к полю `_1`, в результате чего получается первый элемент `99`. Символ точки (`.`) во второй строке аналогичен той точке, которая используется для доступа к полю или вызова метода. В данном случае выполняется доступ к полю по имени `_1`. Если запустим этот сценарий на выполнение, получим следующий результат:

```
99
Luftballons
```

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

Обращение к элементам кортежа

Возникает вопрос: а почему к элементам кортежа нельзя обратиться точно так же, как к элементам списка, например `pair(0)`? Дело в том, что используемый в списках метод `apply` всегда возвращает один и тот же тип, а в кортеже все элементы могут быть разных типов: у `_1` может быть один получаемый тип, у `_2` — другой и т. д. Эти числа вида `_N` начинаются с единицы, а не с нуля, поскольку начало отсчета с единицы традиционно используется в языках со статически типизированными кортежами, например `Haskell` и `ML`.

Шаг 10. Использование наборов и отображений

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

Например, в API Scala содержится основной *трейт* для наборов, который аналогичен Java-интерфейсу. (Более подробно трейты рассматриваются в главе 12.) Затем Scala предоставляет два подчиненных трейта — один для изменяемых, а второй для неизменяемых наборов.

[1] Хотя концептуально можно создавать кортежи любой длины, на данный момент библиотека Scala определяет их только до Tuple22.

На рис. 3.2 показано, что для всех трех трейтов используется одно и то же простое имя `Set`. Но их полные имена отличаются друг от друга, поскольку все трейты размещаются в разных пакетах. Конкретный набор классов в Scala API, такой как набор, содержащий классы `HashSet` (см. рис. 3.2), является расширением либо изменяемого, либо неизменяемого трейта `Set`. (В то время как в Java вы реализуете интерфейсы, в Scala вы расширяете, или подмешиваете, трейты.) Следовательно, если нужно воспользоваться `HashSet`, можно в зависимости от потребностей выбирать между его изменяемой и неизменяемой разновидностями. Исходный способ создания набора показан в листинге 3.5.

Листинг 3.5. Создание, инициализация и использование неизменяемого набора

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

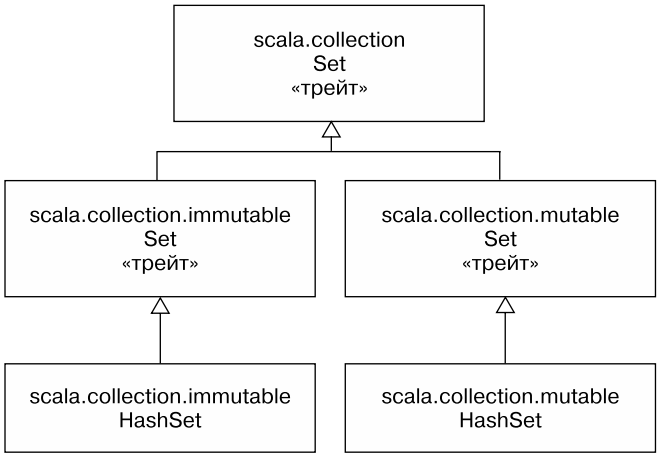

Рис. 3.2. Иерархия классов для наборов Scala

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

Чтобы добавить к набору новый элемент, в отношении набора вызывается метод `+`, которому передается этот новый элемент. Как для изменяемых, так и для неизменяемых наборов метод `+` создает и возвращает новый набор с добавленным элементом. В листинге 3.5 работа ведется с неизменяемым набором. В изменяемых наборах предоставляется конкретный метод `+=`, однако для неизменяемых наборов он не предлагается.

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

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

Следовательно, во второй строке кода листинга 3.5 `var`-переменной `jetSet` присваивается новый набор, содержащий `"Boeing"`, `"Airbus"` и `"Lear"`. И наконец, в последней строке кода листинга 3.5 выводятся данные о том, содержится ли в наборе строка `"Cessna"`. (Как и ожидалось, выводится `false`.)

Если нужен изменяемый набор, следует, как показано в листинге 3.6, воспользоваться инструкцией `import`.

Листинг 3.6. Создание, инициализация и использование изменяемого набора

```
import scala.collection.mutable
val movieSet = mutable.Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

В первой строке листинга 3.6 импортируется изменяемый набор `Set`. Как и в Java, инструкция `import` позволяет использовать простое имя, например `Set`, вместо длинного полного имени. В результате при указании `Set` во второй строке компилятор знает, что подразумевается под `scala.collection.mutable.Set`. В этой строке `movieSet` инициализируется новым изменяемым набором, содержащим строки `"Hitch"` и `"Poltergeist"`. В следующей строке к изменяемому набору добавляется `"Shrek"`, для чего в отношении набора вызывается метод `+=` с передачей ему строки `"Shrek"`. Как уже упоминалось, `+=` является методом, определенным в изменяемых наборах. Если хотите, можете вместо кода `movieSet += "Shrek"` воспользоваться кодом `movieSet.+=("Shrek")`[1].

Хотя рассмотренной исходной реализации наборов, выполняемых изменяемыми и неизменяемыми фабричными методами `Set`, скорее всего, будет достаточно для большинства ситуаций, временами может потребоваться явный класс набора. К счастью, при этом используется аналогичный синтаксис. Нужно просто импортировать нужный класс и воспользоваться фабричным методом в отношении его объекта-спутника. Например, если нужен неизменяемый `HashSet`, можно сделать следующее:

```
import scala.collection.immutable.HashSet

val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")
```

Еще одним полезным классом коллекций в Scala является отображение — `Map`. Как и для наборов, Scala предоставляет изменяемые и неизменяемые версии `Map` с использованием иерархии классов. Как показано на рис. 3.3, иерархия

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

классов для отображений во многом похожа на иерархию для наборов. В пакете `scala.collection` есть основной трейт `Map` и два подчиненных трейта отображений `Map`: изменяемый вариант в `scala.collection.mutable` и неизменяемый вариант в `scala.collection.immutable`.

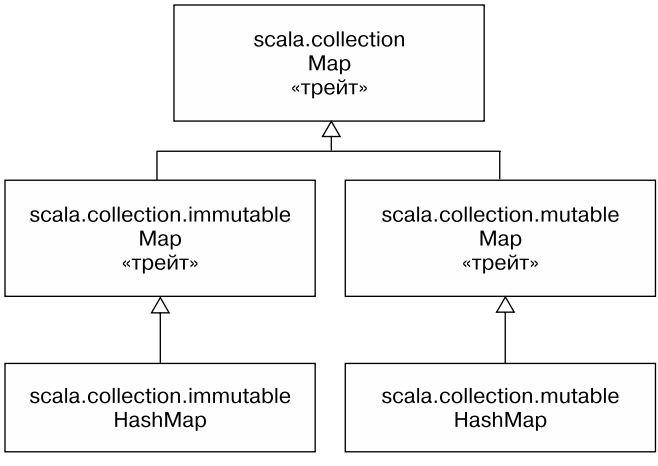

Рис. 3.3. Иерархия классов для Scala-отображений

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

Листинг 3.7. Создание, инициализация и использование изменяемого отображения

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

Например, в листинге 3.7 показана работа с изменяемым отображением. В первой строке импортируется изменяемое отображение. Затем определяется `val`-переменная `treasureMap`, которая инициализируется пустым изменяемым отображением, имеющим целочисленные ключи и строковые значения. Отображение задается пустым, поскольку фабричному методу ничего не передается (в круглых скобках в `Map[Int, String]()` ничего не указано)[1]. В следующих трех

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

строках к отображению добавляются пары «ключ — значение», для чего используются методы -> и +=. Как уже было показано, компилятор Scala преобразует выражения бинарных операций вида 1 -> "Go to island." в код (1).->("Go to island."). Следовательно, когда указывается 1 -> "Go to island.", фактически в отношении объекта 1 вызывается метод по имени ->, которому передается строка со значением "Go to island.". Метод ->, который можно вызвать в отношении любого объекта в программе Scala, возвращает двухэлементный кортеж, содержащий ключ и значение[1]. Затем этот кортеж передается методу += объекта отображения, на который ссылается treasureMap. И наконец, в последней строке выводится значение, соответствующее в treasureMap ключу 2.

При запуске этот код выведет следующие данные:

```
Find big X on ground.
```

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

Листинг 3.8. Создание, инициализация и использование неизменяемого отображения

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

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

Шаг 11. Обучение распознаванию функционального стиля

Как упоминалось в главе 1, Scala позволяет программировать в императивном стиле, но подталкивает вас перейти преимущественно к функциональному стилю. Если к Scala вы пришли с опытом работы в императивном стиле, к примеру, если приходилось программировать на Java, то одной из основных сложностей, с которой можно столкнуться, станет программирование в функциональном стиле. Мы понимаем, что поначалу этот стиль может быть неизвестен, и в данной книге стараемся перевести вас из одного состояния в другое. От вас также потребуются

[1] Используемый в Scala механизм, позволяющий вызывать метод -> в отношении любого объекта, — неявное преобразование — будет рассмотрен в главе 21.

усилия, которые мы настоятельно рекомендуем приложить. Мы уверены, что при наличии опыта работы в императивном стиле изучение программирования в функциональном стиле не только позволит вам стать более квалифицированным программистом на Scala, но и расширит кругозор, сделав вас более ценным программистом в общем смысле.

Сначала нужно усвоить разницу между двумя стилями, отражающуюся в коде. Один верный признак заключается в том, что если код содержит `var`-переменные, то он, вероятнее всего, написан в императивном стиле. Если в коде вообще не содержатся `var`-переменные, то есть в нем только `val`-переменные, то он, вероятнее всего, написан в функциональном стиле. Следовательно, одним из способов приближения к функциональному стилю является попытка обойтись в программах без `var`-переменных.

Обладая багажом императивности, то есть опытом работы с такими языками, как Java, C++ или C#, `var`-переменные можно рассматривать в качестве обычных переменных, а `val`-переменные — в качестве переменных особого вида. В то же время, если у вас имеется опыт работы в функциональном стиле на таких языках, как Haskell, OCaml или Erlang, `val`-переменные можно представлять как обычные, а `var`-переменные — как некое кощунственное обращение с кодом. Но с точки зрения Scala `val`- и `var`-переменные — всего лишь два разных инструмента в вашем арсенале средств и оба они одинаково полезны. Scala склоняет вас к применению `val`-переменных, но, по сути, дает возможность воспользоваться тем инструментом, который лучше подходит для решаемой задачи. И, тем не менее, даже если вы согласны с подобной философией, то поначалу можете испытывать трудности, связанные с избавлением от `var`-переменных в коде.

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

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

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

```
def printArgs(args: Array[String]): Unit = {
    for (arg <- args)
        println(arg)
}
```

или вот так:

```
def printArgs(args: Array[String]): Unit = {
    args.foreach(println)
}
```

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

Но вы можете пойти еще дальше. Реорганизованный метод printArgs нельзя отнести к чисто функциональным, поскольку у него имеются побочные эффекты. В данном случае таким эффектом является вывод в поток стандартного устройства вывода. Признаком функции, имеющей побочные эффекты, является то, что типом результата у нее выступает Unit. Если функция не возвращает никакого интересного значения, о чем, собственно, и свидетельствует тип результата Unit, то единственным способом внесения этой функцией какого-либо изменения в окружающий мир является проявление какого-то побочного эффекта. Более функциональным подходом будет определение метода, форматирующего передаваемые аргументы с целью их последующего вывода на стандартное устройство и, как показано в листинге 3.9, просто возвращающего отформатированную строку.

Листинг 3.9. Функция без побочных эффектов или var-переменных

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

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

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

У каждой полезной программы, вероятнее всего, будут какие-либо побочные эффекты. Отдавая предпочтение методам без побочных эффектов, вы будете стремиться к разработке программ, в которых такие эффекты сведены к минимуму. Одним из преимуществ подобного подхода станет упрощение тестирования ваших программ.

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

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

Имеющийся в Scala метод assert проверяет переданное ему булево выражение и, если оно вычисляется в false, выдает ошибку AssertionError. Если передан-

ное булево выражение вычисляется в `true`, метод просто возвращает управление вызвавшему его коду. Более подробно о тестах, проводимых с помощью `assert`, и тестировании речь пойдет в главе 14.

И все-таки нужно иметь в виду, что ни `var`-переменные, ни побочные эффекты не следует рассматривать как нечто абсолютно неприемлемое. Scala не является чисто функциональным языком, заставляющим вас программировать все в функциональном стиле. Scala представляет собой гибрид императивного и функционального языка. Может оказаться, что в некоторых ситуациях для решения текущей задачи больше подойдет императивный стиль, и тогда вы должны им воспользоваться без всяких колебаний. Но чтобы помочь вам разобраться в программировании без применения `var`-переменных, в главе 7 будет показано множество конкретных примеров кода с использованием `var`-переменных и рассмотрены способы их преобразования в `val`-переменные.

Сбалансированная позиция для программистов, работающих на Scala

Отдавайте предпочтение `val`-переменным, неизменяемым объектам и методам без побочных эффектов. Старайтесь применять их в первую очередь. Используйте `var`-переменные, изменяемые объекты и методы с побочными эффектами при необходимости и наличии четкой обоснованности их применения.

Шаг 12. Считывание строк из файла

Сценарии, выполняющие небольшие повседневные задачи, часто нуждаются в обработке строк, взятых из файлов. В этом разделе будет создан сценарий, считывающий строки из файла и выводящий их на стандартное устройство, предваряя каждую строку количеством содержащихся в ней символов. Первая версия сценария показана в листинге 3.10.

Листинг 3.10. Считывание строк из файла

```
import scala.io.Source

if (args.length > 0) {
  for (line <- Source.fromFile(args(0)).getLines())
    println(line.length + " " + line)
}
else
  Console.err.println("Please enter filename")
```

Сценарий начинается с импорта класса `Source` из пакета `scala.io`. Затем он проверяет, указан ли в командной строке хотя бы один аргумент. Если да, то первый аргумент рассматривается как имя открываемого и обрабатываемого файла. Выражение `Source.fromFile(args(0))` пробует открыть указанный файл и возвращает

объект типа `Source`, в отношении которого вызывается метод `getLines`. Этот метод возвращает значение типа `Iterator[String]`, в котором при каждой итерации предоставляется по одной строке с исключением символа конца строки.

Выражение `for` выполняет последовательный перебор этих строк и выводит длину каждой строки, затем пробел, а затем саму строку. Если аргументы в командной строке не указаны, финальное условие `else` выведет сообщение в поток стандартного устройства. При добавлении этого кода в файл countchars1.scala и запуске его с указанием самого этого файла:

```
$ scala countchars1.scala countchars1.scala
```

вы увидите следующий текст:

```
22 import scala.io.Source
0
22 if (args.length > 0) {
0
51     for (line <- Source.fromFile(args(0)).getLines())
37         println(line.length + " " + line)
1 }
4 else
46     Console.err.println("Please enter filename")
```

Хотя сценарий в его текущем виде выводит необходимую информацию, может быть, вам захочется выстроить числа, выровняв их по правому краю, и добавить символ вертикальной черты, чтобы вид выводимой информации стал таким:

```
22 | import scala.io.Source
 0 |
22 | if (args.length > 0) {
 0 |
51 |   for (line <- Source.fromFile(args(0)).getLines())
37 |     println(line.length + " " + line)
 1 | }
 4 | else
46 |   Console.err.println("Please enter filename")
```

Чтобы реализовать задуманное, можно дважды выполнить последовательный перебор строк. При первом переборе определить максимальную ширину, требуемую какому-либо количеству символов в строке. А при втором переборе вывести данные, используя вычисленную ранее максимальную ширину. Поскольку перебор строк будет выполняться дважды, вы можете также присвоить эти строки переменной:

```
val lines = Source.fromFile(args(0)).getLines().toList
```

Завершающий выражение вызов метода `toList` нужен потому, что метод `getLines` возвращает итератор. По мере проведения итерации через итератор он истощается. Преобразование в список посредством вызова `toList` дает возможность выполнять итерацию любое количество раз без многократного выделения памяти для хранения всех строк из файла. Получается, что переменная `lines` ссылается на список строк с содержимым файла, указанного в командной строке.

Затем, поскольку вам нужно вычислять ширину позиции для количества символов в каждой строке дважды — по одному разу за каждую итерацию, из этого выражения можно вывести небольшую функцию, подсчитывающую ширину, необходимую для символов, отображающих длину переданной строки:

```
def widthOfLength(s: String) = s.length.toString.length
```

Применяя эту функцию, максимальную ширину можно вычислить следующим образом:

```
var maxWidth = 0
for (line <- lines)
  maxWidth = maxWidth.max(widthOfLength(line))
```

Здесь с помощью выражения `for` выполняется последовательный перебор всех строк, вычисляется ширина символов, показывающих длину строки, и, если она больше текущего максимума, ее значение присваивается `var`-переменной `maxWidth`, для которой было установлено начальное значение `0`. (Метод `max`, который можно вызвать в отношении любого объекта типа `Int`, возвращает самое большое число, сравнивая значение, в отношении которого он был вызван, и значение, которое ему было передано.) В качестве альтернативного варианта, если предпочтительнее искать максимум без использования `var`-переменных, можно сначала определить самую длинную строку:

```
val longestLine = lines.reduceLeft(
  (a, b) => if (a.length > b.length) a else b
)
```

Метод `reduceLeft` применяет переданную ему функцию к первым двум элементам в списке `lines`, затем — к результату первого применения и к следующему элементу в `lines` и так далее до конца списка. Результатом каждого применения будет самая длинная строка из встреченных до сих пор, поскольку переданная функция `(a, b) => if (a.length > b.length) a else b` возвращает самую длинную из двух переданных строк. Метод `reduceLeft` вернет результат последнего применения функции, который в данном случае станет самым длинным строковым элементом списка `lines`.

Получив результат, можно вычислить максимальную ширину, передав самую длинную строку функции `widthOfLength`:

```
val maxWidth = widthOfLength(longestLine)
```

Теперь останется только вывести строки с надлежащим форматированием. Это можно сделать следующим образом:

```
for (line <- lines) {
  val numSpaces = maxWidth - widthOfLength(line)
  val padding = " " * numSpaces
  println(padding + line.length + " | " + line)
}
```

В этом выражении `for` еще раз выполняет последовательный перебор элементов списка `lines`. Для каждой строки сначала вычисляется количество пробелов,

устанавливаемых перед указанием длины строки, и это количество присваивается переменной `numSpaces` с помощью выражения `" " * numSpaces`. И наконец, выводится информация с надлежащим форматированием. Весь сценарий показан в листинге 3.11.

Листинг 3.11. Вывод отформатированного количества символов каждой строки файла

```
import scala.io.Source

def widthOfLength(s: String) = s.length.toString.length

if (args.length > 0) {

  val lines = Source.fromFile(args(0)).getLines().toList

  val longestLine = lines.reduceLeft(
      (a, b) => if (a.length > b.length) a else b
  )
  val maxWidth = widthOfLength(longestLine)

  for (line <- lines) {
    val numSpaces = maxWidth - widthOfLength(line)
    val padding = " " * numSpaces
    println(padding + line.length + " | " + line)
  }
}
else
  Console.err.println("Please enter filename")
```

Резюме

Знания, полученные в этой главе, позволят вам приступить к применению Scala для решения небольших задач, в особенности тех, для которых используются сценарии. В последующих главах рассмотренные темы мы изучим глубже, также будут представлены другие, не затронутые здесь темы.