Функциональное программирование

Кей Хорстман

SCALA

для нетерпеливых

Второе издание,
переработанное и дополненное

ДМК
издательство

УДК 004.432.42Scala
ББК 32.973-018.1
Х84

Хорстманн К.

Х84 Scala для нетерпеливых / пер. с англ. А. Н. Киселева – 2-е изд. – М.: ДМК Пресс, 2017. – 414 с.: ил.

ISBN 978-5-97060-536-3

Книга в сжатой форме описывает, что можно делать на языке Scala, и как это делать. Кей Хорстманн, 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.

Второе издание было обновлено до поддержки версии Scala 2.12 и демонстрирует самые современные приемы использования языка. В него было добавлено описание последних нововведений в Scala, включая интерполяцию строк, динамический вызов, неявные классы и объекты Future.

Издание предназначено для программистов на Java, C++ и C#, которые желают освоить язык Scala и в целом функциональное программирование.

УДК 004.432.42Scala
ББК 32.973-018.1

ISBN 978-0-13-454056-6 (анг.)
ISBN 978-5-97060-536-3 (рус.)

Содержание

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 в формате 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 командной оболочки в своей операционной системе;
- ввести команду `scala` и нажать клавишу **Enter**.

Теперь попробуйте ввести следующие команды, завершая каждую нажатием клавиши **Enter**. Каждый раз интерпретатор будет выводить ответ, как показано на рис. 1.1. 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 сообщает тип результата – `Int`, `Double` и `java.lang.String`.

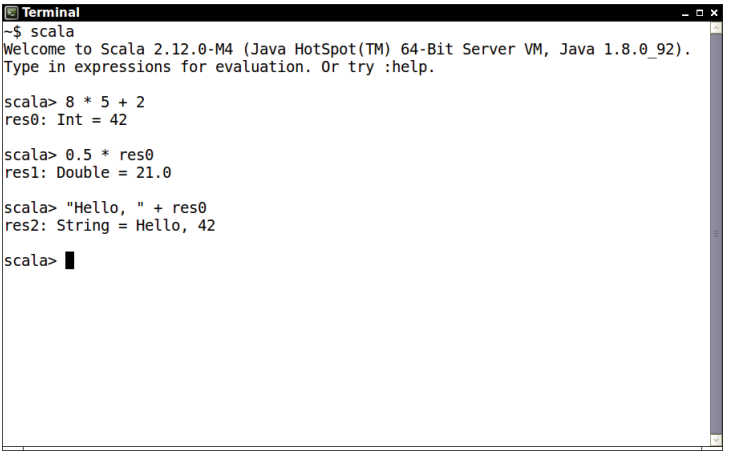

Рис. 1.1. Интерпретатор Scala

Совет. Не любите пользоваться командной оболочкой? Существует несколько IDE (Integrated Development Environment – интегрированная среда разработки), поддерживающих язык Scala, в которых имеется встроенная интерактивная оболочка, куда можно вводить выражения и получать результаты. На рис. 1.2 показана такая интерактивная оболочка в Eclipse.

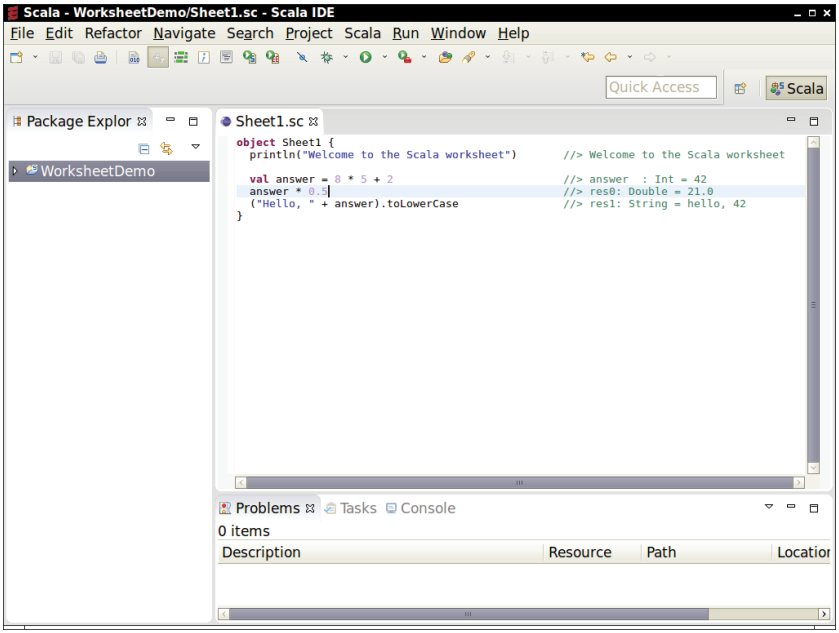

Рис. 1.2. Интерактивная оболочка Scala

Вызывая методы, попробуйте использовать функцию *автоматического дополнения*, привязанную к клавише **Tab**. Попробуйте ввести `res2.to` и нажать **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 **Tab**. 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 циклом *чтения–вычисления–вывода* (read-eval-print loop, REPL).

Строго говоря, программа `scala` *не* является интерпретатором. За кулисами она быстро компилирует введенные команды в байт-код и выполняет его в виртуальной машине Java. По этой причине большинство программистов на Scala предпочитает 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, если экспериментировать с более сложными примерами, их можно сначала компоновать в редакторе, а затем копировать в окно REPL.

Совет. В интерактивной оболочке REPL введите `:help`, чтобы увидеть список полезных команд. Все команды начинаются с двоеточия. Например, команда `:warnings` выводит подробную информацию о последних предупреждениях компилятора. Кроме того, вы можете вводить лишь часть команды. Например, команда `:w` действует точно так же, как `:warnings`, – по крайней мере пока, потому что нет другой команды, начинающейся с символа `w`.

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 считается 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 в языке 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», для работы 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 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 + – это имя 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 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 целочисленного значения.) 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

Вы уже видели, как вызывать методы объектов, такие как:

```
"Hello".intersect("World")
```

Если метод не имеет параметров, круглые скобки можно опустить. Например, в классе `StringOps` имеется метод `sorted`, который можно вызывать без `()`. Он возвращает новую строку, в которой все буквы отсортированы в алфавитном порядке. Попробуйте выполнить такой вызов:

```
"Bonjour".sorted // Вернет строку "Bjnooru"
```

Главное правило: метод без параметров и не изменяющий объектов может вызываться без круглых скобок. Подробнее мы обсудим эту особенность в главе 5.

Математические методы в Java, такие как `sqrt`, объявлены как статические методы класса `Math`. В Scala такие методы определены в объектах-одиночках (singleton objects), которые будут обсуждаться в главе 6. Пакет может иметь *объект пакета*. В этом случае можно импортировать пакет и использовать методы этого пакета без использования префикса:

```
import scala.math._ // Символ _ в Scala считается "групповым",
                    // как звездочка (*) в Java
sqrt(2) // Вернет 1.4142135623730951
pow(2, 4) // Вернет 16.0
min(3, Pi) // Вернет 3.0
```

Пакет `scala.math` можно не импортировать, но тогда придется указать полное имя пакета перед методом:

```
scala.math.sqrt(2)
```

> **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)`. Однако в этой книге мы всегда будем использовать префикс `scala.` для большей ясности.

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

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)
```

Здесь `Random` – это объект-одиночка, генератор случайных чисел, определенный в пакете `scala.util`. Попробуйте выполнить эту команду в REPL – вы получите число вида `1039447980491200275486540240713`.

1.6. Метод apply

В Scala 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 записывается как `s[i]`; в Java – как `s.charAt(i)`.) Попробуйте выполнить в REPL следующую команду:

```
val s = "Hello"
s(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 `s(4)` фактически является краткой формой записи

```
s.apply(4)
```

Почему не используется оператор `[]`? Дело в следующем: последовательность `s` элементов типа `T` можно представить как функцию от {0, 1, . . . , *n* – 1}, возвращающую `T`, которая отображает *i* в *s(i)*, т. е. в *i*-й элемент коллекции.

Этот аргумент выглядит еще более убедительным применительно к ассоциативным массивам. В главе 4 вы увидите, что поиск значения по ключу в ассоциативном массиве *map* выполняется как *map*(*key*). Концептуально *map* – это функция, преобразующая ключи в значения, поэтому правильнее записывать ее с использованием нотации функций.

> **Внимание.** Иногда нотация `()` конфликтует с другой особенностью Scala: неявными параметрами. Например, выражение
>
> `"Bonjour".sorted(3)`
>
> вызовет ошибку, потому что метод `sorted` с необязательным аргументом, определяющим порядок сортировки, но число `3` таковым не является. Чтобы преодолеть эту проблему, можно заключить выражение в круглые скобки:
>
> `("Bonjour".sorted)(3)`
>
> или явно вызвать метод `apply`:
>
> `"Bonjour".sorted.apply(3)`

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`.

> **Примечание.** На всем протяжении главы мы предполагали, что код на Scala выполняется в виртуальной машине Java. Это действительно так для стандартного дистрибутива Scala. Однако проект Scala.js (www.scala-js.org) предлагает инструменты для преобразования кода на Scala в код на JavaScript. Воспользовавшись ими, можно написать серверную и клиентскую части веб-приложения на Scala.

1.7. Scaladoc

Для исследования Java API программисты на Java пользуются системой генерации документации Javadoc. В Scala есть аналогичный инструмент – Scaladoc (рис. 1.3).

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, предназначенные скорее для разработчиков библиотек, чем для прикладных 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 http://scala-lang.org/download/all.html и установить ее локально.

В отличие от Javadoc, где список классов приводится в алфавитном порядке, классы в Scaladoc отсортированы по именам пакетов. Если имя класса или метода известно, а имя пакета нет, воспользуйтесь строкой поиска вверху (рис. 1.4).

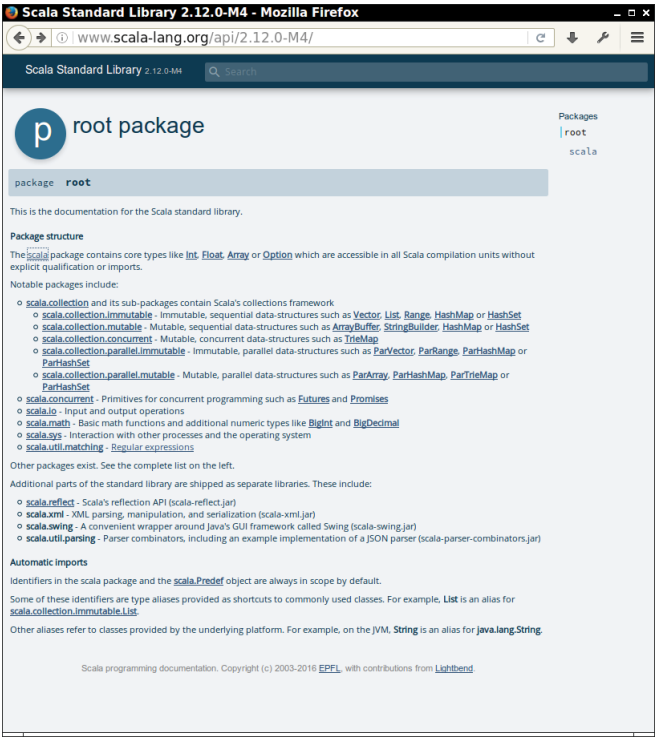

Рис. 1.3. Страница со статьей в окне Scaladoc

Щелкните на значке **X**, чтобы очистить фильтр, или на имени искомого класса или метода (рис. 1.5).

Обратите внимание на символы «O» и «C» рядом с именем каждого класса. Они позволяют исследовать класс (C) или объект-компаньон (O). Для трейтов (traits, которые напоминают интерфейсы Java) отображаются символы «t» и «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.

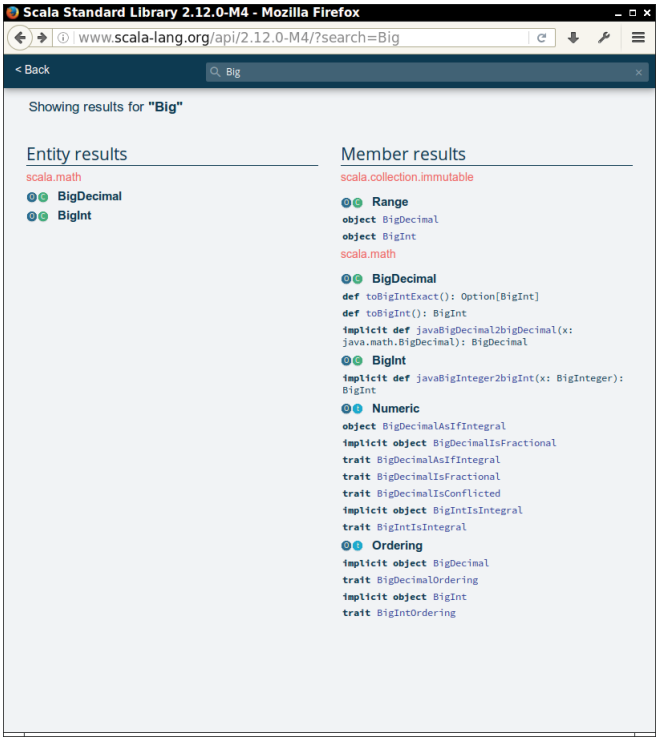

Рис. 1.4. Строка поиска в окне Scaladoc

- 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`.
- 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.