

Функциональное программирование

Кей Хорстман

SCALA

для нетерпеливых

*Второе издание,
переработанное и дополненное*

УДК 004.432.42Scala

ББК 32.973-018.1

X84

Хорстманн К.

X84 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

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-13-454056-6 (анг.) Copyright © 2017 Pearson Education Inc.

ISBN 978-5-97060-536-3 (рус.) © Оформление, перевод ДМК Пресс, 2017



Содержание

Предисловие к первому изданию	15
Вступление	17
Об авторе	19
 Глава 1. Основы	 20
1.1. Интерпретатор Scala.....	20
1.2. Объявление значений и переменных	23
1.3. Часто используемые типы.....	25
1.4. Арифметика и перегрузка операторов	26
1.5. Вызов функций и методов.....	28
1.6. Метод apply	29
1.7. Scaladoc	31
Упражнения.....	35
 Глава 2. Управляющие структуры и функции	 37
2.1. Условные выражения	38
2.2. Завершение инструкций	40
2.3. Блочные выражения и присваивание	41
2.4. Ввод и вывод	42
2.5. Циклы	44
2.6. Расширенные циклы for	45
2.7. Функции	47
2.8. Аргументы по умолчанию и именованные аргументы L1 ...	48
2.9. Переменное количество аргументов L1	49
2.10. Процедуры.....	50

2.11. Ленивые значения L1	51
2.12. Исключения	52
Упражнения	55

Глава 3. Работа с массивами

57

3.1. Массивы фиксированной длины	58
3.2. Массивы переменной длины: буферы	58
3.3. Обход массивов и буферов	59
3.4. Преобразование массивов	60
3.5. Типичные алгоритмы	62
3.6. Расшифровываем Scaladoc	64
3.7. Многомерные массивы	66
3.8. Взаимодействие с Java	66
Упражнения	68

Глава 4. Ассоциативные массивы и кортежи

70

4.1. Конструирование ассоциативных массивов	71
4.2. Доступ к значениям в ассоциативных массивах	72
4.3. Изменение значений в ассоциативных массивах	72
4.4. Обход элементов ассоциативных массивов	74
4.5. Сортированные ассоциативные массивы	74
4.6. Взаимодействие с Java	75
4.7. Кортежи	76
4.8. Функция zip	77
Упражнения	77

Глава 5. Ассоциативные массивы и кортежи

79

5.1. Простые классы и методы без параметров	80
5.2. Свойства с методами доступа	81
5.3. Свойства только с методами чтения	83
5.4. Приватные поля объектов	85
5.5. Свойства компонентов L1	86
5.6. Дополнительные конструкторы	87
5.7. Главный конструктор	88
5.8. Вложенные классы L1	91
Упражнения	93

Глава 6. Объекты	96
6.1. Объекты-одиночки	96
6.2. Объекты-компаньоны	97
6.3. Объекты, расширяющие классы или трейты	98
6.4. Метод apply	99
6.5. Объект, представляющий приложение	100
6.6. Перечисления	101
Упражнения	103
 Глава 7. Пакеты и импортирование	 104
7.1. Пакеты	105
7.2. Правила видимости	106
7.3. Объявления цепочек пакетов	108
7.4. Объявления в начале файла	108
7.5. Объекты пакетов	109
7.6. Видимость внутри пакетов	110
7.7. Импортирование	111
7.8. Импортирование возможно в любом месте	112
7.9. Переименование и сокрытие членов	112
7.10. Неявный импорт	113
Упражнения	113
 Глава 8. Наследование	 115
8.1. Наследование классов	115
8.2. Переопределение методов	116
8.3. Проверка и приведение типов	117
8.4. Защищенные поля и методы	118
8.5. Создание суперклассов	118
8.6. Переопределение полей	120
8.7. Анонимные подклассы	121
8.8. Абстрактные классы	122
8.9. Абстрактные поля	122
8.10. Порядок создания и опережающие определения L3	123
8.11. Иерархия наследования в Scala	125
8.12. Равенство объектов L1	128
8.13. Классы-значения L2	129
Упражнения	131

Глава 9. Файлы и регулярные выражения	133
9.1. Чтение строк	134
9.2. Чтение символов	134
9.3. Чтение лексем и чисел	135
9.4. Чтение из URL и других источников	136
9.5. Чтение двоичных файлов	136
9.6. Запись в текстовые файлы	136
9.7. Обход каталогов	137
9.8. Сериализация	137
9.9. Управление процессами A2	138
9.10. Регулярные выражения	141
9.11. Группы в регулярных выражениях	142
Упражнения	143
 Глава 10. Трейты	 145
10.1. Почему не поддерживается множественное наследование?	146
10.2. Трейты как интерфейсы	148
10.3. Трейты с конкретными реализациями	149
10.4. Объекты с трейтами	150
10.5. Многоуровневые трейты	150
10.6. Переопределение абстрактных методов в трейтах	152
10.7. Трейты с богатыми интерфейсами	153
10.8. Конкретные поля в трейтах	154
10.9. Абстрактные поля в трейтах	155
10.10. Порядок конструирования трейтов	156
10.11. Инициализация полей трейтов	158
10.12. Трейты, наследующие классы	160
10.13. Собственные типы L2	161
10.14. За кулисами	162
Упражнения	164
 Глава 11. Операторы	 167
11.1. Идентификаторы	168
11.2. Инфиксные операторы	169
11.3. Унарные операторы	170



11.4. Операторы присваивания	171
11.5. Приоритет	171
11.6. Ассоциативность	172
11.7. Методы apply и update	173
11.8. Экстракторы L2	174
11.9. Экстракторы с одним аргументом или без аргументов L2	177
11.10. Метод unapplySeq L2	177
11.11. Динамический вызов L2	178
Упражнения	182

Глава 12. Функции высшего порядка

12.1. Функции как значения	186
12.2. Анонимные функции	187
12.3. Функции с функциональными параметрами	188
12.4. Вывод типов	189
12.5. Полезные функции высшего порядка	190
12.6. Замыкания	192
12.7. Преобразование функций в SAM	193
12.8. Карринг	194
12.9. Абстракция управляющих конструкций	196
12.10. Выражение return	198
Упражнения	199

Глава 13. Коллекции

13.1. Основные трейты коллекций	202
13.2. Изменяемые и неизменяемые коллекции	204
13.3. Последовательности	205
13.4. Списки	207
13.5. Множества	208
13.7. Операторы добавления и удаления элементов	210
13.7. Общие методы	212
13.8. Функции map и flatMap	214
13.9. Функции reduce, fold и scan A3	216
13.10. Функция zip	220
13.11. Итераторы	222

13.12. Потоки A3	223
13.13. Ленивые представления A3	225
13.14. Взаимодействие с коллекциями Java	226
13.15. Параллельные коллекции	227
Упражнения	229

Глава 14. Сопоставление с образцом

и case-классы	232
14.1. Лучше, чем switch	233
14.2. Ограничители	234
14.3. Переменные в образцах	235
14.4. Сопоставление с типами	236
14.5. Сопоставление с массивами, списками и кортежами	237
14.6. Экстракторы	238
14.7. Образцы в объявлениях переменных	239
14.8. Образцы в выражениях for	240
14.9. Case-классы	241
14.10. Метод сору и именованные параметры	242
14.11. Инфиксная нотация в предложениях case	243
14.12. Сопоставление с вложенными структурами	244
14.13. Так ли необходимы case-классы?	245
14.14. Запечатанные классы	247
14.15. Имитация перечислений	247
14.16. Тип Option	248
14.17. Частично определенные функции L2	249
Упражнения	251

Глава 15. Аннотации

15.1. Что такое аннотации?	255
15.2. Что можно аннотировать?	256
15.3. Аргументы аннотаций	257
15.4. Реализация аннотаций	258
15.5. Аннотации для элементов Java	259
15.6. Аннотации для оптимизации	262
15.7. Аннотации ошибок и предупреждений	267
Упражнения	269

Глава 16. Обработка XML	271
16.1. Литералы XML	272
16.2. Узлы XML	273
16.3. Атрибуты элементов	274
16.4. Встроенные выражения	276
16.5. Выражения в атрибутах	277
16.6. Необычные типы узлов	278
16.7. XPath-подобные выражения	279
16.8. Сопоставление с образцом	281
16.9. Модификация элементов и атрибутов	282
16.10. Трансформация XML	283
16.11. Загрузка и сохранение	284
16.12. Пространства имен	287
Упражнения	288
 Глава 17. Объекты Future	290
17.1. Запуск асинхронных заданий в объектах Future	291
17.2. Ожидание результатов	294
17.3. Класс Try	295
17.4. Обратные вызовы	296
17.5. Комбинирование заданий в объектах Future	297
17.6. Другие преобразования объектов Future	300
17.7. Методы объекта Future	302
17.8. Объекты Promise	304
17.9. Контексты выполнения	306
Упражнения	307
 Глава 18. Параметризованные типы	310
18.1. Обобщенные классы	311
18.2. Обобщенные функции	312
18.3. Границы изменения типов	312
18.4. Границы представления	314
18.5. Границы контекста	314
18.6. Границы контекста ClassTag	315
18.7. Множественные границы	316
18.8. Ограничение типов L3	316

18.9. Вариантность.....	318
18.10. Ко- и контравариантные позиции	320
18.11. Объекты не могут быть обобщенными	322
18.12. Подстановочный символ	323
Упражнения.....	324

Глава 19. Дополнительные типы

19.1. Типы-одиночки.....	327
19.2. Проекция типов	329
19.3. Цепочки	330
19.4. Псевдонимы типов.....	331
19.5. Структурные типы	332
19.6. Составные типы	332
19.7. Инфиксные типы	334
19.8. Экзистенциальные типы	334
19.9. Система типов языка Scala	336
19.10. Собственные типы	337
19.11. Внедрение зависимостей	338
19.12. Абстрактные типы L3	341
19.13. Родовой полиморфизм L3	343
19.14. Типы высшего порядка L3	346
Упражнения.....	350

Глава 20. Парсинг

20.1. Грамматики.....	354
20.2. Комбинирование операций парсера	356
20.3. Преобразование результатов парсинга.....	358
20.4. Отбрасывание лексем.....	360
20.5. Создание деревьев синтаксического анализа	361
20.6. Уход от левой рекурсии.....	361
20.7. Дополнительные комбинаторы	363
20.8. Уход от возвратов	366
20.9. Raskrat-парсеры	367
20.10. Что такое парсеры?.....	368
20.11. Парсеры на основе регулярных выражений.....	369
20.12. Парсеры на основе лексем.....	370

20.13. Обработка ошибок	372
Упражнения	374

Глава 21. Неявные параметры и преобразования376

21.1. Неявные преобразования	377
21.2. Использование неявных преобразований для расширения существующих библиотек	378
21.3. Импорт неявных преобразований	379
21.4. Правила неявных преобразований	381
21.5. Неявные параметры	382
21.6. Неявные преобразования с неявными параметрами	383
21.7. Границы контекста	385
21.8. Классы типов	386
21.9. Неявный параметр подтверждения	389
21.10. Аннотация @implicitNotFound	390
21.11. Тайна CanBuildFrom	391
Упражнения	393

Предметный указатель.....395



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**.

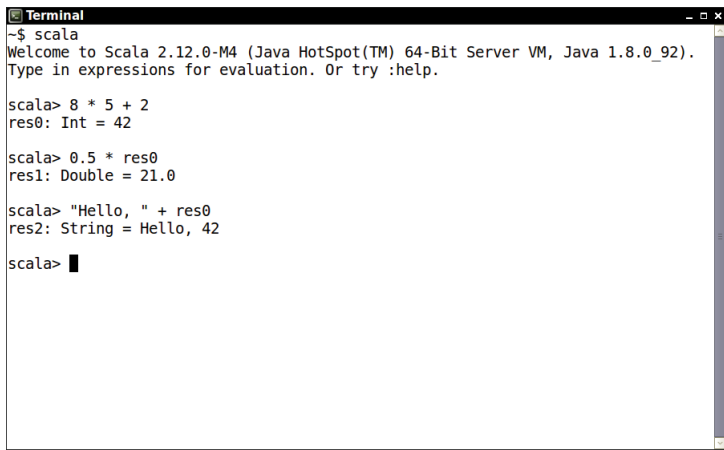
Теперь попробуйте ввести следующие команды, завершая каждую нажатием клавиши **Enter**. Каждый раз интерпретатор будет выводить ответ, как показано на рис. 1.1. Например, если ввести `8 * 5 + 2` (как показано ниже), интерпретатор выведет ответ 42.

```
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`.



```
Terminal
~$ scala
Welcome to Scala 2.12.0-M4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_92).
Type in expressions for evaluation. Or try :help.

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

scala> 0.5 * res0
res1: Double = 21.0

scala> "Hello, " + res0
res2: String = Hello, 42

scala> █
```

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.

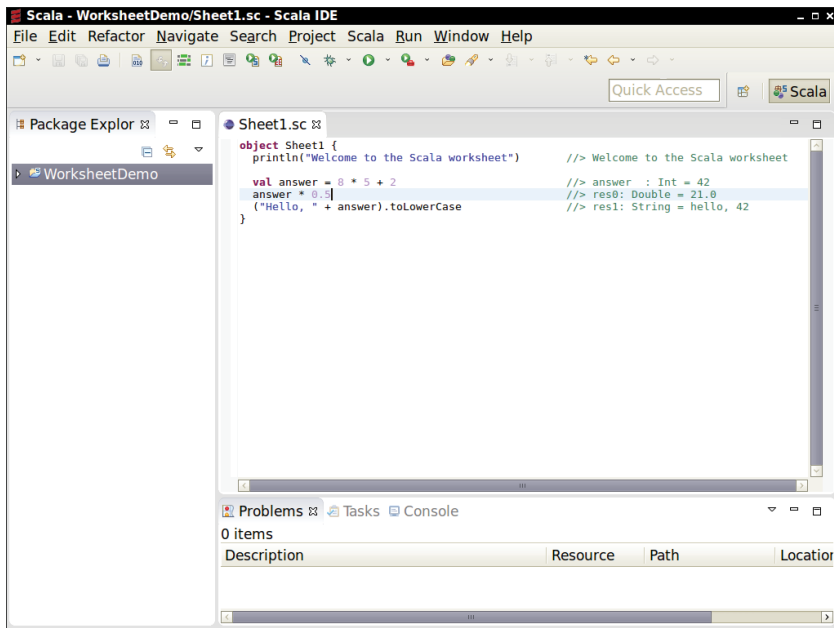


Рис. 1.2. Интерактивная оболочка Scala

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 \uparrow и \downarrow . 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 прочесть выражение, вычислить его, вывести результат и прочесть 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 получают 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 `qxuwz`. 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 $\{0, 1, \dots, n - 1\}$, 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 неизвестные вам возможности, предназначенные скорее для разработчиков библиотек, чем для прикладных программистов.

Ниже приводятся 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 сверху (рис. 1.4).

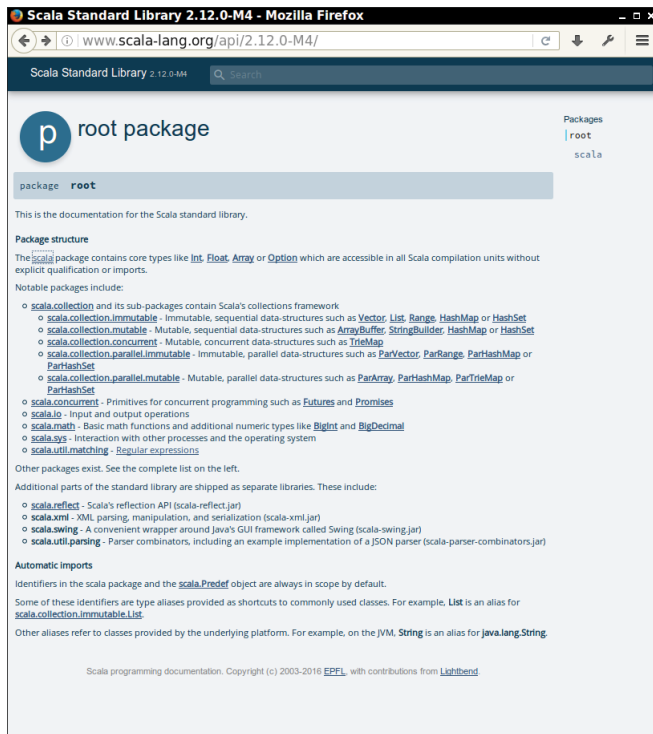


Рис. 1.3. Страница со статьей в окне Scaladoc

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.

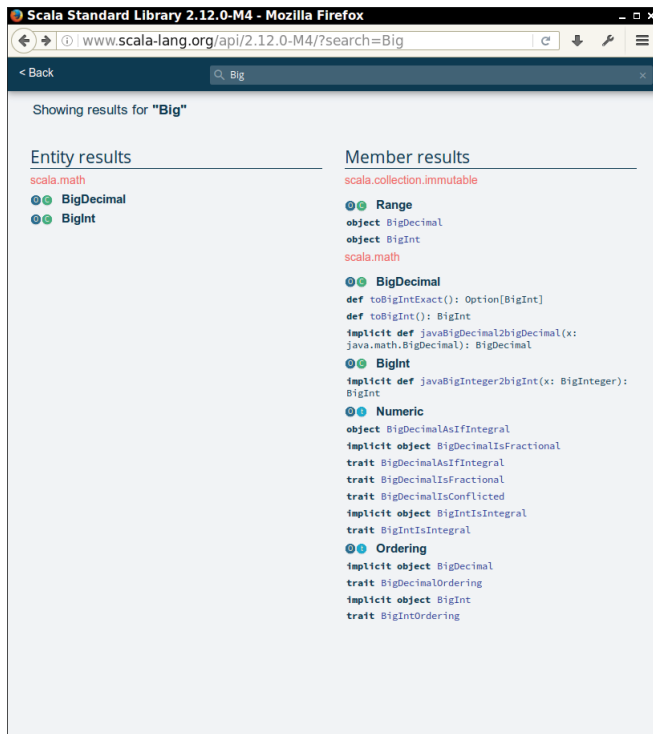


Рис. 1.4. Строка поиска в окне Scaladoc

- ❑ 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.