SCALA for the Impatient

Cay S. Horstmann

Foreword by Martin Odersky

Operators

- Infix notation `x` *op* `y` is `x.`*op*`(y)`, postfix notation `x` *op* is `x.`*op*`()`
- Only `+ - ! ~` can be prefix—define method `unary_`*op*
- Assignment `x` *op*`= y` is `x = x` *op* `y` unless defined separately
- Precedence depends on *first* character, except for assignments

Highest: Other operator char	* / %	+ -	:	< >	! =	&	^	|	Not operator char	Lowest: Assignments

- Right associative if *last* character is a colon :
- `x(i) = x(j)` is `x.update(i, x.apply(j))`
- There is no `++` or `--` for numbers. Use `x += 1; y -= 1`
- Use `x == y` to compare objects—it calls `equals`

Functions

```
def triple(x: Int) = 3 * x // Parameter name: Type
val f = (x: Int) => 3 * x // Anonymous function
(1 to 10).map(3 * _) // Function with anonymous parameter
def greet(x: Int) { // Without =, return type is Unit
  println("Hello, " + x) }
def greet(x: Int, salutation: String = "Hello") { // Default argument
  println(salutation + ", " + x) }
// Call as greet(42), greet(42, "Hi"), greet(salutation = "Hi", x = 42)
def sum(xs: Int*) = { // * denotes varargs
  var r = 0; for (x <- xs) r += x // Semicolon separates statements on same line
  r // No return. Last expression is value of block
}
def sum(xs: Int*): Int = // Return type required for recursive functions
  if (xs.length == 0) 0 else xs.head + sum(xs.tail : _*) // Sequence as varargs
```

`for` Loops

```
for (i <- 1 to n) println(i) // i iterates through all values in 1 to n
for (i <- 1 to 9; j <- 1 to 9) println(i * 10 + j) // Multiple iterates
for (i <- 1 to 9 if i != 5; j <- 1 to 9 if i != j) println(i * 10 + j) // Guards
for (i <- 1 to 3; from = 4 - i; j <- from to 3) println(i * 10 + j) // Variable
val r = for (i <- 1 to n) yield i * i // r is a sequence 1, 4, 9, ...
for ((x, y) <- pairs) println(x + " " + y) // Destructures pairs and other values with extractors
```

Pattern Matching

```
val x = r match {
  case '0' => ... // Match value
  case ch if someProperty(ch) => ... // Guard
  case e: Employee => ... // Match runtime type
  case (x, y) => ... // Destructures pairs and other values with extractors
  case Some(v) => ... // Case classes have extractors
  case 0 :: tail => ... // Infix notation for extractors yielding a pair
  case _ => ... // Default case
}

try { ... } catch { // Use the same syntax for catch clauses
 case _: MalformedURLException => println("Bad URL")
 case ex: IOException => ex.printStackTrace()
}
```

Scala for the Impatient

This page intentionally left blank

Scala for the Impatient

Cay S. Horstmann

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382–3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Horstmann, Cay S., 1959-
Scala for the impatient / Cay S. Horstmann.
p. cm.
Includes index.
ISBN 978-0-321-77409-5 (pbk. : alk. paper)—ISBN 0-321-77409-4 (pbk. : alk. paper) 1. Scala (Computer program language) 2. Programming languages (Electronic computers) 3. Computer programming. I. Title.
QA76.73.S28H67 2012
005.13’3—dc23

2011052136

ISBN-13: 978-0-321-77409-5
ISBN-10: 0-321-77409-4
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Third printing, February 2014

To my wife, who made writing this book possible,
and to my children, who made it necessary.

This page intentionally left blank

Contents

Foreword

When I met Cay Horstmann some years ago he told me that Scala needed a better introductory book. My own book had come out a little bit earlier, so of course I had to ask him what he thought was wrong with it. He responded that it was great but too long; his students would not have the patience to read through the eight hundred pages of *Programming in Scala*. I conceded that he had a point. And he set out to correct the situation by writing *Scala for the Impatient*.

I am very happy that his book has finally arrived because it really delivers on what the title says. It gives an eminently practical introduction to Scala, explains what's particular about it, how it differs from Java, how to overcome some common hurdles to learning it, and how to write good Scala code.

Scala is a highly expressive and flexible language. It lets library writers use highly sophisticated abstractions, so that library users can express themselves simply and intuitively. Therefore, depending on what kind of code you look at, it might seem very simple or very complex.

A year ago, I tried to provide some clarification by defining a set of levels for Scala and its standard library. There were three levels each for application programmers and for library designers. The junior levels could be learned quickly and would be sufficient to program productively. Intermediate levels would make programs more concise and more functional and would make libraries more flexible to use. The highest levels were for experts solving specialized tasks. At the time I wrote:

> I hope this will help newcomers to the language decide in what order to pick subjects to learn, and that it will give some advice to teachers and book authors in what order to present the material.

Cay's book is the first to have systematically applied this idea. Every chapter is tagged with a level that tells you how easy or hard it is and whether it's oriented towards library writers or application programmers.

As you would expect, the first chapters give a fast-paced introduction to the basic Scala capabilities. But the book does not stop there. It also covers many of the more "senior" concepts and finally progresses to very advanced material which is not commonly covered in a language introduction, such as how to write parser combinators or make use of delimited continuations. The level tags serve as a guideline for what to pick up when. And Cay manages admirably to make even the most advanced concepts simple to understand.

I liked the concept of *Scala for the Impatient* so much that I asked Cay and his editor, Greg Doench, whether we could get the first part of the book as a free download on the Typesafe web site. They have gracefully agreed to my request, and I would like to thank them for that. That way, everybody can quickly access what I believe is currently the best compact introduction to Scala.

Martin Odersky

January 2012

Preface

The evolution of Java and C++ has slowed down considerably, and programmers who are eager to use more modern language features are looking elsewhere. Scala is an attractive choice; in fact, I think it is by far the most attractive choice for programmers who want to move beyond Java or C++. Scala has a concise syntax that is refreshing after the Java boilerplate. It runs on the Java virtual machine, providing access to a huge set of libraries and tools. It embraces the functional programming style without abandoning object orientation, giving you an incremental learning path to a new paradigm. The Scala interpreter lets you run quick experiments, which makes learning Scala very enjoyable. Last but not least, Scala is statically typed, enabling the compiler to find errors, so that you don't waste time finding them—or not—later in the running program.

I wrote this book for *impatient* readers who want to start programming in Scala right away. I assume you know Java, C#, or C++, and I don't bore you with explaining variables, loops, or classes. I don't exhaustively list all the features of the language, I don't lecture you about the superiority of one paradigm over another, and I don't make you suffer through long and contrived examples. Instead, you will get the information that you need in compact chunks that you can read and review as needed.

Scala is a big language, but you can use it effectively without knowing all of its details intimately. Martin Odersky, the creator of Scala, has identified levels of expertise for application programmers and library designers—as shown in the following table.

Application Programmer	Library Designer	Overall Scala Level
Beginning A1		Beginning
Intermediate A2	Junior L1	Intermediate
Expert A3	Senior L2	Advanced
	Expert L3	Expert

For each chapter (and occasionally for individual sections), I indicate the experience level required. The chapters progress through levels A1, L1, A2, L2, A3, L3. Even if you don't want to design your own libraries, knowing about the tools that Scala provides for library designers can make you a more effective library user.

I hope you enjoy learning Scala with this book. If you find errors or have suggestions for improvement, please visit http://horstmann.com/scala and leave a comment. On that page, you will also find a link to an archive file containing all code examples from the book.

I am very grateful to Dmitry Kirsanov and Alina Kirsanova who turned my manuscript from XHTML into a beautiful book, allowing me to concentrate on the content instead of fussing with the format. Every author should have it so good!

Reviewers include Adrian Cumiskey, Mike Davis, Rob Dickens, Daniel Sobral, Craig Tataryn, David Walend, and William Wheeler. Thanks so much for your comments and suggestions!

Finally, as always, my gratitude goes to my editor, Greg Doench, for encouraging me to write this book, and for his insights during the development process.

Cay Horstmann

San Francisco, 2012

About the Author

Cay S. Horstmann is principal author of *Core Java™, Volumes I & II, Eighth Edition* (Sun Microsystems Press, 2008), as well as a dozen other books for professional programmers and computer science students. He is a professor of computer science at San Jose State University and a Java Champion.

The Basics

Topics in This Chapter A1

Chapter 1

In this chapter, you will learn how to use Scala as an industrial-strength pocket calculator, working interactively with numbers and arithmetic operations. We introduce a number of important Scala concepts and idioms along the way. You will also learn how to browse the Scaladoc documentation at a beginner's level.

Highlights of this introduction are:

- Using the Scala interpreter
- Defining variables with `var` and `val`
- Numeric types
- Using operators and functions
- Navigating Scaladoc

1.1 The Scala Interpreter

To start the Scala interpreter:

- Install Scala.
- Make sure that the `scala/bin` directory is on the PATH.
- Open a command shell in your operating system.
- Type `scala` followed by the Enter key.

TIP: Don't like the command shell? There are other ways of running the interpreter—see http://horstmann.com/scala/install.

Now type commands followed by Enter. Each time, the interpreter displays the answer. For example, if you type `8 * 5 + 2` (as shown in boldface below), you get `42`.

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

The answer is given the name `res0`. You can use that name in subsequent computations:

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

As you can see, the interpreter also displays the type of the result—in our examples, `Int`, `Double`, and `java.lang.String`.

You can call methods. Depending on how you launched the interpreter, you may be able to use *tab completion* for method names. Try typing `res2.to` and then hit the Tab key. If the interpreter offers choices such as

```
toCharArray   toLowerCase   toString     toUpperCase
```

this means tab completion works. Type a `U` and hit the Tab key again. You now get a single completion:

```
res2.toUpperCase
```

Hit the Enter key, and the answer is displayed. (If you can't use tab completion in your environment, you'll have to type the complete method name yourself.)

Also try hitting the ↑ and ↓ arrow keys. In most implementations, you will see the previously issued commands, and you can edit them. Use the ←, →, and Del keys to change the last command to

```
res2.toLowerCase
```

As you can see, the Scala interpreter reads an expression, evaluates it, prints it, and reads the next expression. This is called the *read-eval-print loop*, or REPL.

Technically speaking, the `scala` program is *not* an interpreter. Behind the scenes, your input is quickly compiled into bytecode, and the bytecode is executed by the Java virtual machine. For that reason, most Scala programmers prefer to call it "the REPL".

> TIP: The REPL is your friend. Instant feedback encourages experimenting, and you will feel good whenever something works.
>
> It is a good idea to keep an editor window open at the same time, so you can copy and paste successful code snippets for later use. Also, as you try more complex examples, you may want to compose them in the editor and then paste them into the REPL.

1.2 Declaring Values and Variables

Instead of using the names `res0`, `res1`, and so on, you can define your own names:

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

You can use these names in subsequent expressions:

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

A value declared with `val` is actually a constant—you can't change its contents:

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

To declare a variable whose contents can vary, use a `var`:

```
var counter = 0
counter = 1 // OK, can change a var
```

In Scala, you are encouraged to use a `val` unless you really need to change the contents. Perhaps surprisingly for Java or C++ programmers, most programs don't need many `var` variables.

Note that you need not specify the type of a value or variable. It is inferred from the type of the expression with which you initialize it. (It is an error to declare a value or variable without initializing it.)

However, you can specify the type if necessary. For example,

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

NOTE: In Scala, the type of a variable or function is always written *after* the name of the variable or function. This makes it easier to read declarations with complex types.

As I move back and forth between Scala and Java, I find that my fingers write Java declarations such as `String greeting` on autopilot, so I have to rewrite them as `greeting: String`. This is a bit annoying, but when I work with complex Scala programs, I really appreciate that I don't have to decrypt C-style type declarations.

NOTE: You may have noticed that there were no semicolons after variable declarations or assignments. In Scala, semicolons are only required if you have multiple statements on the same line.

You can declare multiple values or variables together:

```
val xmax, ymax = 100 // Sets xmax and ymax to 100
var greeting, message: String = null
  // greeting and message are both strings, initialized with null
```

1.3 Commonly Used Types

You have already seen some of the data types of the Scala language, such as `Int` and `Double`. Like Java, Scala has seven numeric types: `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, and `Double`, and a `Boolean` type. However, unlike Java, these types are *classes*. There is no distinction between primitive types and class types in Scala. You can invoke methods on numbers, for example:

```
1.toString() // Yields the string "1"
```

or, more excitingly,

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

(We will discuss the `Range` class in Chapter 13. For now, just view it as a collection of numbers.)

In Scala, there is no need for wrapper types. It is the job of the Scala compiler to convert between primitive types and wrappers. For example, if you make an array of `Int`, you get an `int[]` array in the virtual machine.

As you saw in Section 1.1, "The Scala Interpreter," on page 1, Scala relies on the underlying `java.lang.String` class for strings. However, it augments that class with well over a hundred operations in the `StringOps` class.

For example, the `intersect` method yields the characters that are common to two strings:

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

In this expression, the `java.lang.String` object `"Hello"` is implicitly converted to a `StringOps` object, and then the `intersect` method of the `StringOps` class is applied.

Therefore, remember to look into the `StringOps` class when you use the Scala documentation (see Section 1.7, "Scaladoc," on page 8).

Similarly, there are classes `RichInt`, `RichDouble`, `RichChar`, and so on. Each of them has a small set of convenience methods for acting on their poor cousins—`Int`, `Double`, or `Char`. The `to` method that you saw above is actually a method of the `RichInt` class. In the expression

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

the `Int` value `1` is first converted to a `RichInt`, and the `to` method is applied to that value.

Finally, there are classes `BigInt` and `BigDecimal` for computations with an arbitrary (but finite) number of digits. These are backed by the `java.math.BigInteger` and `java.math.BigDecimal` classes, but, as you will see in the next section, they are much more convenient because you can use them with the usual mathematical operators.

NOTE: In Scala, you use methods, not casts, to convert between numeric types. For example, `99.44.toInt` is `99`, and `99.toChar` is `'c'`. Of course, as in Java, the `toString` method converts any object to a string.

To convert a string containing a number into the number, use `toInt` or `toDouble`. For example, `"99.44".toDouble` is `99.44`.

1.4 Arithmetic and Operator Overloading

Arithmetic operators in Scala work just as you would expect in Java or C++:

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

The `+ - * / %` operators do their usual job, as do the bit operators `& | ^ >> <<`. There is just one surprising aspect: These operators are actually methods. For example,

```
a + b
```

is a shorthand for

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

Here, `+` is the name of the method. Scala has no silly prejudice against non-alphanumeric characters in method names. You can define methods with just about any symbols for names. For example, the `BigInt` class defines a method called `/%` that returns a pair containing the quotient and remainder of a division.

In general, you can write

```
a method b
```

as a shorthand for

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

where *method* is a method with two parameters (one implicit, one explicit). For example, instead of

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

you can write

```
1 to 10
```

Use whatever you think is easier to read. Beginning Scala programmers tend to stick to the Java syntax, and that is just fine. Of course, even the most hardened Java programmers seem to prefer `a + b` over `a.+(b)`.

There is one notable difference between Scala and Java or C++. Scala does not have `++` or `--` operators. Instead, simply use `+=1` or `-=1`:

```
counter+=1 // Increments counter—Scala has no ++
```

Some people wonder if there is any deep reason for Scala's refusal to provide a `++` operator. (Note that you can't simply implement a method called `++`. Since the `Int` class is immutable, such a method cannot change an integer value.) The Scala designers decided it wasn't worth having yet another special rule just to save one keystroke.

You can use the usual mathematical operators with `BigInt` and `BigDecimal` objects:

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

That's much better than Java, where you would have had to call `x.multiply(x).multiply(x)`.

NOTE: In Java, you cannot overload operators, and the Java designers claimed this is a good thing because it stops you from inventing crazy operators like `!@$&*` that would make your program impossible to read. Of course, that's silly; you can make your programs just as hard to read by using crazy method names like `qxywz`. Scala allows you to define operators, leaving it up to you to use this feature with restraint and good taste.

1.5 Calling Functions and Methods

Scala has functions in addition to methods. It is simpler to use mathematical functions such as `min` or `pow` in Scala than in Java—you need not call static methods from a class.

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

The mathematical functions are defined in the `scala.math` package. You can import them with the statement

```
import scala.math._ // In Scala, the _ character is a "wildcard," like * in Java
```

> NOTE: To use a package that starts with `scala.`, you can omit the `scala` prefix. For example, `import math._` is equivalent to `import scala.math._`, and `math.sqrt(2)` is the same as `scala.math.sqrt(2)`.

We discuss the `import` statement in more detail in Chapter 7. For now, just use `import` *packageName*`._` whenever you need to import a particular package.

Scala doesn't have static methods, but it has a similar feature, called *singleton objects*, which we will discuss in detail in Chapter 6. Often, a class has a *companion object* whose methods act just like static methods do in Java. For example, the `BigInt` companion object to the `BigInt` class has a method `probablePrime` that generates a random prime number with a given number of bits:

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

Try this in the REPL; you'll get a number such as `1039447980491200275486540240713`. Note that the call `BigInt.probablePrime` is similar to a static method call in Java.

> NOTE: Here, `Random` is a singleton random number generator object, defined in the `scala.util` package. This is one of the few situations where a singleton object is better than a class. In Java, it is a common error to construct a new `java.util.Random` object for each random number.

Scala methods without parameters often don't use parentheses. For example, the API of the `StringOps` class shows a method `distinct`, without `()`, to get the distinct letters in a string. You call it as

```
"Hello".distinct
```

The rule of thumb is that a parameterless method that doesn't modify the object has no parentheses. We discuss this further in Chapter 5.

1.6 The apply Method

In Scala, it is common to use a syntax that looks like a function call. For example, if `s` is a string, then `s(i)` is the `i`th character of the string. (In C++, you would write `s[i]`; in Java, `s.charAt(i)`.) Try it out in the REPL:

```
"Hello"(4) // Yields 'o'
```

You can think of this as an overloaded form of the `()` operator. It is implemented as a method with the name `apply`. For example, in the documentation of the `StringOps` class, you will find a method

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

That is, `"Hello"(4)` is a shortcut for

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

When you look at the documentation for the `BigInt` companion object, you will see `apply` methods that let you convert strings or numbers to `BigInt` objects. For example, the call

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

is a shortcut for

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

It yields a new `BigInt` object, *without having to use* `new`. For example:

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

Using the `apply` method of a companion object is a common Scala idiom for constructing objects. For example, `Array(1, 4, 9, 16)` returns an array, thanks to the `apply` method of the `Array` companion object.

1.7 Scaladoc

Java programmers use Javadoc to navigate the Java API. Scala has its own variant, called Scaladoc (see Figure 1–1).

Navigating Scaladoc is a bit more challenging than Javadoc. Scala classes tend to have many more convenience methods than Java classes. Some methods use features that you haven't learned yet. Finally, some features are exposed as they are implemented, not as they are used. (The Scala team is working on improving the Scaladoc presentation, so that it can be more approachable to beginners in the future.)

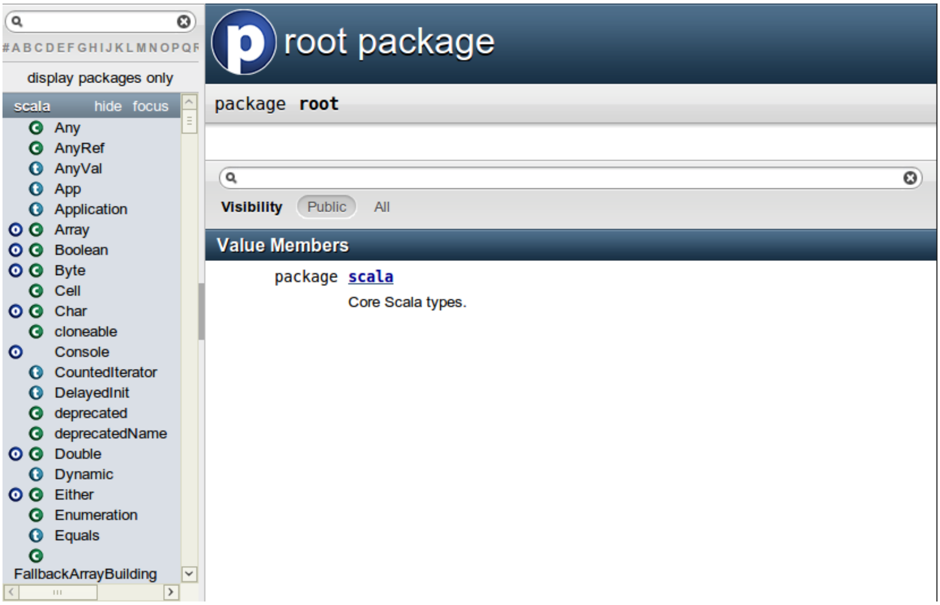

Figure 1–1 The entry page for Scaladoc

Here are some tips for navigating Scaladoc, for a newcomer to the language.

You can browse Scaladoc online at www.scala-lang.org/api, but it is a good idea to download a copy from www.scala-lang.org/downloads#api and install it locally.

Unlike Javadoc, which presents an alphabetical listing of classes, Scaladoc's class list is sorted by packages. If you know the class name but not the package name, use the filter in the top left corner (see Figure 1–2).

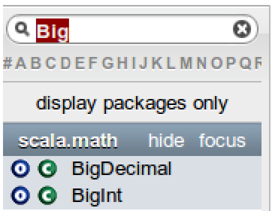

Figure 1–2 The filter box in Scaladoc

Click on the X symbol to clear the filter.

Note the O and C symbols next to each class name. They let you navigate to the class (C) or the companion object (O).

Scaladoc can be a bit overwhelming. Keep these tips in mind.

- Remember to look into `RichInt`, `RichDouble`, and so on, if you want to know how to work with numeric types. Similarly, to work with strings, look into `StringOps`.
- The mathematical functions are in the *package* `scala.math`, not in any class.
- Sometimes, you'll see functions with funny names. For example, `BigInt` has a method `unary_-`. As you will see in Chapter 11, this is how you define the prefix negation operator `-x`.
- A method tagged as `implicit` is an automatic conversion. For example, the `BigInt` object has conversions from `int` and `long` to `BigInt` that are automatically called when needed. See Chapter 21 for more information about implicit conversions.
- Methods can have functions as parameters. For example, the `count` method in `StringOps` requires a function that returns `true` or `false` for a `Char`, specifying which characters should be counted:

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

  You supply a function, often in a very compact notation, when you call the method. As an example, the call `s.count(_.isUpper)` counts the number of uppercase characters. We will discuss this style of programming in much more detail in Chapter 12.
- You'll occasionally run into classes such as `Range` or `Seq[Char]`. They mean what your intuition tells you—a range of numbers, a sequence of characters. You will learn all about these classes as you delve more deeply into Scala.
- Don't get discouraged that there are so many methods. It's the Scala way to provide lots of methods for every conceivable use case. When you need to solve a particular problem, just look for a method that is useful. More often than not, there is one that addresses your task, which means you don't have to write so much code yourself.
- Finally, don't worry if you run into the occasional indecipherable incantation, such as this one in the `StringOps` class:

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

  Just ignore it. There is another version of `patch` that looks more reasonable:

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

If you think of GenSeq[Char] and StringOps[A] as String, the method is pretty easy to understand from the documentation. And it's easy to try it out in the REPL:

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

Exercises

1. In the Scala REPL, type 3. followed by the Tab key. What methods can be applied?
2. In the Scala REPL, compute the square root of 3, and then square that value. By how much does the result differ from 3? (Hint: The res variables are your friend.)
3. Are the res variables val or var?
4. Scala lets you multiply a string with a number—try out "crazy" * 3 in the REPL. What does this operation do? Where can you find it in Scaladoc?
5. What does 10 max 2 mean? In which class is the max method defined?
6. Using BigInt, compute 2^{1024}.
7. What do you need to import so that you can get a random prime as probablePrime(100, Random), without any qualifiers before probablePrime and Random?
8. One way to create random file or directory names is to produce a random BigInt and convert it to base 36, yielding a string such as "qsnvbevtomcj38o06kul". Poke around Scaladoc to find a way of doing this in Scala.
9. How do you get the first character of a string in Scala? The last character?
10. What do the take, drop, takeRight, and dropRight string functions do? What advantage or disadvantage do they have over using substring?

Control Structures and Functions

Topics in This Chapter A1

Chapter 2

In this chapter, you will learn how to implement conditions, loops, and functions in Scala. You will encounter a fundamental difference between Scala and other programming languages. In Java or C++, we differentiate between *expressions* (such as `3 + 4`) and *statements* (for example, an `if` statement). An expression has a value; a statement carries out an action. In Scala, almost all constructs have values. This feature can make programs more concise and easier to read.

Here are the highlights of this chapter:

- An `if` expression has a value.
- A block has a value—the value of its last expression.
- The Scala `for` loop is like an "enhanced" Java `for` loop.
- Semicolons are (mostly) optional.
- The `void` type is `Unit`.
- Avoid using `return` in a function.
- Beware of missing `=` in a function definition.
- Exceptions work just like in Java or C++, but you use a "pattern matching" syntax for `catch`.
- Scala has no checked exceptions.

2.1 Conditional Expressions

Scala has an `if/else` construct with the same syntax as in Java or C++. However, in Scala, an `if/else` has a value, namely the value of the expression that follows the `if` or `else`. For example,

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

has a value of `1` or `-1`, depending on the value of `x`. You can put that value in a variable:

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

This has the same effect as

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

However, the first form is better because it can be used to initialize a `val`. In the second form, `s` needs to be a `var`.

(As already mentioned, semicolons are mostly optional in Scala—see Section 2.2, "Statement Termination," on page 15.)

Java and C++ have a `?:` operator for this purpose. The expression

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

is equivalent to the Scala expression `if (x > 0) 1 else -1`. However, you can't put statements inside a `?:` expression. The Scala `if/else` combines the `if/else` and `?:` constructs that are separate in Java and C++.

In Scala, every expression has a type. For example, the expression `if (x > 0) 1 else -1` has the type `Int` because both branches have the type `Int`. The type of a mixed-type expression, such as

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

is the common supertype of both branches. In this example, one branch is a `java.lang.String`, and the other an `Int`. Their common supertype is called `Any`. (See Section 8.11, "The Scala Inheritance Hierarchy," on page 94 for details.)

If the `else` part is omitted, for example in

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

then it is possible that the `if` statement yields no value. However, in Scala, every expression is supposed to have *some* value. This is finessed by introducing a class `Unit` that has one value, written as `()`. The `if` statement without an `else` is equivalent to

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

Think of `()` as a placeholder for "no useful value," and think of `Unit` as the analog of `void` in Java or C++.

(Technically speaking, `void` has no value whereas `Unit` has one value that signifies "no value". If you are so inclined, you can ponder the difference between an empty wallet and a wallet with a bill labeled "no dollars".)

NOTE: Scala has no `switch` statement, but it has a much more powerful pattern matching mechanism that we will discuss in Chapter 14. For now, just use a sequence of `if` statements.

CAUTION: The REPL is more nearsighted than the compiler—it only sees one line of code at a time. For example, when you type

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

the REPL executes `if (x > 0) 1` and shows the answer. Then it gets confused about the `else` keyword.

If you want to break the line before the `else`, use braces:

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

This is only a concern in the REPL. In a compiled program, the parser will find the `else` on the next line.

TIP: If you want to paste a block of code into the REPL without worrying about its nearsightedness, use *paste mode*. Type

```
:paste
```

Then paste in the code block and type Ctrl+K. The REPL will then analyze the block in its entirety.

2.2 Statement Termination

In Java and C++, every statement ends with a semicolon. In Scala—like in JavaScript and other scripting languages—a semicolon is never required if it falls just before the end of the line. A semicolon is also optional before an `}`, an `else`, and similar locations where it is clear from context that the end of a statement has been reached.

However, if you want to have more than one statement on a single line, you need to separate them with semicolons. For example,

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

A semicolon is needed to separate `r = r * n` and `n -= 1`. Because of the `}`, no semicolon is needed after the second statement.

If you want to continue a long statement over two lines, you need to make sure that the first line ends in a symbol that *cannot be* the end of a statement. An operator is often a good choice:

```
s = s0 + (v - v0) * t + // The + tells the parser that this is not the end
  0.5 * (a - a0) * t * t
```

In practice, long expressions usually involve function or method calls, and then you don't need to worry much—after an opening `(`, the compiler won't infer the end of a statement until it has seen the matching `)`.

In the same spirit, Scala programmers favor the Kernighan & Ritchie brace style:

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

The line ending with a `{` sends a clear signal that there is more to come.

Many programmers coming from Java or C++ are initially uncomfortable about omitting semicolons. If you prefer to have them, just put them in—they do no harm.

2.3 Block Expressions and Assignments

In Java or C++, a block statement is a sequence of statements enclosed in `{ }`. You use a block statement whenever you need to put multiple actions in the body of a branch or loop statement.

In Scala, a `{ }` block contains a sequence of *expressions*, and the result is also an expression. The value of the block is the value of the last expression.

This feature can be useful if the initialization of a `val` takes more than one step. For example,

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

The value of the `{ }` block is the last expression, shown here in bold. The variables `dx` and `dy`, which were only needed as intermediate values in the computation, are neatly hidden from the rest of the program.

In Scala, assignments have no value—or, strictly speaking, they have a value of type `Unit`. Recall that the `Unit` type is the equivalent of the `void` type in Java and C++, with a single value written as `()`.

A block that ends with an assignment, such as

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

has a `Unit` value. This is not a problem, just something to be aware of when defining functions—see Section 2.7, "Functions," on page 20.

Since assignments have `Unit` value, don't chain them together.

```
x = y = 1 // No
```

The value of `y = 1` is `()`, and it's highly unlikely that you wanted to assign a `Unit` to `x`. (In contrast, in Java and C++, the value of an assignment is the value that is being assigned. In those languages, chained assignments are useful.)

2.4 Input and Output

To print a value, use the `print` or `println` function. The latter adds a newline character after the printout. For example,

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

yields the same output as

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

There is also a `printf` function with a C-style format string:

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

You can read a line of input from the console with the `readLine` function. To read a numeric, Boolean, or character value, use `readInt`, `readDouble`, `readByte`, `readShort`, `readLong`, `readFloat`, `readBoolean`, or `readChar`. The `readLine` method, but not the other ones, takes a prompt string:

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

2.5 Loops

Scala has the same while and do loops as Java and C++. For example,

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

Scala has no direct analog of the for (*initialize*; *test*; *update*) loop. If you need such a loop, you have two choices. You can use a while loop. Or, you can use a for statement like this:

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

You saw the to method of the RichInt class in Chapter 1. The call 1 to n returns a Range of the numbers from 1 to n (inclusive).

The construct

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

makes the variable i traverse all values of the expression to the right of the <-. Exactly how that traversal works depends on the type of the expression. For a Scala collection, such as a Range, the loop makes i assume each value in turn.

NOTE: There is no val or var before the variable in the for loop. The type of the variable is the element type of the collection. The scope of the loop variable extends until the end of the loop.

When traversing a string or array, you often need a range from 0 to $n - 1$. In that case, use the until method instead of the to method. It returns a range that doesn't include the upper bound.

```
val s = "Hello"
var sum = 0
for (i <- 0 until s.length) // Last value for i is s.length - 1
  sum += s(i)
```

In this example, there is actually no need to use indexes. You can directly loop over the characters:

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

In Scala, loops are not used as often as in other languages. As you will see in Chapter 12, you can often process the values in a sequence by applying a function to all of them, which can be done with a single method call.

NOTE: Scala has no `break` or `continue` statements to break out of a loop. What to do if you need a `break`? Here are a few options:

1. Use a Boolean control variable instead.
2. Use nested functions—you can `return` from the middle of a function.
3. Use the `break` method in the `Breaks` object:

```
import scala.util.control.Breaks._
breakable {
   for (...) {
      if (...) break; // Exits the breakable block
      ...
   }
}
```

Here, the control transfer is done by throwing and catching an exception, so you should avoid this mechanism when time is of the essence.

2.6 Advanced for Loops and for Comprehensions

In the preceding section, you saw the basic form of the `for` loop. However, this construct is much richer in Scala than in Java or C++. This section covers the advanced features.

You can have multiple *generators* of the form *variable* <- *expression*. Separate them by semicolons. For example,

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

Each generator can have a *guard*, a Boolean condition preceded by `if`:

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

Note that there is no semicolon before the `if`.

You can have any number of *definitions*, introducing variables that can be used inside the loop:

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

When the body of the `for` loop starts with `yield`, then the loop constructs a collection of values, one for each iteration:

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

This type of loop is called a `for` *comprehension*.

The generated collection is compatible with the first generator.

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

NOTE: If you prefer, you can enclose the generators, guards, and definitions of a `for` loop inside braces, and you can use newlines instead of semicolons to separate them:

```
for { i <- 1 to 3
  from = 4 - i
  j <- from to 3 }
```

2.7 Functions

Scala has functions in addition to methods. A method operates on an object, but a function doesn't. C++ has functions as well, but in Java, you have to imitate them with static methods.

To define a function, you specify the function's name, parameters, and body like this:

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

You must specify the types of all parameters. However, as long as the function is not recursive, you need not specify the return type. The Scala compiler determines the return type from the type of the expression to the right of the `=` symbol.

If the body of the function requires more than one expression, use a block. The last expression of the block becomes the value that the function returns. For example, the following function returns the value of `r` after the `for` loop.

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

There is no need for the `return` keyword in this example. It is possible to use `return` as in Java or C++, to exit a function immediately, but that is not commonly done in Scala.

TIP: While there is nothing wrong with using `return` in a named function (except the waste of seven keystrokes), it is a good idea to get used to life without `return`. Pretty soon, you will be using lots of *anonymous functions*, and there, `return` doesn't return a value to the caller. It breaks out to the enclosing named function. Think of `return` as a kind of `break` statement for functions, and only use it when you want that breakout functionality.

With a recursive function, you must specify the return type. For example,

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

Without the return type, the Scala compiler couldn't verify that the type of `n * fac(n - 1)` is an `Int`.

NOTE: Some programming languages (such as ML and Haskell) *can* infer the type of a recursive function, using the Hindley-Milner algorithm. However, this doesn't work well in an object-oriented language. Extending the Hindley-Milner algorithm so it can handle subtypes is still a research problem.

2.8 Default and Named Arguments L1

You can provide default arguments for functions that are used when you don't specify explicit values. For example,

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

This function has two parameters, `left` and `right`, with default arguments `"["` and `"]"`.

If you call `decorate("Hello")`, you get `"[Hello]"`. If you don't like the defaults, supply your own: `decorate("Hello", "<<<", ">>>")`.

If you supply fewer arguments than there are parameters, the defaults are applied from the end. For example, `decorate("Hello", ">>>[")` uses the default value of the `right` parameter, yielding `">>>[Hello]"`.

You can also specify the parameter names when you supply the arguments. For example,

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

The result is "<<<Hello>>>". Note that the named arguments need not be in the same order as the parameters.

Named arguments can make a function call more readable. They are also useful if a function has many default parameters.

You can mix unnamed and named arguments, provided the unnamed ones come first:

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

2.9 Variable Arguments L1

Sometimes, it is convenient to implement a function that can take a variable number of arguments. The following example shows the syntax:

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

You can call this function with as many arguments as you like.

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

The function receives a single parameter of type `Seq`, which we will discuss in Chapter 13. For now, all you need to know is that you can use a `for` loop to visit each element.

If you already have a sequence of values, you cannot pass it directly to such a function. For example, the following is not correct:

```
val s = sum(1 to 5) // Error
```

If the `sum` function is called with one argument, that must be a single integer, not a range of integers. The remedy is to tell the compiler that you want the parameter to be considered an argument sequence. Append `: _*`, like this:

```
val s = sum(1 to 5: _*) // Consider 1 to 5 as an argument sequence
```

This call syntax is needed in a recursive definition:

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

Here, the `head` of a sequence is its initial element, and `tail` is a sequence of all other elements. That's again a `Seq`, and we have to use `: _*` to convert it to an argument sequence.

CAUTION: When you call a Java method with variable arguments of type `Object`, such as `PrintStream.printf` or `MessageFormat.format`, you need to convert any primitive types by hand. For example,

```
val str = MessageFormat.format("The answer to {0} is {1}",
  "everything", 42.asInstanceOf[AnyRef])
```

This is the case for any `Object` parameter, but I mention it here because it is most common with varargs methods.

2.10 Procedures

Scala has a special notation for a function that returns no value. If the function body is enclosed in braces *without a preceding* = *symbol*, then the return type is `Unit`. Such a function is called a *procedure*. A procedure returns no value, and you only call it for its side effect. For example, the following procedure prints a string inside a box, like

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

Because the procedure doesn't return any value, we omit the = symbol.

```
def box(s : String) { // Look carefully: no =
  val border = "-" * s.length + "--\n"
  println(border + "|" + s + "|\n" + border)
}
```

Some people (not me) dislike this concise syntax for procedures and suggest that you always use an explicit return type of `Unit`:

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

CAUTION: The concise procedure syntax can be a surprise for Java and C++ programmers. It is a common error to accidentally omit the = in a function definition. You then get an error message at the point where the function is called, and you are told that `Unit` is not acceptable at that location.

2.11 Lazy Values L1

When a `val` is declared as `lazy`, its initialization is deferred until it is accessed for the first time. For example,

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

(We will discuss file operations in Chapter 9. For now, just take it for granted that this call reads all characters from a file into a string.)

If the program never accesses `words`, the file is never opened. To verify this, try it out in the REPL, but misspell the file name. There will be no error when the initialization statement is executed. However, when you access `words`, you will get an error message that the file is not found.

Lazy values are useful to delay costly initialization statements. They can also deal with other initialization issues, such as circular dependencies. Moreover, they are essential for developing lazy data structures—see Section 13.13, "Streams," on page 173.

You can think of lazy values as halfway between `val` and `def`. Compare

```
val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
  // Evaluated as soon as words is defined
lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
  // Evaluated the first time words is used
def words = scala.io.Source.fromFile("/usr/share/dict/words").mkString
  // Evaluated every time words is used
```

NOTE: Laziness is not cost-free. Every time a lazy value is accessed, a method is called that checks, in a threadsafe manner, whether the value has already been initialized.

2.12 Exceptions

Scala exceptions work the same way as in Java or C++. When you throw an exception, for example

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

the current computation is aborted, and the runtime system looks for an exception handler that can accept an `IllegalArgumentException`. Control resumes with the innermost such handler. If no such handler exists, the program terminates.

As in Java, the objects that you throw need to belong to a subclass of `java.lang.Throwable`. However, unlike Java, Scala has no "checked" exceptions—you never have to declare that a function or method might throw an exception.

NOTE: In Java, "checked" exceptions are checked at compile time. If your method might throw an `IOException`, you must declare it. This forces programmers to think where those exceptions should be handled, which is a laudable goal. Unfortunately, it can also give rise to monstrous method signatures such as `void doSomething() throws IOException, InterruptedException, ClassNotFoundException`. Many Java programmers detest this feature and end up defeating it by either catching exceptions too early or using excessively general exception classes. The Scala designers decided against checked exceptions, recognizing that thorough compile-time checking isn't *always* a good thing.

A `throw` expression has the special type `Nothing`. That is useful in `if/else` expressions. If one branch has type `Nothing`, the type of the `if/else` expression is the type of the other branch. For example, consider

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

The first branch has type `Double`, the second has type `Nothing`. Therefore, the `if/else` expression also has type `Double`.

The syntax for catching exceptions is modeled after the pattern matching syntax (see Chapter 14).

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

As in Java or C++, the more general exception types should come after the more specific ones.

Note that you can use `_` for the variable name if you don't need it.

The `try/finally` statement lets you dispose of a resource whether or not an exception has occurred. For example:

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

The `finally` clause is executed whether or not the `process` function throws an exception. The `reader` is always closed.

This code is a bit subtle, and it raises several issues.

- What if the `URL` constructor or the `openStream` method throws an exception? Then the `try` block is never entered, and neither is the `finally` clause. That's just as well—`in` was never initialized, so it makes no sense to invoke `close` on it.
- Why isn't `val in = new URL(...).openStream()` inside the `try` block? Then the scope of `in` would not extend to the `finally` clause.
- What if `in.close()` throws an exception? Then that exception is thrown out of the statement, superseding any earlier one. (This is just like in Java, and it isn't very nice. Ideally, the old exception would stay attached to the new one.)

Note that `try/catch` and `try/finally` have complementary goals. The `try/catch` statement handles exceptions, and the `try/finally` statement takes some action (usually cleanup) when an exception is not handled. It is possible to combine them into a single `try/catch/finally` statement:

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

This is the same as

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

However, that combination is rarely useful.

Exercises

1. The *signum* of a number is 1 if the number is positive, –1 if it is negative, and 0 if it is zero. Write a function that computes this value.
2. What is the value of an empty block expression `{}`? What is its type?
3. Come up with one situation where the assignment `x = y = 1` is valid in Scala. (Hint: Pick a suitable type for `x`.)
4. Write a Scala equivalent for the Java loop

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

5. Write a procedure `countdown(n: Int)` that prints the numbers from *n* to 0.
6. Write a `for` loop for computing the product of the Unicode codes of all letters in a string. For example, the product of the characters in `"Hello"` is `9415087488L`.
7. Solve the preceding exercise without writing a loop. (Hint: Look at the `StringOps` Scaladoc.)
8. Write a function `product(s : String)` that computes the product, as described in the preceding exercises.

9. Make the function of the preceding exercise a recursive function.
10. Write a function that computes x^n, where n is an integer. Use the following recursive definition:
    - $x^n = y^2$ if n is even and positive, where $y = x^{n / 2}$.
    - $x^n = x \cdot x^{n-1}$ if n is odd and positive.
    - $x^0 = 1$.
    - $x^n = 1 / x^{-n}$ if n is negative.

    Don't use a `return` statement.

This page intentionally left blank

Index

O

P

U

V