SCALA

for the Impatient

Cay Horstmann

Scala for the Impatient

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. And, last but not least, Scala is statically typed, enabling the compiler to find errors, so that you don't waste time finding them later in running programs (or worse, don't find them).

I wrote this book for *impatient* readers who want to start programming with Scala right away. I assume you know Java, C#, or C++, and I won't bore you with explaining variables, loops, or classes. I won't exhaustively list all features of the language, I won't lecture you about the superiority of one paradigm over another, and I won'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 the following levels of expertise for application programmers and library designers:

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. 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 (Agile Owl Software), Michael Davis (Collaborative Consulting), 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

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 3, 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 10).

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.)

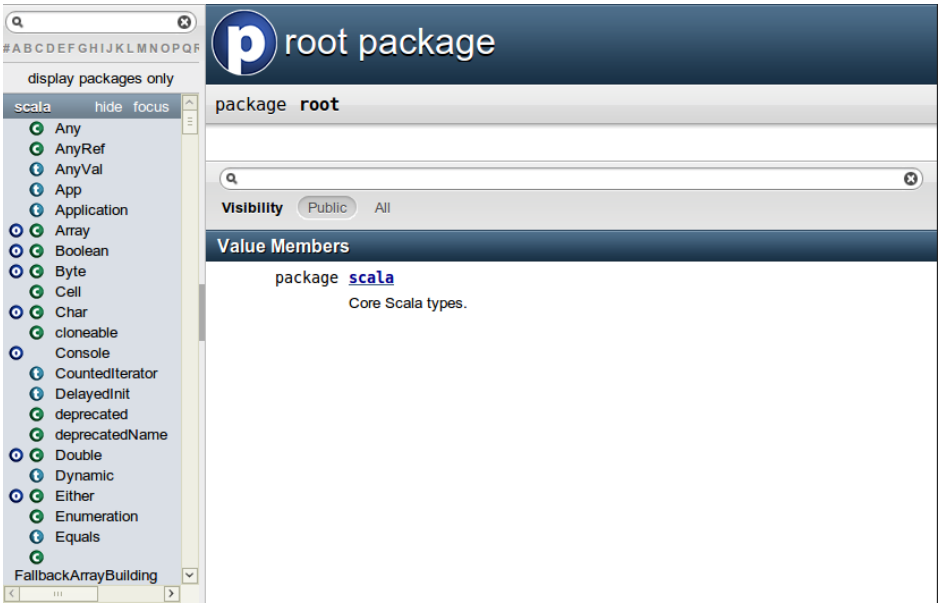

Figure 1–1 The entry page for Scaladoc

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).

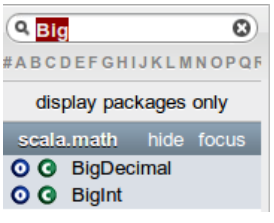

Figure 1–2 The filter box in Scaladoc

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 17.)

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 96 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 `else -1`.

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 * x` 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 statement, 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 22.

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 new line 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, take 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 sum(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).

```
try {
  process(new URL("http://horstmann.com/fred-tiny.gif"))
} 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:

```
var 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 `825152896`.
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.

Working with Arrays

Topics in This Chapter A1

Chapter 3

In this chapter, you will learn how to work with arrays in Scala. Java and C++ programmers usually choose an array or its close relation (such as array lists or vectors) when they need to collect a bunch of elements. In Scala, there are other choices (see Chapter 13), but for now, I'll assume you are impatient and just want to get going with arrays.

Key points of this chapter:

- Use an `Array` if the length is fixed, and an `ArrayBuffer` if the length can vary.
- Don't use `new` when supplying initial values.
- Use `()` to access elements.
- Use `for (elem <- arr)` to traverse the elements.
- Use `for (elem <- arr if ...) ... yield ...` to transform into a new array.
- Scala and Java arrays are interoperable; with `ArrayBuffer`, use `scala.collection.JavaConversions`.

3.1 Fixed-Length Arrays

If you need an array whose length doesn't change, use the `Array` type in Scala. For example,

```
val nums = new Array[Int](10)
  // An array of ten integers, all initialized with zero
val a = new Array[String](10)
  // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
  // An Array[String] of length 2—the type is inferred
  // Note: No new when you supply initial values
s(0) = "Goodbye"
  // Array("Goodbye", "World")
  // Use () instead of [] to access elements
```

Inside the JVM, a Scala `Array` is implemented as a Java array. The arrays in the preceding example have the type `java.lang.String[]` inside the JVM. An array of `Int`, `Double`, or another equivalent of the Java primitive types is a primitive type array. For example, `Array(2,3,5,7,11)` is an `int[]` in the JVM.

3.2 Variable-Length Arrays: Array Buffers

Java has `ArrayList` and C++ has `vector` for arrays that grow and shrink on demand. The equivalent in Scala is the `ArrayBuffer`.

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
  // Or new ArrayBuffer[Int]
  // An empty array buffer, ready to hold integers
b += 1
  // ArrayBuffer(1)
  // Add an element at the end with +=
b += (1, 2, 3, 5)
  // ArrayBuffer(1, 1, 2, 3, 5)
  // Add multiple elements at the end by enclosing them in parentheses
b ++= Array(8, 13, 21)
  // ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)
  // You can append any collection with the ++= operator
b.trimEnd(5)
  // ArrayBuffer(1, 1, 2)
  // Removes the last five elements
```

Adding or removing elements at the end of an array buffer is an efficient ("amortized constant time") operation.

You can also insert and remove elements at an arbitrary location, but those operations are not as efficient—all elements after that location must be shifted. For example:

```
b.insert(2, 6)
  // ArrayBuffer(1, 1, 6, 2)
  // Insert before index 2
b.insert(2, 7, 8, 9)
  // ArrayBuffer(1, 1, 7, 8, 9, 6, 2)
  // You can insert as many elements as you like
b.remove(2)
  // ArrayBuffer(1, 1, 8, 9, 6, 2)
b.remove(2, 3)
  // ArrayBuffer(1, 1, 2)
  // The second parameter tells how many elements to remove
```

Sometimes, you want to build up an `Array`, but you don't yet know how many elements you will need. In that case, first make an array buffer, then call

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

Conversely, call `a.toBuffer` to convert the array `a` to an array buffer.

3.3 Traversing Arrays and Array Buffers

In Java and C++, there are several syntactical differences between arrays and array lists/vectors. Scala is much more uniform. Most of the time, you can use the same code for both.

Here is how you traverse an array or array buffer with a `for` loop:

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

The variable `i` goes from `0` to `a.length - 1`.

The `until` method belongs to the `RichInt` class, and it returns all numbers up to (but not including) the upper bound. For example,

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

Note that `0 until 10` is actually a method call `0.until(10)`.

The construct

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

makes the variable `i` traverse all values of the range. In our case, the loop variable `i` assumes the values `0`, `1`, and so on until (but not including) `a.length`.

To visit every second element, let `i` traverse

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

To visit the elements starting from the end of the array, traverse

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

If you don't need the array index in the loop body, visit the array elements directly, like this:

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

This is very similar to the "enhanced" `for` loop in Java or the "range-based" `for` loop in C++. The variable `elem` is set to `a(0)`, then `a(1)`, and so on.

3.4 Transforming Arrays

In the preceding sections, you saw how to work with arrays just like you would in Java or C++. But in Scala, you can go further. It is very easy to take an array (or array buffer) and transform it in some way. Such transformations don't modify the original array, but they yield a new one.

Use a `for` comprehension like this:

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

The `for (...) yield` loop creates a new collection of the same type as the original collection. If you started with an array, you get another array. If you started with an array buffer, that's what you get from `for (...) yield`.

The result contains the expressions after the `yield`, one for each iteration of the loop.

Oftentimes, when you traverse a collection, you only want to process the elements that match a particular condition. This is achieved with a *guard*: an `if` inside the `for`. Here we double every even element, dropping the odd ones:

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

Keep in mind that the result is a new collection—the original collection is not affected.

NOTE: Alternatively, you could write

```
a.filter(_ % 2 == 0).map(2 * _)
```

or even

```
a filter { _ % 2 == 0 } map { 2 * _ }
```

Some programmers with experience in functional programming prefer `filter` and `map` to guards and `yield`. That's just a matter of style—the `for` loop does exactly the same work. Use whichever you find easier.

Consider the following example. Given a sequence of integers, we want to remove all but the first negative number. A traditional sequential solution would set a flag when the first negative number is called, then remove all elements beyond.

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

But wait—that's actually not so good. It's inefficient to remove variables from the front. We should start from the back after finding the first match.

In Scala, your life can be easier. First, let's use a `for`/`yield` loop to find all matching index values.

```
val indexes = for (i <- 0 until a.length if a(i) < 0) yield i
```

Then we visit the indexes in reverse, except for `indexes(0)`.

```
for (j <- (1 until indexes.length).reverse) a.remove(indexes(j))
```

The key observation is that it is better to have *all index values together* instead of seeing them one by one.

3.5 Common Algorithms

It is often said that a large percentage of business computations are nothing but computing sums and sorting. Fortunately, Scala has built-in functions for these tasks.

```
Array(1, 7, 2, 9).sum
  // 19
  // Works for ArrayBuffer too
```

In order to use the `sum` method, the element type must be a numeric type: either an integral or floating-point type or `BigInteger`/`BigDecimal`.

Similarly, the `min` and `max` methods yield the smallest and largest element in an array or array buffer.

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

The `sorted` method sorts an array or array buffer and *returns* the sorted array or array buffer, without modifying the original:

```
val b = ArrayBuffer(1, 7, 2, 9)
val bSorted = b.sorted(_ < _)
  // b is unchanged; bSorted is ArrayBuffer(1, 2, 7, 9)
```

You pass the comparison function as a parameter—see Chapter 12 for the function syntax.

You can sort an array, but not an array buffer, in place:

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

For the `min`, `max`, and `quickSort` methods, the element type must have a comparison operation. This is the case for numbers, strings, and other types with the `Ordered` trait.

Finally, if you want to display the contents of an array or array buffer, the `mkString` method lets you specify the separator between elements. A second variant has parameters for the prefix and suffix. For example,

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

Contrast with `toString`:

```
a.toString
  // "[I@b73e5"
  // This is the useless toString method from Java
b.toString
  // "ArrayBuffer(1, 7, 2, 9)"
  // The toString method reports the type, which is useful for debugging
```

3.6 Deciphering Scaladoc

There are lots of useful methods on arrays and array buffers, and it is a good idea to browse the Scala documentation to get an idea of what's there.

NOTE: The methods for the `Array` class are listed under `ArrayOps`. Technically, an array is converted to an `ArrayOps` object before any of the operations is applied.

Because Scala has a richer type system than Java, you may encounter some strange-looking syntax as you browse the Scala documentation. Fortunately, you don't have to understand all nuances of the type system to do useful work. Use Table 3–1 as a "decoder ring."

Table 3–1 Scaladoc Decoder Ring

Scaladoc	Explanation
`def count(p: (A) => Boolean): Int`	This method takes a *predicate*, a function from `A` to Boolean. It counts for how many elements the function is true. For example, `a.count(_ > 0)` counts how many elements of `a` are positive.
`def append(elems: A*): Unit`	This method takes *zero or more* arguments of type `A`. For example, `b.append(1, 7, 2, 9)` appends four elements to `b`.
`def appendAll(xs: TraversableOnce[A]): Unit`	The `xs` parameter can be any collection with the `TraversableOnce` trait, the most general trait in the Scala collections hierarchy. Other common traits that you may encounter in Scaladoc are `Traversable` and `Iterable`. All Scala collections implement these traits, and the difference between them is academic for library users. Simply think "any collection" when you see one of these. However, the `Seq` trait requires element access by an integer index. Think "array, list, or string."

Table 3–1 Scaladoc Decoder Ring *(Continued)*

Scaladoc	Explanation
`def += (elem: A): ArrayBuffer.this.type`	This method returns `this`, which allows you to chain calls, for example: `b += 4 -= 5`. When you work with an `ArrayBuffer[A]`, you can just think of the method as `def += (elem: A) : ArrayBuffer[A]`. If someone forms a subclass of `ArrayBuffer`, then the return type of `+=` is that subclass.
`def copyToArray[B >: A] (xs: Array[B]): Unit`	Note that the function copies an `ArrayBuffer[A]` into an `Array[B]`. Here, `B` is allowed to be a *supertype* of `A`. For example, you can copy from an `ArrayBuffer[Int]` to an `Array[Any]`. At first reading, just ignore the `[B >: A]` and replace `B` with `A`.
`def max[B >: A] (implicit cmp: Ordering[B]): A`	`A` must have a supertype `B` for which an "implicit" object of type `Ordering[B]` exists. Such an ordering exists for numbers, strings, and other types with the `Ordered` trait, as well as for classes that implement the Java `Comparable` interface.
`def padTo[B >: A, That](len: Int, elem: B) (implicit bf: CanBuildFrom[ArrayBuffer[A], B, That]): That`	This declaration happens when the method creates a new collection. Skip it and look for the simpler alternative, in this case `def padTo (len: Int, elem: A) : ArrayBuffer[A]` A future version of Scaladoc will hide these declarations.

3.7 Multidimensional Arrays

Like in Java, multidimensional arrays are implemented as arrays of arrays. For example, a two-dimensional array of `Double` values has the type `Array[Array[Double]]`. To construct such an array, use the `ofDim` method:

```
val matrix = Array.ofDim[Double](3, 4) // Three rows, four columns
```

To access an element, use two pairs of parentheses:

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

You can make ragged arrays, with varying row lengths:

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

3.8 Interoperating with Java

Since Scala arrays are implemented as Java arrays, you can pass them back and forth between Java and Scala.

If you call a Java method that receives or returns a `java.util.List`, you could, of course, use a Java `ArrayList` in your Scala code—but that is unattractive. Instead, import the implicit conversion methods in `scala.collection.JavaConversions`. Then you can use Scala buffers in your code, and they automatically get wrapped into Java lists when calling a Java method.

For example, the `java.lang.ProcessBuilder` class has a constructor with a `List<String>` parameter. Here is how you can call it from Scala:

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

The Scala buffer is wrapped into an object of a Java class that implements the `java.util.List` interface.

Conversely, when a Java method returns a `java.util.List`, you can have it automatically converted into a `Buffer`:

```
import scala.collection.JavaConversions.asScalaBuffer
import scala.collection.mutable.Buffer
val cmd : Buffer[String] = pb.command() // Java to Scala
  // You can't use ArrayBuffer—the wrapped object is only guaranteed to be a Buffer
```

If the Java method returns a wrapped Scala buffer, then the implicit conversion unwraps the original object. In our example, `cmd == command`.

Exercises

1. Write a code snippet that sets `a` to an array of `n` random integers between `0` (inclusive) and `n` (exclusive).
2. Write a loop that swaps adjacent elements of an array of integers. For example, `Array(1, 2, 3, 4, 5)` becomes `Array(2, 1, 4, 3, 5)`.
3. Repeat the preceding assignment, but produce a new array with the swapped values. Use `for/yield`.

4. Given an array of integers, produce a new array that contains all positive values of the original array, in their original order, followed by all values that are zero or negative, in their original order.
5. How do you compute the average of an `Array[Double]`?
6. How do you rearrange the elements of an `Array[Int]` so that they appear in reverse sorted order? How do you do the same with an `ArrayBuffer[Int]`?
7. Write a code snippet that produces all values from an array with duplicates removed. (Hint: Look at Scaladoc.)
8. Rewrite the example at the end of Section 3.4, "Transforming Arrays," on page 34 using the `drop` method for dropping the index of the first match. Look the method up in Scaladoc.
9. Make a collection of all time zones returned by `java.util.TimeZone.getAvailableIDs` that are in America. Strip off the `"America/"` prefix and sort the result.
10. Import `java.awt.datatransfer._` and make an object of type `SystemFlavorMap` with the call

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

    Then call the `getNativesForFlavor` method with parameter `DataFlavor.imageFlavor` and get the return value as a Scala buffer. (Why this obscure class? It's hard to find uses of `java.util.List` in the standard Java library.)

Maps and Tuples

Topics in This Chapter A1

Chapter 4

A classic programmer's saying is, "If you can only have one data structure, make it a hash table." Hash tables—or, more generally, maps—are among the most versatile data structures. As you will see in this chapter, Scala makes it particularly easy to use them.

Maps are collections of key/value pairs. Scala has a general notation of tuples—aggregates of *n* objects, not necessarily of the same type. A pair is simply a tuple with $n = 2$. Tuples are useful whenever you need to aggregate two or more values together, and we briefly discuss the syntax at the end of this chapter.

Highlights of the chapter are:

- Scala has a pleasant syntax for creating, querying, and traversing maps.
- You need to choose between mutable and immutable maps.
- By default, you get a hash map, but you can also get a tree map.
- You can easily convert between Scala and Java maps.
- Tuples are useful for aggregating values.

4.1 Constructing a Map

You can construct a map as

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

This constructs an immutable `Map[String, Int]` whose contents can't be changed. If you want a mutable map, use

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

If you want to start out with a blank map, you have to pick a map implementation and supply type parameters:

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

In Scala, a map is a collection of *pairs*. A pair is simply a grouping of two values, not necessarily of the same type, such as `("Alice", 10)`.

The -> operator makes a pair. The value of

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

is

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

You could have equally well defined the map as

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

The -> operator is just a little easier on the eyes than the parentheses. It also supports the intuition that a map data structure is a kind of function that maps keys to values. The difference is that a function computes values, and a map just looks them up.

4.2 Accessing Map Values

In Scala, the analogy between functions and maps is particularly close because you use the `()` notation to look up key values.

```
val bobsScore = scores("Bob") // Like scores.get("Bob") in Java
```

If the map doesn't contain a value for the requested key, an exception is thrown.

To check whether there is a key with the given value, call the `contains` method:

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

Since this call combination is so common, there is a shortcut:

```
val bobsScore = scores.getOrElse("Bob", 0)
  // If the map contains the key "Bob", return the value; otherwise, return 0.
```

Finally, the call *map*`.get(`*key*`)` returns an `Option` object that is either `Some(`*value for key*`)` or `None`. We discuss the `Option` class in Chapter 14.

4.3 Updating Map Values

In a mutable map, you can update a map value, or add a new one, with a () to the left of an = sign:

```
scores("Bob") = 10
  // Updates the existing value for the key "Bob" (assuming scores is mutable)
scores("Fred") = 7
  // Adds a new key/value pair to scores (assuming it is mutable)
```

Alternatively, you can use the += operation to add multiple associations:

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

To remove a key and its associated value, use the -= operator:

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

You can't update an immutable map, but you can do something that's just as useful—obtain a new map that has the desired update:

```
val newScores = scores + ("Bob" -> 10, "Fred" -> 7) // New map with update
```

The newScores map contains the same associations as scores, except that "Bob" has been updated and "Fred" added.

Instead of saving the result as a new value, you can update a var:

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

Similarly, to remove a key from an immutable map, use the - operator to obtain a new map without the key:

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

You might think that it is inefficient to keep constructing new maps, but that is not the case. The old and new maps share most of their structure. (This is possible because they are immutable.)

4.4 Iterating over Maps

The following amazingly simple loop iterates over all key/value pairs of a map:

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

The magic here is that you can use pattern matching in a Scala for loop. (Chapter 14 has all the details.) That way, you get the key and value of each pair in the map without any tedious method calls.

If for some reason you just want to visit the keys or values, use the `keySet` and `values` methods, as you would in Java. The `values` method returns an `Iterable` that you can use in a `for` loop.

```
scores.keySet // A set such as Set("Bob", "Cindy", "Fred", "Alice")
for (v <- scores.values) println(v) // Prints 10 8 7 10 or some permutation thereof
```

To reverse a map—that is, switch keys and values—use

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

4.5 Sorted Maps

When working with a map, you need to choose an implementation—a hash table or a balanced tree. By default, Scala gives you a hash table. You might want a tree map if you don't have a good hash function for the keys, or if you need to visit the keys in sorted order.

To get an immutable tree map instead of a hash map, use

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

Unfortunately, there is (as of Scala 2.9) no mutable tree map. Your best bet is to adapt a Java `TreeMap`, as described in Chapter 13.

TIP: If you want to visit the keys in insertion order, use a `LinkedHashMap`. For example,

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

4.6 Interoperating with Java

If you get a Java map from calling a Java method, you may want to convert it to a Scala map so that you can use the pleasant Scala map API. This is also useful if you want to work with a mutable tree map, which Scala doesn't provide.

Simply add an `import` statement:

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

Then trigger the conversion by specifying the Scala map type:

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

In addition, you can get a conversion from `java.util.Properties` to a `Map[String, String]`:

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

Conversely, to pass a Scala map to a method that expects a Java map, provide the opposite implicit conversion. For example:

```
import scala.collection.JavaConversions.mapAsJavaMap
import java.awt.font.TextAttribute._ // Import keys for map below
val attrs = Map(FAMILY -> "Serif", SIZE -> 12) // A Scala map
val font = new java.awt.Font(attrs) // Expects a Java map
```

4.7 Tuples

Maps are collections of key/value pairs. Pairs are the simplest case of *tuples*—aggregates of values of different types.

A tuple value is formed by enclosing individual values in parentheses. For example,

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

is a tuple of type

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

which is also written as

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

If you have a tuple, say,

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

then you can access its components with the methods _1, _2, _3, for example:

```
val second = t._2 // Sets second to 3.14
```

Unlike array or string positions, the component positions of a tuple start with 1, not 0.

NOTE: You can write `t._2` as `t _2` (with a space instead of a period), but not `t_2`.

Usually, it is better to use pattern matching to get at the components of a tuple, for example

```
val (first, second, third) = t // Sets first to 1, second to 3.14, third to "Fred"
```

You can use a _ if you don't need all components:

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

Tuples are useful for functions that return more than one value. For example, the `partition` method of the `StringOps` class returns a pair of strings, containing the characters that fulfill a condition and those that don't:

```
"New York".partition(_.isUpper) // Yields the pair ("NY", "ew ork")
```

4.8 Zipping

One reason for using tuples is to bundle together values so that they can be processed together. This is commonly done with the `zip` method. For example, the code

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

yields an array of pairs

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

The pairs can then be processed together:

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

TIP: The `toMap` method turns a collection of pairs into a map.

If you have a collection of keys and a parallel collection of values, then zip them up and turn them into a map like this:

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

Exercises

1. Set up a map of prices for a number of gizmos that you covet. Then produce a second map with the same keys and the prices at a 10 percent discount.
2. Write a program that reads words from a file. Use a mutable map to count how often each word appears. To read the words, simply use a `java.util.Scanner`:

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

   Or look at Chapter 9 for a Scalaesque way.

   At the end, print out all words and their counts.
3. Repeat the preceding exercise with an immutable map.
4. Repeat the preceding exercise with a sorted map, so that the words are printed in sorted order.

5. Repeat the preceding exercise with a `java.util.TreeMap` that you adapt to the Scala API.
6. Define a linked hash map that maps `"Monday"` to `java.util.Calendar.MONDAY`, and similarly for the other weekdays. Demonstrate that the elements are visited in insertion order.
7. Print a table of all Java properties, like this:

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

   You need to find the length of the longest key before you can print the table.
8. Write a function `minmax(values: Array[Int])` that returns a pair containing the smallest and largest values in the array.
9. Write a function `lteqgt(values: Array[Int], v: Int)` that returns a triple containing the counts of values less than `v`, equal to `v`, and greater than `v`.
10. What happens when you zip together two strings, such as `"Hello".zip("World")`? Come up with a plausible use case.

Classes

Topics in This Chapter A1

Chapter 5

In this chapter, you will learn how to implement classes in Scala. If you know classes in Java or C++, you won't find this difficult, and you will enjoy the much more concise notation of Scala.

The key points of this chapter are:

- Fields in classes automatically come with getters and setters.
- You can replace a field with a custom getter/setter without changing the client of a class—that is the "uniform access principle."
- Use the `@BeanProperty` annotation to generate the JavaBeans `getXxx`/`setXxx` methods.
- Every class has a primary constructor that is "interwoven" with the class definition. Its parameters turn into the fields of the class. The primary constructor executes all statements in the body of the class.
- Auxiliary constructors are optional. They are called `this`.

5.1 Simple Classes and Parameterless Methods

In its simplest form, a Scala class looks very much like its equivalent in Java or C++:

```
class Counter {
  private var value = 0 // You must initialize the field
  def increment() { value += 1 } // Methods are public by default
  def current() = value
}
```

In Scala, a class is not declared as `public`. A Scala source file can contain multiple classes, and all of them have public visibility.

To use this class, you construct objects and invoke methods in the usual way:

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

You can call a parameterless method (such as `current`) with or without parentheses:

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

Which form should you use? It is considered good style to use `()` for a *mutator* method (a method that changes the object state), and to drop the `()` for an *accessor* method (a method that does not change the object state).

That's what we did in our example:

```
myCounter.increment() // Use () with mutator
println(myCounter.current) // Don't use () with accessor
```

You can enforce this style by declaring `current` without `()`:

```
class Counter {
  ...
  def current = value // No () in definition
}
```

Now the class user must use `myCounter.current`, without parentheses.

5.2 Properties with Getters and Setters

When writing a Java class, we don't like to use public fields:

```
public class Person { // This is Java
  public int age; // Frowned upon in Java
}
```

With a public field, anyone could write to `fred.age`, making Fred younger or older. That's why we prefer to use getter and setter methods:

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

A getter/setter pair such as this one is often called a *property*. We say that the class `Person` has an `age` property.

Why is this any better? By itself, it isn't. Anyone can call `fred.setAge(21)`, keeping him forever twenty-one.

But if that becomes a problem, we can guard against it:

```
public void setAge(int newValue) { if (newValue > age) age = newValue; }
  // Can't get younger
```

Getters and setters are better than public fields because they let you start with simple get/set semantics and evolve them as needed.

NOTE: Just because getters and setters are better than public fields doesn't mean they are always good. Often, it is plainly bad if every client can get or set bits and pieces of an object's state. In this section, I show you how to implement properties in Scala. It is up to you to choose wisely when a gettable/settable property is an appropriate design.

Scala provides getter and setter methods for every field. Here, we define a public field:

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

Scala generates a class for the JVM with a *private* `age` field and getter and setter methods. These methods are public because we did not declare `age` as `private`. (For a private field, the getter and setter methods are private.)

In Scala, the getter and setter methods are called `age` and `age_=`. For example,

```
println(fred.age) // Calls the method fred.age()
fred.age = 21 // Calls fred.age_=(21)
```

NOTE: To see these methods with your own eyes, compile the `Person` class and then look at the bytecode with `javap`:

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

As you can see, the compiler created methods `age` and `age_$eq`. (The `=` symbol is translated to `$eq` because the JVM does not allow an `=` in a method name.)

NOTE: In Scala, the getters and setters are not named `getXxx` and `setXxx`, but they fulfill the same purpose. Section 5.5, "Bean Properties," on page 57 shows how to generate Java-style `getXxx` and `setXxx` methods, so that your Scala classes can interoperate with Java tools.

At any time, you can redefine the getter and setter methods yourself. For example,

```
class Person {
  private var privateAge = 0 // Make private and rename

  def age = privateAge
  def age_=(newValue: Int) {
    if (newValue > privateAge) privateAge = newValue; // Can't get younger
  }
}
```

The user of your class still accesses `fred.age`, but now Fred can't get younger:

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

NOTE: Bertrand Meyer, the inventor of the influential Eiffel language, formulated the *Uniform Access Principle* that states: "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation." In Scala, the caller of `fred.age` doesn't know whether `age` is implemented through a field or a method. (Of course, in the JVM, the service is *always* implemented through a method, either synthesized or programmer-supplied.)

TIP: It may sound scary that Scala generates getter and setter methods for every field. But you have some control over this process.

- If the field is private, the getter and setter are private.
- If the field is a `val`, only a getter is generated.
- If you don't want any getter or setter, declare the field as `private[this]` (see Section 5.4, "Object-Private Fields," on page 56).

5.3 Properties with Only Getters

Sometimes you want a *read-only property* with a getter but no setter. If the value of the property never changes after the object has been constructed, use a `val` field:

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

Scala makes a private `final` field and a getter method, but no setter.

Sometimes, however, you want a property that a client can't set at will, but that is mutated in some other way. The `Counter` class from Section 5.1, "Simple Classes and Parameterless Methods," on page 51 is a good example. Conceptually, the counter has a `current` property that is updated when the `increment` method is called, but there is no setter for the property.

You can't implement such a property with a `val`—a `val` never changes. Instead, provide a private field and a property getter, like this:

```
class Counter {
  private var value = 0
  def increment() { value += 1 }
  def current = value // No () in declaration
}
```

Note that there are no () in the definition of the getter method. Therefore, you *must* call the method without parentheses:

```
val n = myCounter.current // Calling myCounter.current() is a syntax error
```

To summarize, you have four choices for implementing properties:

1. `var foo`: Scala synthesizes a getter and a setter.
2. `val foo`: Scala synthesizes a getter.
3. You define methods `foo` and `foo_=`.
4. You define a method `foo`.

NOTE: In Scala, you cannot have a write-only property (that is, a property with a setter and no getter).

TIP: When you see a field in a Scala class, remember that it is not the same as a field in Java or C++. It is a private field *together with* a getter (for a `val` field) or a getter and a setter (for a `var` field).

5.4 Object-Private Fields

In Scala (as well as in Java or C++), a method can access the private fields of *all* objects of its class. For example,

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

  def isLess(other : Counter) = value < other.value
    // Can access private field of other object
}
```

Accessing `other.value` is legal because `other` is also a `Counter` object.

Scala allows an even more severe access restriction, with the `private[this]` qualifier:

```
private[this] var value = 0 // Accessing someObject.value is not allowed
```

Now, the methods of the `Counter` class can only access the `value` field of the current object, not of other objects of type `Counter`. This access is sometimes called *object-private*, and it is common in some OO languages, such as SmallTalk.

With a class-private field, Scala generates private getter and setter methods. However, for an object-private field, no getters and setters are generated at all.

NOTE: Scala allows you to grant access rights to specific classes. The `private[`*ClassName*`]` qualifier states that only methods of the given class can access the given field. Here, the *ClassName* must be the name of the class being defined or an enclosing class. (See Section 5.8, "Nested Classes," on page 62 for a discussion of inner classes.)

In this case, the implementation will generate auxiliary getter and setter methods that allow the enclosing class to access the field. These methods will be public because the JVM does not have a fine-grained access control system, and they will have implementation-dependent names.

5.5 Bean Properties L1

As you saw in the preceding sections, Scala provides getter and setter methods for the fields that you define. However, the names of these methods are not what Java tools expect. The JavaBeans specification (www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html) defines a Java property as a pair of `getFoo`/`setFoo` methods (or just a `getFoo` method for a read-only property). Many Java tools rely on this naming convention.

When you annotate a Scala field with `@BeanProperty`, then such methods are automatically generated. For example,

```
import scala.reflect.BeanProperty

class Person {
  @BeanProperty var name: String = _
}
```

generates *four* methods:

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

Table 5–1 shows which methods are generated in all cases.

NOTE: If you define a field as a primary constructor parameter (see Section 5.7, "The Primary Constructor," on page 59), and you want JavaBeans getters and setters, annotate the constructor parameter like this:

```
class Person(@BeanProperty var name: String)
```

Table 5–1 Generated Methods for Fields

Scala Field	Generated Methods	When to Use
`val/var name`	public `name` `name_=` (`var` only)	To implement a property that is publicly accessible and backed by a field.
`@BeanProperty val/var name`	public `name` `getName()` `name_=` (`var` only) `setName(...)` (`var` only)	To interoperate with JavaBeans.
`private val/var name`	private `name` `name_=` (`var` only)	To confine the field to the methods of this class, just like in Java. Use `private` unless you really want a public property.
`private[this] val/var name`	none	To confine the field to methods invoked on the same object. Not commonly used.
`private[`*ClassName*`] val/var name`	implementation-dependent	To grant access to an enclosing class. Not commonly used.

5.6 Auxiliary Constructors

As in Java or C++, a Scala class can have as many constructors as you like. However, a Scala class has one constructor that is more important than all the others, called the *primary constructor*. In addition, a class may have any number of *auxiliary constructors*.

We discuss auxiliary constructors first because they are easier to understand. They are very similar to constructors in Java or C++, with just two differences.

1. The auxiliary constructors are called `this`. (In Java or C++, constructors have the same name as the class—which is not so convenient if you rename the class.)
2. Each auxiliary constructor *must* start with a call to a previously defined auxiliary constructor or the primary constructor.

Here is a class with two auxiliary constructors.

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

  def this(name: String) { // An auxiliary constructor
    this() // Calls primary constructor
    this.name = name
  }

  def this(name: String, age: Int) { // Another auxiliary constructor
    this(name) // Calls previous auxiliary constructor
    this.age = age
  }
}
```

We will look at the primary constructor in the next section. For now, it is sufficient to know that a class for which you don't define a primary constructor has a primary constructor with no arguments.

You can construct objects of this class in three ways:

```
val p1 = new Person // Primary constructor
val p2 = new Person("Fred") // First auxiliary constructor
val p3 = new Person("Fred", 42) // Second auxiliary constructor
```

5.7 The Primary Constructor

In Scala, every class has a primary constructor. The primary constructor is not defined with a `this` method. Instead, it is interwoven with the class definition.

1. The parameters of the primary constructor are placed *immediately after the class name*.

   ```
   class Person(val name: String, val age: Int) {
     // Parameters of primary constructor in (...)
     ...
   }
   ```

   Parameters of the primary constructor turn into fields that are initialized with the construction parameters. In our example, `name` and `age` become fields of the `Person` class. A constructor call such as `new Person("Fred", 42)` sets the `name` and `age` fields.

   Half a line of Scala is the equivalent of seven lines of Java:

```
public class Person { // This is Java
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String name() { return this.name; }
  public int age() { return this.age; }
  ...
}
```

2. The primary constructor executes *all statements in the class definition*. For example, in the following class

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

the `println` statement is a part of the primary constructor. It is executed whenever an object is constructed.

This is useful when you need to configure a field during construction. For example:

```
class MyProg {
  private val props = new Properties
  props.load(new FileReader("myprog.properties"))
    // The statement above is a part of the primary constructor
  ...
}
```

NOTE: If there are no parameters after the class name, then the class has a primary constructor with no parameters. That constructor simply executes all statements in the body of the class.

TIP: You can often eliminate auxiliary constructors by using default arguments in the primary constructor. For example:

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

Primary constructor parameters can have any of the forms in Table 5–1. For example,

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

declares and initializes fields

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

Construction parameters can also be regular method parameters, without `val` or `var`. How these parameters are processed depends on their usage inside the class.

- If a parameter without `val` or `var` is used inside at least one method, it becomes a field. For example,

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

  declares and initializes immutable fields `name` and `age` that are object-private.

  Such a field is the equivalent of a `private[this] val` field (see Section 5.4, "Object-Private Fields," on page 56).

- Otherwise, the parameter is not saved as a field. It's just a regular parameter that can be accessed in the code of the primary constructor. (Strictly speaking, this is an implementation-specific optimization.)

Table 5–2 summarizes the fields and methods that are generated for different kinds of primary constructor parameters.

Table 5–2 Fields and Methods Generated for Primary Constructor Parameters

Primary Constructor Parameter	Generated Field/Methods
`name: String`	object-private field, or no field if no method uses `name`
`private val/varname: String`	private field, private getter/setter
`val/var name: String`	private field, public getter/setter
`@BeanProperty val/varname: String`	private field, public Scala and JavaBeans getters/setters

If you find the primary constructor notation confusing, you don't need to use it. Just provide one or more auxiliary constructors in the usual way, but remember to call `this()` if you don't chain to another auxiliary constructor.

However, many programmers like the concise syntax. Martin Odersky suggests to think about it this way: In Scala, classes take parameters, just like methods do.

NOTE: When you think of the primary constructor's parameters as class parameters, parameters without `val` or `var` become easier to understand. The scope of such a parameter is the entire class. Therefore, you can use the parameter in methods. If you do, it is the compiler's job to save it in a field.

TIP: The Scala designers think that *every keystroke is precious*, so they let you combine a class with its primary constructor. When reading a Scala class, you need to disentangle the two. For example, when you see

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

take this definition apart into a class definition:

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

and a constructor definition:

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

NOTE: To make the primary constructor private, place the keyword `private` like this:

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

A class user must then use an auxiliary constructor to construct a `Person` object.

5.8 Nested Classes L1

In Scala, you can nest just about anything inside anything. You can define functions inside other functions, and classes inside other classes. Here is a simple example of the latter.

```
import scala.collection.mutable.ArrayBuffer
class Network {
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Member]
  }

  private val members = new ArrayBuffer[Member]

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

Consider two networks:

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

In Scala, each *instance* has its own class `Member`, just like each instance has its own field `members`. That is, `chatter.Member` and `myFace.Member` are *different classes*.

NOTE: This is different from Java, where an inner class belongs to the outer class.

The Scala approach is more regular. For example, to make a new inner object, you simply use `new` with the type name: `new chatter.Member`. In Java, you need to use a special syntax, `chatter.new Member()`.

In our network example, you can add a member within its own network, but not across networks.

```
val fred = chatter.join("Fred")
val wilma = chatter.join("Wilma")
fred.contacts += wilma // OK
val barney = myFace.join("Barney") // Has type myFace.Member
fred.contacts += barney
  // No—can't add a myFace.Member to a buffer of chatter.Member elements
```

For networks of people, this behavior probably makes sense. If you don't want it, there are two solutions.

First, you can move the `Member` type somewhere else. A good place would be the `Network` companion object. (Companion objects are described in Chapter 6.)

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

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

Alternatively, you can use a *type projection* `Network#Member`, which means "a `Member` of *any* `Network`." For example,

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

You would do that if you want the fine-grained "inner class per object" feature in some places of your program, but not everywhere. See Chapter 18 for more information about type projections.

NOTE: In a nested class, you can access the `this` reference of the enclosing class as *EnclosingClass*`.this`, like in Java. If you like, you can establish an alias for that reference with the following syntax:

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

The `class Network { outer =>` syntax makes the variable `outer` refer to `Network.this`. You can choose any name for this variable. The name `self` is common, but perhaps confusing when used with nested classes.

This syntax is related to the "self type" syntax that you will see in Chapter 18.

Exercises

1. Improve the `Counter` class in Section 5.1, "Simple Classes and Parameterless Methods," on page 51 so that it doesn't turn negative at `Int.MaxValue`.
2. Write a class `BankAccount` with methods `deposit` and `withdraw`, and a read-only property `balance`.
3. Write a class `Time` with read-only properties `hours` and `minutes` and a method `before(other: Time): Boolean` that checks whether this time comes before the other. A `Time` object should be constructed as `new Time(hrs, min)`, where `hrs` is in military time format (between 0 and 23).
4. Reimplement the `Time` class from the preceding exercise so that the internal representation is the number of minutes since midnight (between 0 and $24 \times 60 - 1$). *Do not* change the public interface. That is, client code should be unaffected by your change.
5. Make a class `Student` with read-write JavaBeans properties `name` (of type `String`) and `id` (of type `Long`). What methods are generated? (Use `javap` to check.) Can you call the JavaBeans getters and setters in Scala? Should you?
6. In the `Person` class of Section 5.1, "Simple Classes and Parameterless Methods," on page 51, provide a primary constructor that turns negative ages to 0.
7. Write a class `Person` with a primary constructor that accepts a string containing a first name, a space, and a last name, such as `new Person("Fred Smith")`. Supply read-only properties `firstName` and `lastName`. Should the primary constructor parameter be a `var`, a `val`, or a plain parameter? Why?
8. Make a class `Car` with read-only properties for manufacturer, model name, and model year, and a read-write property for the license plate. Supply four constructors. All require the manufacturer and model name. Optionally, model year and license plate can also be specified in the constructor. If not, the model year is set to -1 and the license plate to the empty string. Which constructor are you choosing as the primary constructor? Why?
9. Reimplement the class of the preceding exercise in Java, C#, or C++ (your choice). How much shorter is the Scala class?
10. Consider the class

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

Rewrite it to use explicit fields and a default primary constructor. Which form do you prefer? Why?

Objects

Topics in This Chapter A1

Chapter 6

In this short chapter, you will learn when to use the `object` construct in Scala. Use it when you need a class with a single instance, or when you want to find a home for miscellaneous values or functions.

The key points of this chapter are:

- Use objects for singletons and utility methods.
- A class can have a companion object with the same name.
- Objects can extend classes or traits.
- The `apply` method of an object is usually used for constructing new instances of the companion class.
- To avoid the `main` method, use an object that extends the `App` trait.
- You can implement enumerations by extending the `Enumeration` object.

6.1 Singletons

Scala has no static methods or fields. Instead, you use the `object` construct. An object defines a single instance of a class with the features that you want. For example,

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

When you need a new unique account number in your application, call `Accounts.newUniqueNumber()`.

The constructor of an object is executed when the object is first used. In our example, the `Accounts` constructor is executed with the first call to `Accounts.newUniqueNumber()`. If an object is never used, its constructor is not executed.

An object can have essentially all the features of a class—it can even extend other classes or traits (see Section 6.3, "Objects Extending a Class or Trait," on page 69). There is just one exception: You cannot provide constructor parameters.

You use an object in Scala whenever you would have used a singleton object in Java or C++:

- As a home for utility functions or constants
- When a single immutable instance can be shared efficiently
- When a single instance is required to coordinate some service (the singleton design pattern)

NOTE: Many people view the singleton design pattern with disdain. Scala gives you the tools for both good and bad design, and it is up to you to use them wisely.

6.2 Companion Objects

In Java or C++, you often have a class with both instance methods and static methods. In Scala, you achieve this by having a class and a "companion" object of the same name. For example,

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

object Account { // The companion object
  private var lastNumber = 0
  private def newUniqueNumber() = { lastNumber += 1; lastNumber }
}
```

The class and its companion object can access each other's private features. They must be located in the *same source file*.

NOTE: The companion object of a class is accessible, but it is not in scope. For example, the `Account` class has to call `Account.newUniqueNumber()` and not just `newUniqueNumber()` to invoke the method of the companion object.

TIP: In the REPL, you must define the class and the object together in paste mode. Type

```
:paste
```

Then type or paste both the class and object definitions, and type Ctrl+D.

6.3 Objects Extending a Class or Trait

An `object` can extend a class and/or one or more traits. The result is an object of a class that extends the given class and/or traits, and in addition has all of the features specified in the object definition.

One useful application is to specify default objects that can be shared. For example, consider a class for undoable actions in a program.

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

A useful default is the "do nothing" action. Of course, we only need one of them.

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

The `DoNothingAction` object can be shared across all places that need this default.

```
val actions = Map("open" -> DoNothingAction, "save" -> DoNothingAction, ...)
  // Open and save not yet implemented
```

6.4 The `apply` Method

It is common to have objects with an `apply` method. The `apply` method is called for expressions of the form

Object(*arg1*, ..., *argN*)

Typically, such an `apply` method returns an object of the companion class.

For example, the Array object defines apply methods that allow array creation with expressions such as

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

Why doesn't one just use a constructor? Not having the new keyword is handy for nested expressions, such as

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

CAUTION: It is easy to confuse Array(100) and new Array(100). The first expression calls apply(100), yielding an Array[Int] with a single element, the integer 100. The second expression invokes the constructor this(100). The result is an Array[Nothing] with 100 null elements.

Here is an example of defining an apply method:

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

object Account { // The companion object
  def apply(initialBalance: Double) =
    new Account(newUniqueNumber(), initialBalance)
  ...
}
```

Now you can construct an account as

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

6.5 Application Objects

Each Scala program must start with an object's main method of type Array[String] => Unit:

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

Instead of providing a main method for your application, you can extend the App trait and place the program code into the constructor body:

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

If you need the command-line arguments, you can get them from the `args` property:

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

If you invoke the application with the `scala.time` option set, then the elapsed time is displayed when the program exits.

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

All this involves a bit of magic. The `App` trait extends another trait, `DelayedInit`, that gets special handling from the compiler. All initialization code of a class with that trait is moved into a `delayedInit` method. The `main` of the `App` trait method captures the command-line arguments, calls the `delayedInit` method, and optionally prints the elapsed time.

NOTE: Older versions of Scala had an `Application` trait for the same purpose. That trait carried out the program's action in the static initializer, which is not optimized by the just-in-time compiler. Use the `App` trait instead.

6.6 Enumerations

Unlike Java or C++, Scala does not have enumerated types. However, the standard library provides an `Enumeration` helper class that you can use to produce enumerations.

Define an object that extends the `Enumeration` class and initialize each value in your enumeration with a call to the `Value` method. For example,

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

Here we define three fields, `Red`, `Yellow`, and `Green`, and initialize each of them with a call to `Value`. This is a shortcut for

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

Each call to the `Value` method returns a new instance of an inner class, also called `Value`.

Alternatively, you can pass IDs, names, or both to the `Value` method:

```
val Red = Value(0, "Stop")
val Yellow = Value(10) // Name "Yellow"
val Green = Value("Go") // ID 11
```

If not specified, the ID is one more than the previously assigned one, starting with zero. The default name is the field name.

You can now refer to the enumeration values as `TrafficLightColor.Red`, `TrafficLightColor.Yellow`, and so on. If that gets too tedious, use a statement

```
import TrafficLightColor._
```

(See Chapter 7 for more information on importing members of a class or object.)

Remember that the type of the enumeration is `TrafficLightColor.Value` and *not* `TrafficLightColor`—that's the type of the object holding the values. Some people recommend that you add a type alias

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

Now the type of the enumeration is `TrafficLightColor.TrafficLightColor`, which is only an improvement if you use an `import` statement. For example,

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

The ID of an enumeration value is returned by the `id` method, and its name by the `toString` method.

The call `TrafficLightColor.values` yields a set of all values:

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

Finally, you can look up an enumeration value by its ID or name. Both of the following yield the object `TrafficLightColor.Red`:

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

Exercises

1. Write an object `Conversions` with methods `inchesToCentimeters`, `gallonsToLiters`, and `milesToKilometers`.
2. The preceding problem wasn't very object-oriented. Provide a general superclass `UnitConversion` and define objects `InchesToCentimeters`, `GallonsToLiters`, and `MilesToKilometers` that extend it.
3. Define an `Origin` object that extends `java.awt.Point`. Why is this not actually a good idea? (Have a close look at the methods of the `Point` class.)
4. Define a `Point` class with a companion object so that you can construct `Point` instances as `Point(3, 4)`, without using `new`.
5. Write a Scala application, using the `App` trait, that prints the command-line arguments in reverse order, separated by spaces. For example, `scala Reverse Hello World` should print `World Hello`.
6. Write an enumeration describing the four playing card suits so that the `toString` method returns ♣, ♦, ♥, or ♠.
7. Implement a function that checks whether a card suit value from the preceding exercise is red.
8. Write an enumeration describing the eight corners of the RGB color cube. As IDs, use the color values (for example, `0xff0000` for `Red`).

Packages and Imports

Topics in This Chapter A1

Chapter 7

In this chapter, you will learn how packages and import statements work in Scala. Both packages and imports are more regular than in Java; they are also a bit more flexible.

The key points of this chapter are:

- Packages nest just like inner classes.
- Package paths are *not* absolute.
- A chain `x.y.z` in a package clause leaves the intermediate packages `x` and `x.y` invisible.
- Package statements without braces at the top of the file extend to the entire file.
- A package object can hold functions and variables.
- Import statements can import packages, classes, and objects.
- Import statements can be anywhere.
- Import statements can rename and hide members.
- `java.lang`, `scala`, and `Predef` are always imported.

7.1 Packages

Packages in Scala fulfill the same purpose as packages in Java or namespaces in C++: to manage names in a large program. For example, the name `Map` can occur in the packages `scala.collection.immutable` and `scala.collection.mutable` without conflict. To access either name, you can use the fully qualified `scala.collection.immutable.Map` or `scala.collection.mutable.Map`. Alternatively, use an `import` statement to provide a shorter alias—see Section 7.7, "Imports," on page 81.

To add items to a package, you can include them in package statements, such as:

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

Then the class name `Employee` can be accessed anywhere as `com.horstmann.impatient.Employee`.

Unlike the definition of an object or a class, a package can be defined in multiple files. The preceding code might be in a file `Employee.scala`, and a file `Manager.scala` might contain

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

NOTE: There is no enforced relationship between the directory of the source file and the package. You don't have to put `Employee.scala` and `Manager.scala` into a `com/horstmann/impatient` directory.

Conversely, you can contribute to more than one package in a single file. The file `Employee.scala` can contain

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

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

7.2 Scope Rules

In Scala, the scope rules for packages are more consistent than those in Java. Scala packages nest just like all other scopes. You can access names from the enclosing scope. For example,

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

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

Note the `Utils.percentOf` qualifier. The `Utils` class was defined in the *parent* package. Everything in the parent package is in scope, and it is not necessary to use `com.horstmann.Utils.percentOf`. (You could, though, if you prefer—after all, `com` is also in scope.)

There is a fly in the ointment, however. Consider

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

This code takes advantage of the fact that the `scala` package is always imported. Therefore, the `collection` package is actually `scala.collection`.

And now suppose someone introduces the following package, perhaps in a different file:

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

Now the `Manager` class no longer compiles. It looks for a `mutable` member inside the `com.horstmann.collection` package and doesn't find it. The intent in the `Manager` class was the `collection` package in the top-level `scala` package, not whatever `collection` subpackage happened to be in some accessible scope.

In Java, this problem can't occur because package names are always *absolute*, starting at the root of the package hierarchy. But in Scala, package names are relative, just like inner class names. With inner classes, one doesn't usually run into problems because all the code is in one file, under control of whoever is in charge of that file. But packages are open-ended. Anyone can contribute to a package at any time.

One solution is to use absolute package names, starting with `_root_`, for example:

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

Another approach is to use "chained" package clauses, as detailed in the next section.

NOTE: Most programmers use complete paths for package names, without the `_root_` prefix. This is safe as long as everyone avoids names `scala`, `java`, `com`, `org`, and so on, for nested packages.

7.3 Chained Package Clauses

A package clause can contain a "chain," or path segment, for example:

```
package com.horstmann.impatient {
  // Members of com and com.horstmann are not visible here
  package people {
    class Person
    ...
  }
}
```

Such a clause limits the visible members. Now a `com.horstmann.collection` package would no longer be accessible as `collection`.

7.4 Top-of-File Notation

Instead of the nested notation that we have used up to now, you can have `package` clauses at the top of the file, without braces. For example:

```
package com.horstmann.impatient
package people

class Person
 ...
```

This is equivalent to

```
package com.horstmann.impatient {
  package people {
    class Person
    ...
    // Until the end of the file
  }
}
```

This is the preferred notation if all the code in the file belongs to the same package (which is the usual case).

Note that in the example above, everything in the file belongs to the package com.horstmann.impatient.people, but the package com.horstmann.impatient has also been opened up so you can refer to its contents.

7.5 Package Objects

A package can contain classes, objects, and traits, but not the definitions of functions or variables. That's an unfortunate limitation of the Java virtual machine. It would make more sense to add utility functions or constants to a package than to some Utils object. Package objects address this limitation.

Every package can have one package object. You define it in the *parent* package, and it has the same name as the child package. For example,

```
package com.horstmann.impatient

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

package people {
  class Person {
    var name = defaultName // A constant from the package
  }
  ...
}
```

Note that the defaultName value didn't need to be qualified because it was in the same package. Elsewhere, it is accessible as com.horstmann.impatient.people.defaultName.

Behind the scenes, the package object gets compiled into a JVM class with static methods and fields, called package.class, inside the package. In our example, that would be a class com.horstmann.impatient.people.package with a static field defaultName. (In the JVM, you can use package as a class name.)

It is a good idea to use the same naming scheme for source files. Put the package object into a file com/horstmann/impatient/people/package.scala. That way, anyone who wants to add functions or variables to a package can find the package object easily.

7.6 Package Visibility

In Java, a class member that isn't declared as public, private, or protected is visible in the package containing the class. In Scala, you can achieve the same effect with qualifiers. The following method is visible in its own package:

```
package com.horstmann.impatient.people

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

You can extend the visibility to an enclosing package:

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

7.7 Imports

Imports let you use short names instead of long ones. With the clause

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

you can write `Color` in your code instead of `java.awt.Color`.

That is the sole purpose of imports. If you don't mind long names, you'll never need them.

You can import all members of a package as

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

This is the same as the `*` wildcard in Java. (In Scala, `*` is a valid character for an identifier. You could define a package `com.horstmann.*.people`, but please don't.)

You can also import all members of a class or object.

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

This is like `import static` in Java. Java programmers seem to live in fear of this variant, but in Scala it is commonly used.

Once you import a package, you can access its subpackages with shorter names. For example:

```
import java.awt._

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

The `event` package is a member of `java.awt`, and the import brings it into scope.

7.8 Imports Can Be Anywhere

In Scala, an import statement can be anywhere, not just at the top of a file. The scope of the import statement extends until the end of the enclosing block. For example,

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

This is a very useful feature, particularly with wildcard imports. It is always a bit worrisome to import lots of names from different sources. In fact, some Java programmers dislike wildcard imports so much that they never use them, but let their IDE generate long lists of imported classes.

By putting the imports where they are needed, you can greatly reduce the potential for conflicts.

7.9 Renaming and Hiding Members

If you want to import a few members from a package, use a *selector* like this:

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

The selector syntax lets you rename members:

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

Now `JavaHashMap` is a `java.util.HashMap` and plain `HashMap` is a `scala.collection.mutable.HashMap`.

The selector `HashMap => _` hides a member instead of renaming it. This is only useful if you import others:

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

Now `HashMap` unambiguously refers to `scala.collection.mutable.HashMap` since `java.util.HashMap` is hidden.

7.10 Implicit Imports

Every Scala program implicitly starts with

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

As with Java programs, `java.lang` is always imported. Next, the `scala` package is imported, but in a special way. Unlike all other imports, this one is allowed to override the preceding import. For example, `scala.StringBuilder` overrides `java.lang.StringBuilder` instead of conflicting with it.

Finally, the `Predef` object is imported. It contains quite a few useful functions. (These could equally well have been placed into the `scala` package object, but `Predef` was introduced before Scala had package objects.)

Since the `scala` package is imported by default, you never need to write package names that start with `scala`. For example,

```
collection.mutable.HashMap
```

is just as good as

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

Exercises

1. Write an example program to demonstrate that

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

   is not the same as

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

2. Write a puzzler that baffles your Scala friends, using a package `com` that isn't at the top level.
3. Write a package `random` with functions `nextInt(): Int`, `nextDouble(): Double`, and `setSeed(seed: Int): Unit`. To generate random numbers, use the linear congruential generator

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

   where $a = 1664525$, $b = 1013904223$, and $n = 32$.
4. Why do you think the Scala language designers provided the `package object` syntax instead of simply letting you add functions and variables to a package?
5. What is the meaning of `private[com] def giveRaise(rate: Double)`? Is it useful?
6. Write a program that copies all elements from a Java hash map into a Scala hash map. Use imports to rename both classes.
7. In the preceding exercise, move all imports into the innermost scope possible.

8. What is the effect of

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

   Is this a good idea?

9. Write a program that imports the `java.lang.System` class, reads the user name from the `user.name` system property, reads a password from the `Console` object, and prints a message to the standard error stream if the password is not `"secret"`. Otherwise, print a greeting to the standard output stream. Do not use any other imports, and do not use any qualified names (with dots).

10. Apart from `StringBuilder`, what other members of `java.lang` does the `scala` package override?

Inheritance

Topics in This Chapter A1

Chapter 8

In this chapter, you will learn the most important ways in which inheritance in Scala differs from its counterparts in Java and C++. The highlights are:

- The `extends` and `final` keywords are as in Java.
- You must use `override` when you override a method.
- Only the primary constructor can call the primary superclass constructor.
- You can override fields.

In this chapter, we only discuss the case in which a class inherits from another class. See Chapter 10 for inheriting *traits*—the Scala concept that generalizes Java interfaces.

8.1 Extending a Class

You extend a class in Scala just like you would in Java—with the `extends` keyword:

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

As in Java, you specify fields and methods that are new to the subclass or that override methods in the superclass.

As in Java, you can declare a class `final` so that it cannot be extended. Unlike Java, you can also declare individual methods or fields `final` so that they cannot be overridden. (See Section 8.6, "Overriding Fields," on page 91 for overriding fields.)

8.2 Overriding Methods

In Scala, you *must* use the `override` modifier when you override a method that isn't abstract. (See Section 8.8, "Abstract Classes," on page 93 for abstract methods.) For example,

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

The `override` modifier can give useful error messages in a number of common situations, such as:

- When you misspell the name of the method that you are overriding
- When you accidentally provide a wrong parameter type in the overriding method
- When you introduce a new method in a superclass that clashes with a subclass method

NOTE: The last case is an instance of the *fragile base class problem*, where a change in the superclass cannot be verified without looking at all the subclasses. Suppose programmer Alice defines a `Person` class, and, unbeknownst to Alice, programmer Bob defines a subclass `Student` with a method `id` yielding the student ID. Later, Alice also defines a method `id` that holds the person's national ID. When Bob picks up that change, something may break in Bob's program (but not in Alice's test cases) since `Student` objects now return unexpected IDs.

In Java, one is often advised to "solve" this problem by declaring all methods as `final` unless they are explicitly designed to be overridden. That sounds good in theory, but programmers hate it when they can't make even the most innocuous changes to a method (such as adding a logging call). That's why Java eventually introduced an optional `@Overrides` annotation.

Invoking a superclass method in Scala works exactly like in Java, with the keyword `super`:

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

The call `super.toString` invokes the `toString` method of the superclass—that is, the `Person.toString` method.

8.3 Type Checks and Casts

To test whether an object belongs to a given class, use the `isInstanceOf` method. If the test succeeds, you can use the `asInstanceOf` method to convert a reference to a subclass reference:

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

The `p.isInstanceOf[Employee]` test succeeds if `p` refers to an object of class `Employee` or its subclass (such as `Manager`).

If `p` is `null`, then `p.isInstanceOf[Employee]` returns `false` and `p.asInstanceOf[Employee]` returns `null`.

If `p` is not an `Employee`, then `p.asInstanceOf[Employee]` throws an exception.

If you want to test whether `p` refers to a `Employee` object, but not a subclass, use

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

The `classOf` method is defined in the `scala.Predef` object that is always imported.

Table 8–1 shows the correspondence between Scala and Java type checks and casts.

Table 8–1 Type Checks and Casts in Scala and Java

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

However, pattern matching is usually a better alternative to using type checks and casts. For example,

```
p match {
  case s: Employee => ... // Process s as a Employee
  case _ => ... // p wasn't a Employee
}
```

See Chapter 14 for more information.

8.4 Protected Fields and Methods

As in Java or C++, you can declare a field or method as `protected`. Such a member is accessible from any subclass, but not from other locations.

Unlike in Java, a `protected` member is *not* visible throughout the package to which the class belongs. (If you want this visibility, you can use a package modifier—see Chapter 7.)

There is also a `protected[this]` variant that restricts access to the current object, similar to the `private[this]` variant discussed in Chapter 5.

8.5 Superclass Construction

Recall from Chapter 5 that a class has one primary constructor and any number of auxiliary constructors, and that all auxiliary constructors must start with a call to a preceding auxiliary constructor or the primary constructor.

As a consequence, an auxiliary constructor can *never* invoke a superclass constructor directly.

The auxiliary constructors of the subclass eventually call the primary constructor of the subclass. Only the primary constructor can call a superclass constructor.

Recall that the primary constructor is intertwined with the class definition. The call to the superclass constructor is similarly intertwined. Here is an example:

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

This defines a subclass

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

and a primary constructor that calls the superclass constructor

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

Intertwining the class and the constructor makes for very concise code. You may find it helpful to think of the primary constructor parameters as parameters of

the class. Here, the Employee class has three parameters: name, age, and salary, two of which it "passes" to the superclass.

In Java, the equivalent code is quite a bit more verbose:

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

NOTE: In a Scala constructor, you can never call super(params), as you would in Java, to call the superclass constructor.

A Scala class can extend a Java class. Its primary constructor must invoke one of the constructors of the Java superclass. For example,

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

8.6 Overriding Fields

Recall from Chapter 5 that a field in Scala consists of a private field *and* accessor/mutator methods. You can override a val (or a parameterless def) with another val field of the same name. The subclass has a private field and a public getter, and the getter overrides the superclass getter (or method).

For example,

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

class SecretAgent(codename: String) extends Person(codename) {
  override val name = "secret" // Don't want to reveal name . . .
  override val toString = "secret" // . . . or class name
}
```

This example shows the mechanics, but it is rather artificial. A more common case is to override an abstract def with a val, like this:

```
abstract class Person { // See Section 8.8 for abstract classes
  def id: Int // Each person has an ID that is computed in some way
  ...
}

class Student(override val id: Int) extends Person
  // A student ID is simply provided in the constructor
```

Note the following restrictions (see also Table 8–2):

- A `def` can only override another `def`.
- A `val` can only override another `val` or a parameterless `def`.
- A `var` can only override an abstract `var` (see Section 8.8, "Abstract Classes," on page 93).

Table 8–2 Overriding `val`, `def`, and `var`

	with `val`	**with** `def`	**with** `var`
Override `val`	• Subclass has a private field (with the same name as the superclass field—that's OK). • Getter overrides the superclass getter.	Error	Error
Override `def`	• Subclass has a private field. • Getter overrides the superclass method.	Like in Java.	A `var` can override a getter/setter pair. Overriding just a getter is an error.
Override `var`	Error	Error	Only if the superclass `var` is abstract (see Section 8.8).

NOTE: In Chapter 5, I said that it's OK to use a `var` because you can always change your mind and reimplement it as a getter/setter pair. However, the programmers extending your class do not have that choice. They cannot override a `var` with a getter/setter pair. In other words, if you provide a `var`, all subclasses are stuck with it.

8.7 Anonymous Subclasses

As in Java, you make an instance of an *anonymous* subclass if you include a block with definitions or overrides, such as

```
val alien = new Person("Fred") {
  def greeting = "Greetings, Earthling! My name is Fred."
}
```

Technically, this creates an object of a *structural type*—see Chapter 18 for details. The type is denoted as `Person{def greeting: String}`. You can use this type as a parameter type:

```
def meet(p: Person{def greeting: String}) {
  println(p.name + " says: " + p.greeting)
}
```

8.8 Abstract Classes

As in Java, you can use the `abstract` keyword to denote a class that cannot be instantiated, usually because one or more of its methods are not defined. For example,

```
abstract class Person(val name: String) {
  def id: Int // No method body—this is an abstract method
}
```

Here we say that every person has an ID, but we don't know how to compute it. Each concrete subclass of `Person` needs to specify an `id` method. In Scala, unlike Java, you do not use the `abstract` keyword for an abstract method. You simply omit its body. As in Java, a class with at least one abstract method must be declared `abstract`.

In a subclass, you need not use the `override` keyword when you define a method that was abstract in the superclass.

```
class Employee(name: String) extends Person(name) {
  def id = name.hashCode // override keyword not required
}
```

8.9 Abstract Fields

In addition to abstract methods, a class can also have abstract fields. An abstract field is simply a field without an initial value. For example,

```
abstract class Person {
  val id: Int
    // No initializer—this is an abstract field with an abstract getter method
  var name: String
    // Another abstract field, with abstract getter and setter methods
}
```

This class defines abstract getter methods for the `id` and `name` fields, and an abstract setter for the `name` field. The generated Java class has *no fields*.

Concrete subclasses must provide concrete fields, for example:

```
class Employee(val id: Int) extends Person { // Subclass has concrete id property
  var name = "" // and concrete name property
}
```

As with methods, no `override` keyword is required in the subclass when you define a field that was abstract in the superclass.

You can always customize an abstract field by using an anonymous type:

```
val fred = new Person {
  val id = 1729
  var name = "Fred"
}
```

8.10 Construction Order and Early Definitions L3

When you overrride a `val` in a subclass *and* use the value in a superclass constructor, the resulting behavior is unintuitive.

Here is an example. A creature can sense a part of its environment. For simplicity, we assume the creature lives in a one-dimensional world, and the sensory data are represented as integers. A default creature can see ten units ahead.

```
class Creature {
  val range: Int = 10
  val env: Array[Int] = new Array[Int](range)
}
```

Ants, however, are near-sighted:

```
class Ant extends Creature {
  override val range = 2
}
```

Unfortunately, we now have a problem. The `range` value is used in the superclass constructor, and the superclass constructor runs *before* the subclass constructor. Specifically, here is what happens:

1. The `Ant` constructor calls the `Creature` constructor before doing its own construction.
2. The `Creature` constructor sets *its* `range` field to `10`.
3. The `Creature` constructor, in order to initialize the `env` array, calls the `range()` getter.
4. That method is overridden to yield the (as yet uninitialized) `range` field of the `Ant` class.
5. The `range` method returns `0`. (That is the initial value of all integer fields when an object is allocated.)
6. `env` is set to an array of length `0`.
7. The `Ant` constructor continues, setting its `range` field to `2`.

Even though it appears as if `range` is either `10` or `2`, `env` has been set to an array of length `0`. The moral is that you should not rely on the value of a `val` in the body of a constructor.

In Java, you have a similar issue when you call a method in a superclass constructor. The method might be overridden in a subclass, and it might not do what you want it to do. (In fact, that is the root cause of our problem—the expression `range` calls the getter method.)

There are several remedies.

- Declare the `val` as `final`. This is safe but not very flexible.
- Declare the `val` as `lazy` in the superclass (see Chapter 2). This is safe but a bit inefficient.
- Use the *early definition syntax* in the subclass—see below.

The "early definition" syntax lets you initialize `val` fields of a subclass *before* the superclass is executed. The syntax is so ugly that only a mother could love it. You place the `val` fields in a block after the `extends` keyword, like this:

```
class Bug extends {
  override val range = 3
} with Creature
```

Note the `with` keyword before the superclass name. This keyword is normally used with traits—see Chapter 10.

The right-hand side of an early definition can only refer to previous early definitions, not to other fields or methods of the class.

TIP: You can debug construction order problems with the `-Xcheckinit` compiler flag. This flag generates code that throws an exception (instead of yielding the default value) when an uninitialized field is accessed.

NOTE: At the root of the construction order problem lies a design decision of the Java language—namely, to allow the invocation of subclass methods in a superclass constructor. In C++, an object's virtual function table pointer is set to the table of the superclass when the superclass constructor executes. Afterwards, the pointer is set to the subclass table. Therefore, in C++, it is not possible to modify constructor behavior through overriding. The Java designers felt that this subtlety was unnecessary, and the Java virtual machine does not adjust the virtual function table during construction.

8.11 The Scala Inheritance Hierarchy

Figure 8–1 shows the inheritance hierarchy of Scala classes. The classes that correspond to the primitive types in Java, as well as the type `Unit`, extend `AnyVal`.

All other classes are subclasses of the `AnyRef` class, which is a synonym for the `Object` class from the Java or .NET virtual machine.

Both `AnyVal` and `AnyRef` extend the `Any` class, the root of the hierarchy.

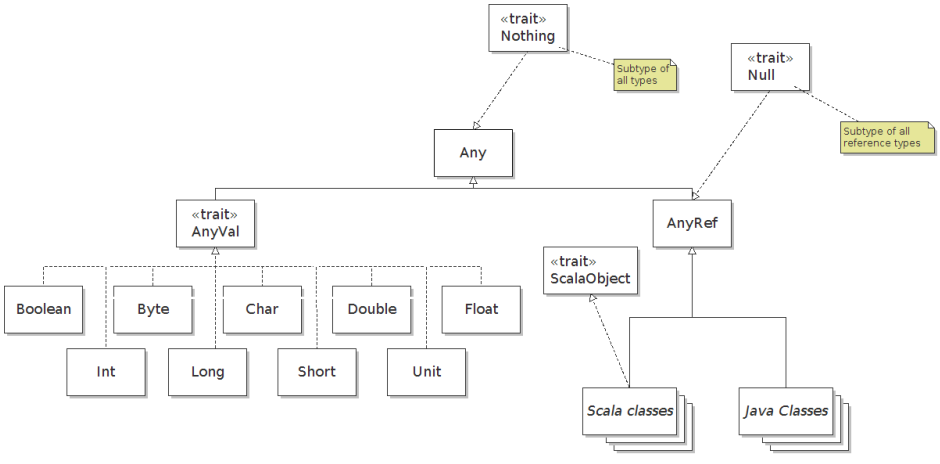

Figure 8–1 The Inheritance Hierarchy of Scala Classes

The `Any` class defines methods `isInstanceOf`, `asInstanceOf`, and the methods for equality and hash codes that we will look at in Section 8.12, "Object Equality," on page 97.

`AnyVal` does not add any methods. It is just a marker for value types.

The AnyRef class adds the monitor methods wait and notify/notifyAll from the Object class. It also provides a synchronized method with a function parameter. That method is the equivalent of a synchronized block in Java. For example,

```
account.synchronized { account.balance += amount }
```

NOTE: Just like in Java, I suggest you stay away from wait, notify, and synchronized unless you have a good reason to use them instead of higher-level concurrency constructs.

All Scala classes implement the marker interface ScalaObject, which has no methods.

At the other end of the hierarchy are the Nothing and Null types.

Null is the type whose sole instance is the value null. You can assign null to any reference, but not to one of the value types. For example, setting an Int to null is not possible. This is better than in Java, where it would be possible to set an Integer wrapper to null.

The Nothing type has no instances. It is occasionally useful for generic constructs. For example, the empty list Nil has type List[Nothing], which is a subtype of List[T] for any T.

CAUTION: The Nothing type is not at all the same as void in Java or C++. In Scala, void is represented by the Unit type, the type with the sole value (). Note that Unit is not a supertype of any other type. However, the compiler still allows any value to be *replaced* by a (). Consider

```
def printAny(x: Any) { println(x) }
def printUnit(x: Unit) { println(x) }
printAny("Hello") // Prints Hello
printUnit("Hello")
  // Replaces "Hello" with () and calls printUnit(()), which prints ()
```

8.12 Object Equality L1

In Scala, the eq method of the AnyRef class checks whether two references refer to the same object. The equals method in AnyRef calls eq. When you implement a class, you should consider overriding the equals method to provide a natural notion of equality for your situation.

For example, if you define a class Item(val description: String, val price: Double), you might want to consider two items equal if they have the same description and price. Here is an appropriate equals method:

```
final override def equals(other: Any) = {
  val that = other.asInstanceOf[Item]
  if (that == null) false
  else description == that.description && price == that.price
}
```

NOTE: We defined the method as `final` because it is generally very difficult to correctly extend equality in a subclass. The problem is symmetry. You want `a.equals(b)` to have the same result as `b.equals(a)`, even when `b` belongs to a subclass.

CAUTION: Be sure to define the `equals` method with parameter type `Any`. The following would be wrong:

```
final def equals(other: Item) = { ... }
```

This is a different method that does not override the `equals` method of `AnyRef`.

When you define `equals`, remember to define `hashCode` as well. The hash code should be computed only from the fields that you use in the equality check. In the `Item` example, combine the hash codes of the fields:

```
final override def hashCode = 13 * description.hashCode + 17 * price.hashCode
```

TIP: You are not compelled to override `equals` and `hashCode`. For many classes, it is appropriate to consider distinct objects unequal. For example, if you have two distinct input streams or radio buttons, you will never consider them equal.

In an application program, you don't generally call `eq` or `equals`. Simply use the `==` operator. For reference types, it calls `equals` after doing the appropriate check for `null` operands.

Exercises

1. Extend the following `BankAccount` class to a `CheckingAccount` class that charges $1 for every deposit and withdrawal.

```
class BankAccount(initialBalance: Double) {
  private var balance = initialBalance
  def deposit(amount: Double) = { balance += amount; balance }
  def withdraw(amount: Double) = { balance -= amount; balance }
}
```

2. Extend the `BankAccount` class of the preceding exercise into a class `SavingsAccount` that earns interest every month (when a method `earnMonthlyInterest` is called) and has three free deposits or withdrawals every month. Reset the transaction count in the `earnMonthlyInterest` method.
3. Consult your favorite Java or C++ textbook that is sure to have an example of a toy inheritance hierarchy, perhaps involving employees, pets, graphical shapes, or the like. Implement the example in Scala.
4. Define an abstract class `Item` with methods `price` and `description`. A `SimpleItem` is an item whose price and description are specified in the constructor. Take advantage of the fact that a `val` can override a `def`. A `Bundle` is an item that contains other items. Its price is the sum of the prices in the bundle. Also provide a mechanism for adding items to the bundle and a suitable `description` method.
5. Design a class `Point` whose x and y coordinate values can be provided in a constructor. Provide a subclass `LabeledPoint` whose constructor takes a label value and x and y coordinates, such as

   ```
   new LabeledPoint("Black Thursday", 1929, 230.07)
   ```

6. Define an abstract class `Shape` with an abstract method `centerPoint` and subclasses `Rectangle` and `Circle`. Provide appropriate constructors for the subclasses and override the `centerPoint` method in each subclass.
7. Provide a class `Square` that extends `java.awt.Rectangle` and has three constructors: one that constructs a square with a given corner point and width, one that constructs a square with corner `(0, 0)` and a given width, and one that constructs a square with corner `(0, 0)` and width `0`.
8. Compile the `Person` and `SecretAgent` classes in Section 8.6, "Overriding Fields," on page 91 and analyze the class files with `javap`. How many `name` fields are there? How many `name` getter methods are there? What do they get? (Hint: Use the `-c` and `-private` options.)
9. In the `Creature` class of Section 8.10, "Construction Order and Early Definitions," on page 94, replace `val range` with a `def`. What happens when you also use a `def` in the `Ant` subclass? What happens when you use a `val` in the subclass? Why?
10. The file `scala/collection/immutable/Stack.scala` contains the definition

    ```
    class Stack[A] protected (protected val elems: List[A])
    ```

    Explain the meanings of the `protected` keywords. (Hint: Review the discussion of private constructors in Chapter 5.)

Files and Regular Expressions

Topics in This Chapter A1

Chapter 9

In this chapter, you will learn how to carry out common file processing tasks, such as reading all lines or words from a file or reading a file containing numbers.

Chapter highlights:

- `Source.fromFile(...).getLines.toArray` yields all lines of a file.
- `Source.fromFile(...).mkString` yields the file contents as a string.
- To convert a string into a number, use the `toInt` or `toDouble` method.
- Use the Java `PrintWriter` to write text files.
- "*regex*".`r` is a `Regex` object.
- Use """..."""" if your regular expression contains backslashes or quotes.
- If a regex pattern has groups, you can extract their contents using the syntax `for (regex(`var_1`, ...,`var_n`) <-` *string*`)`.

9.1 Reading Lines

To read all lines from a file, call the `getLines` method on a `scala.io.Source` object:

```
import scala.io.Source
val source = Source.fromFile("myfile.txt", "UTF-8")
  // The first argument can be a string or a java.io.File
  // You can omit the encoding if you know that the file uses
  // the default platform encoding
val lineIterator = source.getLines
```

The result is an iterator (see Chapter 13). You can use it to process the lines one at a time:

```
for (l <- lineIterator) process l
```

Or you can put the lines into an array or array buffer by applying the `toArray` or `toBuffer` method to the iterator:

```
val lines = source.getLines.toArray
```

Sometimes, you just want to read an entire file into a string. That's even simpler:

```
val contents = source.mkString
```

CAUTION: Call `close` when you are done using the `Source` object.

9.2 Reading Characters

To read individual characters from a file, you can use a `Source` object directly as an iterator since the `Source` class extends `Iterator[Char]`:

```
for (c <- source) process c
```

If you want to be able to peek at a character without consuming it (like `istream::peek` in C++ or a `PushbackInputStreamReader` in Java), call the `buffered` method on the source. Then you can peek at the next input character with the `head` method without consuming it.

```
val source = Source.fromFile("myfile.txt", "UTF-8")
val iter = source.buffered
while (iter.hasNext) {
  if (iter.head is nice)
    process iter.next
  else
    ...
}
source.close()
```

Alternatively, if your file isn't large, you can just read it into a string and process that:

```
val contents = source.mkString
```

9.3 Reading Tokens and Numbers

Here is a quick-and-dirty way of reading all whitespace-separated tokens in a source:

```
val tokens = source.mkString.split("\\s+")
```

To convert a string into a number, use the `toInt` or `toDouble` method. For example, if you have a file containing floating-point numbers, you can read them all into an array by

```
val numbers = for (w <- tokens) yield w.toDouble
```

or

```
val numbers = tokens.map(_.toDouble)
```

TIP: Remember—you can always use the `java.util.Scanner` class to process a file that contains a mixture of text and numbers.

Finally, note that you can read numbers from the *console*:

```
print("How old are you? ")
  // Console is imported by default, so you don't need to qualify print and readInt
val age = readInt()
  // Or use readDouble or readLong
```

CAUTION: These methods assume that the next input line contains a single number, without leading or trailing whitespace. Otherwise, a `NumberFormatException` occurs.

9.4 Reading from URLs and Other Sources

The Source object has methods to read from sources other than files:

```
val source1 = Source.fromURL("http://horstmann.com", "UTF-8")
val source2 = Source.fromString("Hello, World!")
  // Reads from the given string—useful for debugging
val source3 = Source.stdin
  // Reads from standard input
```

CAUTION: When you read from a URL, you need to know the character set in advance, perhaps from an HTTP header. See www.w3.org/International/O-charset for more information.

9.5 Reading Binary Files

Scala has no provision for reading binary files. You'll need to use the Java library. Here is how you can read a file into a byte array:

```
val file = new File(filename)
val in = new FileInputStream(file)
val bytes = new Array[Byte](file.length.toInt)
in.read(bytes)
in.close()
```

9.6 Writing Text Files

Scala has no built-in support for writing files. To write a text file, use a java.io.PrintWriter, for example:

```
val out = new PrintWriter("numbers.txt")
for (i <- 1 to 100) out.println(i)
out.close()
```

Everything works as expected, except for the printf method. When you pass a number to printf, the compiler will complain that you need to convert it to an AnyRef:

```
out.printf("%6d %10.2f",
  quantity.asInstanceOf[AnyRef], price.asInstanceOf[AnyRef]) // Ugh
```

Instead, use the format method of the String class:

```
out.print("%6d %10.2f".format(quantity, price))
```

NOTE: The `printf` method of the `Console` class does not suffer from this problem. You can use

```
printf("%6d %10.2f", quantity, price)
```

for printing a message to the console.

9.7 Visiting Directories

There are currently no "official" Scala classes for visiting all files in a directory, or for recursively traversing directories. In this section, we discuss a couple of alternatives.

NOTE: A prior version of Scala had `File` and `Directory` classes. You can still find them in the `scala.tools.nsc.io` package inside scala-compiler.jar.

It is simple to write a function that produces an iterator through all subdirectories of a directory:

```
import java.io.File
def subdirs(dir: File): Iterator[File] = {
  val children = dir.listFiles.filter(_.isDirectory)
  children.toIterator ++ children.toIterator.flatMap(subdirs _)
}
```

With this function, you can visit all subdirectories like this:

```
for (d <- subdirs(dir)) process d
```

Alternatively, if you use Java 7, you can adapt the `walkFileTree` method of the `java.nio.file.Files` class. That class makes use of a `FileVisitor` interface. In Scala, we generally prefer to use function objects, not interfaces, for specifying work (even though in this case the interface allows more fine-grained control—see the Javadoc for details). The following implicit conversion adapts a function to the interface:

```
import java.nio.file._
implicit def makeFileVisitor(f: (Path) => Unit) = new SimpleFileVisitor[Path] {
  override def visitFile(p: Path, attrs: attribute.BasicFileAttributes) = {
    f(p)
    FileVisitResult.CONTINUE
  }
}
```

Then you can print all subdirectories with the call

```
Files.walkFileTree(dir.toPath, (f: Path) => println(f))
```

Of course, if you don't just want to print the files, you can specify other actions in the function that you pass to the `walkFileTree` method.

9.8 Serialization

In Java, serialization is used to transmit objects to other virtual machines or for short-term storage. (For long-term storage, serialization can be awkward—it is tedious to deal with different object versions as classes evolve over time.)

Here is how you declare a serializable class in Java and Scala.

Java:

```
public class Person implements java.io.Serializable {
  private static final long serialVersionUID = 42L;
  ...
}
```

Scala:

```
@SerialVersionUID(42L) class Person extends Serializable
```

The `Serializable` trait is defined in the `scala` package and does not require an import.

NOTE: You can omit the `@SerialVersionUID` annotation if you are OK with the default ID.

You serialize and deserialize objects in the usual way:

```
val fred = new Person(...)
import java.io._
val out = new ObjectOutputStream(new FileOutputStream("/tmp/test.obj"))
out.writeObject(fred)
out.close()
val in = new ObjectInputStream(new FileInputStream("/tmp/test.obj"))
val savedFred = in.readObject().asInstanceOf[Person]
```

The Scala collections are serializable, so you can have them as members of your serializable classes:

```
class Person extends Serializable {
  private val friends = new ArrayBuffer[Person] // OK—ArrayBuffer is serializable
  ..
}
```

9.9 Process Control A2

Traditionally, programmers use shell scripts to carry out mundane processing tasks, such as moving files from one place to another, or combining a set of files. The shell language makes it easy to specify subsets of files, and to pipe the output of one program into the input of another. However, as programming languages, most shell languages leave much to be desired.

Scala was designed to scale from humble scripting tasks to massive programs. The `scala.sys.process` package provides utilities to interact with shell programs. You can write your shell scripts in Scala, with all the power that the Scala language puts at your disposal.

Here is a simple example:

```
import sys.process._
"ls -al .." !
```

As a result, the `ls -al ..` command is executed, showing all files in the parent directory. The result is printed to standard output.

The `sys.process` package contains an implicit conversion from strings to `ProcessBuilder` objects. The `!` operator *executes* the `ProcessBuilder` object.

The result of the `!` operator is the exit code of the executed program: `0` if the program was successful, or a nonzero failure indicator otherwise.

If you use `!!` instead of `!`, the output is returned as a string:

```
val result = "ls -al .." !!
```

You can pipe the output of one program into the input of another, using the `#|` operator:

```
"ls -al .." #| "grep sec" !
```

NOTE: As you can see, the process library uses the commands of the underlying operating system. Here, I use `bash` commands because `bash` is available on Linux, Mac OS X, and Windows.

To redirect the output to a file, use the `#>` operator:

```
"ls -al .." #> new File("output.txt") !
```

To append to a file, use `#>>` instead:

```
"ls -al .." #>> new File("output.txt") !
```

To redirect input from a file, use `#<`:

```
"grep sec" #< new File("output.txt") !
```

You can also redirect input from a URL:

```
"grep Scala" #< new URL("http://horstmann.com/index.html") !
```

You can combine processes with `p #&& q` (execute `q` if `p` was successful) and `p #|| q` (execute `q` if `p` was unsuccessful). But frankly, Scala is better at control flow than the shell, so why not implement the control flow in Scala?

NOTE: The process library uses the familiar shell operators `| > >> < && ||`, but it prefixes them with a `#` so that they all have the same precedence.

If you need to run a process in a different directory, or with different environment variables, construct the `ProcessBuilder` with the `apply` method of the `Process` object. Supply the command, the starting directory, and a sequence of (name, value) pairs for environment settings.

```
val p = Process(cmd, new File(dirName), ("LANG", "en_US"))
```

Then execute it with the `!` operator:

```
"echo 42" #| p !
```

9.10 Regular Expressions

When you process input, you often want to use regular expressions to analyze it. The `scala.util.matching.Regex` class makes this simple. To construct a `Regex` object, use the `r` method of the `String` class:

```
val numPattern = "[0-9]+".r
```

If the regular expression contains backslashes or quotation marks, then it is a good idea to use the "raw" string syntax, `"""..."""`. For example:

```
val wsnumwsPattern = """\s+[0-9]+\s+""".r
  // A bit easier to read than "\\s+[0-9]+\\s+".r
```

The `findAllIn` method returns an iterator through all matches. You can use it in a `for` loop:

```
for (matchString <- numPattern.findAllIn("99 bottles, 98 bottles"))
  process matchString
```

or turn the iterator into an array:

```
val matches = numPattern.findAllIn("99 bottles, 98 bottles").toArray
  // Array(99, 98)
```

To find the first match anywhere in a string, use `findFirstIn`. You get an `Option[String]`. (See Chapter 14 for the `Option` class.)

```
val m1 =  wsnumwsPattern.findFirstIn("99 bottles, 98 bottles")
  // Some(" 98 ")
```

To check whether the beginning of a string matches, use `findPrefixOf`:

```
numPattern.findPrefixOf("99 bottles, 98 bottles")
  // Some(99)
wsnumwsPattern.findPrefixOf("99 bottles, 98 bottles")
  // None
```

You can replace the first match, or all matches:

```
numPattern.replaceFirstIn("99 bottles, 98 bottles", "XX")
  // "XX bottles, 98 bottles"
numPattern.replaceAllIn("99 bottles, 98 bottles", "XX")
  // "XX bottles, XX bottles"
```

9.11 Regular Expression Groups

Groups are useful to get subexpressions of regular expressions. Add parentheses around the subexpressions that you want to extract, for example:

```
val numitemPattern = "([0-9]+) ([a-z]+)".r
```

To match the groups, use the regular expression object as an "extractor" (see Chapter 14), like this:

```
val numitemPattern(num, item) = "99 bottles"
  // Sets num to "99", item to "bottles"
```

If you want to extract groups from multiple matches, use a `for` statement like this:

```
for (numitemPattern(num, item) <- numitemPattern.findAllIn("99 bottles, 98 bottles"))
  process num and item
```

Exercises

1. Write a Scala code snippet that reverses the lines in a file (making the last line the first one, and so on).
2. Write a Scala program that reads a file with tabs, replaces each tab with spaces so that tab stops are at *n*-column boundaries, and writes the result to the same file.
3. Write a Scala code snippet that reads a file and prints all words with more than 12 characters to the console. Extra credit if you can do this in a single line.

4. Write a Scala program that reads a text file containing only floating-point numbers. Print the sum, average, maximum, and minimum of the numbers in the file.
5. Write a Scala program that writes the powers of 2 and their reciprocals to a file, with the exponent ranging from 0 to 20. Line up the columns:

```
1                1
2                0.5
4                0.25
...              ...
```

6. Make a regular expression searching for quoted strings `"like this, maybe with \" or \\"` in a Java or C++ program. Write a Scala program that prints out all such strings in a source file.
7. Write a Scala program that reads a text file and prints all tokens in the file that are *not* floating-point numbers. Use a regular expression.
8. Write a Scala program that prints the `src` attributes of all `img` tags of a web page. Use regular expressions and groups.
9. Write a Scala program that counts how many files with `.class` extension are in a given directory and its subdirectories.
10. Expand the example with the serializable `Person` class that stores a collection of friends. Construct a few `Person` objects, make some of them friends of another, and then save an `Array[Person]` to a file. Read the array back in and verify that the friend relations are intact.