Table of Contents for Programming Scala, 2nd Edition

Chapter 2. Type Less, Do More

We ended the first chapter with a “teaser” example of an Akka actor application. This chapter continues our tour of Scala features, focusing on features that promote succinct, flexible code. We’ll discuss organization of files and packages, importing other types, variable and method declarations, a few particularly useful types, and miscellaneous syntax conventions.

Semicolons

Semicolons are expression delimiters and they are inferred. Scala treats the end of a line as the end of an expression, except when it can infer that the expression continues to the next line, as in this example:

```
// src/main/scala/progscala2/typelessdomore/semicolon-example.sc

// Trailing equals sign indicates more code on the next
line.
def equalsign(s: String) =
          "equalsign:
  println("             + s)

// Trailing opening curly brace indicates more code on the next
line.
def equalsign2(s: String) = {
          "equalsign2:
  println("              + s)
}

// Trailing commas, periods, and operators indicate more code on the next
line.
def commas(s1: String,
           s2: String) = Console.
          "comma:
  println("         + s1 +
          ",
          "    + s2)
```

Compared to the compiler, the REPL is more aggressive at interpreting the end of a line as the end of an expression, so when entering a multiline expression, it’s safest to end each line (except for the last) with one of the indicators shown in the previous script.

Conversely, you can put multiple expressions on the same line, separated by semicolons.

Tip

Use the REPL’s `:paste` mode when semicolon inference is too aggressive or multiple expressions need to be parsed as a whole, not individually. Enter `:paste`, followed by the code you want to enter, then finish with Ctrl-D.

Variable Declarations

Scala allows you to decide whether a variable is immutable (read-only) or not (read-write) when you declare it. We've already seen that an immutable "variable" is declared with the keyword `val` (think *value object*):

```
val array: Array[String] = new Array(5)
```

Scala is like Java in that most variables are actually references to heap-allocated objects. Hence, the `array` reference cannot be changed to point to a different `Array`, but the array elements themselves are mutable, so the elements can be modified:

```
scala> val array: Array[String] = new Array(5)
array: Array[String] = Array(null, null, null, null, null)

scala> array = new Array(2)
<console>:8: error: reassignment to val
       array = new Array(2)

scala> array(0) = "Hello"

scala> array
res1: Array[String] = Array(Hello, null, null, null, null)
```

A `val` must be initialized when it is declared.

Similarly, a mutable variable is declared with the keyword `var` and it must also be initialized immediately, even though it can be changed later, because it is mutable:

```
scala> var stockPrice: Double = 100.0
stockPrice: Double = 100.0

scala> stockPrice = 200.0
stockPrice: Double = 200.0
```

To be clear, we changed the *value* of `stockPrice` itself. However, the "object" that `stockPrice` refers to can't be changed, because `Doubles` in Scala are immutable.

In Java, so-called *primitive* types, `char`, `byte`, `short`, `int`, `long`, `float`, `double`, and `boolean`, are fundamentally different than reference objects. Indeed, there is no object and no reference, just the "raw" value. Scala tries to be consistently more object-oriented, so these types *are* actually objects with methods, like reference types (see Reference Versus Value Types). However, Scala compiles to primitives where possible, giving you the performance benefit they provide (we'll discuss this in depth in Specialization for Value Types).

There are a few exceptions to the rule that you must initialize `val`s and `var`s when they are declared. For example, either keyword can be used with a constructor parameter, turning it into a field of the type. It will be immutable if `val` is used and mutable if `var` is used.

Consider the following REPL session, where we define a `Person` class with immutable first and last names, but a mutable age (because people age, I guess):

```
// src/main/scala/progscala2/typelessdomore/person.sc
scala> class Person(val name: String, var age: Int)
defined class Person

                          "Dean
scala> val p = new Person(Wampler"      , 29)
p: Person = Person@165a128d

scala> p.name
res0: String = Dean Wampler     // Show the value of
firstName.

scala> p.age
res2: Int = 29                  // Show the value of age.

scala> p.name = "Buck Trends"
<console>:9: error: reassignment to val   // Disallowed!
       p.name = "Buck Trends"
              ^

scala> p.age = 30
p.age: Int = 30                 // Okay!
```

Note

The `var` and `val` keywords only specify whether the reference can be changed to refer to a different object (`var`) or not (`val`). They don't specify whether or not the object they reference is mutable.

Use immutable values whenever possible to eliminate a class of bugs caused by mutability.

For example, a mutable object is dangerous as a key in hash-based maps. If the object is mutated, the output of the `hashCode` method will change, so the corresponding value won't be found at the original location.

More common is unexpected behavior when an object you are using is being changed by someone else. Borrowing a phrase from Quantum Physics, these bugs are *spooky action at a distance*, because nothing you are doing locally accounts for the unexpected behavior; it's coming from somewhere else.

These are the most pernicious bugs in multithreaded programs, where synchronized access to shared, mutable state is required, but difficult to get right.

Using immutable values eliminates these issues.

Ranges

We're going to discuss method declarations next, but some of our examples will use the concept of a `Range`, so let's discuss that first.

Sometimes we need a sequence of numbers from some start to finish. A `Range` literal is just what we need. The following examples show how to create ranges for the types that support them, `Int`, `Long`, `Float`, `Double`, `Char`, `BigInt` (which represents integers of arbitrary size), and `BigDecimal` (which represents floating-point numbers of arbitrary size).

You can create ranges with an inclusive or exclusive upper bound, and you can specify an interval not equal to one:

```
scala> 1 to 10                // Int range inclusive, interval of 1, (1 to 10)
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> 1 until 10             // Int range exclusive, interval of 1, (1 to 9)
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> 1 to 10 by 3           // Int range inclusive, every third.
res2: scala.collection.immutable.Range = Range(1, 4, 7, 10)

scala> 10 to 1 by -3          // Int range inclusive, every third, counting
down.
res2: scala.collection.immutable.Range = Range(10, 7, 4, 1)

scala> 1L to 10L by 3         // Long
res3: scala.collection.immutable.NumericRange[Long] = NumericRange(1, 4, 7, 10)

scala> 1.1f to 10.3f by 3.1f  // Float with an interval != 1
res4: scala.collection.immutable.NumericRange[Float] =
  NumericRange(1.1, 4.2, 7.2999997)

scala> 1.1f to 10.3f by 0.5f  // Float with an interval < 1
res5: scala.collection.immutable.NumericRange[Float] =
  NumericRange(1.1, 1.6, 2.1, 2.6, 3.1, 3.6, 4.1, 4.6, 5.1, 5.6, 6.1, 6.6,
    7.1, 7.6, 8.1, 8.6, 9.1, 9.6, 10.1)

scala> 1.1 to 10.3 by 3.1     // Double
res6: scala.collection.immutable.NumericRange[Double] =
  NumericRange(1.1, 4.2, 7.300000000000001)

scala> 'a' to 'g' by 3         // Char
res7: scala.collection.immutable.NumericRange[Char] = NumericRange(a, d, g)

scala> BigInt(1) to BigInt(10) by 3
res8: scala.collection.immutable.NumericRange[BigInt] =
  NumericRange(1, 4, 7, 10)

scala> BigDecimal(1.1) to BigDecimal(10.3) by 3.1
res9: scala.collection.immutable.NumericRange.Inclusive[scala.math.BigDecimal]
  = NumericRange(1.1, 4.2, 7.3)
```

I wrapped some of the output lines to fit the page.

Partial Functions

Let's discuss the properties of `PartialFunction`. We learned that partial functions are *partial* in the sense that they aren't defined for all possible inputs, only those inputs that match at least one of the specified case clauses.

Only case clauses can be specified in a partial function and the entire function must be enclosed in curly braces. In contrast, "regular" function literals can be wrapped in parentheses or curly braces.

If the function is called with an input that doesn't match one of the `case` clauses, a `MatchError` is thrown at runtime.

You can test if a `PartialFunction` will match an input using the `isDefinedAt` method. This function avoids the risk of throwing a `MathError` exception.

You can "chain" `PartialFunctions` together: `pf1 orElse pf2 orElse pf3` … . If `pf1` doesn't match, then `pf2` is tried, then `pf3`, etc. A `MathError` is only thrown if none of them matches.

The following example illustrates these points:

```
// src/main/scala/progscala2/typelessdomore/partial-
functions.sc

val pf1: PartialFunction[Any,String] = { case s:String => "YES" }    //
❶
val pf2: PartialFunction[Any,String] = { case d:Double => "YES" }    //
❷

val pf = pf1 orElse pf2
// ❸

def tryPF(x: Any, f: PartialFunction[Any,String]): String =          //
❹
  try { f(x).toString } catch { case _: MatchError => "ERROR!" }

def d(x: Any, f: PartialFunction[Any,String]) =                      //
❺
  f.isDefinedAt(x).toString

        "      |   pf1 - String  |   pf2 - Double  |    pf -
println(All"                                                  )   // ❻
        "x     | def?  |  pf1(x) | def?  |  pf2(x) | def?  |
println(pf(x)"                                                   )
println("++++++++++++++++++++++++++++++++++++++++++++++++++++++++")
List("str", 3.14, 10) foreach { x =>
         "%-5s | %-5s | %-6s  | %-5s | %-6s  | %-5s | %-
  printf(6s\n"                                              , x.toString,
    d(x,pf1), tryPF(x,pf1), d(x,pf2), tryPF(x,pf2), d(x,pf), tryPF(x,pf))
}
```

❶

A partial function that only matches on strings.

❷

A partial function that only matches on doubles.

❸

Combine the two functions to construct a new partial function that matches on strings *and* doubles.

❹

A helper function to try the partial function and catch possible `MatchError` exceptions. With either success or failure, a string is returned.

❺

A helper function to try `isDefinedAt` and return a string result.

❻

Try the combinations and output a table of results!

The rest of the code tries different values with the three partial functions, first calling `isDefinedAt` (the `def?` column in the output) and then trying the function itself:

```
      |   pf1 - String  |   pf2 - Double  |    pf - All
x     | def?  |  pf1(x) | def?  |  pf2(x) | def?  |
pf(x)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++
str   | true  | YES     | false | ERROR!  | true  | YES
3.14  | false | ERROR!  | true  | YES     | true  | YES
10    | false | ERROR!  | false | ERROR!  | false |
ERROR!
```

Note that `pf1` fails unless a string is given, while `pf2` fails unless a double is given. Both fail if an integer is given. The combined function `pf` succeeds for both doubles and strings, but also fails for integers.

Method Declarations

Let's continue our examination of method definitions, using a modified version of our `Shapes` hierarchy from before. (We'll drop `Triangle` for simplicity.)

Method Default and Named Arguments

Here is an updated `Point` case class:

```
//
src/main/scala/progscala2/typelessdomore/shapes/Shapes.scala
package progscala2.typelessdomore.shapes

case class Point(x: Double = 0.0, y: Double = 0.0) {                  //
❶

  def shift(deltax: Double = 0.0, deltay: Double = 0.0) =             //
❷
    copy (x + deltax, y + deltay)
}
```

❶

Define `Point` with default initialization values (as before).

❷

A new `shift` method for creating a new `Point` offset from the existing `Point`. It uses the `copy` method that is also created automatically for case classes.

The `copy` method allows you to construct new instances of a case class while specifying just the fields that are changing. This is very useful for larger case classes:

```
scala> val p1 = new Point(x = 3.3, y = 4.4)     // Used named arguments
explicitly.
p1: Point = Point(3.3,4.4)

scala> val p2 = p1.copy(y = 6.6)  // Copied with a new y value.
p2: Point = Point(3.3,6.6)
```

Named arguments make client code more readable. They also help avoid bugs when a long argument list has several fields of the same type. It's easy to pass values in the wrong order. Of course, it's better to avoid long argument lists in the first place.

Methods with Multiple Argument Lists

Next, consider the changes to `Shape`, specifically the changes to the `draw` method:

```
abstract class Shape() {
  /**
   * Draw takes TWO argument LISTS, one list with an offset for
drawing,
   * and the other list that is the function argument we used
previously.

*/
  def draw(offset: Point = Point(0.0, 0.0))(f: String => Unit): Unit =
       "draw(offset = $offset),
    f(s${this.toString}"                          )
}

case class Circle(center: Point, radius: Double) extends Shape

case class Rectangle(lowerLeft: Point, height: Double, width: Double)
  extends Shape
```

Yes, `draw` now has *two* argument *lists*, each of which has a single argument, rather than a single argument list with two arguments. The first argument list lets you specify an offset point where the shape will be drawn. It has a default value of `Point(0.0, 0.0)`, meaning no offset. The second argument list is the same as in the original version of `draw`, a function that does the drawing.

You can have as many argument lists as you want, but it's rare to use more than two.

So, why allow more than one argument list? Multiple lists promote a very nice block-structure syntax when the last argument list takes a single function. Here's how we might invoke this new `draw` method:

```
s.draw(Point(1.0, 2.0))(str => println(s"ShapesDrawingActor: $str"))
```

Scala lets us replace parentheses with curly braces around an argument list. So, this line can also be written this way:

```
s.draw(Point(1.0, 2.0)){str => println(s"ShapesDrawingActor: $str")}
```

Suppose the function literal is too long for one line? We might rewrite it this way:

```
s.draw(Point(1.0, 2.0)) { str =>
  println(s"ShapesDrawingActor: $str")
}
```

Or equivalently:

```
s.draw(Point(1.0, 2.0)) {
  str => println(s"ShapesDrawingActor: $str")
}
```

It looks like a typical block of code we're used to writing with constructs like `if` and `for` expressions, method bodies, etc. However, the `{…}` block is still a function we're passing to `draw`.

So, this "syntactic sugar" of using `{…}` instead of `(…)` looks better with longer function literals; they look more like the block structure syntax we know and love.

If we use the default offset, the first set of parentheses is still required:

```
s.draw() {
  str => println(s"ShapesDrawingActor: $str")
}
```

To be clear, `draw` could just have a single argument list with two values, like Java methods. If so, the client code would look like this:

```
s.draw(Point(1.0, 2.0),
  str => println(s"ShapesDrawingActor: $str")
)
```

That's not nearly as clear and elegant. It's also less convenient to use the default value for the `offset`. We would have to name the function argument:

```
s.draw(f = str => println(s"ShapesDrawingActor: $str"))
```

A second advantage is type inference in the subsequent argument lists. Consider the following example:

```
scala> def m1[A](a: A, f: A => String) = f(a)
m1: [A](a: A, f: A => String)String

scala> def m2[A](a: A)(f: A => String) = f(a)
m2: [A](a: A)(f: A => String)String

                      "$i +
scala> m1(100, i => s$i"      )
<console>:12: error: missing parameter type
                             "$i +
              m1(100, i => s$i"      )
                       ^

                      "$i +
scala> m2(100)(i => s$i"      )
res0: String = 100 + 100
```

The functions `m1` and `m2` look almost identical, but notice what happens when we call each one with the same arguments, an `Int` and a function `Int => String`. For `m1`, Scala can't infer the type of the `i` argument for the function. For `m2`, it can.

A third advantage of allowing extra argument lists is that we can use the *last* argument list for *implicit* arguments. These are arguments declared with the `implicit` keyword. When the methods are called, we can either explicitly specify these arguments, or we can let the compiler fill them in using a suitable value that's in scope. Implicit arguments are a more flexible alternative to arguments with default values. Let's explore an example from the Scala library that uses implicit arguments, `Futures`.

A Taste of Futures

The `scala.concurrent.Future` API uses implicit arguments to reduce boilerplate in your code. They are another tool for concurrency provided by Scala. Akka uses `Future`s, but you can use them separately when you don't need the full capabilities of actors.

When you wrap some work in a `Future`, the work is executed asynchronously and the `Future` API provides various ways to process the results, such as providing callbacks that will be invoked when the result is ready. Let's use callbacks here and defer discussion of the rest of the API until Chapter 17.

The following example fires off five work items concurrently and handles the results as they finish:

```
// src/main/scala/progscala2/typelessdomore/futures.sc
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def sleep(millis: Long) = {
  Thread.sleep(millis)
}

// Busy work ;)
def doWork(index: Int) = {
  sleep((math.random * 1000).toLong)
  index
}

(1 to 5) foreach { index =>
  val future = Future {
    doWork(index)
  }
  future onSuccess {
                                  "Success! returned:
    case answer: Int => println(s$answer"
)
  }
  future onFailure {
    case th: Throwable => println(s"FAILURE! returned: $th")
  }
}

             // Wait long enough for the "work" to
sleep(1000)  finish.
println("Finito!")
```

After the imports, we use a `sleep` method to simulate staying busy for an arbitrary amount of time. The `doWork` method simply calls it with a randomly generated number of milliseconds between 0 and 1000.

The interesting bit comes next. We iterate through a `Range` of integers from 1 to 5, inclusive, with `foreach` and call `scala.concurrent.Future.apply`, the “factory” method on the singleton object `Future`. In this case `Future.apply` is passed an anonymous function of work to do. We use curly braces instead of parentheses to wrap the no-argument anonymous function we pass to it:

```
val future = Future {
  doWork(index)
}
```

`Future.apply` returns a new `Future` object, which executes the `doWork(index)` body on another thread; control then immediately returns to the loop. We then use `onSuccess` to register a callback to be invoked if the `future` completes successfully. This callback function is a `PartialFunction`.

Similarly, we use `onFailure` to register a callback for failure handling, where the failure will be encapsulated in a `Throwable`.

A final `sleep` call waits before exiting to allow the work to finish.

A sample run might go like this, where the results come out in arbitrary order, as you would expect:

```
$ scala src/main/scala/progscala2/typelessdomore/futures.sc
Success! returned: 1
Success! returned: 3
Success! returned: 4
Success! returned: 5
Success! returned: 2
Finito!
```

Okay, what does all this have to do with implicit arguments? A clue is the second import statement:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

I've been cavalier when I said that each future runs on its own thread. In fact, the Future API lets us configure how these concurrent operations are performed by specifying an `ExecutionContext`. That import brings in a default `ExecutionContext` that uses Java's `ForkJoinPool` facilities for managing pools of Java threads. Our example calls *three* methods that have a second argument list where an implicit `ExecutionContext` is expected. We aren't specifying those three argument lists explicitly, so the default `ExecutionContext` is used.

Here is the declaration for one of the methods, `apply`, in the `Future.apply` Scaladocs:

```
apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]
```

Note the `implicit` keyword in the second argument list.

Here are their declarations from the `Future.onSuccess` and `Future.onFailure` Scaladocs.

```
def onSuccess[U](func: (Try[T]) => U)(
  implicit executor: ExecutionContext): Unit
def onFailure[U](callback: PartialFunction[Throwable, U
])(
  implicit executor: ExecutionContext): Unit
```

The `Try` construct is a tool for handling `try {…} catch {…}` clauses, which we'll discuss later.

So how do you declare a value to be implicit? The import, `scala.concurrent.ExecutionContext.Implicits.global`, is the default `ExecutionContext` typically used with `Futures`. It is declared with the `implicit` keyword, so the compiler will use it when these methods are called without an `ExecutionContext` specified explicitly, as in our example. Only objects declared `implicit` *and* visible in the current scope will be considered for use, and only function arguments declared implicit are allowed to be unspecified and subject to this substitution.

Here is the actual declaration of `Implicits.global` in the Scala source code for `scala.concurrent.ExecutionContext`. It demonstrates using the `implicit` keyword to declare an `implicit` value (some details omitted):

```
object Implicits {
  implicit val global: ExecutionContextExecutor =
    impl.ExecutionContextImpl.fromExecutor(null: Executor
)
}
```

We'll create our own implicit values in other examples later on.

Nesting Method Definitions and Recursion

Method definitions can also be nested. This is useful when you want to refactor a lengthy method body into smaller methods, but the "helper" methods aren't needed outside the original method. Nesting them inside the original method means they are invisible to the rest of the code base, including other methods in the type.

Here is an implementation of a factorial calculator, where we call a second, nested method to do the work:

```
// src/main/scala/progscala2/typelessdomore/factorial.sc

def factorial(i: Int): Long = {
  def fact(i: Int, accumulator: Int): Long = {
    if (i <= 1) accumulator
    else fact(i - 1, i * accumulator)
  }

  fact(i, 1)
}

(0 to 5) foreach ( i => println(factorial(i)) )
```

This is the output it produces:

```
1
1
2
6
24
120
```

The second method calls itself recursively, passing an `accumulator` parameter, where the result of the calculation is "accumulated." Note that we return the accumulated value when the counter `i` reaches 1. (We're ignoring negative integer arguments, which would be invalid. The function just returns 1 for `i <= 1`.) After the definition of the nested method, `factorial` calls it with the passed-in value `i` and the initial accumulator "seed" value of 1.

Tip

It's easy to forget to call the nested function! If you get a compiler error that `Unit` is found, but `Long` is required, you probably forgot to call the function.

Did you notice that we use `i` as a parameter name twice, first in the `factorial` method and again in the nested `fact` method? The use of `i` as a parameter name for `fact` "shadows" the outer use of `i` as a parameter name for

`factorial`. This is fine, because we don't need the outer value of `i` inside `fact`. We only use it the first time we call `fact`, at the end of `factorial`.

Like a local variable declaration in a method, a nested method is also only visible inside the enclosing method.

Look at the return types for the two functions. We used `Long` because factorials grow in size quickly. So, we didn't want Scala to infer `Int` as the return type. Otherwise, we don't need the type annotation on `factorial`. However, we *must* declare the return type for `fact`, because it is recursive and Scala's local-scope type inference can't infer the return type of recursive functions.

You might be a little nervous about a recursive function. Aren't we at risk of blowing up the stack? The JVM and many other language environments don't do *tail-call optimizations*, which would convert a tail-recursive function into a loop. (The term *tail-call* means that the recursive call is the last thing done in the expression. The return value of the expression is the value returned by the call.)

Recursion is a hallmark of functional programming and a powerful tool for writing elegant implementations of many algorithms. Hence, the Scala compiler does limited tail-call optimizations itself. It will handle functions that call themselves, but not so-called "trampoline" calls, i.e., "a calls b calls a calls b," etc.

Still, you want to know if you got it right and the compiler did in fact perform the optimization. No one wants a blown stack in production. Fortunately, the compiler can tell you if you got it wrong, if you add an *annotation*, `tailrec`, as shown in this refined version of `factorial`:

```
// src/main/scala/progscala2/typelessdomore/factorial-tailrec.sc
import scala.annotation.tailrec

def factorial(i: Int): Long = {
  @tailrec
  def fact(i: Int, accumulator: Int): Long = {
    if (i <= 1) accumulator
    else fact(i - 1, i * accumulator)
  }

  fact(i, 1)
}

(0 to 5) foreach ( i => println(factorial(i)) )
```

If `fact` is not actually tail recursive, the compiler will throw an error. Consider this attempt to write a recursive Fibonacci function in the REPL:

```
scala> import scala.annotation.tailrec

scala> @tailrec
     | def fibonacci(i: Int): Long =
{
     |   if (i <= 1)
1L
     |   else fibonacci(i - 2) + fibonacci(i - 1
)
     |
}

<console>:16: error: could not optimize @tailrec annotated method fibonacci:
         it contains a recursive call not in tail position
         else fibonacci(i - 2) + fibonacci(i - 1)
                               ^
```

That is, we make not one, but two recursive calls and *then* do something with the returned values, so this function is not tail recursive.

Finally, the nested function can see anything in scope, including arguments passed to the outer function. Note the use of `n` in `count`:

```
// src/main/scala/progscala2/typelessdomore/count-
to.sc

def countTo(n: Int): Unit = {
  def count(i: Int): Unit = {
    if (i <= n) { println(i); count(i + 1) }
  }
  count(1)
}
```

Inferring Type Information

Statically typed languages can be very verbose. Consider this typical declaration in Java, before Java 7:

```
import java.util.HashMap;
...
HashMap<Integer, String> intToStringMap = new HashMap<Integer, String>();
```

We have to specify the type parameters `<Integer, String>` twice. Scala uses the term *type annotations* for explicit type declarations like `HashMap<Integer, String>`.

Java 7 introduced the *diamond operator* to infer the generic types on the righthand side, reducing the verbosity a bit:

```
HashMap<Integer, String> intToStringMap = new HashMap<>();
```

We've already seen some examples of Scala's support for type inference. The compiler can discern quite a bit of

type information from the context, without explicit type annotations. Here's the same declaration rewritten in Scala, with inferred type information:

```
val intToStringMap: HashMap[Integer, String] = new
HashMap
          HashMap[Integer,
If we specify String]                    on the righthand side of the equals sign, it's even more concise:

val intToStringMap2 = new HashMap[Integer, String
]
```

Some functional programming languages, like Haskell, can infer almost all types, because they can perform global type inference. Scala can't do this, in part because Scala has to support *subtype polymorphism* (inheritance), which makes type inference much harder.

Here is a summary of the rules for when explicit type annotations are required in Scala.

The last case is somewhat rare, fortunately.

Note

The `Any` type is the root of the Scala type hierarchy (we'll explore Scala's types in The Scala Type Hierarchy). If a block of code returns a value of type `Any` unexpectedly, chances are good that the code is more general than you intended so that only the `Any` type can fit all possible values.

Let's look at a few examples of cases we haven't seen yet where explicit types are required.

Overloaded methods require an explicit return type when one such method calls another, as in this example:

```
// src/main/scala/progscala2/typelessdomore/method-overloaded-return-v1.scX
// Version 1 of "StringUtil" (with a compilation
error).
// ERROR: Won't compile: needs a String return type on the second
"joiner".

object StringUtilV1 {
  def joiner(strings: String*): String = strings.mkString("-")

  def joiner(strings: List[String]) = joiner(strings :_*)
// COMPILATION
ERROR
}

println( StringUtilV1.joiner(List("Programming", "Scala")) )
```

In this contrived example, the two `joiner` methods concatenate a `List` of strings together, separated by `"-"`, but the code won't compile. They also illustrate a few useful idioms, but let's fix the compilation error first.

If you run this script, you get the following error:

```
...
<console>:10: error: overloaded method joiner needs result type
         def joiner(strings: List[String]) = joiner(strings :_*)
// ERROR
                                             ^
...
```

Because the *second* `joiner` method calls the first, it requires an explicit `String` return type. It should look like this:

```
def joiner(strings: List[String]): String = joiner(strings :_*)
```

Now let's return to the implementation idioms this script demonstrates. Scala supports methods that take *variable argument lists* (sometimes just called *variadic methods*). The first `joiner` method has this signature:

```
def joiner(strings: String*): String = strings.mkString("-")
```

The "*" after the `String` in the argument list says "zero or more Strings." A method can have other arguments, but they would have to come before the variable argument and only one variable argument is allowed.

However, sometimes a user might already have a list of strings ready to pass to `joiner`. The second method is a convenience method for this case. It simply calls the first `joiner` method, but it uses a special syntax to tell the compiler to convert the input `List` into the variable argument list expected by the first `joiner` method:

```
def joiner(strings: List[String]): String = joiner(strings :_*)
```

The way I read this curious expression, `strings :_*`, is to think of it as a hint to the compiler that you want the list `strings` to be treated as a variable argument list ("*") of some unspecified, but inferred type (":_"). Scala doesn't let you write `strings :String*`, even though that's the actual type we need for the other `joiner` method.

We don't actually need this convenience version of `joiner`. The user could also pass a list to the first `joiner` method, the one that takes a variable argument list, and use this same syntax to do the conversion.

The final scenario for return types can be subtle, especially when a more general return type is inferred than what you expected. You usually see this issue when you assign a value returned from a function to a variable with a more specific type. For example, you were expecting a `String`, but the function inferred an `Any` for the return type. Let's see a contrived example that reflects a bug where this scenario can occur:

```
// src/main/scala/progscala2/typelessdomore/method-broad-inference-return.scX
// ERROR: Won't compile. Method actually returns List[Any], which is too
"broad".

def makeList(strings: String*) = {
  if (strings.length == 0)
    List(0)  // #1
  else
    strings.toList
}

                                      // COMPILATION
val list: List[String] = makeList()   ERROR
```

Running this script triggers the following error:

```
...8: error: type mismatch;
 found   : List[Any]
 required: List[String]
val list: List[String] = makeList()   //
ERROR
                                  ^
```

We intended for `makeList` to return a `List[String]`, but when `strings.length` equals zero, we return `List(0)`, incorrectly "assuming" that this expression is the correct way to create an empty list. In fact, we are returning a `List[Int]` with one element, `0`.

Instead, we should return `List.empty[String]` or the special "marker" type for empty lists, `Nil`. Either choice results in the correct inference for the return type.

When the `if` clause returns `List[Int]` and the `else` clause returns `List[String]` (the result of `strings.toList`), the only possible inferred return type for the method is the closest common supertype of `List[Int]` and `List[String]`, which is `List[Any]`.

It's important to note that the code *compiled* without error. We only noticed the problem at runtime because we cautiously declared the value `list` to have the type `List[String]`.

In this case, adding an explicit return type of `List[String]` to the method would have triggered the compiler to catch the bug. Of course, a return type annotation also provides documentation for readers.

There are two other scenarios to watch for if you omit a return type annotation where you might get an unexpected inferred type. First, it could be that a function called by your function is triggering the unexpected inference. Perhaps that function was modified recently in a way that changed its return type, triggering a change in your function's inferred type on recompilation.

A second, related scenario is more often seen as a project grows and different modules are built at different times. If you see a `java.lang.NoSuchMethodError` during integration testing—or worse, production runs—for a method call that you *know* exists in another module, it could be that the return type of that function has been changed, either explicitly or through type inference, and the calling module is out of date and expecting the obsolete return type.

Note

When developing APIs that are built separately from their clients, declare method return types explicitly and use the most general return type you can. This is especially important when APIs declare *abstract* methods (see, e.g., Chapter 9).

Let's finish this section by looking at a common typing mistake that is easy to make, especially when you're new to Scala. Consider the following REPL session:

```
scala> def double(i: Int) { 2 * i }
double: (i: Int)Unit

scala> println(double(2))
()
```

Why did the second command print `()` instead of `4`? Look carefully at what the `scala` interpreter said about the `double` method signature: `(Int)Unit`. We thought we defined a method named `double` that takes an `Int` argument and returns a new `Int` with the value "doubled," but it actually returns `Unit`. Why?

The cause of this unexpected behavior is a missing equals sign in the method definition. Here is the definition we actually intended:

```
scala> def double(i: Int) = { 2 * i }
double: (i: Int)Int

scala> println(double(2))
4
```

Note the equals sign before the body of `double`. Now, the output says we have defined `double` to return an `Int` and the second command does what we expect it to do.

There is a reason for this behavior. Scala regards a method with the equals sign before the body as a function definition and a function always returns a value in functional programming. On the other hand, when Scala sees a method body without the leading equals sign, it assumes the programmer intended the method to be a "procedure" definition, meant for performing side effects only with the return value `Unit`. In practice, it is more likely that the programmer simply forgot to insert the equals sign!

Warning

When the return type of a method is inferred and you don't use an equals sign before the opening curly brace for the method body, Scala infers a `Unit` return type, even when the last expression in the method is a value of another type.

However, this behavior is too subtle and the mistake is easy to make. Because it is easy enough to define a function that returns `Unit`, the "procedure" syntax is now deprecated as of Scala 2.11. Don't use it!

I didn't tell you where the `()` came from that was printed before we fixed the bug. It is actually the real name of the *single* instance of the `Unit` type! We said before that `Unit` behaves like `void` in other languages. However, unlike `void`, `Unit` is actually a type with a single value, named `()`, which is a historical convention in functional languages—we'll explain why in Sum Types Versus Product Types.

Reserved Words

Table 2-1 lists the reserved keywords in Scala. Some of them we've seen already. Many are found in Java and they usually have the same meanings in both languages. For the keywords that we haven't encountered already, we'll learn about them as we proceed.

Table 2-1. Reserved words

Word	Description	See …
`abstract`	Makes a declaration abstract.	Class and Object Basics
`case`	Start a case clause in a match expression. Define a "case class."	Chapter 4
`catch`	Start a clause for catching thrown exceptions.	Using try, catch, and finally Clauses
`class`	Start a class declaration.	Class and Object Basics
`def`	Start a method declaration.	Method Declarations
`do`	Start a `do…while` loop.	Other Looping Constructs
`else`	Start an `else` clause for an `if` clause.	Scala if Statements
`extends`	Indicates that the class or trait that follows is the parent type of the class or trait being declared.	Parent Types
`false`	`Boolean` *false*.	Boolean Literals
`final`	Applied to a class or trait to prohibit deriving child types from it. Applied to a member to prohibit overriding it in a derived class or trait.	Attempting to Override final Declarations
`finally`	Start a clause that is executed after the corresponding `try` clause, whether or not an exception is thrown by the `try` clause.	Using try, catch, and finally Clauses
`for`	Start a `for` comprehension (loop).	Scala for Comprehensions
`forSome`	Used in *existential type* declarations to constrain the allowed concrete types that can be used.	Existential Types
`if`	Start an `if` clause.	Scala if Statements
`implicit`	Marks a method or value as eligible to be used as an *implicit* type converter or value. Marks a method parameter as optional, as long as a type-compatible substitute object is in the scope where the method is called.	Implicit Conversions
`import`	Import one or more types or members of types into the current scope.	Importing Types and Their Members

Word	Description	See …
`lazy`	Defer evaluation of a `val`.	lazy val
`match`	Start a pattern-matching clause.	Chapter 4
`new`	Create a new instance of a class.	Class and Object Basics
`null`	Value of a reference variable that has not been assigned a value.	The Scala Type Hierarchy
`object`	Start a *singleton* declaration: a `class` with only one instance.	A Taste of Scala
`override`	Override a *concrete* member of a type, as long as the original is not marked `final`.	Overriding Members of Classes and Traits
`package`	Start a package scope declaration.	Organizing Code in Files and Namespaces
`private`	Restrict visibility of a declaration.	Chapter 13
`protected`	Restrict visibility of a declaration.	Chapter 13
`requires`	Deprecated. Was used for *self-typing*.	Self-Type Annotations
`return`	Return from a function.	A Taste of Scala
`sealed`	Applied to a parent type. Requires all derived types to be declared in the same source file.	Sealed Class Hierarchies
`super`	Analogous to `this`, but binds to the parent type.	Overriding Abstract and Concrete Methods
`this`	How an object refers to itself. The method name for *auxiliary constructors*.	Class and Object Basics
`throw`	Throw an exception.	Using try, catch, and finally Clauses
`trait`	A *mixin module* that adds additional state and behavior to an instance of a class. Can also be used to just declare methods, but not define them, like a Java interface.	Chapter 9
`try`	Start a block that may throw an exception.	Using try, catch, and finally Clauses
`true`	`Boolean` *true*.	Boolean Literals

Word	Description	See …
`type`	Start a *type* declaration.	Abstract Types Versus Parameterized Types
`val`	Start a read-only “variable” declaration.	Variable Declarations
`var`	Start a read-write variable declaration.	Variable Declarations
`while`	Start a `while` loop.	Other Looping Constructs
`with`	Include the trait that follows in the class being declared or the object being instantiated.	Chapter 9
`yield`	Return an element in a `for` comprehension that becomes part of a sequence.	Yielding
`_`	A placeholder, used in imports, function literals, etc.	Many
`:`	Separator between identifiers and type annotations.	A Taste of Scala
`=`	Assignment.	A Taste of Scala
`=>`	Used in *function literals* to separate the argument list from the function body.	Anonymous Functions, Lambdas, and Closures
`<-`	Used in `for` comprehensions in *generator* expressions.	Scala for Comprehensions
`<:`	Used in *parameterized* and *abstract type* declarations to constrain the allowed types.	Type Bounds
`<%`	Used in *parameterized* and *abstract type* “view bounds” declarations.	View Bounds
`>:`	Used in *parameterized* and *abstract type* declarations to constrain the allowed types.	Type Bounds
`#`	Used in *type projections.*	Type Projections
`@`	Marks an *annotation.*	Annotations
⇒	(Unicode \u21D2) Same as =>.	Anonymous Functions, Lambdas, and Closures
→	(Unicode \u2192) Same as ->.	Implicit Conversions
←	(Unicode \u2190) Same as <-.	Scala for Comprehensions

Notice that `break` and `continue` are not listed. These control keywords don’t exist in Scala. Instead, Scala

encourages you to use functional programming idioms that are usually more succinct and less error-prone, as we'll see.

Some Java methods use names that are reserved words by Scala, for example, `java.util.Scanner.match`. To avoid a compilation error, surround the name with single back quotes ("back ticks"), e.g., `java.util.Scanner.`match`.

Literal Values

We've seen a few *literal values* already, such as `val book = "Programming Scala"`, where we initialized a `val book` with a `String` literal, and `(s: String) => s.toUpperCase`, an example of a function literal. Let's discuss all the literals supported by Scala.

Integer Literals

Integer literals can be expressed in decimal, hexadecimal, or octal. The details are summarized in Table 2-2.

Table 2-2. Integer literals

Kind	Format	Examples
Decimal	0 *or* a nonzero digit followed by zero or more digits (0–9)	0, 1, 321
Hexadecimal	0x followed by one or more hexadecimal digits (0–9, A–F, a–f)	0xFF, 0x1a3b
Octal	0 followed by one or more octal digits (0–7)	013, 077

You indicate a negative number by prefixing the literal with a `-` sign.

For `Long` literals, it is necessary to append the `L` or `l` character at the end of the literal, unless you are assigning the value to a variable declared to be `Long`. Otherwise, `Int` is inferred. The valid values for an integer literal are bounded by the type of the variable to which the value will be assigned. Table 2-3 defines the limits, which are inclusive.

Table 2-3. Ranges of allowed values for integer literals (boundaries are inclusive)

Target type	Minimum (inclusive)	Maximum (inclusive)
`Long`	-2^{63}	$2^{63} - 1$
`Int`	-2^{31}	$2^{31} - 1$
`Short`	-2^{15}	$2^{15} - 1$
`Char`	0	$2^{16} - 1$
`Byte`	-2^{7}	$2^{7} - 1$

A compile-time error occurs if an integer literal number is specified that is outside these ranges, as in the following examples:

```
scala> val i = 1234567890123
<console>:1: error: integer number too large
       val i = 12345678901234567890
               ^

scala> val i = 1234567890123L
i: Long = 1234567890123

scala> val b: Byte = 128
<console>:19: error: type mismatch;
 found   : Int(128)
 required: Byte
       val b: Byte = 128
                     ^

scala> val b: Byte = 127
b: Byte = 127
```

Floating-Point Literals

Floating-point literals are expressions with an optional minus sign, zero or more digits, followed by a period (`.`), followed by *one* or more digits. For `Float` literals, append the `F` or `f` character at the end of the literal. Otherwise, a `Double` is assumed. You can optionally append a `D` or `d` for a `Double`.

Floating-point literals can be expressed with or without exponentials. The format of the exponential part is `e` or `E`, followed by an optional `+` or `-`, followed by one or more digits.

Here are some example floating-point literals, where `Double` is inferred unless the declared variable is `Float` or an `f` or `F` suffix is used:

```
.14
3.14
3.14f
3.14F
3.14d
3.14D
3e5
3E5
3.14e+5
3.14e-5
3.14e-5
3.14e-5f
3.14e-5F
3.14e-5d
3.14e-5D
```

`Float` consists of all IEEE 754 32-bit, single-precision binary floating-point values. `Double` consists of all IEEE 754 64-bit, double-precision binary floating-point values.

Before Scala 2.10, floating-point literals without a digit after the period were allowed, e.g., `3.` and `3.e5`. This syntax leads to some ambiguities, where it can be interpreted as the period before a method name. How should `1.toString` be parsed? Is it `1` the `Int` or `1.0` the `Double`? Hence, literals without at least one digit after the period are deprecated in 2.10 and disallowed in 2.11.

Boolean Literals

The Boolean literals are `true` and `false`. The type of the variable to which they are assigned will be inferred to be `Boolean`:

```
scala> val b1 = true
b1: Boolean = true

scala> val b2 = false
b2: Boolean = false
```

Character Literals

A character literal is either a *printable* Unicode character or an escape sequence, written between single quotes. A character with a Unicode value between 0 and 255 may also be represented by an octal escape, i.e., a backslash (`\`) followed by a sequence of up to three octal characters. It is a compile-time error if a backslash character in a character or string literal does not start a valid escape sequence.

Here are some examples:

```
'A'
'\u0041'
// 'A' in
Unicode
'\n'
          // '\n' in
'\012'    octal
'\t'
```

The valid escape sequences are shown in Table 2-4.

Table 2-4. Character escape sequences

Sequence	Meaning
`\b`	Backspace (BS)
`\t`	Horizontal tab (HT)
`\n`	Line feed (LF)
`\f`	Form feed (FF)
`\r`	Carriage return (CR)
`\"`	Double quote (")
`\'`	Single quote (')
`\\`	Backslash (\)

Note that *nonprintable* Unicode characters like `\u0009` (tab) are not allowed. Use the equivalents like `\t`. Recall

that three Unicode characters were mentioned in Table 2-1 as valid replacements for corresponding ASCII sequences, ⇒ for `=>`, → for `->`, and ← for `<-`.

String Literals

A string literal is a sequence of characters enclosed in double quotes or *triples* of double quotes, i.e., `"""…"""`.

For string literals in double quotes, the allowed characters are the same as the character literals. However, if a double quote (`"`) character appears in the string, it must be "escaped" with a `\` character. Here are some examples:

```
"Programming\nScala"
"He exclaimed, \"Scala is
great!\""
"First\tSecond"
```

The string literals bounded by triples of double quotes are also called *multiline* string literals. These strings can cover several lines; the line feeds will be part of the string. They can include any characters, including one or two double quotes together, but not three together. They are useful for strings with `\` characters that don't form valid Unicode or escape sequences, like the valid sequences listed in Table 2-4. Regular expressions are a good example, which use lots of escaped characters with special meanings. Conversely, if escape sequences appear, they aren't interpreted.

Here are three example strings:

```
"""Programming\nScala"""
"""He exclaimed, "Scala is great!"
"""
"""First
line\n
Second
line\t

Fourth
line"""
```

Note that we had to add a space before the trailing `"""` in the second example to prevent a parse error. Trying to escape the second `"` that ends the `"Scala is great!"` quote, i.e., `"Scala is great!\"`, doesn't work.

When using multiline strings in code, you'll want to indent the substrings for proper code formatting, yet you probably don't want that extra whitespace in the actual string output. `String.stripMargin` solves this problem. It removes all whitespace in the substrings up to and including the first occurrence of a vertical bar, `|`. If you want some whitespace indentation, put the whitespace you want after the `|`. Consider this example:

```
// src/main/scala/progscala2/typelessdomore/multiline-strings.sc

def hello(name: String) = s"""Welcome!
  Hello,
$name!
  * (Gratuitous
Star!!)
  |We're glad you're
here.
  |  Have some extra
whitespace."""                    .stripMargin

      "Programming
hello(Scala"             )
```

It prints the following:

```
Welcome!
  Hello, Programming Scala!
  * (Gratuitous Star!!)
We're glad you're here.
  Have some extra
whitespace.
```

Note where leading whitespace is removed and where it isn't.

If you want to use a different leading character than `|`, use the overloaded version of `stripMargin` that takes a `Char` (character) argument. If the whole string has a prefix or suffix you want to remove (but not on individual lines), there are corresponding `stripPrefix` and `stripSuffix` methods:

```
// src/main/scala/progscala2/typelessdomore/multiline-strings2.sc

def goodbye(name: String) =
   """xxxGoodbye,
  s${name}yyy
  xxxCome
again!yyy"""          .stripPrefix("xxx").stripSuffix("yyy")

        "Programming
goodbye(Scala"             )
```

This example prints the following:

```
Goodbye, Programming
Scalayyy
  xxxCome again!
```

Symbol Literals

Scala supports symbols, which are *interned* strings, meaning that two symbols with the same "name" (i.e., the same

character sequence) will actually refer to the same object in memory. Symbols are used less often in Scala compared to some other languages, like Ruby and Lisp.

A symbol literal is a single quote (`'`), followed by one or more digits, letters, or underscores ("_"), except the first character can't be a digit. So, an expression like `'1symbol` is invalid.

A symbol literal `'id` is a shorthand for the expression `scala.Symbol("id")`. If you want to create a symbol that contains whitespace, use `Symbol.apply`, e.g., `Symbol(" Programming Scala ")`. All the whitespace is preserved.

Function Literals

We've seen function literals already, but to recap, `(i: Int, s: String) => s+i` is a function literal of type `Function2[Int,String,String]` (`String` is returned).

You can even use the literal syntax for a type declaration. The following declarations are equivalent:

```
val f1: (Int,String) => String       = (i, s) => s+
i
val f2: Function2[Int,String,String] = (i, s) => s+i
```

Tuple Literals

How many times have you wanted to return *two* or more values from a method? In many languages, like Java, you only have a few options, none of which is very appealing. You could pass in parameters to the method that will be modified for use as the "return" values, which is ugly. (Some languages even use keywords to indicate which parameters are input versus output.) You could declare some small "structural" class that holds the two or more values, then return an instance of that class.

The Scala library includes `TupleN` classes (e.g., `Tuple2`), for grouping *N* items, with the literal syntax of a comma-separated list of the items inside parentheses. There are separate `TupleN` classes for `N` between 1 and 22, inclusive (though this upper bound may be eliminated eventually in a future version of Scala).

For example, `val tup = ("Programming Scala", 2014)` defines a `Tuple2` instance with `String` inferred for the first element and `Int` inferred for the second element. Tuple instances are immutable, *first-class* values (because they are objects like any custom type you define), so you can assign them to variables, pass them as values, and return them from methods.

You can also use the literal syntax for `Tuple` type declarations:

```
val t1: (Int,String)       = (1, "two"
)
val t2: Tuple2[Int,String] = (1, "two")
```

The following example demonstrates working with tuples:

```
// src/main/scala/progscala2/typelessdomore/tuple-
example.sc

val t = ("Hello", 1, 2.3)
// ❶
println( "Print the whole tuple:
         "                          + t )
println( "Print the first item:
         "                          + t._1 )
// ❷
println( "Print the second item:
         "                          + t._2 )
println( "Print the third item:
         "                          + t._3 )

val (t1, t2, t3) = ("World", '!', 0x22)
// ❸
println( t1 + "    + t2 + "    + t3 )
              ",          ",

val (t4, t5, t6) = Tuple3("World", '!', 0x22)
// ❹
println( t4 + "    + t5 + "    + t6 )
              ",          ",
```

❶

Use the literal syntax to construct a three-element tuple of type `Tuple3`.

❷

Extract the first element of the tuple (counting from 1, not 0). The next two lines extract the second and third elements.

❸

On the fly declare three values, `t1`, `t2`, and `t3`, that are assigned the three corresponding fields from the tuple.

❹

Use the `Tuple3` "factory" method to construct a tuple.

Running this script produces the following output:

```
Print the whole tuple:
(Hello,1,2.3)
Print the first item:  Hello
Print the second item: 1
Print the third item:  2.3
World, !, 34
World, !, 34
```

The expression `t._n` retrieves the n^{th} item from tuple `t`, starting at one, *not* zero, following historical conventions.

There are several ways to define a two-element tuple, which is sometimes called a *pair* for short. In addition to the syntax that uses a parenthesized list of values, you can also use the "arrow operator" between two values, as well as special factory methods on the tuple-related classes:

```
(1, "one")
1 -> "one"
1 → "one"
// Using → character instead of -
>
Tuple2(1, "one")
```

The arrow operator is only available for two-element tuples.

Option, Some, and None: Avoiding nulls

Let's discuss three useful types that express a very useful concept, when we may or may not have a value.

Most languages have a special keyword or type instance that's assigned to reference variables when there's nothing else for them to refer to. In Java, it's `null`, which is a keyword, not an instance of a type. Thus, it's illegal to call any methods on it. But this is a confusing choice on the language designer's part. Why return a keyword when the programmer expects an instance of a type?

Of course, the real problem is that `null` is a giant source of nasty bugs. What `null` really signals is that we don't have a value in a given situation. If the value is not `null`, we do have a value. Why not express this situation explicitly with the type system and exploit type checking to avoid `NullPointerExceptions`?

`Option` lets us express this situation explicitly without the `null` "hack." `Option` is an abstract class and its two concrete subclasses are `Some`, for when we have a value, and `None`, when we don't.

You can see `Option`, `Some`, and `None` in action in the following example, where we create a map of state capitals in the United States:

```
// src/main/scala/progscala2/typelessdomore/state-capitals-
subset.sc

val stateCapitals = Map(
  "Alabama" -> "Montgomery",
  "Alaska"  -> "Juneau",
  //
  ...
  "Wyoming" -> "Cheyenne")

         "Get the capitals wrapped in
println( Options:"                              )
         "Alabama:
println( "           + stateCapitals.get("Alabama") )
         "Wyoming:
println( "           + stateCapitals.get("Wyoming") )
         "Unknown:
println( "           + stateCapitals.get("Unknown") )

         "Get the capitals themselves out of the
println( Options:"                                      )
         "Alabama:
println( "           + stateCapitals.get("Alabama").get )
         "Wyoming:
println( "           + stateCapitals.get("Wyoming").getOrElse("Oops!") )
         "Unknown:
println( "           + stateCapitals.get("Unknown").getOrElse("Oops2!") )
```

Notice what happens when we run the script:

```
Get the capitals wrapped in Options:
Alabama: Some(Montgomery)
Wyoming: Some(Cheyenne)
Unknown: None
Get the capitals themselves out of the
Options:
Alabama: Montgomery
Wyoming: Cheyenne
Unknown: Oops2!
```

The `Map.get` method returns an `Option[T]`, where `T` is `String` in this case. In contrast, Java's `Map.get` returns `T`, where `null` or a real value will be returned. By returning an `Option`, we can't "forget" that we have to verify that something was returned. In other words, the fact that a value may not exist for a given key is enshrined in the return type for the method declaration.

The first group of `println` statements invoke `toString` implicitly on the instances returned by `get`. We are calling `toString` on `Some` or `None` instances because the values returned by `Map.get` are automatically wrapped in a `Some`, when there is a value in the map for the specified key. Conversely, when we ask for a map entry that doesn't exist, the `None` object is returned, rather than `null`. This occurred in the last `println` of the three.

The second group of `println` statements goes a step further. After calling `Map.get`, they call `get` or `getOrElse` on each `Option` instance to retrieve the value it contains.

`Option.get` is a bit dangerous. If the `Option` is a `Some`, `Some.get` returns the value. However, if the `Option` is actually `None`, then `None.get` throws a `NoSuchElementException`.

We also see the alternative, safer method, `getOrElse`, in the last two `println` statements. This method returns either the value in the `Option`, if it is a `Some` instance, or it returns the argument passed to `getOrElse`, if it is a `None` instance. In other words, the `getOrElse` argument behaves as the default return value.

So, `getOrElse` is the more defensive of the two methods. It avoids a potential thrown exception.

To reiterate, because the `Map.get` method returns an `Option`, it automatically documents for the reader that there may not be an item matching the specified key. The map handles this situation by returning a `None`. Most languages would return `null` (or the equivalent). It's true that you learn from experience to expect a possible `null` in these languages, but using `Option` makes the behavior more explicit in the method signature and more self-documenting.

Also, thanks to Scala's static typing, you can't make the mistake of "forgetting" that an `Option` is returned and attempting to call a method supported by the type of the value inside the `Option` (if there is a value). In Java, when a method returns a value, it's easy to forget to check for `null` before calling a method on the value. When a Scala method returns `Option`, the type checking done by the compiler forces you to extract the value from the `Option` first before calling a method on it. That "reminds" you to check if the `Option` is actually `None`. So, the use of `Option` strongly encourages more resilient programming.

Because Scala runs on the JVM and it must interoperate with other libraries, Scala has to support `null`. Some devious soul could hand you a `Some(null)`, too. Still, you now have an alternative and you should avoid using `null` in your code, except for interoperating with Java libraries that require it. Tony Hoare, who invented the null reference in 1965 while working on a language called ALGOL W, called its invention his "billion dollar" mistake. Use `Option` instead.

Sealed Class Hierarchies

While we're discussing `Option`, let's discuss a useful design feature it uses. A key point about `Option` is that there are really only two valid subtypes. Either we have a value, the `Some` case, or we don't, the `None` case. There are *no* other subtypes of `Option` that would be valid. So, we would really like to prevent users from creating their own.

Scala has a keyword `sealed` for this purpose. `Option` is declared like this (eliding some details):

```
sealed abstract class Option[+A] ... { ... }
```

The `sealed` keyword tells the compiler that all subclasses must be declared *in the same source file*. `Some` and `None` are declared in the same file with `Option` in the Scala library. This technique effectively prevents additional subtypes of `Option`.

You can also declare a type `final` if you want to prevent users from subtyping it.

Organizing Code in Files and Namespaces

Scala adopts the `package` concept that Java uses for namespaces, but Scala offers more flexibility. Filenames don't have to match the type names and the package structure does not have to match the directory structure. So, you can define packages in files independent of their "physical" location.

The following example defines a class `MyClass` in a package `com.example.mypkg` using the conventional Java syntax:

```
// src/main/scala/progscala2/typelessdomore/package-
example1.scala
package com.example.mypkg

class MyClass {
  //
  ...
}
```

Scala also supports a block-structured syntax for declaring package scope, which is similar to the `namespace` syntax in C# and the use of `modules` as namespaces in Ruby:

```
// src/main/scala/progscala2/typelessdomore/package-
example2.scala
package com {
  package example {
    package pkg1 {
      class Class11 {
        def m = "m11"
      }
      class Class12 {
        def m = "m12"
      }
    }

    package pkg2 {
      class Class21 {
        def m = "m21"
        def makeClass11 = {
          new pkg1.Class11
        }
        def makeClass12 = {
          new pkg1.Class12
        }
      }
    }

    package pkg3.pkg31.pkg311 {
      class Class311 {
        def m = "m21"
      }
    }
  }
}
```

Two packages, `pkg1` and `pkg2`, are defined under the `com.example` package. A total of three classes are defined between the two packages. The `makeClass11` and `makeClass12` methods in `Class21` illustrate how to reference a type in the “sibling” package, `pkg1`. You can also reference these classes by their full paths, `com.example.pkg1.Class11` and `com.example.pkg1.Class12`, respectively.

The package `pkg3.pkg31.pkg311` shows that you can “chain” several packages together in one statement. It is not necessary to use a separate `package` statement for each package.

However, there is one situation where you might use separate statements. Let's call it the *successive package statement idiom*:

```
// src/main/scala/progscala2/typelessdomore/package-
example3.scala
// Bring into scope all package level declarations in
"example".
package com.example
// Bring into scope all package level declarations in
"mypkg".
package mypkg

class MyPkgClass {
  //
  ...
}
```

If you have package-level declarations, like types, in each of several parent packages that you want to bring into scope, use separate package statements as shown for each level of the package hierarchy with these declarations. Each subsequent package statement is interpreted as a subpackage of the previously specified package, as if we used the block-structure syntax shown previously. The first statement is interpreted as an absolute path.

Following the convention used by Java, the root package for Scala's library classes is named `scala`.

Although the package declaration syntax is flexible, one limitation is that packages cannot be defined within classes and objects, which wouldn't make much sense anyway.

Warning

Scala does not allow package declarations in scripts, which are implicitly wrapped in an `object`, where package declarations are not permitted.

Importing Types and Their Members

To use declarations in packages, you have to import them, just as you do in Java. However, Scala offers additional options, as shown in the following examples that import Java types:

```
import java.awt._
import java.io.File
import java.io.File._
import java.util.{Map, HashMap}
```

You can import all types in a package, using the underscore (`_`) as a wildcard, as shown on the first line. You can also import individual Scala or Java types, as shown on the second line.

Note

Java uses the "star" character (`*`) as the wildcard for imports. In Scala, this character is allowed as a method name, so `_` is used instead to avoid ambiguity. For example, if `Foo` `object` defined a `*` method, as well as other methods,

then what should `import Foo.*` mean?

The third line imports all the static methods and fields in `java.io.File`. The equivalent Java import statement would be `import static java.io.File.*;`. Scala doesn't have an `import static` construct because it treats `object` types uniformly like other types.

Selective importing has a very handy syntax, as shown on the fourth line, where we import just `java.util.Map` and `java.util.HashMap`.

Finally, you can put import statements almost anywhere, so you can scope their visibility to just where they are needed, you can rename types as you import them, and you can suppress the visibility of unwanted types:

```
def stuffWithBigInteger() = {

  import java.math.BigInteger.{
    ONE => _,
    TEN,
    ZERO => JAVAZERO }

  // println( "ONE: "+ONE )     // ONE is effectively
  undefined
          "TEN:
  println( "      +TEN )
          "ZERO:
  println( "       +JAVAZERO )
}
```

Because this import statement is inside `stuffWithBigInteger`, the imported types and values are not visible outside the function.

Renaming the `java.math.BigInteger.ONE` constant to underscore (`_`) makes it invisible and unavailable. Use this technique when you want to import everything except a few items.

Next, the `java.math.BigInteger.TEN` constant is imported without renaming, so it can be referenced simply as `TEN`.

Finally, the `java.math.BigInteger.ZERO` constant is given the "alias" `JAVAZERO`.

Aliasing is useful if you want to give an item a more convenient name or you want to avoid ambiguities with other items in scope that have the same name. It is used a lot to rename imported Java types so they don't conflict with Scala types of the same name, e.g., `java.util.List` and `java.util.Map`, for which we have Scala library types with the same names.

Imports Are Relative

Another important difference compared to Java imports is that Scala imports are *relative*. Note the comments for the following imports:

```
// src/main/scala/progscala2/typelessdomore/relative-imports.scala
import scala.collection.mutable._
import collection.immutable._
// Since "scala" is already
imported
                                          // full path from real
import _root_.scala.collection.parallel._  "root"
```

It's fairly rare that you'll have problems with relative imports, but sometimes surprises occur. If you get a mystifying compiler error that a package wasn't found, verify that you are using relative versus absolute import statements correctly. On rare occasions, you'll need the `_root_` prefix. Usually the top-level package, like `com`, `org`, or `scala` will suffice. Also, be sure that the library in question is visible on the `CLASSPATH`.

Package Objects

A design decision for library writers is where to expose public entry points for APIs that clients will import and use. As for Java libraries, it's common to import some or all of the types defined within a package. For example, the Java `import` statement `java.io.*;` imports all the types in the `io` package. Java 5 added "static imports" to allow individual static members of classes to be imported. Though convenient, it still presents a slightly inconvenient syntax. Consider a fictitious JSON parsing library you might use with a top-level package `json` and static API access methods in a class named `JSON`:

```
static import com.example.json.JSON.*;
```

In Scala, you can at least omit the `static` keyword, but wouldn't it be nice to write something like the following, which can expose with a single import statement all the types, methods, and values needed by the API user:

```
import com.example.json._
```

Scala supports *package objects*, a special kind of `object` scoped at the level of the package, `json` in this case. It is declared just like a normal `object`, but with with the differences illustrated by the following example:

```
// src/com/example/json/package.scala                    ❶

package com.example
// ❷

package object json {
// ❸
  class JSONObject {...}
// ❹
  def fromString(string: String): JSONObject = {...}
  ...
}
```

❶

The name of the file must be `package.scala`. By convention, it is located in the same package directory as

the package object it is defining, `com/example/json` in this case.

❷

The parent package scope.

❸

Use the `package` keyword and name the object after the package, `json` in this case.

❹

Appropriate members to expose to clients.

Now the client can import all these definitions with `import com.example.json._` or import individual elements in the usual way.

Abstract Types Versus Parameterized Types

We mentioned in A Taste of Scala that Scala supports *parameterized types*, which are very similar to *generics* in Java. (We could use the two terms *parameterized types* and *generics* interchangeably, but it's more common to use *parameterized types* in the Scala community and *generics* in the Java community.) Java uses angle brackets (`<…>`), while Scala uses square brackets (`[…]`), because `<` and `>` are often used for method names.

For example, here is a declaration of a list of strings:

```
val strings: List[String] = List("one", "two", "three")
```

Because we can plug in almost any type for a type parameter `A` in a collection like `List[A]`, this feature is called *parametric polymorphism*, because generic implementations of the `List` methods can be used with instances of any type `A`.

Let's discuss the most important details to learn about parameterized types, especially when you're trying to understand type signatures in Scaladocs, such as the entry for `List`, where you'll see that the declaration is written as `sealed abstract class List[+A]`.

The `+` in front of the `A` means that `List[B]` is a *subtype* of `List[A]` for any `B` that is a subtype of `A`. This is called *covariant typing*. It is a reasonably intuitive idea. If we have a function `f(list: List[Any])`, it makes sense that passing a `List[String]` to it should work fine.

If there is a `-` in front of a type parameter, the relationship goes the other way; `Foo[B]` would be a *supertype* of `Foo[A]`, if `B` is a *subtype* of `A` and the declaration is `Foo[-A]` (called *contravariant typing*). This is less intuitive, but we'll wait until Parameterized Types to explain it along with the other details of parameterized types.

Scala supports another type abstraction mechanism called *abstract types*, which can be applied to many of the same design problems for which parameterized types are used. However, while the two mechanisms overlap, they are not redundant. Each has strengths and weaknesses for certain design problems.

These types are declared as members of other types, just like methods and fields. Here is an example that uses an abstract type in a parent class, then makes the type member concrete in child classes:

```
// src/main/scala/progscala2/typelessdomore/abstract-types.sc
import java.io._

abstract class BulkReader {
  type In
  val source: In
                    // Read source and return a
  def read: String  String
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read: String = source
}

class FileBulkReader(val source: File) extends BulkReader {
  type In = File
  def read: String = {
    val in = new BufferedInputStream(new FileInputStream(source
))
    val numBytes = in.available()
    val bytes = new Array[Byte](numBytes)
    in.read(bytes, 0, numBytes)
    new String(bytes)
  }
}

                            "Hello
println(new StringBulkReader(Scala!"       ).read)
// Assumes the current directory is
src/main/scala:
println(new FileBulkReader(
  new File("TypeLessDoMore/abstract-types.sc")).read)
```

It produces the following output:

```
Hello Scala!
// src/main/scala/progscala2/typelessdomore/abstract-types.sc

import java.io._

abstract class BulkReader {
...
```

The `BulkReader` *abstract* class declares three abstract members: a `type` named `In`, a `val` field `source` of type `In`, and a `read` method.

The derived classes, `StringBulkReader` and `FileBulkReader`, provide concrete definitions for these abstract members.

Note that the `type` field works very much like a type parameter in a parameterized type. In fact, we could rewrite this example as follows, where we show only what would be different:

```
abstract class BulkReader[In] {
  val source: In
  ...
}

class StringBulkReader(val source: String) extends BulkReader[String] {...}

class FileBulkReader(val source: File) extends BulkReader[File] {...}
```

Just as for parameterized types, if we define the `In` type to be `String`, the `source` field must also be defined as a `String`. Note that the `StringBulkReader`'s `read` method simply returns the `source` field, while the `FileBulkReader`'s `read` method reads the contents of the file.

So what's the advantage here of using type members instead of parameterized types? The latter are best for the case where the type parameter has no relationship with the parameterized type, like `List[A]` when `A` is `Int`, `String`, `Person`, etc. A type member works best when it "evolves" in parallel with the enclosing type, as in our `BulkReader` example, where the type member needed to match the "behaviors" expressed by the enclosing type. Sometimes this characteristic is called *family polymorphism* or *covariant specialization*.

Recap and What's Next

We covered a lot of practical ground, such as literals, keywords, file organization, and imports. We learned how to declare variables, methods, and classes. We learned about `Option` as a better tool than `null`, plus other useful techniques. In the next chapter, we will finish our fast tour of the Scala "basics" before we dive into more detailed explanations of Scala's features.