Table of Contents for Programming Scala, 2nd Edition

Chapter 3. Rounding Out the Basics

Let's finish our survey of essential "basics" in Scala.

Operator Overloading?

Almost all "operators" are actually methods. Consider this most basic of examples:

```
1 + 2
```

That plus sign between the numbers? It's a method.

First, note that all the types that are special "primitives" in Java are actually regular objects in Scala, meaning they can have methods: `Float`, `Double`, `Int`, `Long`, `Short`, `Byte`, `Char`, and `Boolean`.

As we've seen, Scala identifiers can have nonalphanumeric characters, with a few exceptions that we'll go over in a moment.

So, `1 + 2` is the same as `1.+(2)`, because of the "infix" notation where we can drop the period and parentheses for single-argument methods.[]

Similarly, a method with no arguments can be invoked without the period. This is called "postfix" notation. However, use of this postfix convention can sometimes be confusing, so Scala 2.10 made it an *optional* feature. We set up the SBT build to trigger a warning if we use this feature without explicitly telling the compiler we want to use it. We do that with an import statement. Consider the following REPL session using the `scala` command (versus SBT `console`):

```
$ scala
...
scala> 1 toString
warning: there were 1 feature warning(s); re-run with -feature for details
res0: String = 1
```

Well, that's not helpful. Let's restart REPL with the `-feature` flag to produce a more informative warning:

```
$ scala -feature
...
scala> 1.toString  // normal invocation
res0: String = 1

scala> 1 toString  // postfix invocation
<console>:8: warning: postfix operator toString should be enabled
by making the implicit value scala.language.postfixOps visible.
This can ... adding the import clause 'import scala.language.postfixOps'
or by setting the compiler option -language:postfixOps.
See the Scala docs for value scala.language.postfixOps for a discussion
why the feature should be explicitly enabled.
              1 toString
                ^
res1: String = 1

scala> import scala.language.postfixOps
import scala.language.postfixOps

scala> 1 toString
res2: String = 1
```

Because I prefer to have this longer warning turned on all the time, I configured the SBT project to use the `-feature` flag. So running the REPL using the `console` task in SBT has this compler flag enabled already.

We can resolve the warning in one of two ways. We showed one way, which is to use `import scala.language.postfixOps`. We can also pass another flag to the compiler to enable the feature globally, `-language:postfixOps`. I tend to prefer case-by-case import statements, to remind the reader which optional features I'm using (we'll list all the optional features in scalac Command-Line Tool).

Dropping the punctuation, when it isn't confusing or ambiguous for the compiler, makes the code cleaner and can help create elegant programs that read more naturally.

So, what characters can you use in identifiers? Here is a summary of the rules for identifiers, used for method and type names, variables, etc:

Characters

Scala allows all the printable ASCII characters, such as letters, digits, the underscore (_), and the dollar sign (`$`), with the exceptions of the "parenthetical" characters, `(`, `)`, `[`, `]`, `{`, and `}`, and the "delimiter" characters, `` ` ``, `’`, `'`, `"`, `.`, `;`, and `,`. Scala allows the other characters between \u0020 and \u007F that are not in the sets just shown, such as mathematical symbols, the so-called *operator characters* like `/` and `<`, and some other symbols.

Reserved words can't be used

As in most languages, you can't reuse reserved words for identifiers. We listed the reserved words in Reserved Words. Recall that some of them are combinations of operator and punctuation characters. For example, a single underscore (_) is a reserved word!

Plain identifiers—combinations of letters, digits, `$`, `_`, and operators

A *plain identifier* can begin with a letter or underscore, followed by more letters, digits, underscores, and dollar signs. Unicode-equivalent characters are also allowed. Scala reserves the dollar sign for internal use, so you shouldn't use it in your own identifiers, although this isn't prevented by the compiler. After an underscore, you can have either letters and digits, *or* a sequence of operator characters. The underscore is important. It tells the compiler to treat all the characters up to the next whitespace as part of the identifier. For example, `val xyz_++= = 1` assigns the variable `xyz_++=` the value `1`, while the expression `val xyz++= = 1` won't compile because the "identifier" could also be interpreted as `xyz ++=`, which looks like an attempt to append something to `xyz`. Similarly, if you have operator characters after the underscore, you can't mix them with letters and digits. This restriction prevents ambiguous expressions like this: `abc_-123`. Is that an identifier `abc_-123` or an attempt to subtract `123` from `abc_`?

Plain identifiers—operators

If an identifier begins with an operator character, the rest of the characters must be operator characters.

"Back-quote" literals

An identifier can also be an arbitrary string between two back quote characters, e.g., `` def `test that addition works` = assert(1 + 1 == 2) ``. (Using this trick for literate test names is the one use of this otherwise-questionable technique you'll see occasionally in production code.) We also saw back quotes used previously to invoke a method or variable in a Java class when the name is identical to a Scala reserved word, e.g., `` java.net.Proxy.`type`() ``.

Pattern-matching identifiers

In pattern-matching expressions (recall the actor example in A Taste of Concurrency), tokens that begin with a lowercase letter are parsed as *variable identifiers*, while tokens that begin with an uppercase letter are parsed as *constant identifiers* (such as class names). This restriction prevents some ambiguities because of the very succinct variable syntax that is used, e.g., no `val` keyword is present.

Syntactic Sugar

Once you know that all operators are methods, it's easier to reason about unfamiliar Scala code. You don't have to worry about special cases when you see new operators. Our actors in A Taste of Concurrency sent asynchronous messages to each other with an exclamation point, `!`, which is actually just an ordinary method.

This flexible method naming gives you the power to write libraries that feel like a natural extension of Scala itself. You can write a new math library with numeric types that accept all the usual mathematical operators. You can write a new concurrent messaging layer that behaves just like actors. The possibilities are constrained by just a few limitations for method names.

Caution

Just because you *can* make up operator symbols doesn't mean you *should*. When designing your own APIs, keep in mind that obscure punctuational operators are hard for users to read, learn, and remember. Overuse of them contributes a "line noise" quality of unreadability to your code. So, stick to established conventions for operators and err on the side of readable method names when an operator shortcut isn't obvious.

Methods with Empty Argument Lists

Along with the infix and postfix invocation options, Scala is flexible about the use of parentheses in methods with no arguments.

If a method takes no parameters, you can define it without parentheses. Callers must invoke the method without parentheses. Conversely, if you add empty parentheses to your definition, callers have the option of adding parentheses or not.

For example, `List.size` has no parentheses, so you write `List(1, 2, 3).size`. If you try

`List(1, 2, 3).size()`, you'll get an error.

However, the `length` method for `java.lang.String` does have parentheses in its definition (because Java requires them), but Scala lets you write both `"hello".length()` and `"hello".length`. That's also true for Scala-defined methods with empty parentheses.

It's because of Java interoperability that the rules are not consistent for the two cases where empty parentheses are part of the definition or not. Scala would prefer that definition and usage remain consistent, but it's more flexible when the definition has empty parentheses, so that calling Java no-argument methods can be consistent with calling Scala no-argument methods.

A convention in the Scala community is to omit parentheses for no-argument methods that have no side effects, like the size of a collection. When the method performs side effects, parentheses are usually added, offering a small "caution signal" to the reader that mutation might occur, requiring extra care. If you use the option `-Xlint` when you invoke `scala` or `scalac`, it will issue a warning if you define a method with no parentheses that performs side effects (e.g., I/O). I've added that flag to our SBT build.

Why bother with optional parentheses in the first place? They make some method call chains read better as expressive, self-explanatory "sentences" of code:

```
// src/main/scala/progscala2/rounding/no-dot-better.sc

def isEven(n: Int) = (n % 2) == 0

List(1, 2, 3, 4) filter isEven foreach println
```

It prints the following output:

```
2
4
```

Now, if you're not accustomed to this syntax, it can take a while to understand what's going on, even though it is quite "clean." So here is the last line repeated four times with progressively less of the details filled in. The last line is the original:

```
List(1, 2, 3, 4).filter((i: Int) => isEven(i)).foreach((i: Int) => println(i))
List(1, 2, 3, 4).filter(i => isEven(i)).foreach(i => println(i))
List(1, 2, 3, 4).filter(isEven).foreach(println)
List(1, 2, 3, 4) filter isEven foreach println
```

The first three versions are more explicit and hence better for the beginning reader to understand. However, once you're familiar with the fact that `filter` is a method on collections that takes a single argument, `foreach` is an implicit loop over a collection, and so forth, the last, "Spartan" implementation is much faster to read and understand. The other two versions have more visual noise that just get in the way, once you're more experienced. Keep that in mind as you learn to read Scala code.

To be clear, this expression works because each method we used took a single argument. If you tried to use a method in the chain that takes zero or more than one argument, it would confuse the compiler. In those cases, put some or all of the punctuation back in.

Precedence Rules

So, if an expression like `2.0 * 4.0 / 3.0 * 5.0` is actually a series of method calls on `Double`s, what are the *operator precedence* rules? Here they are in order from lowest to highest precedence:

1. *All letters*
2. `|`
3. `^`
4. `&`
5. `< >`
   `=`
6. `!`
7. `:`
8. `+ -`
9. `* / %`
10. *All other special characters*

Characters on the same line have the same precedence. An exception is `=` when it's used for assignment, in which case it has the lowest precedence.

Because `*` and / have the same precedence, the two lines in the following `scala` session behave the same:

```
scala> 2.0 * 4.0 / 3.0 * 5.0
res0: Double = 13.333333333333332

scala> (((2.0 * 4.0) / 3.0) * 5.0)
res1: Double = 13.333333333333332
```

In a sequence of left-associative method invocations, they simply bind in left-to-right order. Aren't all methods "left-associative"? No. In Scala, any method with a name that ends with a colon (`:`) binds to the *right*, while all other methods bind to the *left*. For example, you can prepend an element to a `List` using the `::` method, called "cons," which is short for "constructor," a term introduced by Lisp:

```
scala> val list = List('b', 'c', 'd')
list: List[Char] = List(b, c, d)

scala> 'a' :: list
res4: List[Char] = List(a, b, c, d)
```

The second expression is equivalent to `list.::('a')`.

Tip

Any method whose name ends with a `:` binds to the *right*, not the *left*.

Domain-Specific Languages

Domain-specific languages, or DSLs, are languages written for a specific problem domain, with the goal of making it easy to express the concepts in that domain in a concise and intuitive manner. For example, SQL could be considered a DSL, because it is programming language to express an interpretation of the Relational Model.

However, the term DSL is usually reserved for ad hoc languages that are *embedded* in a host language or parsed with a custom parser. The term *embedded* means that you write code in the host language using idiomatic conventions that express the DSL. Embedded DSLs are also called *internal* DSLs, as distinct from *external* DSLs that require a custom parser.

Internal DSLs allow the developer to leverage the entirety of the host language for edge cases that the DSL does not cover (or from a negative point of view, the DSL can be a “leaky abstraction”). Internal DSLs also save the work of writing lexers, parsers, and the other tools for a custom language.

Scala provides excellent support for both kinds of DSLs. Scala’s flexible rules for identifiers, such as operator names, and the support for infix and postfix method calling syntax, provide building blocks for writing embedded DSLs using normal Scala syntax.

Consider this example of a style of test writing called *Behavior-Driven Development* using the ScalaTest library. The Specs2 library is similar.

```
// src/main/scala/progscala2/rounding/scalatest.scX
// Example fragment of a ScalaTest. Doesn't run
standalone.

import org.scalatest.{ FunSpec, ShouldMatchers }

class NerdFinderSpec extends FunSpec with ShouldMatchers {

  describe ("nerd finder") {
        "identify nerds from a
    it (List"                      ) {
                        "Rick            "James         "Woody
      val actors = List(Moranis"       , Dean"        , Allen"       )
      val finder = new NerdFinder(actors)
                                        "Rick            "Woody
      finder.findNerds shouldEqual List(Moranis"      , Allen"
)
    }
  }
}
```

This is just a taste of the power of Scala for writing DSLs. We’ll see more examples in Chapter 20 and learn how to write our own.

Scala if Statements

Superficially, Scala’s `if` statement looks like Java’s. The `if` conditional expression is evaluated. If it’s true, then the corresponding block is evaluated. Otherwise, the next branches are tested and so forth. A simple example:

```
// src/main/scala/progscala2/rounding/if.sc

if (2 + 2 == 5) {
          "Hello from
  println(1984."            )
} else if (2 + 2 == 3) {
            "Hello from Remedial Math
    println(class?"                          )
} else {
          "Hello from a non-Orwellian
  println(future."                            )
}
```

What's different in Scala is that `if` statements and almost all other statements in Scala are actually expressions that return values. So, we can assign the result of an `if` expression, as shown here:

```
// src/main/scala/progscala2/rounding/assigned-
if.sc

val configFile = new java.io.File("somefile.txt")

val configFilePath = if (configFile.exists()) {
  configFile.getAbsolutePath()
} else {
  configFile.createNewFile()
  configFile.getAbsolutePath()
}
```

The type of the value will be the so-called *least upper bound* of all the branches, the closest parent type that matches all the potential values from each clause. In this example, the value `configFilePath` is the result of an `if` expression that handles the case of a configuration file not existing internally, then returns the absolute path to that file. This value can now be reused throughout an application. Its type will be `String`.

Because `if` statements are expressions, the ternary conditional expression that exists in C-derived languages, e.g., `predicate ? trueHandler() : falseHandler()`, isn't supported, because it would be redundant.

Scala for Comprehensions

Another familiar control structure that's particularly feature-rich in Scala is the `for` loop, called the `for` *comprehension* or `for` *expression*.

Actually, the term *comprehension* comes from functional programming. It expresses the idea that we are traversing one or more collections of some kind, "comprehending" what we find, and computing something new from it, often another collection.

for Loops

Let's start with a basic `for` expression:

```
// src/main/scala/progscala2/rounding/basic-for.sc

                                 "Yorkshire
val dogBreeds = List("Doberman", Terrier"           , "Dachshund",
                     "Scottish          "Great
                     Terrier"         , Dane"       ,
"Portuguese Water
Dog"                     )

for (breed <- dogBreeds)
  println(breed)
```

As you might guess, this code says, “For every element in the list `dogBreeds`, create a temporary variable called `breed` with the value of that element, then print it.” The output is the following:

```
Doberman
Yorkshire Terrier
Dachshund
Scottish Terrier
Great Dane
Portuguese Water
Dog
```

Because this form doesn’t return anything, it only performs side effects. These kinds of `for` comprehensions are sometimes called `for` *loops*, analogous to Java `for` loops.

Generator Expressions

The expression `breed <- dogBreeds` is called a *generator expression*, so named because it’s *generating* individual values from a collection. The left arrow operator (`<-`) is used to iterate through a collection, such as a `List`.

We can also use it with a `Range` to write a more traditional-looking `for` loop:

```
//
src/main/scala/progscala2/rounding/generator.sc

for (i <- 1 to 10) println(i)
```

Guards: Filtering Values

What if we want to get more granular? We can add `if` expressions to filter for just elements we want to keep. These expressions are called *guards*. To find all terriers in our list of dog breeds, we modify the previous example to the following:

```
// src/main/scala/progscala2/rounding/guard-for.sc

                                  "Yorkshire
val dogBreeds = List("Doberman", Terrier"           , "Dachshund",
                     "Scottish           "Great
                     Terrier"          , Dane"       ,
"Portuguese Water
Dog"                     )
for (breed <- dogBreeds
  if breed.contains("Terrier")
) println(breed)
```

Now the output is this:

```
Yorkshire Terrier
Scottish Terrier
```

You can have more than one guard:

```
// src/main/scala/progscala2/rounding/double-guard-
for.sc

                                  "Yorkshire
val dogBreeds = List("Doberman", Terrier"           , "Dachshund",
                     "Scottish           "Great
                     Terrier"          , Dane"       ,
"Portuguese Water
Dog"                     )

for (breed <- dogBreeds
  if breed.contains("Terrier")
  if !breed.startsWith("Yorkshire")
) println(breed)

for (breed <- dogBreeds
  if breed.contains("Terrier") && !breed.startsWith("Yorkshire")
) println(breed)
```

The second `for` comprehension combines the two `if` statements into one. The combined output of both `for` comprehensions is this:

```
Scottish
Terrier
Scottish
Terrier
```

Yielding

What if, rather than printing your filtered collection, you needed to hand it off to another part of your program? The `yield` keyword is your ticket to generating new collections with `for` expressions.

Also, we're going to switch to curly braces, which can be used instead of parentheses, in much the same way that method argument lists can be wrapped in curly braces when a block-structure format is more visually appealing:

```
// src/main/scala/progscala2/rounding/yielding-
for.sc

                                 "Yorkshire
val dogBreeds = List("Doberman", Terrier"           , "Dachshund",
                     "Scottish           "Great
                     Terrier"          , Dane"       ,
"Portuguese Water
Dog"                     )
val filteredBreeds = for {
  breed <- dogBreeds
  if breed.contains("Terrier") && !breed.startsWith("Yorkshire")
} yield breed
```

Every time through the `for` expression, the filtered result is yielded as a value named `breed`. These results accumulate with every run, and the resulting collection is assigned to the value `filteredBreeds`. The type of the collection resulting from a `for-yield` expression is inferred from the type of the collection being iterated over. In this case, `filteredBreeds` is of type `List[String]`, because it is derived from the original `dogBreeds` list, which is of type `List[String]`.

Tip

An informal convention is to use parentheses when the `for` comprehension has a single expression and curly braces when multiple expressions are used. Note that older versions of Scala required semicolons between expressions when parentheses were used.

When a `for` comprehension doesn't use `yield`, but performs side effects like printing instead, the comprehension is called a `for` *loop*, because this behavior is more like the `for` loops you know from Java and other languages.

Expanded Scope and Value Definitions

Another useful feature of Scala's `for` comprehensions is the ability to define values inside the first part of your `for` expressions that can be used in the later expressions, as in this example:

```
// src/main/scala/progscala2/rounding/scoped-
for.sc

                                 "Yorkshire
val dogBreeds = List("Doberman", Terrier"           , "Dachshund",
                     "Scottish           "Great
                     Terrier"          , Dane"       ,
"Portuguese Water
Dog"                     )
for {
  breed <- dogBreeds
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)
```

Note that `upcasedBreed` is an immutable value, but the `val` keyword isn't required.[] The result is:

```
DOBERMAN
YORKSHIRE TERRIER
DACHSHUND
SCOTTISH TERRIER
GREAT DANE
PORTUGUESE WATER
DOG
```

If you recall `Option`, which we discussed as a better alternative to using `null`, it's useful to recognize that it is a special kind of collection, limited to zero or one elements. We can "comprehend" it too:

```
// src/main/scala/progscala2/patternmatching/scoped-option-
for.sc

                                                  "Yorkshire
val dogBreeds = List(Some("Doberman"), None, Some(Terrier"           ),
                                                   "Scottish
                     Some("Dachshund"), None, Some(Terrier"          ),
                                "Great               "Portuguese Water
                     None, Some(Dane"       ), Some(Dog"
))
        "first
println(pass:"      )
for {
  breedOption <- dogBreeds
  breed <- breedOption
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)

        "second
println(pass:"       )
for {
  Some(breed) <- dogBreeds
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)
```

Imagine that we called some services to return various breed names. The services returned `Options`, because some of the services couldn't return anything, so they returned `None`. In the first expression of the first `for` comprehension, each element extracted is an `Option` this time. The next line uses the arrow to extract the value in the option.

But wait! Doesn't `None` throw an exception if you try to extract an object from it? Yes, but the comprehension effectively checks for this case and skips the `None`s. It's as if we added an explicit `if breedOption != None` before the second line.

The second `for` comprehension makes this even cleaner, using *pattern matching*. The expression `Some(breed) <- dogBreeds` only succeeds when the `breedOption` is a `Some` and it extracts the `breed`, all in one step. `None` elements are not processed further.

When do you use the left arrow (`<-`) versus the equals sign (`=`)? You use the arrow when you're iterating through a collection or other "container," like an `Option`, and extracting values. You use the equals sign when you're assigning a value that doesn't involve iteration. A limitation is that the first expression in a `for` comprehension has to be an extraction/iteration using the arrow.

When working with loops in most languages, you can `break` out of a loop or `continue` the iterations. Scala doesn't have either of these statements, but when writing idiomatic Scala code, they are rarely necessary. Use conditional expressions to test if a loop should continue, or make use of recursion. Better yet, filter your collections ahead of time to eliminate complex conditions within your loops.[]

Note

Scala `for` comprehensions do not offer a `break` or `continue` feature. Other features make them unnecessary.

Other Looping Constructs

Scala provides several other looping constructs, although they are not widely used, because `for` comprehensions are so flexible and powerful. Still, sometimes a `while` loop is just what you need.

Scala while Loops

The `while` loop executes a block of code as long as a condition is true. For example, the following code prints out a complaint once a day until the next Friday the 13th has arrived:

```
// src/main/scala/progscala2/rounding/while.sc
// WARNING: This script runs for a LOOOONG
time!
import java.util.Calendar

def isFridayThirteen(cal: Calendar): Boolean = {
  val dayOfWeek = cal.get(Calendar.DAY_OF_WEEK)
  val dayOfMonth = cal.get(Calendar.DAY_OF_MONTH)

  // Scala returns the result of the last expression in a
  method
  (dayOfWeek == Calendar.FRIDAY) && (dayOfMonth == 13)
}

while (!isFridayThirteen(Calendar.getInstance())) {
          "Today isn't Friday the 13th.
  println(Lame."                             )
  // sleep for a
  day
  Thread.sleep(86400000)
}
```

Scala do-while Loops

Like the `while` loop, a `do-while` loop executes some code while a conditional expression is true. That is, the `do-while` checks to see if the condition is true *after* running the block. To count up to 10, we could write this:

```
// src/main/scala/progscala2/rounding/do-while.sc

var count = 0

do {
  count += 1
  println(count)
} while (count < 10)
```

Conditional Operators

Scala borrows most of the conditional operators from Java and its predecessors. You'll find the ones listed in Table 3-1 in `if` statements, `while` loops, and everywhere else conditions apply.

Table 3-1. Conditional operators

Operator	Operation	Description
`&&`	and	The values on the left and right of the operator are true. The righthand side is *only* evaluated if the lefthand side is *true*.
`||`	or	At least one of the values on the left or right is true. The righthand side is *only* evaluated if the lefthand side is *false*.
`>`	greater than	The value on the left is greater than the value on the right.
`>=`	greater than or equals	The value on the left is greater than or equal to the value on the right.
`<`	less than	The value on the left is less than the value on the right.
`<=`	less than or equals	The value on the left is less than or equal to the value on the right.
`==`	equals	The value on the left is the same as the value on the right.
`!=`	not equals	The value on the left is not the same as the value on the right.

Note that `&&` and `||` are "short-circuiting" operators. They stop evaluating expressions as soon as the answer is known.

Most of the operators behave as they do in Java and other languages. An exception is the behavior of `==` and its negation, `!=`. In Java, `==` compares object references only. It doesn't perform a logical equality check, i.e., comparing field values. You use the `equals` method for that purpose. So, if two *different* objects are of the same type and have the same field values (that is, the same state), `==` will still return false in Java.

In contrast, Scala uses `==` for logical equality, but it calls the `equals` method. A new method, `eq`, is available when you want to compare references, but not test for logical equality (we'll discuss the details of object equality in Equality of Objects).

Using try, catch, and finally Clauses

Through its use of functional constructs and strong typing, Scala encourages a coding style that lessens the need for exceptions and exception handling. But exceptions are still used, especially where Scala interacts with Java code, where use of exceptions is more prevalent.

Note

Unlike Java, Scala does not have checked exceptions, which are now regarded as an unsuccessful design. Java’s checked exceptions are treated as unchecked by Scala. There is also no `throws` clause on method declarations. However, there is a `@throws` annotation that is useful for Java interoperability. See the section Annotations.

Scala treats exception handling as just another pattern match, allowing us to implement concise handling of many different kinds of exceptions.

Let’s see this in action in a common application scenario, resource management. We want to open files and process them in some way. In this case, we’ll just count the lines. However, we must handle a few error scenarios. The file might not exist, especially since we’ll ask the user to specify the filenames. Also, something might go wrong while processing the file. (We’ll trigger an arbitrary failure to test what happens.) We need to ensure that we close all open file handles, whether or not the we process the files successfully:

```
//
src/main/scala/progscala2/rounding/TryCatch.scala
package progscala2.rounding

object TryCatch {
  /** Usage: scala rounding.TryCatch filename1 filename2 ...
  */
  def main(args: Array[String]) = {
    args foreach (arg => countLines(arg))
// ❶
  }

  import scala.io.Source
// ❷
  import scala.util.control.NonFatal

  def countLines(fileName: String) = {
// ❸
                // Add a blank line for
    println()   legibility
    var source: Option[Source] = None
// ❹
    try {
// ❺
      source = Some(Source.fromFile(fileName))                       //
❻
      val size = source.get.getLines.size
               "file $fileName has $size
      println(slines"                         )
    } catch {
                                    "Non fatal exception!
      case NonFatal(ex) => println(s$ex"                     )      //
❼
    } finally {
      for (s <- source) {
// ❽
                 "Closing
        println(s$fileName..."         )
        s.close
      }
    }
  }
}
```

❶

Use `foreach` to loop through the list of arguments and operate on each. `Unit` is returned by each pass and by `foreach` as the final result of the expression.

❷

Import `scala.io.Source` for reading input and `scala.util.control.NonFatal` for matching on “nonfatal” exceptions.

❸

For each filename, count the lines.

❹

Declare the `source` to be an `Option`, so we can tell in the `finally` clause if we have an actual instance or not.

❺

Start of `try` clause.

❻

`Source.fromFile` will throw a `java.io.FileNotFoundException` if the file doesn't exist. Otherwise, wrap the returned `Source` instance in a `Some`. Calling `get` on the next line is safe, because if we're here, we know we have a `Some`.

❼

Catch nonfatal errors. For example, out of memory would be fatal.

❽

Use a `for` comprehension to extract the `Source` instance from the `Some` and close it. If `source` is `None`, then nothing happens!

Note the `catch` clause. Scala uses pattern matches to pick the exceptions you want to catch. This is more compact and more flexible than Java's use of separate `catch` clauses for each exception. In this case, the clause `case NonFatal(ex) => …` uses `scala.util.control.NonFatal` to match any exception that isn't considered fatal.

The `finally` clause is used to ensure proper resource cleanup in one place. Without it, we would have to repeat the logic at the end of the `try` clause and the `catch` clause, to ensure our file handles are closed. Here we use a `for` comprehension to extract the `Source` from the option. If the option is actually a `None`, nothing happens; the block with the `close` call is not invoked.

Tip

This is a widely used idiom; use a `for` comprehension when you need to test whether an `Option` is a `Some`, in which case you do some work, or is a `None`, in which case you ignore it.

This program is already compiled by `sbt` and we can run it from the `sbt` prompt using the `run-main` task, which lets us pass arguments. I've wrapped some lines to fit the page, using a `\` to indicate line continuations, and elided some text:

```
> run-main progscala2.rounding.TryCatch foo/bar \
  src/main/scala/progscala2/rounding/TryCatch.
scala
[info] Running rounding.TryCatch foo/bar .../rounding/TryCatch.scala

... java.io.FileNotFoundException: foo/bar (No such file or directory)

file src/main/scala/progscala2/rounding/TryCatch.scala has 30 lines
Closing src/main/scala/progscala2/rounding/TryCatch.scala...
[success] ...
```

The first file doesn't exist. The second file is the source for the program itself. The `scala.io.Source` API is a convenient way to process a stream of data from a file and other sources. Like many such APIs, it throws an exception if the file doesn't exist. So, the exception for `foo/bar` is expected.

Tip

When resources need to be cleaned up, whether or not the resource is used successfully, put the cleanup logic in a `finally` clause.

Pattern matching aside, Scala's treatment of exception handling is similar to the approaches used in most popular languages. You throw an exception by writing `throw new MyBadException(…)`, as in Java. If your custom exception is a `case` class, you can omit the `new`. That's all there is to it.

Automatic resource management is a common pattern. There is a separate Scala project by Joshua Suereth called *ScalaARM* for this purpose. Let's try writing our own.

Call by Name, Call by Value

Here is our implementation of a reusable *application resource manager*:

```
//
src/main/scala/progscala2/rounding/TryCatchArm.scala
package progscala2.rounding
import scala.language.reflectiveCalls
import scala.util.control.NonFatal

object manage {
  def apply[R <: { def close():Unit }, T](resource: => R)(f: R => T) =
{
    var res: Option[R] = None
    try {
                                   // Only reference "resource"
      res = Some(resource)         once!!
      f(res.get)
    } catch {
                                    "Non fatal exception!
      case NonFatal(ex) => println(s$ex"                      )
    } finally {
      if (res != None) {
                 "Closing
        println(sresource..."         )
        res.get.close
      }
    }
  }
}

object TryCatchARM {
  /** Usage: scala rounding.TryCatch filename1 filename2 ...
  */
  def main(args: Array[String]) = {
    args foreach (arg => countLines(arg))
  }

  import scala.io.Source

  def countLines(fileName: String) = {
               // Add a blank line for
    println()  legibility
    manage(Source.fromFile(fileName)) { source =>
      val size = source.getLines.size
               "file $fileName has $size
      println(slines"                         )
                                                "Big
      if (size > 20) throw new RuntimeException(file!"     )
    }
  }
}
```

You can run it like the previous example, substituting `TryCatchARM` for `TryCatch`. The output will be similar.

This is a lovely little bit of *separation of concerns*, but to implement it, we used a few new power tools.

First, we named our object `manage` rather than `Manage`. Normally, you follow the convention of using a leading uppercase letter for type names, but in this case we will use manage like a function. We want client code to look like we're using a built-in operator, similar to a while loop. See how it's used in countLines. This is another example of Scala's tools for building little *domain-specific languages* (DSLs).

That manage.apply method declaration is hairy looking. Let's deconstruct it. Here is the signature again, spread over several lines and annotated:

```
def apply[
  R <: { def close():Unit },
❶
  T ]
❷
  (resource: => R)
❸
  (f: R => T) = {...}
❹
```

❶

Two new things are shown here. R is the type of the resource we'll manage. The <: means R is a subclass of something else, in this case a *structural type* with a close():Unit method. What would be more intuitive, especially if you haven't seen structural types before, would be for all resources to implement a Closable interface that defines a close():Unit method. Then we could say R <: Closable. Instead, structural types let us use reflection and plug in any type that has a close():Unit method (like Source). Reflection has a lot of overhead and structural types are a bit scary, so reflection is another *optional feature*, like postfix expressions, which we saw earlier. So we add the import statement to tell the compiler we know what we're doing.

❷

T will be the type returned by the anonymous function passed in to do work with the resource.

❸

It looks like resource is a function with an unusual declaration. Actually, resource is a *by-name* parameter. For the moment, think of this as a function that we call without parentheses.

❹

Finally we pass a second argument list containing the work to do with the resource, an anonymous function that will take the resource as an argument and return a result of type T.

Recapping point 1, here is how the apply method declaration would look if we could assume that all resources implement a Closable abstraction:

```
object manage {
  def apply[R <: Closable, T](resource: => R)(f: R => T) =
{...}
  ...
}
```

The line, val res = Some(resource), is the *only* place resource is evaluated, which is essential. Because `resource` behaves like a function, it is evaluated every time it's referenced, just like a function would be evaluated repeatedly. We don't want to evaluate `Source.fromFile(fileName)` every time we reference `resource`, because we would reopen new `Source` instances each time!

Next, this `res` value is passed to the work function `f`.

See how `manage` is used in `TryCatchARM.countLines`. It looks like a built-in control structure with one argument list that creates the `Source` and a second argument list that is a block of code that works with the `Source`. So, `manage` looks something like a conventional `while` statement, for example.

To recap a bit, the first expression that creates the `Source` is not evaluated immediately, *before* execution moves to `manage`. The evaluation of the expression is delayed until the line `val res = Some(resource)` within `manage`. This is what the *by-name* parameter `resource` gives us. We can write a function `manage.apply` that accepts as a parameter an arbitrary expression, but we defer evaluation until later.

Scala, like most languages, normally uses *call-by-value* semantics. If we write `manage(Source.fromFile(fileName))` in a call-by-value context, the `Source.fromFile` method is called and the value it returns is passed to `manage`.

By deferring evaluation until the line `val res = Some(resource)` within `apply`, this line is effectively the following:

```
val res = Some(Source.fromFile(fileName))
```

Supporting idioms like this is the reason that Scala offers *by-name* parameters.

What if we didn't have *by-name* parameters? We could use an anonymous function, but it would be a bit uglier.

The call to `manage` would now look like this:

```
manage(() => Source.fromFile(fileName)) { source =>
```

Within `apply`, our reference to `resource` would now be an "obvious" function call:

```
val res = Some(resource())
```

Okay, that's not a terrible burden, but *call by name* enables a syntax for building our own control structures, like our `manage` utility.

Remember that by-name parameters behave like functions; the expression is evaluated every time it is used. In our ARM example, we only wanted to evaluate it *once*, but that's not the general case.

Here is a simple implementation of a while-like loop construct, called `continue`:

```
// src/main/scala/progscala2/rounding/call-by-name.sc

@annotation.tailrec
// ❶
def continue(conditional: => Boolean)(body: => Unit) {                //
❷
  if (conditional) {
// ❸
    body
// ❹
    continue(conditional)(body)
  }
}

var count = 0
// ❺
continue(count < 5) {
           "at
  println(s$count"    )
  count += 1
}
```

❶

Ensure the implementation is tail recursive.

❷

Define a `continue` function that accepts two argument lists. The first list takes a single, by-name parameter that is the conditional. The second list takes a single, by-name value that is the body to be evaluated for each iteration.

❸

Test the condition.

❹

If still true, evaluate the body, then call the `continue` recursively.

❺

Try it!

It's important to note that the by-name parameters are evaluated every time they are referenced. (By the way, this implementation shows how "loop" constructs can be replaced with recursion). So, by-name parameters are in a sense *lazy*, because evaluation is deferred, but possibly repeated over and over again. Scala also provides lazy values.

lazy val

A related scenario to *by-name* parameters is the case where you want to evaluate an expression *once* to initialize a value, not repeatedly, but you want to defer that invocation. There are some common scenarios where this is useful:

- The expression is expensive (e.g., opening a database connection) and we want to avoid the overhead until the value is actually needed.

- Improve startup times for modules by deferring work that isn't needed immediately.
- Sometimes a field in an object needs to be initialized lazily so that other initializations can happen first. We'll explore these scenarios when we discuss Scala's object model in Overriding fields in traits.

Here is an example using a `lazy val`:

```
// src/main/scala/progscala2/rounding/lazy-init-val.sc

object ExpensiveResource {
  lazy val resource: Int = init()
  def init(): Int = {
    // do something
    expensive
    0
  }
}
```

The `lazy` keyword indicates that evaluation should be deferred until the value is needed.

So, how is a `lazy val` different from a method call? In a method call, the body is executed *every* time the method is invoked. For a `lazy val`, the initialization "body" is evaluated only once, when the value is used for the first time. This one-time evaluation makes little sense for a mutable field. Therefore, the `lazy` keyword is not allowed on `var`s.

So, why not mark object field values `lazy` all the time, so that creating objects is always faster? Actually, it may not be faster except for truly expensive operations.

Lazy values are implemented with a *guard*. When client code references a lazy value, the reference is intercepted by the guard to check if initialization is required. This guard step is really only essential the *first* time the value is referenced, so that the value is initialized first before the access is allowed to proceed. Unfortunately, there is no easy way to eliminate these checks for subsequent calls. So, lazy values incur overhead that "eager" values don't. Therefore, you should only use them when the guard overhead is outweighed by the expense of initialization or in certain circumstances where careful ordering of initialization dependencies is most easily implemented by making some values lazy (see Overriding fields in traits).

Enumerations

Remember our examples involving various dog breeds? In thinking about the types in these programs, we might want a top-level `Breed` type that keeps track of a number of breeds. Such a type is called an *enumerated type*, and the values it contains are called *enumerations*.

While enumerations are a built-in part of many programming languages, Scala takes a different route and implements them as an `Enumeration` class in its standard library. This means there is no special syntax for enumerations baked into Scala's grammar, as there is for Java. Instead, you just define an object that extends the `Enumeration` class and follow its idioms. So, at the byte code level, there is no connection between Scala enumerations and the `enum` constructs in Java.

Here is an example:

```
//
src/main/scala/progscala2/rounding/enumeration.sc

object Breed extends Enumeration {
  type Breed = Value
  val doberman = Value("Doberman Pinscher")
                      "Yorkshire
  val yorkie   = Value(Terrier"           )
                      "Scottish
  val scottie  = Value(Terrier"          )
                      "Great
  val dane     = Value(Dane"       )
                      "Portuguese Water
  val portie   = Value(Dog"                  )
}
import Breed._

// print a list of breeds and their
IDs
println("ID\tBreed")
for (breed <- Breed.values) println(s"${breed.id}\t$breed")

// print a list of Terrier
breeds
println("\nJust Terriers:")
Breed.values filter (_.toString.endsWith("Terrier")) foreach println

def isTerrier(b: Breed) = b.toString.endsWith("Terrier")

        "\nTerriers
println(Again??"            )
Breed.values filter isTerrier foreach println
```

It prints the following:

```
ID      Breed
0       Doberman Pinscher
1       Yorkshire Terrier
2       Scottish Terrier
3       Great Dane
4       Portuguese Water
Dog

Just Terriers:
Yorkshire Terrier
Scottish Terrier

Terriers Again??
Yorkshire Terrier
Scottish Terrier
```

We can see that our `Breed` enumerated type contains several values of type `Value`, as in the following example:

```
val doberman = Value("Doberman Pinscher")
```

Each declaration is actually calling a method named `Value` that takes a string argument. We use this method to assign a long-form breed name to each enumeration value, which is what the `Value.toString` method returned in the output.

The type `Breed` is an alias that lets us reference `Breed` instead of `Value`. The only place we actually use this is the argument to the `isTerrier` method. If you comment out the `type` definition, this function won't compile.

There is no namespace collision between the type and method that both have the name `Value`. The compiler maintains separate namespaces for values and methods.

There are other overloaded versions of the `Value` method. We're using the one that takes a `String`. Another one takes no arguments, so the string will just be the name of the value, e.g., "doberman." Another `Value` method takes an `Int` ID value, so the default string is used and the integer `id` is a value we assign explicitly. Finally, the last `Value` method takes both an `Int` and `String`.

Because we're not calling one of the methods that takes an explicit ID value, the values are incremented and assigned automatically starting at 0, in declaration order. These `Value` methods return a `Value` object, and they add the value to the enumeration's collection of values.

To work with the values as a collection, we call the `values` method. So, we can easily iterate through the breeds with a `for` comprehension and `filter` them by name or see the `ids` that are automatically assigned if we don't explicitly specify a number.

You'll often want to give your enumeration values human-readable names, as we did here. However, sometimes you may not need them. Here's another enumeration example adapted from the Scaladoc entry for `Enumeration`:

```
// src/main/scala/progscala2/rounding/days-enumeration.sc

object WeekDay extends Enumeration {
  type WeekDay = Value
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}
import WeekDay._

def isWorkingDay(d: WeekDay) = ! (d == Sat || d == Sun)

WeekDay.values filter isWorkingDay foreach println
```

Running this script yields the following output (v2.7):

```
Mon
Tue
Wed
Thu
Fri
```

Note that we imported `WeekDay._`. This made each enumeration value (`Mon`, `Tues`, etc.) in scope. Otherwise, you would have to write `WeekDay.Mon`, `WeekDay.Tues`, etc. We can iterate through the values by calling the `values` method. In this case, we filter the values for "working days" (weekdays).

You don’t actually see enumerations used a lot in Scala code, especially compared to Java code. They are lightweight, but they are also limited to the case where you know in advance the exact set of values to define. Clients can’t add more values.

Instead, case classes are often used when an “enumeration of values” is needed. They are a bit more heavyweight, but they have two advantages. First, they offer greater flexibility to add methods and fields, and to use pattern matching on them. The second advantage is that they aren’t limited to a fixed set of known values. Client code can add more case classes to a base set that your library defines, when useful.

Interpolated Strings

We introduced *interpolated* strings in A Taste of Concurrency. Let’s explore them further.

A `String` of the form `s"foo ${bar}"` will have the value of expression `bar`, converted to a `String` and inserted in place of `${bar}`. If the expression `bar` returns an instance of a type other than `String`, a `toString` method will be invoked, if one exists. It is an error if it can’t be converted to a `String`.

If `bar` is just a variable reference, the curly braces can be omitted. For example:

```
val name = "Buck Trends"
         "Hello,
println(s$name"        )
```

When using interpolated strings, to write a literal dollar sign `$`, use two of them, `$$`.

There are two other kinds of interpolated strings. The first kind provides *printf* formatting and uses the prefix `f`. The second kind is called “raw” interpolated strings. It doesn’t expand escape characters, like `\n`.

Suppose we’re generating financial reports and we want to show floating-point numbers to two decimal places.[]
Here’s an example:

```
val gross   = 100000F
val net     = 64000F
val percent = (net / gross) * 100
         "$$${gross}%.2f vs. $$${net}%.2f or
println(f${percent}%.1f%%"                                    )
```

The output of the last line is the following:

```
$100000.00 vs. $64000.00 or 64.0%
```

Scala uses Java’s `Formatter` class for `printf` formatting. The embedded references to expressions use the same `${…}` syntax as before, but `printf` formatting directives trail them with no spaces.

In this example, we use two dollar signs, $$, to print a literal US dollar sign, and two percent signs, `%%`, to print a literal percent sign. The expression `${gross}%.2f` formats the value of `gross` as a floating-point number with two digits after the decimal point.

The types of the variables used must match the format expressions, but some implicit conversions are performed. The following attempt to use an `Int` expression in a `Float` context is allowed. It just pads with zeros. However, the second expression, which attempts to render a `Double` as an `Int`, causes a compilation error:

```
scala> val i = 200
i: Int = 200

scala> f"${i}%.2f"
res4: String = 200.00

scala> val d = 100.22
d: Double = 100.22

scala> f"${d}%2d"
<console>:9: error: type mismatch;
 found   : Double
 required: Int
              f"${d}%2d"
                  ^
```

As an aside, you can still format strings using `printf`-style formatting with Java’s static method `String.format`. It takes as arguments a format string and a variable argument list of values to substitute into the final string. There is a second version of this method where the first argument is the locale.

While Scala uses Java strings, in certain contexts the Scala compiler will wrap a Java `String` with extra methods defined in `scala.collection.immutable.StringLike`. One of those extra methods is an *instance* method called `format`. You call it on the format string itself, then pass as arguments the values to be incorporated into the string. For example:

```
               "%02d: name =              "Dean
scala> val s = %s"               .format(5, Wampler"      )
            "05: name = Dean
s: String = Wampler"
```

In this example, we asked for a two-digit integer, padded with leading zeros.

The final version of the built-in string interpolation capabilities is the “raw” format that doesn’t expand control characters. Consider these two examples:

```
                  "Dean
scala> val name = Wampler"
               "Dean
name: String = Wampler"

scala> s"123\n$name\n456"
res0: String =
123
Dean Wampler
456

scala> raw"123\n$name\n456"
res1: String = 123\nDean Wampler\n456
```

Finally, we can actually define our own string interpolators, but we’ll need to learn more about *implicits* first. See Build Your Own String Interpolator for details.

Traits: Interfaces and “Mixins” in Scala

Here we are, about 100 pages in and we haven’t discussed one of the most basic features of any object-oriented language: how abstractions are defined, the equivalent of Java’s *interfaces*. What about inheritance of classes, too?

I put this off deliberately to emphasize the powerful capabilities that functional programming brings to Scala, but now is a good time to provide an overview of this important topic.

I’ve used vague terms like abstractions before. Some of our examples used `abstract` classes as “parent” classes already. I didn’t dwell on them, assuming that you’ve seen similar constructs in other languages before.

Java has `interfaces`, which let you *declare*, but not *define* methods, at least you couldn’t define them before Java 8. You can also declare and define `static` variables and nested types.

Scala replaces interfaces with *traits*. We’ll explore them in glorious detail in Chapter 9, but for now, think of them of interfaces that also give you the option of defining the methods you declare. Traits can also declare and optionally define *instance* fields (not just *static* fields, as in Java interfaces), and you can declare and optionally define *type* values, like the types we just saw in our enumeration examples.

It turns out that these extensions fix many limitations of Java’s object model, where only classes can define methods and fields. Traits enable true composition of behavior (“mixins”) that simply isn’t possible in Java before Java 8.

Let’s see an example that every enterprise Java developer has faced; mixing in logging. First, let’s start with a service:

```
//
src/main/scala/progscala2/rounding/traits.sc

class ServiceImportante(name: String) {
  def work(i: Int): Int = {
             "ServiceImportante: Doing important work!
    println(s$i"                                       )
    i + 1
  }
}

val service1 = new ServiceImportante("uno")
                                "Result:
(1 to 3) foreach (i => println(s${service1.work(i)}"         ))
```

We ask the service to do some work and get this output:

```
ServiceImportante: Doing important work!
1
Result: 2
ServiceImportante: Doing important work!
2
Result: 3
ServiceImportante: Doing important work!
3
Result: 4
```

Now we want to mix in a standard logging library. For simplicity, we'll just use `println`.

Here are two traits, one that defines the abstraction (that has no concrete members) and the other that implements the abstraction for "logging" to standard output:

```
trait Logging {
  def info    (message: String): Unit
  def warning(message: String): Unit
  def error   (message: String): Unit
}

trait StdoutLogging extends Logging {
                                          "INFO:
  def info    (message: String) = println(s$message"          )
  def warning(message: String) = println(s"WARNING: $message")
                                          "ERROR:
  def error   (message: String) = println(s$message"          )
}
```

Note that `Logging` is *exactly* like a Java interface. It is even implemented the same way in JVM byte code.

Finally, let's declare a service that "mixes in" logging:

```
val service2 = new ServiceImportante("dos") with StdoutLogging {
  override def work(i: Int): Int = {
          "Starting work: i =
    info(s$i"                    )
    val result = super.work(i)
          "Ending work: i = $i, result =
    info(s$result"                              )
    result
  }
}
                                "Result:
(1 to 3) foreach (i => println(s${service2.work(i)}"         ))
```

```
INFO:    Starting work: i = 1
ServiceImportante: Doing important work!
1
INFO:    Ending work: i = 1, result = 2
Result: 2
INFO:    Starting work: i = 2
ServiceImportante: Doing important work!
2
INFO:    Ending work: i = 2, result = 3
Result: 3
INFO:    Starting work: i = 3
ServiceImportante: Doing important work!
3
INFO:    Ending work: i = 3, result = 4
Result: 4
```

Now we log when we enter and leave `work`.

To mix in traits, we use the `with` keyword. We can mix in as many as we want. Some traits might not modify existing behavior at all, and just add new useful, but independent methods.

In this example, we're actually *modifying* the behavior of `work`, in order to inject logging, but we are not changing its "contract" with clients, that is, its external behavior.[]

If we needed multiple instances of `ServiceImportante with StdoutLogging`, we could declare a class:

```
class LoggedServiceImportante(name: String)
  extends ServiceImportante(name) with StdoutLogging {...}
```

Note how we pass the `name` argument to the parent class `ServiceImportante`. To create instances, `new LoggedServiceImportante("tres")` works as you would expect it to work.

However, if we need just one instance, we can mix in `StdoutLogging` as we define the variable.

To use the logging enhancment, we have to override the `work` method. Scala requires the `override` keyword when you override a concrete method in a parent class. Note how we access the parent class `work` method, using `super.work`, as in Java and many other languages.

There is a lot more to discuss about traits and object composition, as we'll see.

Recap and What's Next

We've covered a lot of ground in these first chapters. We learned how flexible and concise Scala code can be. In this chapter, we learned some powerful constructs for defining DSLs and for manipulating data, such as `for` comprehensions. Finally, we learned how to encapsulate values in enumerations and the basic capabilities of `traits`.

You should now be able to read quite a bit of Scala code, but there's plenty more about the language to learn. Now we'll begin a deeper dive into Scala features.