Table of Contents for Programming Scala, 2nd Edition

We described `for` comprehensions in Scala for Comprehensions. At this point, they look like a nice, more flexible version of the venerable `for` *loop*, but not much more. In fact, lots of sophistication lies below the surface, sophistication that has useful benefits for writing concise code with elegant solutions to a number of design problems.

In this chapter, we'll dive below the surface to really understand `for` comprehensions. We'll see how they are implemented in Scala and how your own container types can exploit them.

We'll finish with an examination of how many Scala container types use `for` comprehensions to address several common design problems, such as error handling during the execution of a sequence of processing steps. As a final step, we'll extract a well-known functional technique for a recurring idiom.

Recap: The Elements of for Comprehensions

`for` comprehensions contain one or more generator expressions, plus optional guard expressions (for filtering), and value definitions. The output can be "yielded" to create new collections or a side-effecting block of code can be executed on each pass, such as printing output. The following example demonstrates all these features. It removes blank lines from a text file:

```
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package progscala2.forcomps

object RemoveBlanks {

  /**
   * Remove blank lines from the specified input
file.

*/
  def apply(path: String, compressWhiteSpace: Boolean = false): Seq[String] =
    for {
      line <- scala.io.Source.fromFile(path).getLines.toSeq          // ❶
      if line.matches("""^\s*$""") == false                           // ❷
                                                               "
      line2 = if (compressWhiteSpace) line replaceAll ("\\s+", "  )  // ❸
              else line
    } yield line2                                                     //
❹

  /**
   * Remove blank lines from the specified input files and echo the
remaining
   * lines to standard output, one after the
other.
   * @param args list of file paths. Prefix each with an optional "-"
to
   *             compress remaining whitespace in the
file.

*/
  def main(args: Array[String]) = for {
    path2 <- args                                                     //
❺
    (compress, path) = if (path2 startsWith "-") (true, path2.substring(1))
                       else (false, path2)                            //
❻
    line <- apply(path, compress)
  } println(line)                                                     //
❼
}
```

❶

Use `scala.io.Source` to open the file and get the lines, where `getLines` returns a `scala.collection.Iterator`, which we must convert to a sequence, because we can't return an `Iterator` from the `for` comprehension and the return type is determined by the initial generator.

❷

Filter for blank lines using a regular expression.

❸

Define a local variable, either each nonblank line unchanged, if whitespace compression is not enabled, or a new line with all whitespace compressed to single spaces.

❹

Use yield to return line, so the for comprehension constructs a Seq[String] that apply returns. We'll return to the actual collection returned by apply in a moment.

❺

The main method also uses a for comprehension to process the argument list, where each argument is treated as a file path to process.

❻

If the file path starts with a - character, enable whitespace compression. Otherwise, only strip blank lines.

❼

Write all the processed lines together to stdout.

This file is compiled by sbt. Try running it at the sbt prompt on the source file itself. First, try it without the - prefix character. Here we show just a few of the lines of output:

```
> run-main progscala2.forcomps.RemoveBlanks \
  src/main/scala/progscala2/forcomps/RemoveBlanks.scala
[info] Running ...RemoveBlanks src/.../forcomps/RemoveBlanks.scala
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package forcomps
object RemoveBlanks {
  /**
   * Remove blank lines from the specified input
file.

*/
  def apply(path: String, compressWhiteSpace: Boolean = false): Seq[String] =
...
```

The blank lines in the original file are removed. Running with the - prefix yields this:

```
> run-main progscala2.forcomps.RemoveBlanks \
  -src/main/scala/progscala2/forcomps/RemoveBlanks.scala
[info] Running ...RemoveBlanks -src/.../forcomps/RemoveBlanks.scala
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package forcomps
object RemoveBlanks {
 /**
 * Remove blank lines from the specified input
file.
 */
 def apply(path: String, compressWhiteSpace: Boolean = false): Seq[String] =
...
```

Now runs of whitespace are compressed to single spaces.

You might try modifying this application to add more options, like prefixing with line numbers, writing the output back to separate files, calculating statistics, etc. How might you allow individual elements in the args array to be command-line options, like a typical Unix-style command line?

Let's return to the actual collection returned by the `apply` method. We can find out if we start the `sbt` console:

```
> console
Welcome to Scala version 2.11.2 (Java HotSpot(TM) ...).
...
scala> val lines = forcomps.RemoveBlanks.apply(
         "src/main/scala/progscala2/forcomps/RemoveBlanks.scala"
     | )
lines: Seq[String] = Stream(
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala,
?)

scala> lines.head
res1: String = // src/main/scala/progscala2/forcomps/RemoveBlanks.scala

scala> lines take 5 foreach println
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package forcomps
object RemoveBlanks {
  /**
   * Remove blank lines from the specified input
file.
```

A lazy `Stream` is returned, which we introduced in Tail Recursion Versus Traversals of Infinite Collections. When the REPL printed the line after the definition of `lines`, the `Stream.toString` method computes the head of the stream (the comment line from the file) and it shows a question mark for the unevaluated tail.

We can ask for the head, then take the first five lines, which forces those lines to be evaluated, etc. `Stream` can be appropriate because we are processing files that could be very large, in which case storing the entire unfiltered contents in memory could consume too much memory. Unfortunately, if we actually do read the entire large data set, we'll still have it all in memory, because `Stream` remembers all the elements it evaluated. Note that each iteration of both `for` comprehensions (in `apply` and in `main`) are stateless, so we don't need to keep more than one line at a time in memory.

In fact, when you call `toSeq` on a `scala.collection.Iterator`, the default implementation in the subtype `scala.collection.TraversableOnce` returns a `Stream`. Other types that subclass `Iterator` might return a strict collection.

for Comprehensions: Under the Hood

The `for` comprehension syntax is actually syntactic sugar provided by the compiler for calling the collection methods `foreach`, `map`, `flatMap`, and `withFilter`.

Why have a second way to invoke these methods? For nontrivial sequences, `for` comprehensions are much easier to read and write than involved expressions with the corresponding API calls.

The method `withFilter` is used for filtering elements just like the `filter` method that we saw previously. Scala will use `filter` if `withFilter` is not defined (with a compiler warning). However, unlike `filter`, it doesn't construct its own output collection. For better efficiency, it works with the other methods to combine filtering with their logic so that one less new collection is generated. Specifically, `withFilter` restricts the domain of the elements allowed to pass through subsequent combinators like `map`, `flatMap`, `foreach`, and other `withFilter` invocations.

To see what the `for` comprehension sugar encapsulates, let's walk through several informal comparisons first, then we'll discuss the details of the precise mapping.

Consider this simple `for` comprehension:

```
// src/main/scala/progscala2/forcomps/for-
foreach.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
} println(s)
//
Results:
// Alabama
//
Alaska
//
Virginia
// Wyoming

states foreach println
// Results the same as
before.
```

The output is shown in the comments. (From this point on, I'll show REPL sessions less often. When I use code listings instead, I'll use comments to show important results.)

A single generator expression *without* a `yield` expression after the comprehension corresponds to an invocation of `foreach` on the collection.

What happens if we use `yield` instead?

```
// src/main/scala/progscala2/forcomps/for-
map.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
} yield s.toUpperCase
// Results: List(ALABAMA, ALASKA, VIRGINIA,
WYOMING)

states map (_.toUpperCase)
// Results: List(ALABAMA, ALASKA, VIRGINIA,
WYOMING)
```

A single generator expression with a `yield` expression corresponds to an invocation of `map`. Note that when `yield` is used to construct a new container, the type of the first generator expression determines the final resulting collection. This makes sense when you look at that corresponding `map` expression. Try changing the input `List` to `Vector` and notice that a new `Vector` is generated.

What if we have more than one generator?

```
// src/main/scala/progscala2/forcomps/for-
flatmap.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
  c <- s
} yield s"$c-${c.toUpper}"
// Results: List("A-A", "l-L", "a-A", "b-B",
...)

states flatMap (_.toSeq map (c => s"$c-${c.toUpper}"))
// Results: List("A-A", "l-L", "a-A", "b-B",
...)
```

The second generator iterates through each character in the string `s`. The contrived `yield` statement returns the character and its uppercase equivalent, separated by a dash.

When there are multiple generators, all but the last are converted to `flatMap` invocations. The last is a `map` invocation. Once again, a `List` is generated. Try using another input collection type.

What if we add a guard?

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

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
  c <- s
  if c.isLower
         "$c-${c.toUpper}
} yield s"
// Results: List("l-L", "a-A", "b-B",
...)

states flatMap (_.toSeq withFilter (_.isLower) map (c => s"$c-${c.toUpper}"))
// Results: List("l-L", "a-A", "b-B",
...)
```

Note that the `withFilter` invocation is injected before the final `map` invocation.

Finally, defining a variable works like this:

```
// src/main/scala/progscala2/forcomps/for-
variable.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
  c <- s
  if c.isLower
        "$c-${c.toUpper}
  c2 = s"
} yield c2
// Results: List("l-L", "a-A", "b-B",
...)

states flatMap (_.toSeq withFilter (_.isLower) map { c =>
            "$c-${c.toUpper}
  val c2 = s"
  c2
})
// Results: List("l-L", "a-A", "b-B",
...)
```

Translation Rules of for Comprehensions

Now that we have an intuitive understanding of how `for` comprehensions are translated to collection methods, let's define the details more precisely.

First, in a generator expression such as `pat <- expr`, `pat` is actually a pattern expression. For example, `(x, y) <- List((1,2), (3,4))`. Similarly, in a value definition `pat2 = expr`, `pat2` is also interpreted as a pattern.

The first step in the translation is to convert `pat <- expr` to the following:

```
// pat <-
expr
pat <- expr.withFilter { case pat => true; case _ => false }
```

After this, the following translations are applied repeatedly until all comprehension expressions have been replaced. Note that some steps generate *new* `for` comprehensions that subsequent iterations will translate.

First, a `for` comprehension with one generator and a final `yield` expression:

```
// for ( pat <- expr1 ) yield
expr2
expr map { case pat => expr2 }
```

A `for` *loop*, where `yield` isn't used, but side effects are performed:

```
// for ( pat <- expr1 )
expr2
expr foreach { case pat => expr2 }
```

A `for` comprehension with more than one generator:

```
// for ( pat1 <- expr1; pat2 <- expr2; ... ) yield
exprN
expr1 flatMap { case pat1 => for (pat2 <- expr2 ...) yield exprN }
```

Note that the nested generators are translated to nested `for` comprehensions. The next cycle of applying the translation rules will convert them to method calls. The elided (…) expressions could be other generators, value definitions, or guards.

A `for` loop with more than one generator:

```
// for ( pat1 <- expr1; pat2 <- expr2; ... )
exprN
expr1 foreach { case pat1 => for (pat2 <- expr2 ...) yield exprN }
```

In the `for` comprehension examples we've seen before that had a guard expression, we wrote the guard on a separate line. In fact, a guard and the expression on the previous line can be written together on a single line, e.g.,
`pat1 <- expr1 if`
`guard`   .

A generator followed by a guard is translated as follows:

```
// pat1 <- expr1 if
guard
pat1 <- expr1 withFilter ((arg1, arg2, ...) => guard)
```

Here, the `argN` variables are the arguments for the appropriate function passed to `withFilter`. For most collections, there will be a single argument.

A generator followed by a value definition has a surprisingly complex translation:

```
// pat1 <- expr1; pat2 =
expr2
(pat1, pat2) <- for {                   //
❶
  x1 @ pat1 <- expr1                    //
❷
} yield {
  val x2 @ pat2 = expr2                 //
❸
  (x1, x2)
// ❹
}
```

❶

We'll return a pair of the two patterns.

❷

`x1 @ pat1` means assign to variable `x1` the value corresponding to the whole expression that `pat1` matches. Inside `pat1`, there might be other variables bound to the constituent parts. If `pat1` is just an immutable variable name, the two assignments to `x1` and `pat1` are redundant.

❸

Assign to `x2` the value of `pat2`.

❹

Return the tuple.

As an example of `x @ pat = expr`, consider the following REPL session:

```
scala> val z @ (x, y) = (1 -> 2)
z: (Int, Int) = (1,2)
x: Int = 1
y: Int = 2
```

The value `z` is the tuple `(1,2)`, while `x` and `y` are the constituents of the pair.

The complete translation is hard to follow, so let's look at a concrete example:

```
// src/main/scala/progscala2/forcomps/for-variable-
translated.sc

val map = Map("one" -> 1, "two" -> 2)

val list1 = for {
  (key, value) <- map   // How is this line and the next translated?
  i10 = value + 10
} yield (i10)
// Result: list1: scala.collection.immutable.Iterable[Int] = List(11,
12)

// Translation:
val list2 = for {
  (i, i10) <- for {
    x1 @ (key, value) <- map
  } yield {
    val x2 @ i10 = value + 10
    (x1, x2)
  }
} yield (i10)
// Result: list2: scala.collection.immutable.Iterable[Int] = List(11,
12)
```

Note how the two expressions inside the outer `for {…}` are translated. Even though we work with a pair `(x1, x2)` inside, only `x2` (equivalent to `i10`) is actually returned. Also, recall that the elements of a `Map` are key-value pairs, so that's what we pattern match against in the generator shown, which iterates through the map.

This completes the translation rules. Whenever you encounter a `for` comprehension, you can apply these rules to translate it into method invocations on containers. You won't need to do this often, but sometimes it's a useful skill for debugging problems.

Let's look at one more example that uses pattern matching to parse a conventional properties file, with a key = value format:

```
// src/main/scala/progscala2/forcomps/for-
patterns.sc

val ignoreRegex = """^\s*(#.*|\s*)$""".r
// ❶
val kvRegex = """^\s*([^=]+)\s*=\s*([^#]+)\s*.*$""".r                //
❷

val properties = """
  |# Book
properties

|
  |book.name = Programming Scala, Second Edition # A
comment
  |book.authors = Dean Wampler and Alex
Payne
  |book.publisher =
O'Reilly
  |book.publication-year =
2014

|"""  .stripMargin
// ❸

val kvPairs = for {
  prop <- properties.split("\n")
// ❹
  if ignoreRegex.findFirstIn(prop) == None
// ❺
  kvRegex(key, value) = prop
// ❻
} yield (key.trim, value.trim)
// ❼
// Returns: kvPairs: Array[(String, String)] =
Array(
//   (book.name,Programming Scala, Second
Edition),
//   (book.authors,Dean Wampler and Alex
Payne),
//
(book.publisher,O'Reilly),
//   (book.publication-
year,2014))
```

❶

A regular expression that looks for lines to “ignore,” i.e., those that are blank or comments, where `#` is the comment marker and must be the first non-whitespace character on the line.

❷

A regular expression for `key = value` pairs, which handles arbitrary whitespace and optional trailing comments.

❸

An example multiline string of properties. Note the use of `StringLike.stripMargin` to remove all leading characters on each line up to and including the `|`. This technique lets us indent those lines without having that whitespace interpreted as part of the string.

❹

Split the properties on line feeds.

❺

Filter for lines that we *don't* want to ignore.

❻

Use of a pattern expression on the lefthand side; extract the key and value from a valid property line, using the regular expression.

❼

Yield the resulting key-value pairs, trimming extraneous whitespace that remains.

An `Array[(String,String)]` is returned. It's an `Array` because the generator called `String.split`, which returns an `Array`.

See Section 6.19 in the *Scala Language Specification* for more examples of `for` comprehensions and their translations.

Options and Other Container Types

We used `List`s, `Array`s, and `Map`s for our examples, but any types that implement `foreach`, `map`, `flatMap`, and `withFilter` can be used in `for` comprehensions and not just the obvious collection types. In other words, any type that can be considered a *container* could support these methods and allow us to use instances of the type in `for` comprehensions.

Let's consider several other container types. We'll see how exploiting `for` comprehensions can transform your code in unexpected ways.

Option as a Container

`Option` is a *binary* container. It has an item or it doesn't. It implements the four methods we need.

Here are the implementations for the methods we need in `Option` (from the Scala 2.11 library source code; some incidental details have been omitted or changed):

```
sealed abstract class Option[+T] { self =>                          // ❶
  ...
                           // Implemented by Some and
  def isEmpty: Boolean  None.

  final def foreach[U](f: A => U): Unit =
    if (!isEmpty) f(this.get)

  final def map[B](f: A => B): Option[B] =
    if (isEmpty) None else Some(f(this.get))

  final def flatMap[B](f: A => Option[B]): Option[B] =
    if (isEmpty) None else f(this.get)

  final def filter(p: A => Boolean): Option[A] =
    if (isEmpty || p(this.get)) this else None

  final def withFilter(p: A => Boolean): WithFilter = new WithFilter(p)

  /** We need a whole WithFilter class to honor the "doesn't create a
  new
   *  collection" contract even though it seems unlikely to matter much in
a
   *  collection with max size
1.

*/
  class WithFilter(p: A => Boolean) {
    def map[B](f: A => B): Option[B] = self filter p map f      // ❷
    def flatMap[B](f: A => Option[B]): Option[B] = self filter p flatMap f
    def foreach[U](f: A => U): Unit = self filter p foreach f
    def withFilter(q: A => Boolean): WithFilter =
      new WithFilter(x => p(x) && q(x))
  }
}
```

❶

The `self =>` expression defines an alias for `this` for the `Option` instance. It is needed inside `WithFilter` later. See Self-Type Annotations for more details.

❷

Use the `self` reference defined above to operate on the enclosing `Option` instance, rather than an instance of `WithFilter`. That is, if we used `this`, we would refer to the `WithFilter` instance.

The `final` keyword prevents subclasses from overriding the implementation. It might be a little shocking to see a base class refer to derived classes. Normally, it would be considered bad object-oriented design for base types to know anything about their derived types, if any.

However, recall from Chapter 2 that the `sealed` keyword means that the only allowed subclasses must be defined in the same file. `Option`s are either empty (`None`) or not (`Some`). So, this code is robust, comprehensive (it covers all cases), concise, and entirely reasonable.

The crucial feature about these `Option` methods is that the function arguments are only applied if the `Option` isn't empty.

This feature allows us to address a common design problem in an elegant way. A common pattern in distributed computation is to break up a computation into smaller tasks, distribute those tasks around a cluster, then gather the results together. For example, Hadoop's MapReduce framework uses this pattern. We would like an elegant way to ignore the task results that are empty and just deal with the nonempty results. For the moment, we'll ignore task errors.

First, let's assume that each task returns an `Option`, where `None` is returned for empty results and `Some` wraps a nonempty result. We want to filter out the `None` results in the most elegant way.

Consider the following example, where we have a list of three results, each of which is an `Option[Int]`:

```
// src/main/scala/progscala2/forcomps/for-options-
seq.sc

val results: Seq[Option[Int]] = Vector(Some(10), None, Some(20))

val results2 = for {
  Some(i) <- results
} yield (2 * i)
// Returns: Seq[Int] = Vector(20,
40)
```

`Some(i) <- list` pattern matches on the elements of `results`, removing the `None` values, and extracting the integers inside the `Some` values. We then yield the final expression we want. The output is `Vector(20, 40)`.

As an exercise, let's work through the translation rules. Here is the first step, where we apply the first rule for converting each `pat <- expr` expression to a `withFilter` expression:

```
// Translation step
#1
val results2b = for {
  Some(i) <- results withFilter {
    case Some(i) => true
    case None => false
  }
} yield (2 * i)
// Returns: results2b: List[Int] = List(20,
40)
```

Finally, we convert the outer `for { x <- y} yield (z)` expression to a `map` call:

```
// Translation step
#2
val results2c = results withFilter {
  case Some(i) => true
  case None => false
} map {
  case Some(i) => (2 * i)
}
// Returns: results2c: List[Int] = List(20,
40)
```

The `map` expression actually generates a compiler warning:

```
<console>:9: warning: match may not be exhaustive.
It would fail on the following input: None
        } map {
              ^
```

Normally, it would be dangerous if the partial function passed to `map` did not have a `case None => ...` clause. If a `None` showed up, a `MatchError` exception would be thrown. However, because the call to `withFilter` has already removed any `None` elements, the error won't happen.

Now let's consider another design problem. Instead of having independent tasks where we ignore the empty results and combine the nonempty results, consider the case where we run a sequence of dependent steps, and we want to stop the whole process as soon as we encounter a `None`.

Note that we have a limitation that using `None` means we receive no feedback about why the step returned nothing, such as a failure. We'll address this limitation in subsequent sections.

We could write tedious conditional logic that tries each case, one at a time, and checks the results,▯ but a `for` comprehension is better:

```
// src/main/scala/progscala2/forcomps/for-options-
good.sc

def positive(i: Int): Option[Int] =
  if (i > 0) Some(i) else None

for {
  i1 <- positive(5)
  i2 <- positive(10 * i1)
  i3 <- positive(25 * i2)
  i4 <- positive(2  * i3)
} yield (i1 + i2 + i3 + i4)
// Returns: Option[Int] =
Some(3805)

for {
  i1 <- positive(5)
  i2 <- positive(-1 * i1)               // ❶    EPIC FAIL
!
  i3 <- positive(25 * i2)               // ❷
  i4 <- positive(-2 * i3)               // EPIC FAIL!
} yield (i1 + i2 + i3 + i4)
// Returns: Option[Int] =
None
```

❶

`None` is returned. What happens with the left arrow?

❷

Is it okay to reference `i2` here?

The `positive` function is our “worker” that returns an `Option[Int]`, a `Some(i)` if the input `i` is positive, or a `None`, otherwise.

Note that the second and third expressions in the two `for` comprehensions use the previous values. As written, they appear to assume that the “happy path” will occur and it’s safe to use the returned `Int` extracted from the `Option[Int]`.

For the first `for` comprehension, the assumption works fine. For the second `for` comprehension, it *still* works fine! Once a `None` is returned, the subsequent expressions are effectively no-ops, because the function literals won’t be evaluated when `map` or `flatMap` are called from that point forward.

Let’s look at three more container types with the same properties, `Either`, `Try`, and a `Validation` type in a popular, third-party library called *Scalaz*.

Either: A Logical Extension to Option

We noted that the use of `Option` has the disadvantage that `None` carries no information that could tell us why no value is available, e.g., because an error occurred. Using `Either` instead is one solution. As the name suggests, `Either` is a container that holds one and only one of two things. In other words, where `Option` handled the case of zero or one items, `Either` handles the case of one item or another.

`Either` is a parameterized type with two parameters, `Either[+A, +B]`, where the `A` and `B` are the two possible types of the element contained in the `Either`. Recall that `+A` indicates that `Either` is *covariant* in the type parameter `A` and similarly for `+B`. This means that if you need a value of type `Either[Any,Any]` (for example, a method argument), you can use an instance of type `Either[String,Int]`, because `String` and `Int` are subtypes of `Any`, therefore `Either[String,Int]` is a subtype of `Either[Any,Any]`.

`Either` is also a sealed abstract class with two subclasses defined, `Left[A]` and `Right[B]`. That's how we distinguish between the two possible elements.

The concept of `Either` predates Scala. It has been used for a long time as an alternative to throwing exceptions. By historical convention, when used to hold either an error indicator or a value, the `Left` value is used to hold the error indicator, such as a message string or even an exception thrown by a lower-level library, and the normal return value is returned as a `Right`. To be clear, though, `Either` can be used for any scenario where you want to hold one object *or* another, possibly of different types.

Before we dive into some of the particulars of `Either`, let's just try porting our previous example. First, if you have a list of `Either` values and just want to discard the errors (`Left`s), a simple `for` comprehension does the trick:

```
// src/main/scala/progscala2/forcomps/for-eithers-
good.sc

def positive(i: Int): Either[String,Int] =
  if (i > 0) Right(i) else Left(s"nonpositive number $i")

for {
  i1 <- positive(5).right
  i2 <- positive(10 * i1).right
  i3 <- positive(25 * i2).right
  i4 <- positive(2  * i3).right
} yield (i1 + i2 + i3 + i4)
// Returns: scala.util.Either[String,Int] =
Right(3805)

for {
  i1 <- positive(5).right
  i2 <- positive(-1 * i1).right   // EPIC FAIL!
  i3 <- positive(25 * i2).right
  i4 <- positive(-2 * i3).right   // EPIC FAIL!
} yield (i1 + i2 + i3 + i4)
// Returns: scala.util.Either[String,Int] = Left(nonpositive number -
5)
```

This version is very similar to the implementation for `Option`, with the appropriate type changes. Like the `Option` implementation, we only see the first error. However, note that we have to call the `right` method on the values returned from `positive`. To understand why, let's discuss the purpose of the `right` and corresponding `left` methods.

Consider these simple examples of `Either`, `Left`, and `Right` adapted from the Scaladocs for `Either`:

```
       val l: Either[String, Int] = Left("boo"
scala> )
l: Either[String,Int] = Left(boo)

       val r: Either[String, Int] = Right(12
scala> )
r: Either[String,Int] = Right(12)
```

We declare two `Either[String, Int]` values and assign a `Left[String]` to the first and a `Right[Int]` to the second.

By the way, you might recall from unapply Method that you can use infix notation for type annotations when a type takes two parameters. So, we can declare `l` two ways:

```
       val l1: Either[String, Int] = Left("boo"
scala> )
l1: Either[String,Int] = Left(boo)

scala> val l2: String Either Int = Left("boohoo")
l2: Either[String,Int] = Left(boohoo)
```

For this reason, I wish `Either` was named `Or` instead! If you *really* prefer `Or` you could use a type alias in your own code:

```
scala> type Or[A,B] = Either[A,B]
defined type alias Or

scala> val l3: String Or Int = Left("better?")
l3: Or[String,Int] = Left(better?)
```

Next, our combinator method friends `map`, `fold`, etc. aren't defined on `Either` itself. Instead, we have to call `Either.left` or `Either.right`. The reason is that our combinators take a single function argument, but we would need the ability to specify two functions, one to invoke if the value is a `Left` and one to invoke if the value is a `Right`. Instead, the `left` and `right` methods create "projections" that have the combinator methods:

```
scala> l.left
res0: scala.util.Either.LeftProjection[String,Int] = \
  LeftProjection(Left(boo))

scala> l.right
res1: scala.util.Either.RightProjection[String,Int] = \
  RightProjection(Left(boo))

scala> r.left
res2: scala.util.Either.LeftProjection[String,Int] = \
  LeftProjection(Right(12))

scala> r.right
res3: scala.util.Either.RightProjection[String,Int] = \
  RightProjection(Right(12))
```

Note that the `Either.LeftProjection` values can hold either a `Left` or `Right` instance, and similarly for `Either.RightProjection`. Let's call `map` on these projections, passing in a single function:

```
scala> l.left.map(_.size)
res4: Either[Int,Int] = Left(3)

scala> r.left.map(_.size)
res5: Either[Int,Int] = Right(12)

scala> l.right.map(_.toDouble)
res6: Either[String,Double] = Left(boo)

scala> r.right.map(_.toDouble)
res7: Either[String,Double] = Right(12.0)
```

When you call `LeftProjection.map` and it holds a `Left` instance, it calls the function on the value held by the `Left`, analogous to how `Option.map` works with a `Some`. However, if you call `LeftProjection.map` and it holds a `Right`, it passes the `Right` instance through without modification, analogous to how `Option.map` works with `None`.

Similarly, calling `RightProjection.map` when it holds a `Right` instance means the function will be called on the value held by the `Right`, while nothing is changed if it actually holds a `Left` instance.

Note the return types. Because `l.left.map(_.size)` converts a `String` to an `Int`, the new `Either` is `Either[Int,Int]`. The second type parameter is not changed, because the function won't be applied to a `Right[Int]`.

Similarly, `r.right.map(_.toDouble)` converts an `Int` to a `Double`, so an `Either[String,Double]` is returned. There is a "String.toDouble" method, which parses the string and returns a double or throws an exception if it can't. However, this method will never be called.

We can also use a `for` comprehension to compute the size of the `String`. Here is the previous expression and the equivalent `for` comprehension:

```
l.left map (_.size)
// Returns:
Left(3)
                               // Returns:
for (s <- l.left) yield s.size Left(3)
```

Throwing exceptions versus returning Either values

While `Either` has its charms, isn't it just easier to throw an exception when things go wrong? There are certainly times when an exception makes sense as a way to abandon a calculation, as long as some object on the call stack catches the exception and performs reasonable recovery.

However, throwing an exception breaks referential transparency. Consider this contrived example:

```
// src/main/scala/progscala2/forcomps/ref-transparency.sc

scala> def addInts(s1: String, s2: String): Int =
         s1.toInt + s2.
     | toInt
addInts: (s1: String, s2: String)Int

scala> for {
         i <- 1 to
     | 3
         j <- 1 to
     | i
     |
           "$i+$j =
} println(s${addInts(i.toString,j.toString)}"         )
1+1 = 2
2+1 = 3
2+2 = 4
3+1 = 4
3+2 = 5
3+3 = 6

scala> addInts("0", "x")
java.lang.NumberFormatException: For input string: "x"
...
```

It appears that we can substitute invocations of `addInts` with values, rather than call the function. We might cache previous calls and return those instead. However, `addInts` throws an exception if we happen to pass a `String` that can't be parsed as an `Int`. Hence, we can't replace the function call with values that can be returned for all argument lists.

Even worse, the type signature of `addInts` provides no indication that trouble lurks. This is a contrived example of course, but parsing string input by end users is certainly a common source of exceptions.

It's true that Java's *checked exceptions* solve this particular problem. Method signatures indicate the possible error conditions in the form of thrown exceptions. However, for various reasons, checked exceptions hasn't worked well in practice. They aren't implemented by other languages, including Scala. Java programmers often avoid using them, throwing subclasses of the unchecked `java.lang.RuntimeException` instead.

Using `Either` restores referential transparency and indicates through the type signature that errors can occur. Consider this rewrite of `addInts`:

```
// src/main/scala/progscala2/forcomps/ref-transparency.sc

scala> def addInts2(s1: String, s2: String): Either[NumberFormatException,Int]=
          try
      | {
           Right(s1.toInt + s2.toInt
      | )
          } catch
      | {
           case nfe: NumberFormatException => Left(nfe
      | )

      | }
addInts2: (s1: String, s2: String)Either[NumberFormatException,Int]

scala> println(addInts2("1", "2"))
Right(3)

scala> println(addInts2("1", "x"))
Left(java.lang.NumberFormatException: For input string: "x")

scala> println(addInts2("x", "2"))
Left(java.lang.NumberFormatException: For input string: "x")
```

The type signature now indicates the possible failure "mode." Instead of grabbing control of the call stack by throwing the exception out of `addInts2`, we've *reified* the error by returning the exception as a value on the call stack.

Now, not only can you substitute values for the method invocations with valid strings, you could even substitute the appropriate `Left[java.lang.NumberFormatException]` values for invocations with invalid strings!

So, `Either` lets you assert control of call stack in the event of a wide class of failures. It also makes the behavior more explicit to users of your APIs.

Look at the implementation of `addInts2` again. Throwing exceptions is quite common in Java and even Scala libraries, so we might find ourselves writing this `try {…} catch {…}` boilerplate a lot to wrap the good and bad results in an `Either`. For handling exceptions, maybe we should encapsulate this boilerplate with types and use names for these types that express more clearly when we have either a "failure" or a "success." The `Try` type does just that.

Try: When There Is No Do

`scala.util.Try` is structurally similar to `Either`. It is a sealed abstract class with two subclasses, `Success` and `Failure`.

`Success` is analogous to the conventional use of `Right`. It holds the normal return value. `Failure` is analogous to `Left`, but `Failure` always holds a `Throwable`.

Here are the signatures of these types (omitting some traits that aren't relevant to the discussion):

```
sealed abstract class Try[+T] extends AnyRef {...}
final case class Success[+T](value: T) extends Try[T] {...}
final case class Failure[+T](exception: Throwable) extends Try[T] {...}
```

Note that there is just one type parameter, `Try[+T]`, compared to two for `Either[+A,+B]`, because the equivalent of the `Left` type is now `Throwable`.

Also, `Try` is clearly asymmetric, unlike `Either`. There is only one “normal” type we care about (`T`) and a `java.lang.Throwable` for the error case. This means that `Try` can define combinator methods like `map` to apply to the `T` value when the `Try` is actually a `Success`.

Let’s see how `Try` is used, again porting our previous example. First, if you have a list of `Try` values and just want to discard the `Failure`s, a simple `for` comprehension does the trick:

```
// src/main/scala/progscala2/forcomps/for-tries-
good.sc
import scala.util.{ Try, Success, Failure }

def positive(i: Int): Try[Int] = Try {
                  "nonpositive number
  assert (i > 0, s$i"                    )
  i
}

for {
  i1 <- positive(5)
  i2 <- positive(10 * i1)
  i3 <- positive(25 * i2)
  i4 <- positive(2  * i3)
} yield (i1 + i2 + i3 + i4)
// Returns: scala.util.Try[Int] =
Success(3805)

for {
  i1 <- positive(5)
  i2 <- positive(-1 * i1)              // EPIC FAIL!
  i3 <- positive(25 * i2)
  i4 <- positive(-2 * i3)              // EPIC FAIL!
} yield (i1 + i2 + i3 + i4)
// Returns: scala.util.Try[Int] =
Failure(
//   java.lang.AssertionError: assertion failed: nonpositive number -
5)
```

Note the concise definition of `positive`. If the assertion fails, the `Try` block will return a `Failure` wrapping the thrown `java.lang.AssertionError`. Otherwise, the result of the `Try` expression is wrapped in a `Success`. A more explicit definition of `positive` showing the boilerplate is the following:

```
def positive(i: Int): Try[Int] =
  if (i > 0) Success(i)
  else Failure(new AssertionError("assertion failed"))
```

The `for` comprehensions look exactly like those for the original `Option` example. With type inference, there is very little boilerplate here, too. You can focus on the “happy path” logic and let `Try` capture errors.

Scalaz Validation

There is one scenario where all of the previous types aren’t quite what we need. The combinators won’t be called for subsequent expressions after an empty result (for `Option`) or failure. Effectively, we stop processing at the first error. However, what if we’re performing several, independent steps and we would actually like to accumulate any and all errors as we go, then decide what to do? A classical scenario is validating user input, e.g., from a web form. You want to return any and all errors at once to the user.

The Scala standard library doesn’t provide a type for this, but the popular, third-party library Scalaz offers a Validation type for this purpose:

```
// src/main/scala/progscala2/forcomps/for-validations-
good.sc
import scalaz._, std.AllInstances._

def positive(i: Int): Validation[List[String], Int] = {
  if (i > 0) Success(i)
// ❶
                     "Nonpositive integer
  else Failure(List(s$i"                     ))
}

for {
  i1 <- positive(5)
  i2 <- positive(10 * i1)
  i3 <- positive(25 * i2)
  i4 <- positive(2  * i3)
} yield (i1 + i2 + i3 + i4)
// Returns: scalaz.Validation[List[String],Int] =
Success(3805)

for {
  i1 <- positive(5)
  i2 <- positive(-1 * i1)              // EPIC FAIL!
  i3 <- positive(25 * i2)
  i4 <- positive(-2 * i3)              // EPIC FAIL!
} yield (i1 + i2 + i3 + i4)
// Returns: scalaz.Validation[List[String],Int]
=
//   Failure(List(Nonpositive integer -5))
//                                                                    ❷

positive(5) +++ positive(10) +++ positive(25)                         //
❸
// Returns: scalaz.Validation[String,Int] =
Success(40)

positive(5) +++ positive(-10) +++ positive(25) +++ positive(-30)      //
❹
// Returns: scalaz.Validation[String,Int]
=
//   Failure(Nonpositive integer -10, Nonpositive integer -
30)
```

❶

Success and Failure here are subclasses of scalaz.Validation. They are not the scala.util.Try subtypes.

❷

Because we use a for comprehension, the evaluation is still short-circuited, so we don’t see the last error for i4.

❸

However, in this and the following expressions, we evaluate all the calls to positive, then “add” the results or accumulate the errors.

❹

Both errors are reported.

Like `Either`, the first of the two type parameters is the type used to report errors. In this case, we use a `List[String]` so we can accumulate multiple errors. However, `String` or any other collection that supports appending values will also work. Scalaz handles the details of invoking the appropriate “concatenation” method.

The second type parameter is for the result returned if validation succeeds. Here we use an `Int`, but it could also be a collection type.

Note that the `for` comprehension still short-circuits the evaluation. That’s still what we want, because each subsequent invocation of `positive` depends on a previous invocation.

However, we then see how to use the `+++` “addition” operator▯ to perform *independent* evaluations, like you might do with web form input, and then aggregate together the results, if all of them validated successfully. Otherwise, all the errors are aggregated together as the result of this expression. We use a list of `String`s for this purpose.

In a web form, you wouldn’t be adding numbers together, but accumulating mixed fields. Let’s adapt this example to be more realistic for form validation. We’ll use as the success type `List[(String,Any)]`, which is a list of key-value tuples. If successful, we could call `toMap` on the `List` to create a `Map` to return to the caller.▯

We’ll validate a user’s first name, last name, and age. The names must be nonempty and contain only alphabetic characters. The age, which will now start out as a string you might retrieve from a web form, must parse to a positive integer:

```
// src/main/scala/progscala2/forcomps/for-validations-good-
form.sc
import scalaz._, std.AllInstances._

/** Validate a user's name; nonempty and alphabetic characters, only.
*/
def validName(key: String, name: String):
    Validation[List[String], List[(String,Any)]] = {
                     // remove
  val n = name.trim  whitespace
  if (n.length > 0 && n.matches("""^\p{Alpha}$""")) Success(List(key -> n))
                     "Invalid $key
  else Failure(List(s<$n>"              ))
}

/** Validate that the string is an integer and greater than zero.
*/
def positive(key: String, n: String):
    Validation[List[String], List[(String,Any)]] = {
  try {
    val i = n.toInt
    if (i > 0) Success(List(key -> i))
                       "Invalid $key
    else Failure(List(s$i"              ))
  } catch {
    case _: java.lang.NumberFormatException =>
                      "$n is not an
```

```
      Failure(List(sinteger"              ))
  }
}

def validateForm(firstName: String, lastName: String, age: String):
    Validation[List[String], List[(String,Any)]] = {
  validName("first-name", firstName) +++ validName("last-name", lastName) +++
    positive("age", age)
}

validateForm("Dean", "Wampler", "29")
// Returns: Success(List((first-name,Dean), (last-name,Wampler),
(age,29)))
validateForm("", "Wampler", "29")
// Returns: Failure(List(Invalid first-name
<>))
             "D e a
validateForm(n"       , "Wampler", "29")
// Returns: Failure(List(Invalid first-name <D e a
n>))
validateForm("D1e2a3n_", "Wampler", "29")
// Returns: Failure(List(Invalid first-name
<D1e2a3n_>))
validateForm("Dean", "", "29")
// Returns: Failure(List(Invalid last-name
<>))
validateForm("Dean", "Wampler", "0")
// Returns: Failure(List(Invalid age
0))
validateForm("Dean", "Wampler", "xx")
// Returns: Failure(List(xx is not an
integer))
validateForm("", "Wampler", "0")
// Returns: Failure(List(Invalid first-name <>, Invalid age
0))
validateForm("Dean", "", "0")
// Returns: Failure(List(Invalid last-name <>, Invalid age
0))
             "D e a
validateForm(n"       , "", "29")
// Returns: Failure(List(Invalid first-name <D e a n>, Invalid last-name
<>))
```

Using `scalaz.Validation` yields beautifully concise code for validating a set of independent values. It returns all the errors found, if there are any, or the values collected in a suitable data structure.

Recap and What's Next

`Either`, `Try`, and `Validation` express through types a fuller picture of how the program actually behaves. Both say that a valid value will (hopefully) be returned, but if not, they also encapsulate the failure information you'll need to know. Similarly, `Option` encapsulates the presence or absence of a value explicitly in the type signature.

By *reifying* the exception using one of these types,□ we also solve an important problem in concurrency. Because asynchronous code isn't guaranteed to be running on the same thread as the "caller," the caller can't catch an exception thrown by the other code. However, by returning an exception the same way we return the normal result, the caller can get the exception. We'll explore the details in Chapter 17.

You probably expected this chapter to be a perfunctory explanation of Scala's fancy `for` *loops*. Instead, we broke through the facade to find a surprisingly powerful set of tools. We saw how a set of functions, `map`, `flatMap`, `foreach`, and `withFilter`, plug into `for` comprehensions to provide concise, flexible, yet powerful tools for building nontrivial application logic.

We saw how to use `for` comprehensions to work with collections, but we also saw how useful they are for other container types, specifically `Option`, `util.Either`, `util.Try`, and `scalaz.Validation`.

We've now finished our exploration of the essential parts of functional programming and their support in Scala. We'll learn more concepts when we discuss the type system in Chapter 14 and Chapter 15 and explore advanced concepts in Chapter 16.

Let's now turn to Scala's support for object-oriented programming. We've already covered many of the details in passing. Now we'll complete the picture.