SAMPLE CHAPTER

Scala IN DEPTH

Joshua D. Suereth

FOREWORD BY
Martin Odersky

MANNING

Scala in Depth

by Joshua D. Suereth

Chapter 2

brief contents

2 The core rules

In this chapter

- Using the Scala Read Eval Print Loop
- Expression-oriented programming
- Immutability
- The Option class

This chapter covers a few topics that every newcomer to Scala needs to know. Not every topic is covered in depth, but we cover enough to allow you to explore the subject. You'll learn about the Read Eval Print Loop and how you can use this to rapidly prototype software. Next we'll learn about expression-oriented programming, and how to look at control flow in a different light. From this, we'll spring into immutability and why it can help to greatly simplify your programs, and help them run better concurrently.

2.1 Learn to use the Read Eval Print Loop (REPL)

Scala provides many materials to learn the core language. You can investigate many tutorials, examples, and projects online. But the single most important thing Scala provides is a *Read Eval Print Loop (REPL)*. The REPL is an interactive shell that

compiles Scala code and returns results/type immediately. The Scala REPL is instantiated by running `scala` on the command line, assuming you have Scala installed on your machine and your path is set correctly. The Scala REPL should output something like the following:

```
$ scala
Welcome to Scala version 2.8.0.r21454-b20100411185142
  (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_15).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

From now on, in code examples I'll use the `scala>` prompt to imply that these were entered into the REPL. The following line will be the output. Let's do a few quick samples in the REPL and see what it shows us.

```
scala> "Hello"
res0: java.lang.String = Hello

scala> "Hello".filter(_ != 'l')
res1: String = Heo

scala> "Hello".map(_.toInt + 4)
res2: scala.collection.immutable.IndexedSeq[Int] =
  Vector(76, 105, 112, 112, 115)

scala> "Hello".r
res3: scala.util.matching.Regex = Hello
```

You'll notice that after every statement we enter into the interpreter, it prints a line like `res0: java.lang.String = Hello` (see figure 2.1). The first part of this expression is a variable name for the expression. In the case of these examples, the REPL is defining a new variable for the result of each expression (`res0` through `res3`). The next part of the result expression (after the `:`) is the static type of the expression. The first example has a type of `java.lang.String`, whereas the last has a type of `scala.util.matching.Regex`. The last part of the result expression is the stringified value of the result. This normally comes from calling the `toString` method defined on all classes within the JVM.

As you can see, the REPL is a powerful way to test the Scala language and its type system. Most build tools also include a mechanism to start the REPL with the same classpath as your current working project. This means libraries and compiled classes from your project are available within the REPL. You can make API calls and remote server hits inside the REPL. This can be a great way to test out a web service or REST API in a quick manner. This leads to what I refer to as *experiment-driven development*.

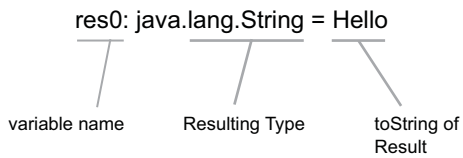


Figure 2.1 REPL return values

2.1.1 Experiment-driven development

Experiment-driven development is where you, the developer, first spend some time experimenting with a live interpreter or REPL before writing tests or production code. This gives you time to fully understand the external pieces of software you're interacting with and get a feel for the comings and goings of data within that API. It's a great way to learn about a new web service or RESTful API that has just been published, that latest Apache library, or even learn about something one of your coworkers have written. After determining the workings of the API, you can then better write your own code. If you also ascribe to test-driven development, this means that you would then write your tests.

Rule 1 **Experiment in the REPL**

Scala provides the REPL tool so every developer can toy around in the language before committing any final code. It's by far the most useful tool in the Scala ecosystem. Development should start inside the REPL in Scala.

There has been a big push for developers to embrace test-driven development (TDD). This is an approach to development where one writes the unit tests first, and then any implementation of those classes. You don't always know what your API should be before you write the tests. Part of TDD is defining the API through the tests. It allows you to see your code in context and get a feel for whether it's something you would want to use. Strongly typed languages can present more issues than dynamic languages with TDD because of expressiveness. Using the REPL, experiment-driven development brings this API definition phase before test generation, allowing a developer to ensure an API is possible in the type system.

Scala is a strongly typed language with flexible syntax, and as such sometimes requires some finagling with the type system to attain the API you desire. Because a lot of developers don't have strong type theory backgrounds, this often requires more experimentation. Experiment-driven development is about experimenting in the REPL with the type system to utilize types as effectively as possible in your API. Experiment-driven design is more about adding larger features or domains into your code, rather than new methods or bug fixes.

Experiment-driven design can also help drastically when defining domain-specific languages (DSLs). A DSL is a pseudo programming language that deals with a particular domain. This language is specific to the domain at hand—for example, querying for data from a database. A DSL may be either internal, as seen in many Scala libraries, or external like SQL. In Scala, it is popular among library developers to create DSLs covering the same domain as the library. For example, the Scala actors library defines a DSL for sending and receiving messages in a thread-safe manner.

One of the challenges when defining a DSL in Scala is to make effective use of the type system. A good type-safe DSL can be expressive and easy to read and can catch many programming errors at compiler time rather then runtime. Also having static knowledge of types can drastically improve performance. The REPL will let you exper-

iment with how to express a particular domain and make sure that expression will compile. When developing Scala, one finds himself adopting the following creative flow:

- Experiment in the REPL with API design
- Copy working API into project files
- Develop unit tests against API
- Adapt code until unit tests pass

When used effectively, experiment-driven development can drastically improve the quality of your API. It will also help you become more comfortable with Scala syntax as you progress. The biggest issue remaining is that not every possible API in Scala is expressible in the REPL. This is because the REPL is interpreted on the fly, and it eagerly parses input.

2.1.2 Working around eager parsing

The Scala REPL attempts to parse input as soon as it possibly can. This, and a few other limitations, means that there are some things that are hard to impossible to express with the REPL. One important function to express are *companion objects and classes*.

A companion object and class are a set of object and class definitions that use the same name. This is easy to accomplish when compiling files; declare the object and class like so:

```
class Foo
```

These statements will also evaluate in the REPL, but they won't function as companions of each other. To prove this, in the following listing let's do something that a companion object can do, that a regular object can't: access private variables on the class.

Listing 2.1 Companion objects in REPL

```
scala>class Foo {
     | private var x = 5
     | }

defined class Foo

scala> object Foo {
     | def im_in_yr_foo(f: Foo) = f.x
     | }
<console>:7: error: variable x cannot be accessed in Foo
       def im_in_yr_foo(f: Foo) = f.x
```

This would compile normally

To fix this issue, we need to embed these objects in some other accessible scope within the interpreter. In the following listing, let's place them inside some scope so we can interpret/compile the class and companion object at the same time:

Listing 2.2 Correct companion object in REPL

```
scala> object holder {
     | class Foo {
     |   private var x = 5
     | }
     | object Foo {
     |   def im_in_yr_foo(f: Foo) = f.x
     | }
     | }
defined module holder

scala> import holder.Foo
import holder.Foo

scala> val x = new Foo
x: holder.Foo = holder$Foo@a5c18ff

scala> Foo.im_in_yr_foo(x)
res0: Int = 5
```

Provides accessible scope

Entire holder object compiled at once

What we've done is create a *holder object*. This gives us our accessible scope, and defers the REPL's compilation until the close of the holder object. We then have to import `Foo` from the holder object. This allows us to test/define companion objects within the REPL.

> **PASTE AND SCALA 2.9.X** Starting in Scala 2.9.x, the REPL supports a `:paste` command, where all code copied into the prompt is compiled in the same run. This provides an alternative to using a container object.

2.1.3 Inexpressible language features

Even working around eager parsing, there are still some language features that the REPL can't reproduce. Most of these issues revolve around packages, package objects, and package visibility restrictions. In particular, you're unable to effectively create a package or package object in the REPL the same way you can within a source file. This also means that other language features dealing with packages, particularly visibility restrictions using the `private` keyword, are also inexpressible. Usually packages are used to namespace your code and separate it from other libraries you might use. This isn't normally needed inside the REPL, but there may be times when you're toying with some advanced feature of Scala—say, package objects and implicit resolution—and you would like to do some experiment-driven development. In this case, you can't express what you want solely in the REPL; see the following listing.

Listing 2.3 Inexpressible language features in the REPL

```
package foo

package object bar {
  private[foo] def baz(...) = ...
}
```

Package definitions

Package objects

Package private

Hope isn't lost. As stated before, most build utilities allow you to create a Scala REPL session against your current project. As a last resort you can toy with some concept in a Scala file, recompile, and restart your REPL session.

A tool known as JRebel (http://mng.bz/8b4t) can dynamically reload class files within a running JVM. The JRebel team has graciously provided free licenses when used with Scala. This tool, combined with some form of continuous compilation, available in most Scala build tools, will allow you to modify your project files and have the changed behavior be immediately available within you REPL session. For the maven-scala-plugin, the details for continuous compilation are located at http://mng.bz/qG78. The Simple Build Tool (http://mng.bz/2f7Q) provides the `cc` target for continuous compilation. Whatever build tool you use to start a REPL session must be integrated with a JRebel classloader so that dynamic class reloading can happen. This technique is a bit detailed and prone to change, so please check your build tool's documentation or the JRebel website for help.

The REPL will allow you to try out Scala code and get a real feel for what you're doing before attempting to create some large complicated system. It's often important in software development to get a slightly more than cursory knowledge of a system before tackling a new feature. The Scala REPL should allow you to do so with a minimal amount of time, allowing you to improve your development skills.

This entire book is enriched with examples of code from the REPL, as it's the best tool to teach and learn Scala. I often find myself running sample programs completely via the REPL before I even create some kind of "main" method, or a unit test, as is standard within Java development. To help encourage this, the book favors demonstrating concepts in the REPL using a few simple scripts. Please feel free to follow along with a REPL of your own.

> **USE THE REPL EVERYWHERE!** No matter what build environment you use, the REPL can dramatically improve your development process. All the major IDEs have support for running the Scala REPL and most of the major build tools. Consult the documentation of the Scala integration for your IDE or build system for details on how to ensure a good REPL experience. For bonus points, use the REPL in combination with a graphical debugger.

The REPL is also a great way to begin learning how to use expressions rather than statements.

Rule 2 **Use expressions not statements**

In Scala, a lot of code can be written as small methods of one expression. This style is not only elegant, but helps in code maintenance.

2.2 Think in expressions

Expression-oriented programming is a term I use to refer to the use of expressions rather than statements in code. What's the difference between an expression and a statement? A statement is something that executes, but an expression is something that evaluates.

What does this mean in practice? Expressions return values. Statements execute code, but there's no value returned. In this section, we'll learn all about expression-oriented programming and how it can help simplify your programs. We'll also look at mutability of objects, and how it interacts with expression-oriented programming.

> **STATEMENT VERSUS EXPRESSION** A statement is something that executes; an expression is something that evaluates to a value.

Expressions are blocks of code that evaluate to a value. In Scala, some control blocks are also expressions. This means that if the control were to branch, each of these branches must evaluate to a value as well. The `if` clause is a great example; this checks a conditional expression and returns one expression or another, depending on the value of the conditional expression. Let's look at a simple REPL session:

```
scala> if(true) "true string" else "false string"
res4: String = true string

scala> if(false) 5 else "hello"
res5: Any = hello
```

As you can see, in Scala an `if` block is an expression. Our first `if` block returns `"true string"`, the true expression. The second `if` block returns *hello*, the result of the false expression. To accomplish something similar in Java, you would use the `?:` syntax as shown in the following:

```
String x = true ? "true string" : "false string"
```

An `if` block in Java is therefore distinct from a `?:` expression in that it doesn't evaluate to a value. You can't assign the result of an `if` block in Java, but Scala has unified the concept of `?:` with its `if` blocks. Scala has no `?:` syntax; you merely use `if` blocks. This is the beginning of expression-oriented programming. In fact, Scala has few statements that do *not* return values from their last expression.

2.2.1 Don't use return

One of the keys to using expressions is realizing that there's no need for a return statement. An expression evaluates to a value, so there's no need to return.

While programming in Java, there was a common practice of having a single point of return for any method. This meant that if there was some kind of conditional logic, the developer would create a variable that contained the eventual return value. As the method flowed, this variable would be updated with what the method should return. The last line in every method would be a return statement. The following listing shows an example.

Listing 2.4 Java idiom: one return statement

```
def createErrorMessage(errorCode: Int) : String = {
   var result : String = _                                  <-- Initialized to default
   errorCode match {
     case 1 =>
```

```
      result = "Network Failure"                    <-- Directly assign result
    case 2 =>
      result = "I/O Failure"
    case _ =>
     result = "Unknown Error"
  }
  return result;
}
```

As you can see, the `result` variable is used to store the final result. The code falls through a pattern match, assigning error strings as appropriate, then returns the `result` variable. We can improve this code slightly by using the expression-oriented syntax that pattern matching allows. A pattern match returns a value. The type of the value is determined as a common super type from all case statement returns. Pattern matching also throws an exception if no pattern is matched, so we're guaranteed a return or error here. The following listing shows the code translated for an expression-oriented pattern match.

Listing 2.5 Updated `createErrorMessage` with expression-oriented pattern match

```
def createErrorMessage(errorCode : Int) : String = {
  val result = errorCode match {                   <-- Assigning pattern match

    case 1 => "Network Failure"                     | Returns
    case 2 => "I/O Failure"                         | expression
    case 3 => "Unknown Error"
  }
  return result
}
```

You'll notice two things. First, we changed the result variable to a `val` and let the type inferencer determine the type. This is because we no longer have to change the `val` after assignment; the pattern match should determine the unique value. Therefore, we reduced the size and complexity of the code, and we increased immutability in the program. *Immutability* refers to the unchanging state of an object or variable; it's the opposite of mutability. *Mutability* is the ability of an object or variable to change or mutate during its lifetime. We'll cover mutability and expression-oriented programming in the next section. You'll frequently find that expression-oriented programming and immutable objects work well together.

The second thing we've done is remove any kind of assignment from the case statements. The last expression in a case statement is the "result" of that case statement. We could have embedded further logic in each case statement if necessary, as long as we eventually had some kind of expression at the bottom. The compiler will also warn us if we accidentally forget to return, or somehow return the wrong type.

The code is looking a lot more concise, but we can still improve it somewhat. In Scala, most developers avoid return statements in their code; they prefer to have the last expression be the return value (similar to all the other expression-oriented styles). In fact, for the `createErrorMessage` method, we can remove the intermediate `result` variable altogether. The following listing shows the final transformation.

Listing 2.6 Final expression-oriented `createErrorMessage` method

```
def createErrorMessage(errorCode: Int) : String = errorCode match {
  case 1 => "Network Failure"
  case 2 => "I/O Failure"
  case _ => "Unknown Error"
}
```

Note how we haven't even opened up a code block for the method? The pattern match is the only statement in the method, and it returns an expression of type `String`. We've completely transformed the method into an expression-oriented syntax. Note how much more concise and expressive the code is. Also note that the compiler will warn us of any type infractions or unreachable case statements.

2.2.2 Mutability

Expression-oriented programming becomes slightly more interesting when mixed with mutability, or the ability to change an object's state during its lifetime. This is because code utilizing mutable objects tends to be written in an imperative style.

Imperative coding is a style that you're probably used to. Many early languages such as C, Fortran, and Pascal are imperative. Imperative code tends to be made of statements, not expressions. Objects are created which have state. Then statements are executed that "mutate" or change the state of an object. In the case of languages that don't have objects, the same mechanisms apply, except with variables and structures. The following listing shows an example of imperative code.

Listing 2.7 Example of imperative style code

```
val x = Vector2D(0.0,0.0)
x.magnify(2.0)
```

Note how a vector is constructed and then mutated via the magnify method. Expression-oriented code prefers having all statements return some expression or value, which would include the move method. In the case of object mutation, what value should be returned? One option is to return the object that was just mutated, as in the following listing.

Listing 2.8 Example mutable expression-oriented method

```
class Vector2D(var x: Double, var y: Double) {
  def magnify(amt: Double) : Vector2D = {
     x *= amt
     y *= amt
     this
  }
}
```

This may seem a great option but has some serious drawbacks. In particular, it can get confusing determining when an object is being mutated, especially when combined with immutable objects. See if you can determine what values should print at the end

of this block of code. Assume that the - method defined on `Vector2D` follows the mathematical definition. Now for the listing.

Listing 2.9 Mixing immutable and mutable objects with expression

```
scala> val x = new Vector2D(1.0, 1.0)
x : Vector2D = Vector2D(1.0,1.0)

scala> val y = new Vector2D(-1.0, 1.0)
y : Vector2D = Vector2D(1.0, 1.0)

scala> x.magnify(3.0) - (x - y).magnify(3.0)
res0 : mutable.Vector2D = ???
```

What is the result of the preceding expression, then? On first look, we would expect it to be the vector (3.0,3.0) minus the vector (6.0,0.0), which is (-3.0,3.0). But each of these variables is mutable. This means that the operations are modifying the variables in the order they're used. Let's evaluate this as it's compiled. First the `x` vector, (1.0,1.0) is magnified by 3 to become (3.0,3.0). Next, we subtract `y` from `x` to give `x` the value (2.0,4.0). Why? Because the right-hand side of the - method must be evaluated next, and `(x-y)` is the first part of this expression. We then magnify `x` by 3.0 again, bringing the value to (6.0,12.0). Finally we subtract `x` from itself, bringing the resulting value to (0.0,0.0). That's right—`x` is subtracted from itself. Why? Because the expression on the left-hand side of the - and the right-hand side of the minus both start with the `x` variable. Because we're using mutability, this means that each expression returns `x` itself. So no matter what we do, we wind up calling `x - x` which results in the vector (0.0, 0.0).

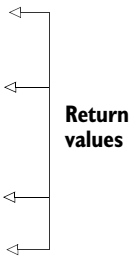
Because of this confusion, it's best to prefer immutability when using objects and expression-oriented programming. This is particularly the case with operator overloading, as with the previous example. Some examples can demonstrate where mutability works well with expression-oriented programming, particularly with pattern matching or `if` statements.

Code has a common task where you need to look up values on an object based on some value. These objects may be immutable or mutable. But expression-oriented programming comes in to simplify the lookup. Let's consider a simple example of looking up the action to perform based on a Menu button click. When we click the Menu button, we receive an event from our event system. This event is marked with the identifier of the button pressed. We want to perform some action and return a status. Let's check out the code in the following listing.

Listing 2.10 Mutable objects and expressions—the right way

```
def performActionForButton(buttonEvent: ButtonEvent,
                       form: Form) : Boolean =
  buttonEvent.getIdentifier match {
     case "SUBMIT" if form.isValid() =>
        try {
          form.submit()
```

```
          true
        } catch {
          case t: FormSubmitError =>
            false
        }
      case "CLEAR" =>
        form.clear()
        true
      case _ =>
        false
  }
```



Note how we're mutating the objects in place and then returning our result. Instead of an explicit return statement, we state the expression we wish to return. You can see the code here is more succinct than creating a variable to hold the result variable. You'll also notice that mixing mutation statements with our expressions has reduced some of the clarity of the code. This is one of the reasons why it's better to prefer immutable code—the topic of our next section.

Expression-oriented programming can reduce boilerplate and provide elegant code. It's accomplished through having all statements return meaningful values. You can now reduce clutter and increase expressiveness within your code.

Expression-oriented programming tends to pair favorably with immutable programming, but less so with mutable objects. Immutability is a term to denote that something doesn't change, in this case the state of an object, once constructed.

2.3 Prefer immutability

Immutability, in programming, refers to the unchanging state of objects after construction. This is one of the capstones of functional programming and a recommended practice for object-oriented design on the JVM. Scala is no exception here and prefers immutability in design, making it the default in many cases. This can be tricky. In this section, you'll learn how immutability can help when dealing with equality issues or concurrent programs.

Rule 3 **Prefer Immutability**

Creating immutable classes drastically reduces the number of potential runtime issues. When in doubt, it's safest to stay immutable.

The most important thing to realize in Scala is that there's a difference between an immutable object and an immutable reference. In Scala, all variables are references to objects. Defining a variable as a `val` means that it's an immutable *reference*. All method parameters are immutable references, and class arguments default to being immutable references. The only way to create a mutable variable is through the `var` syntax. The immutability of the reference doesn't affect whether the object referred to is immutable. You can have a mutable reference to an immutable object and vice versa. This means it's important to know whether the object itself is immutable or mutable.

Determining immutability constraints on objects isn't obvious. In general, it's safe to assume that if the documentation states an object is immutable, then it is;

otherwise, be careful. The Scala standard library helps make the delineation obvious in its collections classes by having parallel package hierarchies, one for immutable classes and one for mutable classes.

In Scala immutability is important because it can help programmers reason through their code. If an object's state doesn't change, then you can determine where objects are created to see where state changes. It can also simplify methods that are based on the state of an object. This benefit is particularly evident when defining equality or writing concurrent programs.

2.3.1 Object equality

One critical reason to prefer immutability is the simplification of object equality. If an object won't change state during its lifetime, one can create an equals implementation that is both deep and correct for any object of that type. This is also critical when creating a hash function for objects. A hash function is one that returns a simplified representation of an object, usually an integer, that can be used to quickly identify the object. A good hash function and equals method are usually paired, if not through code, then in logical definition. If state changes during the lifetime of an object, it can ruin any hash code that was generated for the object. This in turn can affect the equality tests of the object. The following listing shows a simple example of a two-dimensional geometric point class.

Listing 2.11 Mutable `Point2` class

```
class Point2(var x: Int, var y: Int) {
  def move(mx: Int, my: Int) : Unit = {
    x = x + mx
    y = y + my
  }
}
```

The `Point2D` class is simple. It consists of *x* and *y* values, corresponding to locations on the x and y axes. It also has a `move` method, which is used to move the point around the two-dimensional plane. Imagine we want to tie labels to particular points on this 2-D plane, where each label is only a string. To do so, we'd like to use a map of `Point2D` to string values. For efficient lookup, we're going to use a hashing function and a `HashMap`. Let's try the simplest possible thing, hashing with the *x* and *y* variables directly, in the following listing.

Listing 2.12 Mutable `Point2` class with hashing function

```
class Point2(var x: Int, var y: Int) {
  def move(mx: Int, my: Int) : Unit = {
    x = x + mx
    y = y + my
  }
  override def hashCode(): Int = y + (31*x)
}
```

```
scala> val x = new Point2(1,1)
x: Point2 = Point2@20

scala> x.##
res1: Int = 32

scala> val y = new Point2(1,2)
y: Point2 = Point2@21

scala> import collection.immutable.HashMap
import collection.immutable.HashMap

scala> val map = HashMap(x -> "HAI", y -> "ZOMG")
map: scala.collection.immutable.HashMap[
  Point2,java.lang.String] =
  Map((Point2@21,ZOMG), (Point2@20,HAI))

scala> map(x)
res4: java.lang.String = HAI

scala> val z = new Point2(1,1)
z: Point2 = Point2@20

scala> map(z)
java.util.NoSuchElementException: key not found: Point2@20
...
```

Things appear to be working exactly as we want—until we attempt to construct a new point object with the same values as point *x*. This point should hash into the same section of the map, but the equality check will fail because we haven't created our own equality method. By default, Scala uses object location equality and hashing, but we've only overridden the hash code. Object location equality is using the address in memory for an object as the only factor to determine if two objects are equal. In our `Point2` case, object location equality can be a quick check for equality, but we can also make use of the *x* and *y* locations to check for equality.

You may have noticed that the `Point2` class overrides the `hashCode` method, but I'm calling the `##` method on the instance `x`. This is a convention in Scala. For compatibility with Java, Scala utilizes the same `equals` and `hashCode` methods defined on `java.lang.Object`. But Scala also abstracts primitives such that they appear as full objects. The compiler will box and unbox the primitives as needed for you. These primitive-like objects are all subtypes of `scala.AnyVal` whereas "standard" objects, those that would have extended `java.lang.Object`, are subtypes of `scala.AnyRef`. `scala.AnyRef` can be considered an alias for `java.lang.Object`. As the `hashCode` and `equals` methods are defined on `AnyRef`, Scala provides the methods `##` and `==` that you can use for both `AnyRef` and `AnyVal`.

HASHCODE AND EQUALS SHOULD ALWAYS BE PAIRED The `equals` and `hashCode` methods should always be implemented such that if `x == y` then `x.## == y.##`.

Let's implement our own equality method in the following listing and see what the results are.

Listing 2.13 Mutable `Point2` class with hashing and equality

```
class Point2(var x: Int, var y: Int) extends Equals {
  def move(mx: Int, my: Int) : Unit = {
    x = x + mx
    y = y + my
  }
  override def hashCode(): Int = y + (31*x)
  def canEqual(that: Any): Boolean = that match {
     case p: Point2 => true
     case _ => false
  }
  override def equals(that: Any): Boolean = {
    def strictEquals(other: Point2) =
       this.x == other.x && this.y == other.y
    that match {
      case a: AnyRef if this eq a => true
      case p: Point2 => (p canEqual this) && strictEquals(p)
      case _ => false
    }
  }
}
```

```
scala> val x = new Point2(1,1)
x: Point2 = Point2@20
```

```
scala> val y = new Point2(1,2)
y: Point2 = Point2@21
```

```
scala> val z = new Point2(1,1)
z: Point2 = Point2@20
```

```
scala> x == z
res6: Boolean = true
```

```
scala> x == y
res7: Boolean = false
```

The implementation of `equals` may look strange, but will be covered in more detail in section 2.5.2. For now, note that the `strictEquals` helper method compares the *x* and *y* values directly. This means that two points are considered equal if they are in the same location. We've now tied our `equals` and `hashCode` methods to the same criteria, the `x` and `y` values. Let's throw our *x* and *y* values into a `HashMap` again, only this time we're going to move the *x* value, and see what happens to the label attached to it.

Listing 2.14 Mutating `Point2` with `HashMap`

```
scala> val map = HashMap(x -> "HAI", y -> "WORLD")
map: scala.collection.immutable.HashMap[Point2,java.lang.String] =
   Map((Point2@21,WORLD), (Point2@20,HAI))
```

```
scala> x.move(1,1)
```

```
scala> map(y)
res9: java.lang.String = WORLD
```

```
scala> map(x)
java.util.NoSuchElementException: key not found: Point2@40
...

scala> map(z)
java.util.NoSuchElementException: key not found: Point2@20
...
```

What happened to the label attached to *x*? We placed it into the `HashMap` when *x* has a value of (1,1). This means it had a hash code of 32. We then move *x* to (2,2), changing its hash code to 64. Now when we try to look up the label in the map using *x*, it can't be found because *x* was encoding with the hash bucket of 32, and it's looking in the hash bucket for 64. Well, what if we try to look up the value using a new point, *z*, that still has a hash code of 32? It also fails, because *x* and *z* aren't equal according to our rules. You see, a `HashMap` uses the hash at the time of insertion to store values but doesn't update when an object's state mutates. This means we've lost our label for *x* when using hash-based lookup, but we can still retrieve the value when traversing the map or using traversal algorithms:

```
scala> map.find( _._1 == x)
res13: Option[(Point2, java.lang.String)] = Some((Point2@40,HAI))
```

As you can see, this behavior is rather confusing, and can cause no end of strife when debugging. As such, it's generally recommended to ensure the following constraints when implementing equality:

- If two objects are equal, they should have the same `hashCode`.
- A `hashCode` computed for an object won't change for the life of the object.
- When sending an object to another JVM, equality should be determined using attributes available in both JVMs.

As you can see, the second constraint implies that all criteria used in creating a `hashCode` should *not* change with the life of an object. The last statement, when applicable, means that an object's hash and equals method should be computed using its own internal state. Combine this with the first statement, and you find that the only way to satisfy these requirements is through the use of immutable objects. If the state of an object never changes, it's acceptable to use it in computing a hash code or when testing equality. You can also serialize the object to another JVM and continue to have a consistent hash code and equality.

You may be wondering, why do I care about sending objects to other JVMs? My software will never run on more than one JVM. In fact, my software runs on a mobile device, where resources are critical. The problem with that thinking is that serializing an object to another JVM need not be done in real time. I could save some program state to disk and read it back later. This is effectively the same as sending something to another JVM. Although you may not be directly sending it over the network, you're sending it through time, where the JVM of today is the writer of data, and the JVM started tomorrow is the user of the data. In these instances, having a hash code and equals implementation is critical.

The last constraint makes immutability a necessity. Remove this constraint, and there are only two simple ways to satisfy the first two constraints:

- Utilize only immutable object internal state in `hashCode` computation
- Use default concepts for equals and `hashCode`

As you can see, this means that *something* in the object must be immutable. Making the entire object immutable simplifies this whole process greatly.

2.3.2 Concurrency

Immutability doesn't merely simplify object equality; it can also simplify concurrent access of data. Programs are becoming increasingly parallelized, and processors are splitting into multiple cores. The need to run concurrent threads of control in programs is growing across all forms of computing. Traditionally, this meant using creative means to protect access to shared data across these various threads of control. Protected mutable data usually means some form of locking. Immutability can help share state while reducing the need for locking.

Locking entails a performance overhead. Threads that wish to read data can't do so unless the lock is available to obtain. Even using read-write locks can cause issues, because a writer may be slow in preventing readers from accessing the data they desire. On the JVM, there are optimizations in the JIT to attempt to avoid locks when they aren't necessary. In general, you want to have as few locks in your software as possible, but you want enough to encourage a high degree of parallelism. The more you can design your code to avoid locking the better. For instance, let's try to measure the effect of locking on an algorithm and see if we can design a new algorithm that reduces the amount of locking.

We'll create an index service that we can query to find particular items by their key. The service will also allow users to add new items into the index. We expect to have many users looking up values and a smaller amount of users adding additional content to the index. Here's the initial interface:

```
trait Service[Key,Value] {
  def lookUp(k: Key): Option[Value]
  def insert(k: Key, v: Value): Unit
}
```

The service is made up of two methods: `lookUp`, which will look up values in the index by the key, and `insert`, which will insert new values into the service. This service is like a map of key-to-value pairs. Let's implement this using a locking and a mutable `HashMap`.

```
import collection.mutable.{HashMap=>MutableHashMap}

class MutableService[Key, Value] extends Service[Key, Value] {
  val currentIndex = new MutableHashMap[Key, Value]
  def lookUp(k: Key): Option[Value] = synchronized(currentIndex.get(k))
  def insert(k: Key, v: Value): Unit = synchronized {
    currentIndex.put(k,v)
  }
}
```

This class contains three members. The first is the `currentIndex`, which is a reference to the mutable `HashMap` that we use to store values. The `lookUp` and `insert` methods are both surrounded by a `synchronized` block, which synchronizes against the `MutableService`. You'll notice that all operations on a `MutableService` require locking. But given what was stated about the usage of this service, the `lookUp` method will be called far more often than the `insert` method. A read-write lock could help in this situation, but let's look at using immutability instead.

We'll change the `currentIndex` to be an `ImmutableHashMap` that get overwritten when the `insert` method is called. The `lookUp` method can then be free of any locking, as shown in the following code:

```
class ImmutableService[Key, Value] extends Service[Key, Value] {
  var currentIndex = new ImmutableHashMap[Key,Value]
  def lookUp(k: Key): Option[Value] = currentIndex.get(k)
  def insert(k: Key, v: Value): Unit = synchronized {
    currentIndex = currentIndex + ((k, v))
  }
}
```

The first thing to notice is that the `currentIndex` is a mutable reference to an immutable variable. We update this reference every time there's an insert operation. The second thing to notice is that this service isn't completely immutable. All that's happened is the reduction of locking by utilizing an immutable `HashMap`. This simple change can cause a drastic improvement in running time.

I've set up a simple micro-performance benchmark suite for these two classes. The basics of the suite are simple. We construct a set of tasks that will write items into the service and a set of tasks that will attempt to read items from the index. We then interleave the two sets of tasks and submit them to a queue of two threads for execution. We time the speed that this entire process takes and record the results. Figure 2.2 shows some worst-case results.

The y-axis is the execution time of running a test. The x-axis corresponds to the number of `insert/lookUp` tasks submitted to the thread pools. You'll notice that the mutable service's execution time grows faster than the immutable service's execution time. This graph certainly shows that extra locking can severely impact performance. But note that the execution times of this test can greatly vary. Due to the uncertainty of parallelism, this graph could look anywhere from the one shown above to a graph where the immutable service and mutable service execution times track relatively the same. In general, the `MutableService` implementation was slower than the `ImmutableService`, but don't judge performance from one graph or on execution alone.

Figure 2.3 shows another graph where you can see, for one particular test, the `MutableService` had all of its stars align and ran with a drastically reduced locking overhead. You can see in the preceding run where a single test case had all its timing align so that the `MutableService` could outperform the `ImmutableService`. Though possible for this specific case, the general case involved the `ImmutableService` outperforming the `MutableService`. If the assumptions stated here hold true for a real-life program,

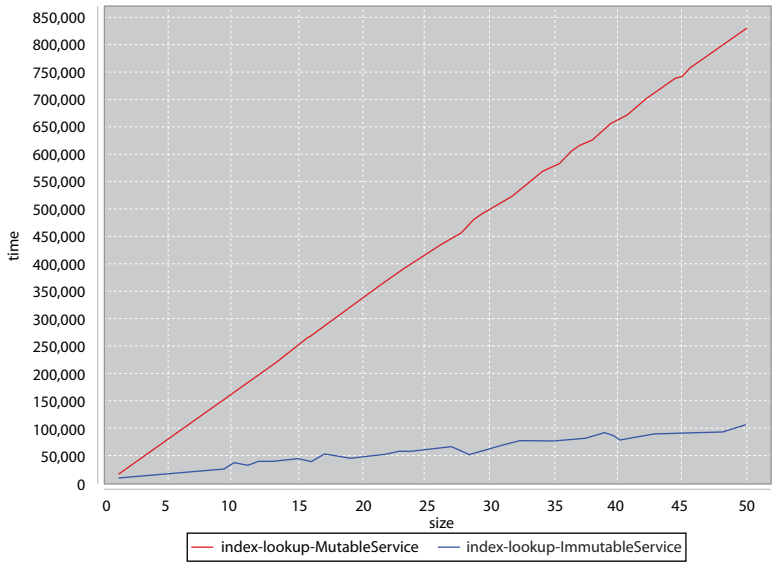


Figure 2.2
Immutable versus mutable service worst-case scenario

it appears that the `ImmutableService` will perform better in the general case and not suffer from random contention slowdowns.

The most important thing to realize is that immutable objects can be passed among many threads without fear of contention. The ability to remove locks, and all the potential bugs associated with them, can drastically improve the stability of a codebase. Combined with the improved reasoning one can get, as seen with the equals method, immutability is something to strive to maintain within a codebase.

Immutability can ease concurrent development by reducing the amount of protection a developer must use when interacting with immutable objects. Scala also

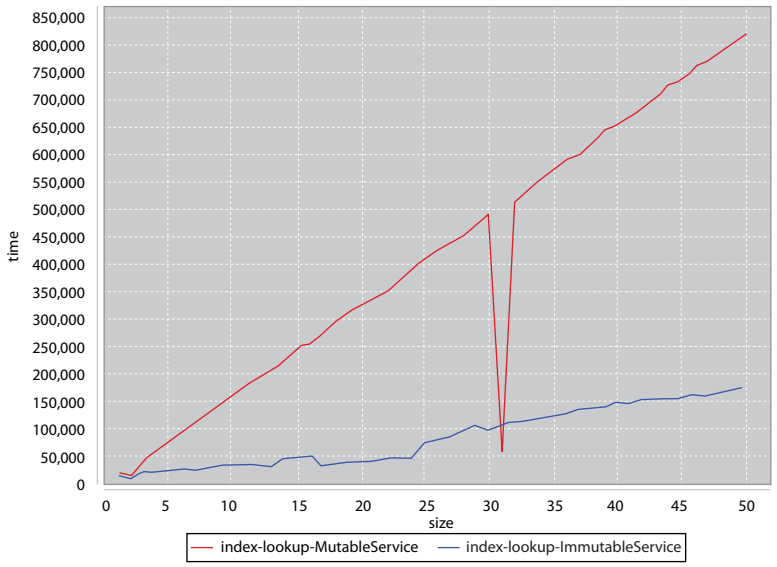


Figure 2.3
Immutable versus mutable service "one golden run" scenario

provides a class called `Option` that allows developers to relax the amount of protection they need when dealing with `null`.

2.4 Use None instead of null

Scala does its best to discourage the use of null in general programming. It does this through the `scala.Option` class found in the standard library. An `Option` can be considered a container of something or nothing. This is done through the two subclasses of `Option`: `Some` and `None`. `Some` denotes a container of exactly one item. `None` denotes an empty container, a role similar to what `Nil` plays for `List`.

Rule 4 **Use None instead of null**

While it was habit in Java to initialize values to null, Scala provides an `Option` type for the same purpose. `Option` is self-documenting for developers and, used correctly, can prevent unintended null pointer exceptions when using Scala.

In Java, and other languages that allow `null`, `null` is often used as a placeholder to denote a nonfatal error as a return value or to denote that a variable isn't yet initialized. In Scala, one can denote this through the `None` subclass of `Option`. Conversely, one can denote an initialized, or nonfatal variable state through the `Some` subclass of `Option`. Let's look at the usage of these two classes in the following listing.

Listing 2.15 Simple usage of `Some` and `None`

```
scala> var x : Option[String] = None
x: Option[String] = None

scala> x.get
java.util.NoSuchElementException: None.get in

scala> x.getOrElse("default")
res0: String = default

scala> x = Some("Now Initialized")
x: Option[String] = Some(Now Initialized)

scala> x.get
res0: java.lang.String = Now Initialized

scala> x.getOrElse("default")
res1: java.lang.String = Now Initialized
```

An `Option` containing no value can be constructed via the `None` object. An `Option` that contains a value is created via the `Some` factory method. `Option` provides many differing ways of retrieving values from its inside. Of particular use are the `get` and `getOrElse` methods. The `get` method will attempt to access the value stored in an `Option` and will throw an exception if it's empty. This is similar to accessing nullable values within other languages. The `getOrElse` method will attempt to access the value stored in an `Option`, if one exists; otherwise it will return the value supplied to the method. You should always prefer `getOrElse` over using `get`.

Scala provides a factory method on the `Option` companion object that will convert from a Java style reference, where `null` implies an empty variable, into an `Option` where this is more explicit. Let's take a quick look in the following listing.

Listing 2.16 Usage of the `Option` factory

```
scala> var x : Option[String] = Option(null)
x: Option[String] = None

scala> x = Option("Initialized")                    <-- Option.apply("Initialized")
x: Option[String] = Some(Initialized)
```

The `Option` factory method will take a variable and create a `None` object if the input was `null`, or it will create a `Some` if the input was initialized. This makes it rather easy to take inputs from an untrusted source—that is, another JVM language—and wrap them into `Options`. You might be asking yourself why you would want to do this. Isn't checking for `null` just as simple in code? `Option` provides advanced features that make it far more ideal than using `null` and `if` checks.

2.4.1 Advanced Option techniques

The greatest feature of `Option` is that you can treat it as a collection. This means you can perform the standard `map`, `flatMap`, and `foreach` methods, as well as utilize it inside a `for` expression. This helps ensure a nice concise syntax, and it opens a variety of different methods to handling uninitialized values. Let's look at some common issues solved using `null` and their solutions using `Option`, starting with creating an object or returning a default.

CREATE AN OBJECT OR RETURN A DEFAULT

You'll have many times in code when you'll need to construct something if some other variable exists, or supply some sort of default. Let's pretend that we have an application that requires some kind of temporary file storage for its execution. The application is designed so that a user may optionally specify a directory to store temporary files on the command line. If the user doesn't specify a new file, if the argument provided by the user is not a real directory, or if they didn't provide a directory, then we want to return a sensible default temporary directory. The following listing shows a method that will return this temporary directory:

Listing 2.17 Creating an object or returning a default

```
def getTemporaryDirectory(tmpArg: Option[String]): java.io.File = {
   tmpArg.map(name => new java.io.File(name)).          <-- Create if defined
       filter(_.isDirectory).                           <-- Only directories
       getOrElse(new java.io.File(                      <-- Specify default
         System.getProperty("java.io.tmpdir")))
}
```

The `getTemporaryDirectory` method takes the command-line parameter as an `Option` containing a `String` and returns a `File` object referencing the temporary directory we should use. The first thing we do is use the `map` method on `Option` to create a `java.io.File` if there was a parameter. Next, we make sure that this newly constructed file object is a directory. To do that, we use the `filter` method. This will check whether the value in an `Option` abides by some predicate, and if not, convert to a `None`. Finally, we check to see if we have a value in the `Option`; otherwise we return the default temporary directory.

This enables a powerful set of checks without resorting to nested `if` statements or blocks. Sometimes we would like a block, such as when we want to execute a block of code based on the availability of a particular parameter.

EXECUTE BLOCK OF CODE IF VARIABLE IS INITIALIZED

`Option` can be used to execute a block of code if the `Option` contains a value. This is done through the `foreach` method, which, as expected, iterates over all the elements in the `Option`. As an `Option` can only contain zero or one value, this means the block either executes or is ignored. This syntax works particularly well with `for` expressions. Let's take a look at the following listing.

Listing 2.18 Executing code if option is defined

```
val username: Option[String] = ...

for(uname <- username) {
   println("User: " + uname)
}
```

As you can see, this looks like a normal "iterate over a collection" control block. The syntax remains similar when we need to iterate over several variables. Let's look at the case where we have some kind of Java servlet framework, and we want to be able to authenticate users. If authentication is possible, we want to inject our security token into the `HttpSession` so that later filters and servlets can check access privileges for this user, as in the following listing.

Listing 2.19 Executing code if several options are defined

```
def authenticateSession(session: HttpSession,
                        username: Option[String],
                        password: Option[Array[Char]]) = {
   for(u <- username;
       p <- password;
       if canAuthenticate(u, p)) {
     val privileges = privilegesFor(u)
     injectPrivilegesIntoSession(session, privileges)
   }
}
```

Conditional logic

No need for Option

Note that you can embed conditional logic in a `for` expression. This helps keep less nested logical blocks within your program. Another important consideration is that all the helper methods do *not* need to use the `Option` class. `Option` works as a great

front-line defense for potentially uninitialized variables, but it doesn't need to pollute the rest of your code. In Scala, `Option` as an argument implies that something may not be initialized. The opposite should be true as well. If a method takes a value that is not labeled as an `Option`, you should not pass it null or uninitialized parameters.

Scala's `for` expression syntax is rather robust, even allowing you to produce values, rather then execute code blocks. This is handy when you have a set of potentially uninitialized parameters that you want to transform into something else.

USING POTENTIAL UNINITIALIZED VARIABLES TO CONSTRUCT ANOTHER VARIABLE

Sometimes we want to transform a set of potentially uninitialized values so that we have to deal with only one. To do this, we need to use a `for` expression again, but this time using a `yield`. The following listing shows a case where a user has input some database credentials, or we attempted to read them from an encrypted location, and we want to create a database connection using these parameters. We don't want to deal with failure in our function, as this is a utility function that won't have access to the user. In this case, we'd like to transform our database connection configuration parameters into a single option containing our database.

Listing 2.20 Merging options

```
def createConnection(conn_url: Option[String],
                     conn_user: Option[String],
                     conn_pw: Option[String]) : Option[Connection] =
  for {
    url <- conn_url
    user <- conn_user
    pw <- conn_pw
  } yield DriverManager.getConnection(url, user, pw)
```

This function does exactly what we need it to. It does seem, though, that we're merely deferring all logic to `DriverManager.getConnection`. What if we wanted to abstract this such that we can take *any* function and create one that's option-friendly in the same manner? The following listing shows what we'll call the "lift" function.

Listing 2.21 Generically converting functions

```
scala>  def lift3[A,B,C,D](
     |  f: Function3[A,B,C,D]): Function3[Option[A], Option[B],
     |                                    Option[C], Option[D]] = {
     |     (oa : Option[A], ob : Option[B], oc : Option[C]) =>
     |        for(a <- oa; b <- ob; c <- oc) yield f(a,b,c)
     |   }
lift3: [A,B,C,D](f: (A, B, C) => D)(Option[A],
                                    Option[B],
                                    Option[C]) => Option[D]

scala> lift3(DriverManager.getConnection)
res4: (Option[java.lang.String],
       Option[java.lang.String],
       Option[java.lang.String]) => Option[java.sql.Connection] =
  <function3>
```

Using lift3 directly

The `lift3` method looks somewhat like our earlier `createConnection` method, except that it takes a function as its sole parameter. The `Function3` trait represents a function that takes three arguments and returns a result. The `lift3` function takes a function of three arguments as input and outputs a new function of three arguments. As you can see from the REPL output, we can use this against existing functions to create option-friendly functions. We've directly taken the `DriverManager.getConnection` method and lifted it into something that's semantically equivalent to our earlier `createConnection` method. This technique works well when used with the "encapsulation" of uninitialized variables. You can write most of your code, even utility methods, assuming that everything is initialized, and then lift these functions into `Option`-friendly variants when needed.

One important thing to mention is that `Option` derives its equality and `hashCode` from what it contains. In Scala, understanding equality and `hashCode`, especially in a polymorphic setting, is very important.

2.5 Polymorphic equality

Let's discuss how to properly implement an `equals` and `hashCode` function in Scala. This can be tricky in a polymorphic language, but can be done by following some basic rules. In general, it's best to avoid having multiple concrete levels with classes that also need equality stronger then referential equality. In some cases, classes only need referential equality, the ability to differentiate two objects to determine if they're the same instance. But if the equality comparison needs to determine if two differing instances are equivalent *and* there are multiple concrete hierarchies, then things get a bit more tricky.

To understand this issue, we'll look at how to write a good equality method.

2.5.1 Example: A timeline library

We'd like to construct a time line, or calendar, widget. This widget needs to display dates, times, and time ranges as well as associated events with each day. The fundamental concept in this library is going to be an `InstantaneousTime`.

`InstantaneousTime` is a class that represents a particular discrete time within the time series. We could use the `java.util.Date` class, but we'd prefer something that's immutable, as we've just learned how this can help simplify writing good `equals` and `hashCode` methods. In an effort to keep things simple, let's have our underlying time storage be an integer of seconds since midnight, January 1, 1970, Greenwich Mean Time on a Gregorian calendar. We'll assume that all other times can be formatted into this representation and that time zones are an orthogonal concern to representation. We're also going to make the following common assumptions about our equality usage in the application:

- When `equals` is called and it will return `true`, it's because both objects are the same reference.
- Most calls to `equals` result in a return of `false`.

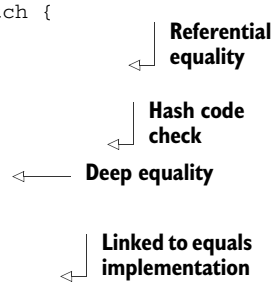
- Our implementation of `hashCode` is sufficiently sparse that for most equality comparisons, the `hashCode`s will be different.
- Computing a `hashCode` is more efficient than a deep equality comparison.
- Testing referential equality is more efficient than a deep equality comparison.

These assumptions are standard for most equality implementations. They might not always hold for your application. Let's take a first crack at the class and a simple equals and `hashCode` method pair in the following listing, and see what this looks like.

Listing 2.22 Simple `InstantaneousTime` class

```
trait InstantaneousTime {
  val repr: Int

  override def equals(other: Any) : Boolean = other match {
    case that: InstantaneousTime =>
       if(this eq that) {
         true
       } else {
         (that.## == this.##) &&
         (repr == that.repr)
       }
    case _ => false
  }
  override def hashCode() : Int = repr.##
}
```



The class contains only one member, `repr`, which is a number representing the seconds since midnight, January 1, 1970 Greenwich Mean Time. As this is the only data value in the class, and it's immutable, `equals` and `hashCode` will be based on this value. When implementing an `equals` method within the JVM, it's usually more performant to test referential equality before doing any sort of deep equality check. In the case of this class, it's not necessary. For a sufficiently complex class, it can drastically help performance, but this class doesn't need it. The next piece to a good `equals` method is usually using the `hashCode` for an early false check. Given a sufficiently sparse and easy to compute `hashCode`, this would be a good idea. Once again, in this class it's not necessary, but in a sufficiently complex class, this can be performant.

> **## AND == VS. EQUALS AND HASHCODE** In Scala, the `##` method is equivalent to the `hashCode` method in Java as the `==` method is equivalent to the `equals` method in Java. In Scala, when *calling* the `equals` or `hashCode` method it's better to use `##` and `==`. These methods provide additional support for value types. But the `equals` and `hashCode` method are used when *overriding* the behavior. This split provides better runtime consistency and still retains Java interoperability.

This class helps us illustrate two principles: the importance of a good equality method and always challenge the assumptions of your code. In this case, the "best practice" equality method, while great for a sufficiently complex class, provides little benefit for this simple class.

NOTE When implementing equality for your own classes, test the assumptions in the standard equality implementation to make sure they hold true.

Our implementation of equals suffers from yet another flaw, that of polymorphism.

2.5.2 Polymorphic equals implementation

In general, it's best to avoid polymorphism with types requiring deep equality. Scala no longer supports subclassing case classes for this very reason. But there are still times in code where this is useful or even necessary. To do so, we need to ensure that we've implemented our equality comparisons correctly, keeping polymorphism in mind and utilizing it in our solution.

Let's create a subclass of `InstantaneousTime` that also stores labels. This is the class we'll use to save events in our timeline, so we'll call it `Event`. We'll make the assumption that events on the same day will hash into the same bucket, and hence have the same `hashCode`, but equality will also include the name of the event. Let's take a crack at an implementation in the following listing.

Listing 2.23 Event subclass of `InstantaneousTime`

```
trait Event extends InstantaneousTime {
  val name: String
  override def equals(other: Any): Boolean = other match {
    case that: Event =>
      if(this eq that) {
        true
      } else {
        (repr == that.repr) &&
        (name == that.name)
      }
    case _ => false
  }
}
```

Quick referential check

Deep equals using all

We've dropped the `hashCode` early exit in our code, as checking the `repr` member is just as performant in our particular class. The other thing you'll notice is that we've changed the pattern match so that only two `Event` objects can be equal to each other. Let's try to use this in the REPL in the following listing.

Listing 2.24 Using `Event` and `InstantaneousTime`

```
scala> val x = new InstantaneousTime {
     | val repr = 2
     | }
x: java.lang.Object with InstantaneousTime = $anon$1@2

scala> val y = new Event {
     | val name = "TestEvent"
     | val repr = 2
     | }
y: java.lang.Object with Event = $anon$1@2

scala> y == x
```

Subclass to original

```
res8: Boolean = false

scala> x == y                                          <-- Original to subclass
res9: Boolean = true
```

Rule 5 **Use scala.Equals for polymorphic equality**

Polymorphic equality is easy to mess up. scala.Equals provides a template to make it easier to avoid mistakes.

What's happened? The old class is using the old implementation of the equality method, and therefore doesn't check for the new name field. We need to modify our original equality method in the base class to account for the fact that subclasses may wish to modify the meaning of equality. In Scala, there's a `scala.Equals` trait that can help us fix this issue. The `Equals` trait defines a `canEqual` method that's used in tandem with the standard `equals` method. The `canEqual` method allows subclasses to opt out of their parent classes' equality implementation. This is done by allowing the `other` parameter in the `equals` method an opportunity to cause an equality failure. To do so, we override `canEqual` in our subclass with whatever rejection criteria our overridden `equals` method has. Let's modify our classes to account for polymorphism using these two methods in the following listing.

Listing 2.25 Using `scala.Equals`

```
trait InstantaneousTime extends Equals {
  val repr: Int
  override def canEqual(other: Any) =
    other.isInstanceOf[InstantaneousTime]                <-- Allows any subclass
  override def equals(other: Any) : Boolean =
    other match {
      case that: InstantaneousTime =>
        if(this eq that) true else {
             (that.## == this.##) &&
             (that canEqual this) &&                     <-- Call other object's canEqual
             (repr == that.repr)
        }
    case _ => false
  }
  override def hashCode(): Int = repr.hashCode
}

trait Event extends InstantaneousTime {
  val name: String
  override def canEqual(other: Any) =                    <-- Subclass opt out of equality canEqual
    other.isInstanceOf[Event]
  override def equals(other: Any): Boolean = other match {
    case that: Event =>
      if(this eq that) {
        true
      } else {
        (that canEqual this) &&
        (repr == that.repr) &&
        (name == that.name)
```

```
      }
    case _ => false
  }
}
```

The first thing to do is implement `canEqual` on `InstantaneousTime` to return `true` if the other object is also an `InstantaneousTime`. Next let's account for the other object's `canEqual` result in the equality implementation. Finally, an overridden `canEqual` in the `Event` class will only allow equality with other `Event`s.

WHEN OVERRIDING EQUALITY OF A PARENT CLASS, ALSO OVERRIDES CANEQUAL
The `canEqual` method is a lever, allowing subclasses to opt out of their parent class's equality implementation. This allows a subclass to do so without the usual dangers associated with a parent class `equals` method returning `true` while a subclass would return `false` for the same two objects.

Let's look at our earlier REPL session and see if the new `equals` methods behave better, as in the following listing.

Listing 2.26 Using new `equals` and `canEquals` methods

```
scala> val x = new InstantaneousTime {
     | val repr = 2
     | }
x: java.lang.Object with InstantaneousTime = $anon$1@2

scala> val y = new Event {
     | val name = "TestEvent"
     | val repr = 2
     | }
y: java.lang.Object with Event = $anon$1@2

scala> y == x
res10: Boolean = false

scala> x == y                                    <-- No longer returns true
res11: Boolean = false
```

We've succeeded in defining an appropriate equality method. We can now write a general `equals` method that performs well with general assumptions about our programs, and we can handle the case where our classes are also polymorphic.

2.6 Summary

In this chapter, we looked at the first crucial items for using Scala. Leveraging the REPL to do rapid prototyping is crucial to any successful Scala developer. Favoring expression-oriented programming and immutability helps simplify a program and improve the ability to reason through code. `Option` can also help improve reasonability of the code by clearly delineating where uninitialized values are accepted. Also, writing a good equality method in the presence of polymorphism can be difficult. All of these practices can help make the first steps in Scala successful. For continued success, let's look at code style and how to avoid running into issues with Scala's parser.

SCALA/PROGRAMMING

Scala IN DEPTH

Joshua D. Suereth

Scala is a powerful JVM language that blends the functional and OO programming models. You'll have no trouble getting introductions to Scala in books or online, but it's hard to find great examples and insights from experienced practitioners. You'll find them in **Scala in Depth**.

What's Inside

- Concise, expressive, and readable **code style**
- **Integrate** Scala into your existing Java projects
- Scala's 2.8.0 **collections** API
- How to use **actors** for concurrent programming
- Mastering the Scala **type system**
- Scala's **OO** features—type member inheritance, multiple inheritance, and composition
- **Functional** concepts and patterns—immutability, applicative functors, and monads

There's little heavy-handed theory here—just dozens of crisp, practical techniques for coding in Scala. Written for readers who know Java, Scala, or another OO language.

Josh Suereth is a software developer with Typesafe. He is a Scala committer and the maintainer of scala-tools.org.

To download their free eBook in PDF and mobile formats, owners of this book should visit manning.com/ScalainDepth

Free eBook
SEE INSERT

"Authorative and understandable."
—From the Foreword by Martin Odersky, Creator of Scala

"Takes you deep into the inner workings of Scala."
—John C Tyler, PROS Pricing

"By far the best examples I've seen in any technical book."
—Eric Weinberg, Wagger Designs

"An eye opener! Now I know why Scala does what it does."
—John Griffin, Coauthor of *Hibernate Search in Action*

ISBN 13: 978-1-935182-70-2
ISBN 10: 1-935182-70-6
54999
9 781935 182702

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]