Chapter 30

Object Equality

Comparing two values for equality is ubiquitous in programming. It is also more tricky than it looks at first glance. This chapter looks at object equality in detail and gives some recommendations to consider when you design your own equality tests.

30.1 EQUALITY IN SCALA

As mentioned in Section 11.2, the definition of equality is different in Scala and Java. Java has two equality comparisons: the == operator, which is the natural equality for value types and object identity for reference types, and the equals method, which is (user-defined) canonical equality for reference types. This convention is problematic because the more natural symbol,==, does not always correspond to the natural notion of equality. When programming in Java, a common pitfall for beginners is to compare objects with == when they should be compared with equals. For instance, comparing two strings x and y using "x == y" might yield false in Java, even if x and y have exactly the same characters in the same order.

Scala also has an equality method signifying object identity, but it is not used much. That kind of equality, written "x eq y", is true if x and y reference the same object. The == equality is reserved in Scala for the "natural" equality of each type. For value types, == is value comparison, just like in Java. For reference types, == is the same as equals in Scala. You can redefine the behavior of == for new types by overriding the equals method, which is always inherited from class Any. The inherited equals, which takes effect unless overridden, is object identity, as is the case in Java. So equals (and with it, ==) is by default the same as eq, but you can change its behavior by overriding the equals method in the classes you define. It is not possible to override == directly, as it is defined as a final method in class Any. That is, Scala treats == as if it were defined as follows in class Any:

```
final def == (that: Any): Boolean =
  if (null eq this) {null eq that} else {this equals that}
```

30.2 WRITING AN EQUALITY METHOD

How should the equals method be defined? It turns out that writing a correct equality method is surprisingly difficult in object-oriented languages. In fact, after studying a large body of Java code, the authors of a 2007 paper concluded that almost all implementations of equalsmethods are faulty.[1] This is problematic because equality is at the basis of many other things. For one, a faulty equality method for a type C might mean that you cannot reliably put an object of type C in a collection.

For example, you might have two elements, elem1 and elem2, of type C which are equal (*i.e.*, "elem1 equals elem2" yields true). Nevertheless, with commonly occurring faulty implementations of the equals method, you could still see behavior like the following:

```
var hashSet: Set[C] = new collection.immutable.HashSet
```

```
hashSet += elem1
hashSet contains elem2    // returns false!
```

Here are four common pitfalls[2] that can cause inconsistent behavior when overridingequals:

1. Defining equals with the wrong signature.
2. Changing equals without also changing hashCode.
3. Defining equals in terms of mutable fields.
4. Failing to define equals as an equivalence relation.

These four pitfalls are discussed in the remainder of this section.

Pitfall #1: Defining equals with the wrong signature

Consider adding an equality method to the following class of simple points:

```
class Point(val x: Int, val y: Int) { ... }
```

A seemingly obvious but wrong way would be to define it like this:

```
// An utterly wrong definition of equals
def equals(other: Point): Boolean =
  this.x == other.x && this.y == other.y
```

What's wrong with this method? At first glance, it seems to work OK:

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@37d7d90f
p2: Point = Point@3beb846d

scala> val q = new Point(2, 3)
q: Point = Point@e0cf182

scala> p1 equals p2
res0: Boolean = true

scala> p1 equals q
res1: Boolean = false
```

However, trouble starts once you start putting points into a collection:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val coll = mutable.HashSet(p1)
coll: scala.collection.mutable.HashSet[Point] =
Set(Point@37d7d90f)

scala> coll contains p2
res2: Boolean = false
```

How to explain that coll does not contain p2, even though p1 was added to it, and p1 and p2 are equal objects? The reason becomes clear in the following interaction, where the precise type of one of the compared points is masked. Define p2a as an alias of p2, but with type Any instead of Point:

```
scala> val p2a: Any = p2
p2a: Any = Point@3beb846d
```

Now, were you to repeat the first comparison, but with the alias p2a instead of p2, you would get:

```
scala> p1 equals p2a
res3: Boolean = false
```

What went wrong? The version of equals given previously does not override the standard method equals because its type is different. Here is the type of the equals method as it is defined in the root class Any:[3]

```
def equals(other: Any): Boolean
```

Because the equals method in Point takes a Point instead of an Any as an argument, it does not override equals in Any. Instead, it is just an overloaded alternative. Now, overloading in Scala and in Java is resolved by the static type of the argument, not the run-time type. So as long as the static type of the argument is Point, the equals method in Point is called. However, once the static argument is of type Any, the equals method in Any is called instead. This method has not been overridden, so it is still implemented by comparing object identity.

That's why the comparison "p1 equals p2a" yields false even though points p1 and p2a have the same x and y values. That's also why the contains method in HashSet returned false. Since that method operates on generic sets, it calls the generic equals method in Object instead of the overloaded variant in Point. Here's a better equals method:

```
// A better definition, but still not perfect
override def equals(other: Any) = other match {
  case that: Point => this.x == that.x && this.y == that.y
  case _ => false
}
```

Now equals has the correct type. It takes a value of type Any as parameter and it yields a Booleanresult. The implementation of this method uses a pattern match. It first tests whether theother object is also of type Point. If it is, it compares the coordinates of the two points and returns the result. Otherwise the result is false.

A related pitfall is to define == with a wrong signature. Normally, if you try to redefine == with the correct signature, which takes an argument of type Any, the compiler will give you an error because you try to override a final method of type Any.

Newcomers to Scala sometimes make two errors at once: They try to override == and they give it the wrong signature. For instance:

```
def ==(other: Point): Boolean = // Don't do this!
```

In this case, the user-defined == method is treated as an overloaded variant of the same-named method class Any and the program compiles. However, the behavior of the program would be just as dubious as if you had defined equals with the wrong signature.

Pitfall #2: Changing equals without also changing hashCode

We'll continue to use the example from pitfall #1. If you repeat the comparison of p1 and p2awith the latest definition of Point, you will get true, as expected. However, if you repeat theHashSet.contains test, you will probably still get false:

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@122c1533
p2: Point = Point@c23d097

scala> collection.mutable.HashSet(p1) contains p2
res4: Boolean = false
```

But this outcome is not 100% certain. You might also get true from the experiment. If you do, you can try with some other points with coordinates 1 and 2. Eventually, you'll get one that is not contained in the set. What goes wrong here is that Point redefined equals without also redefining hashCode.

Note that the collection in the example here is a HashSet. This means elements of the collection are put in "hash buckets" determined by their hash code. The contains test first determines a hash bucket to look in and then compares the given elements with all elements in that bucket. Now, the last version of class Point did redefine equals, but it did not redefine hashCode at the same time. So hashCode is still what it was in its version in class AnyRef: some transformation of the address of the allocated object.

The hash codes of p1 and p2 are almost certainly different, even though the fields of both points are the same. Different hash codes mean, with high probability, different hash buckets in the set.
The contains test will look for a matching element in the bucket which corresponds to p2's hash code. In most cases, point p1 will be in another bucket, so it will never be found. p1and p2 might also end up by chance in the same hash bucket. In that case the test would return true. The problem is that the last implementation of Point violated the contract onhashCode as defined for class Any:[4]

If two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result.

In fact, it's well known in Java that hashCode and equals should always be redefined together. Furthermore, hashCode may only depend on fields that equals depends on. For the Point class, the following would be a suitable definition of hashCode:

```
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
```

This is just one of many possible implementations of hashCode. Recall that the ## method is a shorthand for computing hash codes that works for primitive values, reference types, andnull. When invoked on a collection or a tuple, it computes a mixed hash that is sensitive to the hash codes of all the elements in the collection. We'll provide more guidance on writinghashCode later in this chapter.

Adding hashCode fixes the problems of equality when defining classes like Point; however, there are other trouble spots to watch out for.

Pitfall #3: Defining equals in terms of mutable fields

Consider the following slight variation of class Point:

```
class Point(var x: Int, var y: Int) { // Problematic
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
```

The only difference is that the fields x and y are now vars instead
of vals. The equals andhashCode methods are now defined in terms of these mutable fields, so their results change when the fields change. This can have strange effects once you put points in collections:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val coll = collection.mutable.HashSet(p)
coll: scala.collection.mutable.HashSet[Point] =
Set(Point@5428bd62)

scala> coll contains p
res5: Boolean = true
```

Now, if you change a field in point p, does the collection still contain the point? We'll try it:

```
scala> p.x += 1

scala> coll contains p
res7: Boolean = false
```

This looks strange. Where did p go? More strangeness results if you check whether the iterator of the set contains p:

```
scala> coll.iterator contains p
res8: Boolean = true
```

So here's a set that does not contain p, yet p is among the elements of the set! What happened is that after the change to the x field, the point p ended up in the wrong hash bucket of the setcoll. That is, its original hash bucket no longer corresponded to the new value of its hash code. In a manner of speaking, point p "dropped out of sight" in the set coll even though it still belonged to its elements.

The lesson to be drawn from this example is that when equals and hashCode depend on mutable state, it causes problems for potential users. If you put such objects into collections, you have to be careful to never modify the depended-on state, and this is tricky. If you need a comparison that takes the current state of an object into account, you should usually name it something else, not equals.

Considering the last definition of Point, it would have been preferable to omit a redefinition ofhashCode and name the comparison method equalContents or some other name different fromequals. Point would then have inherited the default implementation of equals and hashCode; pwould have stayed locatable in coll even after the modification to its x field.

Pitfall #4: Failing to define equals as an equivalence relation

The contract of the equals method in scala.Any specifies that equals must implement an equivalence relation on non-null objects:[5]

- It is reflexive: For any non-null value x , the expression x.equals(x) should return true.
- It is symmetric: For any non-null values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: For any non-null values x, y, and z, if x.equals(y) returns true andy.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: For any non-null values x and y, multiple invocations of x.equals(y)should consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null value x, x.equals(null) should return false.

The definition of equals developed for class Point up to now satisfies the contract for equals. However, things become more complicated once subclasses are considered. Say there is a subclass ColoredPoint of Point that adds a field color of type Color. Assume Color is defined as an enumeration, as presented in Section 20.9:

```
object Color extends Enumeration {
  val Red, Orange, Yellow, Green, Blue, Indigo, Violet = Value
}
```

ColoredPoint overrides equals to take the new color field into account:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) { // Problem: equals not symmetric

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      this.color == that.color && super.equals(that)
    case _ => false
  }
}
```

This is what many programmers would likely write. Note that in this case, class ColoredPointneed not override hashCode. Because the new definition of equals on ColoredPoint is stricter than the overridden definition in Point (meaning it equates fewer pairs of objects), the contract forhashCode stays valid. If two colored points are equal, they must have the same coordinates, so their hash codes are guaranteed to be equal as well.

Taking the class ColoredPoint by itself, its definition of equals looks OK. However, the contract for equals is broken once points and colored points are mixed. Consider:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val cp = new ColoredPoint(1, 2, Color.Red)
cp: ColoredPoint = ColoredPoint@5428bd62

scala> p equals cp
res9: Boolean = true

scala> cp equals p
res10: Boolean = false
```

The comparison "p equals cp" invokes p's equals method, which is defined in class Point. This method only takes into account the coordinates of the two points. Consequently, the comparison yields true. On the other hand, the comparison "cp equals p" invokes cp's equalsmethod, which is defined in class ColoredPoint. This method returns false because p is not aColoredPoint. So the relation defined by equals is not symmetric.

The loss in symmetry can have unexpected consequences for collections. Here's an example:

```
scala> collection.mutable.HashSet[Point](p) contains cp
res11: Boolean = true

scala> collection.mutable.HashSet[Point](cp) contains p
res12: Boolean = false
```

Even though p and cp are equal, one contains test succeeds whereas the other one fails:

How can you change the definition of equals so that it becomes symmetric? Essentially there are two ways. You can either make the relation more general or more strict. Making it more general means that a pair of two objects, x and y, is taken to be equal if either comparing xwith y or comparing y with x yields true. Here's code that does this:

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) { // Problem: equals not transitive

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case that: Point =>
      that equals this
    case _ =>
      false
  }
}
```

The new definition of equals in ColoredPoint has one more case than the old one: If the otherobject is a Point but not a ColoredPoint, the method forwards to the equals method of Point. This has the desired effect of making equals symmetric. Now, both "cp equals p" and "p equals cp" result in true. However, the contract for equals is still broken. The problem is that the new relation is no longer transitive!

Here's a sequence of statements that demonstrates this. Define a point and two colored points of different colors, all at the same position:

```
scala> val redp = new ColoredPoint(1, 2, Color.Red)
redp: ColoredPoint = ColoredPoint@5428bd62

scala> val bluep = new ColoredPoint(1, 2, Color.Blue)
bluep: ColoredPoint = ColoredPoint@5428bd62
```

Taken individually, redp is equal to p and p is equal to bluep:

```
scala> redp == p
res13: Boolean = true

scala> p == bluep
res14: Boolean = true
```

However, comparing redp and bluep yields false:

```
scala> redp == bluep
res15: Boolean = false
```

Hence, the transitivity clause of the equals's contract is violated.

Making the equals relation more general seems to lead to a dead end. We'll try to make it stricter instead. One way to make equals stricter is to always treat objects of different classes as different. This could be achieved by modifying the equals methods in classes Point andColoredPoint. In class Point, you could add an extra comparison that checks whether the run-time class of the other Point is exactly the same as this Point's class:

```
// A technically valid, but unsatisfying, equals method
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point =>
      this.x == that.x && this.y == that.y &&
      this.getClass == that.getClass
    case _ => false
  }
}
```

You can then revert class ColoredPoint's implementation back to the version that previously had violated the symmetry requirement:[6]

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (this.color == that.color) && super.equals(that)
    case _ => false
  }
}
```

Here, an instance of class Point is considered to be equal to some other instance of the same class, only if the objects have the same coordinates and they have the same run-time class, meaning getClass on either object returns the same value. The new definitions satisfy symmetry and transitivity because now every comparison between objects of different classes yields false. So a colored point can never be equal to a point. This convention looks reasonable, but one could argue that the new definition is too strict.

Consider the following slightly roundabout way to define a point at coordinates (1, 2):

```
scala> val pAnon = new Point(1, 1) { override val y = 2 }
pAnon: Point = $anon$1@5428bd62
```

Is pAnon equal to p? The answer is no because the java.lang.Class objects associated with p andpAnon are different. For p it is Point, whereas for pAnon it is an anonymous subclass of Point. But clearly, pAnon is just another point at coordinates (1, 2). It does not seem reasonable to treat it as being different from p.

So it seems we are stuck. Is there a sane way to redefine equality on several levels of the class hierarchy while keeping its contract? In fact, there is such a way, but it requires one more method to redefine together with equals and hashCode. The idea is that as soon as a class redefines equals (and hashCode), it should also explicitly state that objects of this class are never equal to objects of some superclass that implement a different equality method. This is achieved by adding a method canEqual to every class that redefines equals. Here's the method's signature:

```
def canEqual(other: Any): Boolean
```

The method should return true if the other object is an instance of the class in which canEqual is (re)defined, false otherwise. It is called from equals to make sure that the objects are comparable both ways. Listing 30.1 shows a new (and final) implementation of class Pointalong these lines:

```
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point =>
      (that canEqual this) &&
      (this.x == that.x) && (this.y == that.y)
    case _ =>
      false
  }
  def canEqual(other: Any) = other.isInstanceOf[Point]
}
```

Listing 30.1 - A superclass equals method that calls canEqual.

The equals method in this version of class Point contains the additional requirement that the other object *can equal* this one, as determined by the canEqual method. The implementation ofcanEqual in Point states that all instances of Point can be equal.

Listing 30.2 shows the corresponding implementation of ColoredPoint.

```
class ColoredPoint(x: Int, y: Int, val color: Color.Value)
    extends Point(x, y) {

  override def hashCode = (super.hashCode, color).##
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (that canEqual this) &&
      super.equals(that) && this.color == that.color
    case _ =>
      false
  }
  override def canEqual(other: Any) =
    other.isInstanceOf[ColoredPoint]
}
```

Listing 30.2 - A subclass equals method that calls canEqual.

It can be shown that the new definition of Point and ColoredPoint keeps the contract of equals. Equality is symmetric and transitive. Comparing a Point to a ColoredPoint always yields false. Indeed, for any point p and colored point cp, "p equals cp" will return false because "cp canEqual p" will return false. The reverse comparison, "cp equals p", will also return falsebecause p is not a ColoredPoint, so the first pattern match in the body of equals in ColoredPointwill fail.

On the other hand, instances of different subclasses of Point can be equal, as long as none of the classes redefines the equality method. For instance, with the new class definitions, the comparison of p and pAnon would yield true. Here are some examples:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val cp = new ColoredPoint(1, 2, Color.Indigo)
cp: ColoredPoint = ColoredPoint@e6230d8f

scala> val pAnon = new Point(1, 1) { override val y = 2 }
pAnon: Point = $anon$1@5428bd62

scala> val coll = List(p)
coll: List[Point] = List(Point@5428bd62)

scala> coll contains p
res16: Boolean = true

scala> coll contains cp
res17: Boolean = false

scala> coll contains pAnon
res18: Boolean = true
```

These examples demonstrate that if a superclass equals implementation defines and callscanEqual, then programmers who implement subclasses can decide whether or not their subclasses may be equal to instances of the superclass. Because ColoredPoint overrides canEqual, for example, a colored point may never be equal to a plain-old point. But because the anonymous subclass referenced from pAnon does not override canEqual, its instance can be equal to a Point instance.

One potential criticism of the canEqual approach is that it violates the Liskov Substitution Principle. For example, the technique of implementing equals by comparing run-time classes, which led to the inability to define a subclass whose instances can equal instances of the superclass, has been described as a violation of the LSP.[7] The LSP states you should be able to use (substitute) a subclass instance where a superclass instance is required.

In the previous example, however, "coll contains cp" returned false even though cp's x and yvalues matched those of the point in the collection. Thus, it may seem like a violation of the LSP because you can't use a ColoredPoint here where a Point is expected. We believe this is the wrong interpretation, because the LSP doesn't require that a subclass behaves identically to its superclass, just that it behaves in a way that fulfills the contract of its superclass.

The problem with writing an equals method that compares run-time classes is not that it violates the LSP, but that it doesn't give you a way to create a subclass whose instances can equal superclass instances. For example, had we used the run-time class technique in the previous example, "coll contains pAnon" would have returned false, and that's not what we wanted. By contrast, we really did want "coll contains cp" to return false, because by overriding equals in ColoredPoint, we were basically saying that an indigo-colored point at coordinates (1, 2) is *not the same thing* as an uncolored point at (1, 2). Thus, in the previous example we were able to pass two different Point subclass instances to the collection's containsmethod, and we got back two different answers, both correct.

30.3 DEFINING EQUALITY FOR PARAMETERIZED TYPES

The equals methods in the previous examples all started with a pattern match that tested whether the type of the operand conformed to the type of the class containing the equalsmethod. When classes are parameterized, this scheme needs to be adapted a little bit.

As an example, consider binary trees. The class hierarchy shown in Listing 30.3 defines an abstract class Tree for a binary tree, with two alternative implementations: an EmptyTree object and a Branch class representing non-empty trees. A non-empty tree is made up of some element elem, and a left and right child tree. The type of its element is given by a type parameter T.

```
trait Tree[+T] {
  def elem: T
  def left: Tree[T]
  def right: Tree[T]
}

object EmptyTree extends Tree[Nothing] {
  def elem =
    throw new NoSuchElementException("EmptyTree.elem")
  def left =
    throw new NoSuchElementException("EmptyTree.left")
  def right =
    throw new NoSuchElementException("EmptyTree.right")
}

class Branch[+T](
  val elem: T,
```

```
    val left: Tree[T],
    val right: Tree[T]
  ) extends Tree[T]
```

Listing 30.3 - Hierarchy for binary trees.

We'll now add equals and hashCode methods to these classes. For class Tree itself there's nothing to do because we assume that these methods are implemented separately for each implementation of the abstract class. For object EmptyTree, there's still nothing to do because the default implementations of equals and hashCode that EmptyTree inherits from AnyRef work just fine. After all, an EmptyTree is only equal to itself, so equality should be reference equality, which is what's inherited from AnyRef.

But adding equals and hashCode to Branch requires some work. A Branch value should only be equal to other Branch values, and only if the two values have equal elem, left and right fields. It's natural to apply the schema for equals that was developed in the previous sections of this chapter. This would give you:

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {

  override def equals(other: Any) = other match {
    case that: Branch[T] => this.elem == that.elem &&
                            this.left == that.left &&
                            this.right == that.right
    case _ => false
  }
}
```

Compiling this example, however, gives an indication that "unchecked" warnings occurred. Compiling again with the -unchecked option reveals the following problem:

```
 $ fsc -unchecked Tree.scala
Tree.scala:14: warning: non variable type-argument T in type
pattern is unchecked since it is eliminated by erasure
    case that: Branch[T] => this.elem == that.elem &&
               ^
```

As the warning says, there is a pattern match against a Branch[T] type, yet the system can only check that the other reference is (some kind of) Branch; it cannot check that the element type of the tree is T. You encountered in Chapter 19 the reason for this: Element types of parameterized types are eliminated by the compiler's erasure phase; they are not available to be inspected at run-time.

So what can you do? Fortunately, it turns out that you need not necessarily check that twoBranches have the same element types when comparing them. It's quite possible that twoBranches with different element types are equal, as long as their fields are the same. A simple example of this would be the Branch that consists of a single Nil element and two empty subtrees. It's plausible to consider any two such Branches to be equal, no matter what static types they have:

```
scala> val b1 = new Branch[List[String]](Nil,
           EmptyTree, EmptyTree)
b1: Branch[List[String]] = Branch@9d5fa4f

scala> val b2 = new Branch[List[Int]](Nil,
           EmptyTree, EmptyTree)
b2: Branch[List[Int]] = Branch@56cdfc29

scala> b1 == b2
res19: Boolean = true
```

The positive result of the comparison above was obtained with the implementation of equalson Branch shown previously. This demonstrates that the element type of the Branch was not checked—if it had been checked, the result would have been false.

We can disagree on which of the two possible outcomes of the comparison would be more natural. In the end, this depends on the mental model of how classes are represented. In a model where type-parameters are present only at compile-time, it's natural to consider the two Branch values b1 and b2 to be equal. In an alternative model where a type parameter forms part of an object's value, it's equally natural to consider them different. Since Scala adopts the type erasure model, type parameters are not preserved at run time, so that b1 and b2 are naturally considered to be equal.

There's only a tiny change needed to formulate an equals method that does not produce anunchecked warning. Instead of an element type T, use a lower case letter, such as t:

```
case that: Branch[t] => this.elem == that.elem &&
                        this.left == that.left &&
                        this.right == that.right
```

Recall from Section 15.2 that a type parameter in a pattern starting with a lower-case letter represents an unknown type. Now the pattern match:

```
case that: Branch[t] =>
```

will succeed for Branch values of any type. The type parameter t represents the unknown element type of the Branch. It can also be replaced by an underscore, as in the following case, which is equivalent to the previous one:

```
case that: Branch[_] =>
```

The only thing that remains is to define for class Branch the other two methods, hashCode andcanEqual, which go with equals. Here's a possible implementation of hashCode:

```
override def hashCode: Int = (elem, left, right).##
```

This is only one of many possible implementations. As shown previously, the principle is to take hashCode values of all fields and combine them. Here's an implementation of methodcanEqual in class Branch:

```
def canEqual(other: Any) = other match {
```

```
    case that: Branch[_] => true
    case _ => false
  }
```

The implementation of the canEqual method used a typed pattern match. It would also be possible to formulate it with isInstanceOf:

```
  def canEqual(other: Any) = other.isInstanceOf[Branch[_]]
```

If you feel like nit-picking—and we encourage you to do so!—you might wonder what the occurrence of the underscore in the type above signifies. After all, Branch[_] is technically a type parameter of a method, not a type pattern. So how is it possible to leave some parts of it undefined?

The answer to this question is discussed in the next chapter. Branch[_] is shorthand for a so-called wildcard type, which is, roughly speaking, a type with some unknown parts in it. So even though technically the underscore stands for two different things in a pattern match and in a type parameter of a method call, in essence, the meaning is the same: It lets you label something that is unknown. The final version of Branch is shown in Listing 30.4.

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {

  override def equals(other: Any) = other match {
    case that: Branch[_] => (that canEqual this) &&
                            this.elem == that.elem &&
                            this.left == that.left &&
                            this.right == that.right
    case _ => false
  }

  def canEqual(other: Any) = other.isInstanceOf[Branch[_]]

  override def hashCode: Int = (elem, left, right).##
}
```

Listing 30.4 - A parameterized type with equals and hashCode.

30.4 RECIPES FOR EQUALS AND HASHCODE

In this section, we'll provide step-by-step recipes for creating equals and hashCode methods that should suffice for most situations. As an illustration, we'll use the methods of class Rational, shown in Listing 30.5.

To create this class, we removed the mathematical operator methods from the version of classRational shown in Listing 6.5 here. We also made a minor enhancement to toString, and modified the initializers of numer and denom to normalize all fractions to have a positive denominator (*i.e.*, to transform 1/-2 to -1/2).

Recipe for equals

Here's the recipe for overriding equals:

1. To override equals in a non-final class, create a canEqual method. If the inherited definition of equals is from AnyRef (that is, equals was not redefined higher up in the class hierarchy), the definition of canEqual should be new; otherwise, it will override a previous definition of a method with the same name. The only exception to this requirement is for final classes that redefine the equals method inherited from AnyRef. For them the subclass anomalies described in Section 30.2 cannot arise; consequently they need not define canEqual. The type of object passed to canEqual should be Any:

   ```
   def canEqual(other: Any): Boolean =
   ```

2. The canEqual method should yield true if the argument object is an instance of the current class (*i.e.*, the class in which canEqual is defined), and false otherwise:

   ```
   other.isInstanceOf[Rational]
   ```

3. In the equals method, make sure you declare the type of the object passed as an Any:

   ```
   override def equals(other: Any): Boolean =
   ```

4. Write the body of the equals method as a single match expression. The selector of the matchshould be the object passed to equals:

   ```
   other match {
     // ...
   }
   ```

5. The match expression should have two cases. The first case should declare a typed pattern for the type of the class on which you're defining the equals method:

   ```
   case that: Rational =>
   ```

6. In the body of this case, write an expression that logical-ands together the individual expressions that must be true for the objects to be equal. If the equals method you are overriding is not that of AnyRef, you will most likely want to include an invocation of the superclass's equals method:

   ```
   super.equals(that) &&
   ```

   If you are defining equals for a class that first introduced canEqual, you should invokecanEqual on the argument to the equality method, passing this as the argument:

   ```
   (that canEqual this) &&
   ```

   Overriding redefinitions of equals should also include the canEqual invocation, unless they contain a call to super.equals. In the latter case, the canEqual test will already be done by the

superclass call. Lastly, for each field relevant to equality, verify that the field in this object is equal to the corresponding field in the passed object:

```
numer == that.numer &&
denom == that.denom
```

7. For the second case, use a wildcard pattern that yields false:

```
case _ => false
```

If you adhere to this recipe for equals, equality is guaranteed to be an equivalence relation, as is required by the equals contract.

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = (if (d < 0) -n else n) / g
  val denom = d.abs / g

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  override def equals(other: Any): Boolean =
    other match {

      case that: Rational =>
        (that canEqual this) &&
        numer == that.numer &&
        denom == that.denom

      case _ => false
    }

  def canEqual(other: Any): Boolean =
    other.isInstanceOf[Rational]

  override def hashCode: Int = (numer, denom).##

  override def toString =
    if (denom == 1) numer.toString else numer + "/" + denom
}
```

Listing 30.5 - Class Rational with equals and hashCode.

Recipe for hashCode

For hashCode, you can usually achieve satisfactory results if you use the following recipe, whichis similar to a recipe recommended for Java classes in *Effective Java*.[8] Include in the calculation each field in your object that is used to determine equality in the equals method (the "relevant" fields). Make a tuple containing the values of all those fields. Then, invoke ##on the resulting tuple.

For example, to implement the hash code for an object that has five relevant fields named a, b,c, d, and e, you would write:

```
override def hashCode: Int = (a, b, c, d, e).##
```

If the equals method invokes super.equals(that) as part of its calculation, you should start yourhashCode calculation with an invocation of super.hashCode. For example, had Rational's equalsmethod invoked super.equals(that), its hashCode would have been:

```
override def hashCode: Int = (super.hashCode, numer, denom).##
```

One thing to keep in mind as you write hashCode methods using this approach is that your hash code will only be as good as the hash codes you build out of it, namely the hash codes you obtain by calling hashCode on the relevant fields of your object. Sometimes you may need to do something extra besides just calling hashCode on the field to get a useful hash code for that field. For example, if one of your fields is a collection, you probably want a hash code for that field that is based on all the elements contained in the collection. If the field is a Vector, List,Set, Map, or tuple, you can simply include it in the list of items you are hashing over, becauseequals and hashCode are overridden in those classes to take into account the contained elements. However the same is not true for Arrays, which do not take elements into account when calculating a hash code. Thus for an array, you should treat each element of the array like an individual field of your object, calling ## on each element explicitly or passing the array to one of the hashCode methods in singleton object java.util.Arrays.

Lastly, if you find that a particular hash code calculation is harming the performance of your program, consider caching the hash code. If the object is immutable, you can calculate the hash code when the object is created and store it in a field. You can do this by simply overriding hashCode with a val instead of a def, like this:

```
override val hashCode: Int = (numer, denom).##
```

This approach trades off memory for computation time, because each instance of the immutable class will have one more field to hold the cached hash code value.

30.5 CONCLUSION

In retrospect, defining a correct implementation of equals has been surprisingly subtle. You must be careful about the type signature; you must override hashCode; you should avoid dependencies on mutable state; and you should implement and use a canEqual method if your class is non-final.

Given how difficult it is to implement a correct equality method, you might prefer to define your classes of comparable objects as case classes. That way, the Scala compiler will addequals and hashCode methods with the right properties automatically.

Footnotes for Chapter 30:

[1] Vaziri, *et. al.*, "Declarative Object Identity Using Relation Types" [Vaz07]

[2] All but the third pitfall are described in the context of Java in the book, *Effective Java Second Edition*, by Joshua Bloch. [Blo08]

[3] If you write a lot of Java, you might expect the argument to this method to be type Objectinstead of type Any. Don't worry about it; it is the same equals method. The compiler simply makes it appear to have type Any.

[4] The text of Any's hashCode contract is inspired by the Javadoc documentation of classjava.lang.Object.

[5] As with hashCode, Any's equals method contract is based on java.lang.Object's equals method contract.

[6] Given the new implementation of equals in Point, this version of ColoredPoint no longer violates the symmetry requirement.

[7] Bloch, *Effective Java Second Edition*, p. 39 [Blo08]

[8] Bloch, *Effective Java Second Edition*. [Blo08]