Chapter 11

Scala's Hierarchy

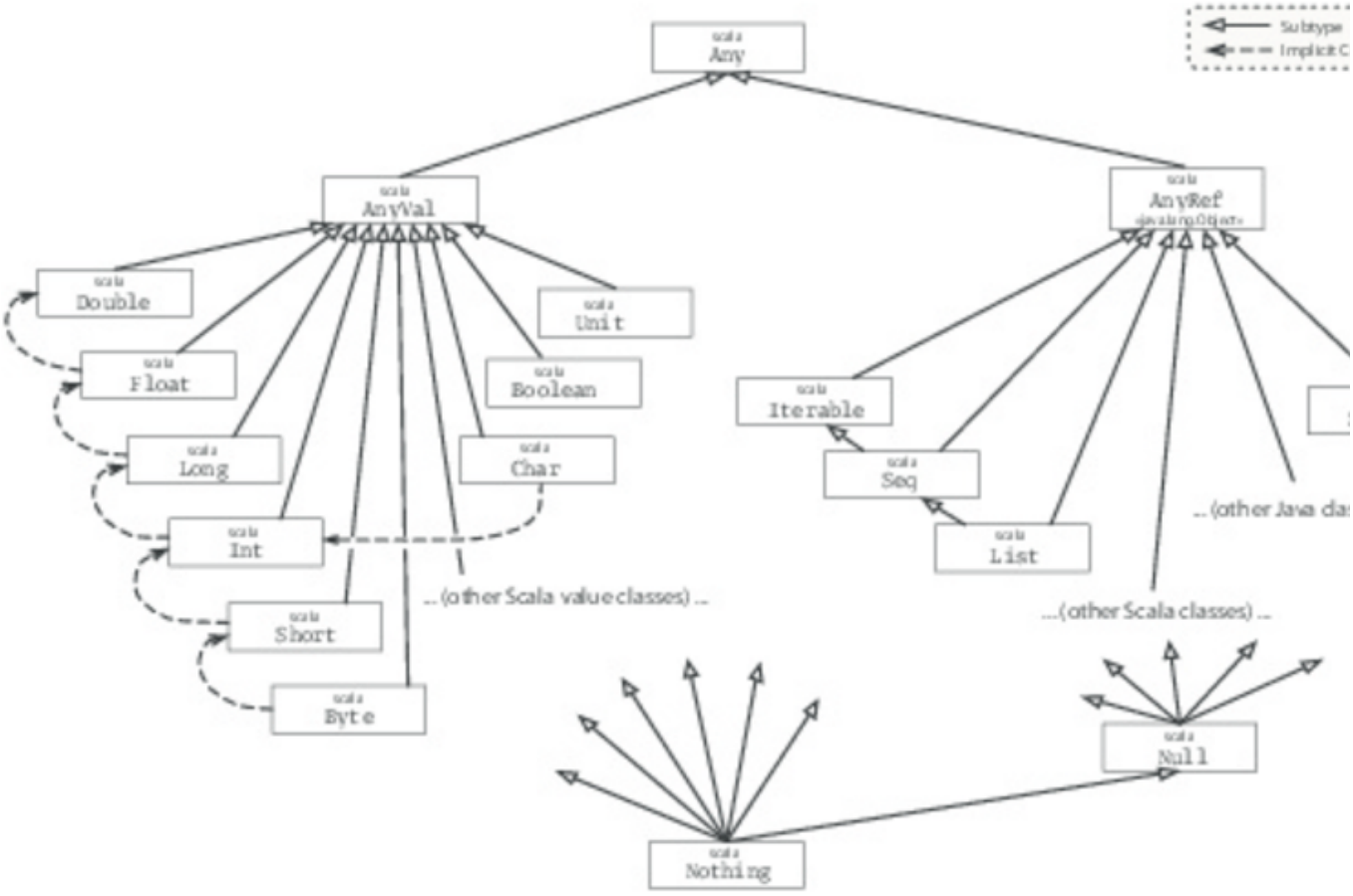
Now that you've seen the details of class inheritance in the previous chapter, it is a good time to take a step back and look at Scala's class hierarchy as a whole. In Scala, every class inherits from a common superclass named Any. Because every class is a subclass of Any, the methods defined in Any are "universal" methods: they may be invoked on any object. Scala also defines some interesting classes at the bottom of the hierarchy, Null and Nothing, which essentially act as common *sub*classes. For example, just as Any is a superclass of every other class, Nothing is a subclass of every other class. In this chapter, we'll give you a tour of Scala's class hierarchy.

11.1 SCALA'S CLASS HIERARCHY

Figure 11.1 shows an outline of Scala's class hierarchy. At the top of the hierarchy is class Any, which defines methods that include the following:

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```

Because every class inherits from Any, every object in a Scala program can be compared using==, !=, or equals; hashed using ## or hashCode; and formatted using toString. The equality and inequality methods, == and !=, are declared final in class Any, so they cannot be overridden in subclasses. The == method is essentially the same as equals and != is always the negation ofequals.[1] So individual classes can tailor what == or != means by overriding the equals method. We'll show an example later in this chapter.



Class hierarchy of Scala.

The root class Any has two subclasses: AnyVal and AnyRef. AnyVal is the parent class of value classes in Scala. While you can define your own value classes (see Section 11.4), there are nine value classes built into Scala: Byte, Short, Char, Int, Long, Float, Double, Boolean, and Unit. The first eight of these correspond to Java's primitive types, and their values are represented at run time as Java's primitive values. The instances of these classes are all written as literals in Scala. For example, 42 is an instance of Int, 'x' is an instance of Char, and false an instance ofBoolean. You cannot create instances of these classes using new. This is enforced by the "trick" that value classes are all defined to be both abstract and final.

So if you were to write:

```
scala> new Int
```

you would get:

```
<console>:5: error: class Int is abstract; cannot be
instantiated
       new Int
       ^
```

The other value class, Unit, corresponds roughly to Java's void type; it is used as the result type of a method that does not otherwise return an interesting result. Unit has a single instance value, which is written (), as discussed in Section 7.2.

As explained in Chapter 5, the value classes support the usual arithmetic and boolean operators as methods. For instance, Int has methods named + and *, and Boolean has methods named || and &&. Value classes also inherit all methods from class Any. You can test this in the interpreter:

```
scala> 42.toString
res1: String = 42

scala> 42.hashCode
res2: Int = 42

scala> 42 equals 42
res3: Boolean = true
```

Note that the value class space is flat; all value classes are subtypes of scala.AnyVal, but they do not subclass each other. Instead there are implicit conversions between different value class types. For example, an instance of class scala.Int is automatically widened (by an implicit conversion) to an instance of class scala.Long when required.

As mentioned in Section 5.10, implicit conversions are also used to add more functionality to value types. For instance, the type Int supports all of the operations below:

```
scala> 42 max 43
res4: Int = 43

scala> 42 min 43
res5: Int = 42

scala> 1 until 5
res6: scala.collection.immutable.Range = Range(1, 2, 3, 4)

scala> 1 to 5
res7: scala.collection.immutable.Range.Inclusive
  = Range(1, 2, 3, 4, 5)

scala> 3.abs
res8: Int = 3

scala> (-3).abs
res9: Int = 3
```

Here's how this works: The methods min, max, until, to, and abs are all defined in a classscala.runtime.RichInt, and there is an implicit conversion from class Int to RichInt. The conversion is applied whenever a method is invoked on an Int that is undefined in Int but defined in RichInt. Similar "booster classes" and implicit conversions exist for the other value classes. Implicit conversions will be discussed in detail in Chapter 21.

The other subclass of the root class Any is class AnyRef. This is the base class of all reference classes in Scala. As mentioned previously, on the Java platform AnyRef is in fact just an alias for

class java.lang.Object. So classes written in Java, as well as classes written in Scala, all inherit from AnyRef.[2] One way to think of java.lang.Object, therefore, is as the way AnyRef is implemented on the Java platform. Thus, although you can use Object and AnyRefinterchangeably in Scala programs on the Java platform, the recommended style is to useAnyRef everywhere.

11.2 HOW PRIMITIVES ARE IMPLEMENTED

How is all this implemented? In fact, Scala stores integers in the same way as Java—as 32-bit words. This is important for efficiency on the JVM and also for interoperability with Java libraries. Standard operations like addition or multiplication are implemented as primitive operations. However, Scala uses the "backup" class java.lang.Integer whenever an integer needs to be seen as a (Java) object. This happens for instance when invoking the toStringmethod on an integer number or when assigning an integer to a variable of type Any. Integers of type Int are converted transparently to "boxed integers" of type java.lang.Integer whenever necessary.

All this sounds a lot like auto-boxing in Java 5 and it is indeed quite similar. There's one crucial difference though: Boxing in Scala is much less visible than boxing in Java. Try the following in Java:

```
// This is Java
boolean isEqual(int x, int y) {
  return x == y;
}
System.out.println(isEqual(421, 421));
```

You will surely get true. Now, change the argument types of isEqual to java.lang.Integer (orObject, the result will be the same):

```
// This is Java
boolean isEqual(Integer x, Integer y) {
  return x == y;
}
System.out.println(isEqual(421, 421));
```

You will find that you get false! What happens is that the number 421 gets boxed twice, so that the arguments for x and y are two different objects. Because == means reference equality on reference types, and Integer is a reference type, the result is false. This is one aspect where it shows that Java is not a pure object-oriented language. There is a difference between primitive types and reference types that can be clearly observed.

Now try the same experiment in Scala:

```
scala> def isEqual(x: Int, y: Int) = x == y
isEqual: (x: Int, y: Int)Boolean

scala> isEqual(421, 421)
res10: Boolean = true

scala> def isEqual(x: Any, y: Any) = x == y
isEqual: (x: Any, y: Any)Boolean

scala> isEqual(421, 421)
```

```
res11: Boolean = true
```

The equality operation == in Scala is designed to be transparent with respect to the type's representation. For value types, it is the natural (numeric or boolean) equality. For reference types other than Java's boxed numeric types, == is treated as an alias of the equals method inherited from Object. That method is originally defined as reference equality, but is overridden by many subclasses to implement their natural notion of equality. This also means that in Scala you never fall into Java's well-known trap concerning string comparisons. In Scala, string comparison works as it should:

```
scala> val x = "abcd".substring(2)
x: String = cd

scala> val y = "abcd".substring(2)
y: String = cd

scala> x == y
res12: Boolean = true
```

In Java, the result of comparing x with y would be false. The programmer should have usedequals in this case, but it is easy to forget.

However, there are situations where you need reference equality instead of user-defined equality. For example, in some situations where efficiency is paramount, you would like to*hash cons* with some classes and compare their instances with reference equality.[3] For these cases, class AnyRef defines an additional eq method, which cannot be overridden and is implemented as reference equality (*i.e.*, it behaves like == in Java for reference types). There's also the negation of eq, which is called ne. For example:

```
scala> val x = new String("abc")
x: String = abc

scala> val y = new String("abc")
y: String = abc

scala> x == y
res13: Boolean = true

scala> x eq y
res14: Boolean = false

scala> x ne y
res15: Boolean = true
```

Equality in Scala is discussed further in Chapter 30.

11.3 BOTTOM TYPES

At the bottom of the type hierarchy in Figure 11.1 you see the two classes scala.Null andscala.Nothing. These are special types that handle some "corner cases" of Scala's object-oriented type system in a uniform way.

Class Null is the type of the null reference; it is a subclass of every reference class (*i.e.*, every class that itself inherits from AnyRef). Null is not compatible with value types. You cannot, for example, assign a null value to an integer variable:

```
scala> val i: Int = null
<console>:7: error: an expression of type Null is ineligible
for implicit conversion
       val i: Int = null
                    ^
```

Type Nothing is at the very bottom of Scala's class hierarchy; it is a subtype of every other type. However, there exist no values of this type whatsoever. Why does it make sense to have a type without values? As discussed in Section 7.4, one use of Nothing is that it signals abnormal termination.

For instance there's the error method in the Predef object of Scala's standard library, which is defined like this:

```
def error(message: String): Nothing =
  throw new RuntimeException(message)
```

The return type of error is Nothing, which tells users that the method will not return normally (it throws an exception instead). Because Nothing is a subtype of every other type, you can use methods like error in very flexible ways. For instance:

```
def divide(x: Int, y: Int): Int =
  if (y != 0) x / y
  else error("can't divide by zero")
```

The "then" branch of the conditional, x / y, has type Int, whereas the else branch, the call toerror, has type Nothing. Because Nothing is a subtype of Int, the type of the whole conditional isInt, as required.

11.4 DEFINING YOUR OWN VALUE CLASSES

As mentioned in Section 11.1, you can define your own value classes to augment the ones that are built in. Like the built-in value classes, an instance of your value class will usually compile to Java bytecode that does not use the wrapper class. In contexts where a wrapper is needed, such as with generic code, the value will get boxed and unboxed automatically.

Only certain classes can be made into value classes. For a class to be a value class, it must have exactly one parameter and it must have nothing inside it except defs. Furthermore, no other class can extend a value class, and a value class cannot redefine equals or hashCode.

To define a value class, make it a subclass of AnyVal, and put val before the one parameter. Here is an example value class:

```
class Dollars(val amount: Int) extends AnyVal {
  override def toString() = "$" + amount
}
```

As described in Section 10.6, the val prefix allows the amount parameter to be accessed as a field. For example, the following code creates an instance of the value class, then retrieves the amount from it:

```
scala> val money = new Dollars(1000000)
money: Dollars = $1000000
scala> money.amount
res16: Int = 1000000
```

In this example, money refers to an instance of the value class. It is of type Dollars in Scala source code, but the compiled Java bytecode will use type Int directly.

This example defines a toString method, and the compiler figures out when to use it. That's why printing money gives $1000000, with a dollar sign, but printing money.amount gives 1000000. You can even define multiple value types that are all backed by the same Int value. For example:

```
class SwissFrancs(val amount: Int) extends AnyVal {
  override def toString() = amount + " CHF"
}
```

Even though both Dollars and SwissFrancs are represented as integers, it works fine to use them in the same scope:

```
scala> val dollars = new Dollars(1000)
dollars: Dollars = $1000
scala> val francs = new SwissFrancs(1000)
francs: SwissFrancs = 1000 CHF
```

Avoiding a types monoculture

To get the most benefit from the Scala class hierarchy, try to define a new class for each domain concept, even when it would be possible to reuse the same class for different purposes. Even if such a class is a so-called *tiny type* with no methods or fields, defining the additional class is a way to help the compiler be helpful to you.

For example, suppose you are writing some code to generate HTML. In HTML, a style name is represented as a string. So are anchor identifiers. HTML itself is also a string, so if you wanted, you could define helper code using strings to represent all of these things, like this:

```
def title(text: String, anchor: String, style: String): String =
  s"<a id='$anchor'><h1 class='$style'>$text</h1></a>"
```

That type signature has four strings in it! Such *stringly typed* code is technically strongly typed, but since everything in sight is of type String, the compiler cannot help you detect the use of one when you meant to write the other. For example, it won't stop you from this travesty:

```
scala> title("chap:vcls", "bold", "Value Classes")
res17: String = <a id='bold'><h1 class='Value
    Classes'>chap:vcls</h1></a>
```

This HTML is mangled. The intended display text "Value Classes" is being used as a style class, and the text being displayed is "chap.vcls," which was supposed to be an anchor. To top it off, the actual

anchor identifier is "bold," which is supposed to be a style class. Despite this comedy of errors, the compiler utters not a peep.

The compiler can be more helpful if you define a tiny type for each domain concept. For example, you could define a small class for styles, anchor identifiers, display text, and HTML. Since these classes have one parameter and no members, they can be defined as value classes:

```
class Anchor(val value: String) extends AnyVal
class Style(val value: String) extends AnyVal
class Text(val value: String) extends AnyVal
class Html(val value: String) extends AnyVal
```

Given these classes, it is possible to write a version of title that has a less trivial type signature:

```
def title(text: Text, anchor: Anchor, style: Style): Html =
  new Html(
    s"<a id='${anchor.value}'>" +
        s"<h1 class='${style.value}'>" +
        text.value +
        "</h1></a>"
  )
```

If you try to use this version with the arguments in the wrong order, the compiler can now detect the error. For example:

```
scala> title(new Anchor("chap:vcls"), new Style("bold"),
           new Text("Value Classes"))
<console>:18: error: type mismatch;
 found   : Anchor
 required: Text
               new Anchor("chap:vcls"),
               ^
<console>:19: error: type mismatch;
 found   : Style
 required: Anchor
               new Style("bold"),
               ^
<console>:20: error: type mismatch;
 found   : Text
 required: Style
               new Text("Value Classes"))
               ^
```

11.5 CONCLUSION

In this chapter we showed you the classes at the top and bottom of Scala's class hierarchy. Now that you've gotten a good foundation on class inheritance in Scala, you're ready to understand mixin composition. In the next chapter, you'll learn about traits.

Footnotes for Chapter 11:

[1] The only case where == does not directly call equals is for Java's boxed numeric classes, such as Integer or Long. In Java, a new Integer(1) does not equal a new Long(1) even though for primitive

values 1 == 1L. Since Scala is a more regular language than Java, it was necessary to correct this discrepancy by special-casing the == method for these classes. Likewise, the ##method provides a Scala version of hashing that is the same as Java's hashCode, except for boxed numeric types, where it works consistently with ==. For instance new Integer(1) andnew Long(1) hash the same with ## even though their Java hashCodes are different.

[2] The reason AnyRef alias exists, instead of just using the name java.lang.Object, is because Scala was originally designed to work on both the Java and .NET platforms. On .NET, AnyRefwas an alias for System.Object.

[3] You hash cons instances of a class by caching all instances you have created in a weak collection. Then, any time you want a new instance of the class, you first check the cache. If the cache already has an element equal to the one you are about to create, you can reuse the existing instance. As a result of this arrangement, any two instances that are equal withequals() are also equal with reference equality.