Lecture 20: Advanced OOP in Scala
March 10, 2017

Contents

1 Traits as Mixins 1
   1.1 Birds: Do they swim or fly?! 2
   1.2 An Example: Decorating Shapes 3

2 Stackable Modifications 5

Objectives

By the end of this lecture, you will know:

- Scala’s more advanced OOP features (mixins and stackable modifications)

1 Traits as Mixins

Sometime back in the 1970s, an entrepreneur named Steve Herrell had the brilliant idea that he could mix candy into ice cream. For example, he could mix heath bar crunch into coffee ice cream. Yum! And from there, the idea of mixins for object-oriented programs was born.\footnote{Well, maybe not, but OOP mixins are named after the dessert.}

So far, we have talked only of pure interfaces—interfaces in which no methods are implemented. In Java, interfaces must be pure. But in Scala, interfaces are implemented using traits, which are not pure; they can include implementation details.

Because Scala has muddied the waters between interfaces and abstract classes, it has also dropped the distinction between the keywords implements and extends. In Scala, only one of these keywords is used, namely extends.

When a Scala class extends a trait (not a class or an abstract class), it does not merely inherit any implementation details in that trait. Instead, it mixes them in, meaning the functionality defined in the trait is essentially copied-and-pasted into the class definition. Then, when an object (i.e., an instance of a class) invokes a method that is not explicitly defined for its class, Scala looks to its mixins for a definition.

Whereas a class can only inherit from one superclass, a class can mix in any number of traits. In this lecture, we discuss how Scala decides on an implementation, when multiple implementations of the same method are mixed in.
1.1 Birds: Do they swim or fly?!

Let’s look at an example.
Consider a simple `Bird` class:

```scala
abstract class Bird
```

Suppose we would like to decorate `Birds` with extra features, such as `Flying` and `Swimming`. These extra features can be implemented using traits, as follows:

```scala
trait Swimming {
  def swim = println("I’m going for a swim..."))
}
```

```scala
trait Flying {
  val flyMessage = "default fly message"
  def fly = println(flyMessage)
}
```

And we can then mix these traits into `Bird` objects, as follows:

```scala
scala> val hawk = new Bird with Flying

scala> hawk.fly
res: String = "default fly message"

scala> val duck = new Bird with Flying with Swimming

scala> duck.swim
res: String = "I’m going for a swim..."

scala> duck.fly
res: String = "default fly message"
```

Notice that mixins are specified using the keyword `with`.

Beyond mixing into objects (dynamically, as shown above), traits can also be mixed into class declarations (statically) to create new classes. For example:

```scala
class Hawk extends Bird with Flying {
  override val flyMessage = "I am a great flyer!"
}
```

```scala
class Duck extends Bird with Swimming with Flying {
  override val flyMessage = "I am a pretty good flyer."
}
```
After mixing in the Flying and Swimming traits to create new classes, it is straightforward to create Hawk and Duck objects:

```scala
scala> val hawk = new Hawk
scala> val duck = new Duck
```

In the Hawk and Duck subclasses of Bird, the flyMessage in Flying is overridden with a class-specific value. Consequently, when the fly method is invoked, it prints the class-specific value instead of the default message. (Isn't that cool!?)

```scala
scala> hawk.fly
res: String = "I am a great flyer!"

scala> duck.fly
res: String = "I am a pretty good flyer."
```

Finally, we define a Penguin class of birds who swim but do not fly.

```scala
class Penguin extends Bird with Swimming
```

```scala
scala> val penguin = new Penguin
scala> penguin.fly
res: String = "default fly message"

scala> penguin.swim
res: String = "I’m going for a swim."
```

### 1.2 An Example: Decorating Shapes

Recall the Rectangle class, which implements IShape

```scala
trait IShape {

/**
 * Gets the area of a shape.
 * @return the area of a shape
 */
def getArea(): Double

/**
 * Gets the perimeter of a shape.
 * @return the perimeter of a shape
 */
```

def getPerimeter: Double

}

Suppose we would like to decorate rectangles with extra features, such as Location and Color. These extra features can be implemented in traits, as follows:

trait Location {
  val x = 10
  val y = 10

  def printLocation() {
    println("Location: " + x + "," + y)
  }
}

trait Color {
  val border = "black"
  val fill = "blue"

  def printColor() {
    println("Color: Border=" + border + " Fill=" + fill)
  }
}

And we can mix these traits into Rectangle objects, as follows:

scala> val rect = new Rectangle(20, 50) extends IShape with Location with Color

scala> rect.printLocation
Location: 10,10

scala> rect.printColor()
Color: Border=black Fill=blue

Beyond mixing into Rectangle objects, which is called dynamically mixing in, traits can also be mixed into class definitions—statically—to create new classes. For example:

class Rectangle(width: Double, height: Double) extends IShape with Location with Color {
  override def getArea: Double = width * height
override def getPerimeter: Double = (2 * width) + (2 * height)

override def printLocation(x: Int, y: Int) {
    print(x + ", " + y)
}

def describeRectangle() {
    print("Upper-left Corner: ")
    print("\n")
    printLocation(x, y)
    print("\n")
    println()

    print("Lower-right Corner: ")
    print("\n")
    printLocation(x + width.toInt, y + height.toInt)
    print("\n")
    println()
    printColor()
}

Observe that this implementation of the Rectangle class includes the method printLocation. Consequently, the printLocation method defined in the Location trait is not mixed in with the rest of the Location trait; that definition is overridden by the definition in Rectangle. However, the other behaviors and attributes of the Location trait are mixed in to Rectangle. In particular, x and y are declared in the Location trait and used freely in the Rectangle class.

Now that the Location and Color traits have been mixed into the Rectangle class, it is straightforward to create Rectangle objects:

scala> val rect = new Rectangle(20, 50)

scala> rect.describeRectangle()
Upper-left Corner: (10, 10)
Lower-right Corner: (30, 30)
Color: Border = black Fill = blue

2 Stackable Modifications

When a subtype inherits from multiple supertypes, this is called multiple inheritance. In object-oriented design, there is a dangerous trap to avoid regarding multiple inheritance, namely the diamond problem. This problem concerns an ambiguity in how a subtype perceives a supertype which is common to two supertypes of the subtype. To make this inheritance pattern more clear, look at the figure below, which, sure enough, is shaped like a diamond:
To restate the diamond problem: “How does TypeD perceive what has been inherited from TypeA if both of TypeB and TypeC inherit from TypeA?” In particular, “If TypeB and TypeC both provide implementations of a method defined in TypeA, to which implementation are we referring when a TypeD object invokes the method?”

Java has a straightforward solution to the diamond problem. It does not support multiple inheritance among classes. On the contrary, every Java class has one and only one base class (i.e., immediate superclass). Consequently, there is never a question of which implementation of an inherited method to invoke.

Scala also does not support multiple inheritance. As in Java, every Scala class has one and only one base class. On the other hand, Scala supports something that is perhaps more powerful than multiple inheritance, called stackable modifications. Stackable modifications are mixins stacked on top of one another.

**Example 1** Let’s start with a straightforward example of mixing in traits:

```scala
class Id {
  def id(x: Int): Int = x
}

trait AddOne {
  def add1(x: Int): Int = x + 1
}

trait MultTwo {
  def mult2(x: Int): Int = x * 2
}

class Mixin extends Id with AddOne with MultTwo
```

2 Except `Object`, which sits at the root of the Java hierarchy and has no base class.

3 A Java class can implement multiple interfaces (as they provide only empty promises!). This is Java’s answer to multiple inheritance.
There is nothing unusual about these declarations. In particular, there are no conflicts. The \texttt{Id} class implements one method, the \texttt{AddOne} class another, and the \texttt{MultTwo} class a third. (See Figure 1.) These methods operate independently, and as expected:

```
scala> val mixin = new Mixin
scala> mixin.id(18)
res: Int = 18
scala> mixin.add1(18)
res: Int = 19
scala> mixin.mult2(18)
res: Int = 36
```

As a matter of Scala style, it is preferable to invoke these methods using infix notation (and less punctuation), as follows:

```
scala> val mixin = new Mixin
scala> mixin id 18
res: Int = 18
scala> mixin add1 18
res: Int = 19
scala> mixin mult2 18
res: Int = 36
```

\textbf{Example 2}  \hspace{1em} Next, imagine that the \texttt{Add} and \texttt{Mult} classes inherit from a variant of \texttt{Id}, which is renamed \texttt{Fun} and that our intent is to modify (i.e., override) the behavior of the \texttt{fun} method depending on which traits we mix in to the \texttt{Fun} class. To accomplish this, we create traits that extend \texttt{Fun} (see Figure 2), as follows:

```scala
class Fun {
  def fun(x: Int): Int = x
}

trait Add extends Fun {
  override def fun(x: Int): Int = x + x
}

Trait Mult extends Fun {
```

\footnote{Isn’t this fun?!}
override def fun(x: Int): Int = x * x
}

class Mixin1 extends Fun with Add with Mult
class Mixin2 extends Fun with Mult with Add

Note: The fact that these traits are extensions of Fun means that they can only be mixed in to the Fun class itself, or classes that extend Fun.

When a method is invoked by an object in a class whose behavior has been modified by mixins, the definition of the method that is used is the one found in the trait closest to the end of the list of the traits being mixed in (i.e., the last one that is mixed in), because each additional mixin overrides the ones that came before it. The reason for this is: when we mix multiple traits in to a class, we are in fact creating a class hierarchy. For example, this class declaration

class MyClass extends IMyClass with A with B with C

creates a class hierarchy with this class at the root:

class MyClass extends IMyClass
Figure 2: Example 2

and this one, inheriting from the one just above:

```scala
class MyClass extends IMyClass with A
```

and this one, again inheriting from the one just above:

```scala
class MyClass extends IMyClass with A with B
```

and this one, again inheriting from the one just above:

```scala
class MyClass extends IMyClass with A with B with C
```

Consequently, a method’s definition is always the one found farthest to the right, meaning deepest in the hierarchy (i.e., in the most specific subclass).

So, in this example, `mixin1 fun 18` invokes `x * x`, while `mixin2 fun 18` invokes `x + x`:

```scala
scala> val mixin1 = new Mixin1
```
scala> mixin1 fun 18
res: Int = 324

scala> val mixin2 = new Mixin2

scala> mixin2 fun 18
res: Int = 36

Example 3  Given these additional details about Scala’s resolution rule, we can now consider a more complicated situation in which the function `fun` is not only overridden in the `Add` and `Mult` traits, but more than that, the overriding definitions call `super`!

```scala
trait Add extends Fun {
  override def fun(x: Int): Int = x + super.fun(x)
}

trait Mult extends Fun {
  override def fun(x: Int): Int = x * super.fun(x)
}
```

![Diagram of Example 3](image)

Figure 3: Example 3

Given these definitions of the `Add` and `Mult` traits, here is how a call to `fun` unfolds for `mixin1`, which mixes in `Add` first and `Mult` second:
mixin1.fun(x)
⇒ x * super.fun(x)
⇒ x * Add.fun(x)
⇒ x * (x + super.fun(x))
⇒ x * (x + Fun.fun(x))
⇒ x * (x + x)

And here is how things unfold for mixin2, which mixes in the two traits in the reverse order:

mixin2.fun(x)
⇒ x + super.fun(x)
⇒ x + Mult.fun(x)
⇒ x + (x * super.fun(x))
⇒ x + (x * Fun.fun(x))
⇒ x + (x * x)

That is, mixin1 multiplies first, and then adds, while mixin2 adds first, and then multiplies. This ordering is determined by the aforementioned resolution rule. In particular:

scala> val mixin1 = new Mixin1
scala> mixin1 fun 1
res: Int = 2

scala> mixin1 fun 10
res: Int = 200

scala> val mixin2 = new Mixin2
scala> mixin2 fun 1
res: Int = 2

scala> mixin2 fun 10
res: Int = 110

This is the sense in which modifications are stackable. They are stacked on top of one another, with conflicts resolved in a very specific order. Scala’s conflict resolution strategy is called linearization, the gory details of which are well beyond the scope of CS 18.

Example 4  Finally, a common use case for stackable modifications is the following: an abstract base class, a core class that extends the base class, and various modifications, which are also extensions of the base class, that encode a way of modifying behaviors in the core class.

Here is an example:

abstract class Fun {
  def fun(x: Int): Int
}

trait Add extends Fun {

abstract override def fun(x: Int): Int = x + super.fun(x)

trait Mult extends Fun {
  abstract override def fun(x: Int): Int = x * super.fun(x)
}

class Id extends Fun {
  def fun(x: Int): Int = x
}

class Mixin1 extends Id with Add with Mult
class Mixin2 extends Id with Mult with Add

Here, in Mixin1 and Mixin2, the abstract base class Fun is extended by the usual two modifications, Add and Mult, but now they override the abstract declaration of fun in Fun. (See Figure 4.)

Note: The keywords abstract override can only be used in this combination within traits. They tell the compiler to check that the method being overridden is actually defined concretely somewhere in the hierarchy rooted at Fun. Otherwise, the calls to super would be meaningless!

Figure 4: Example 4

This simple example executes exactly as the code in the previous example does. But note the additional potential for abstraction exhibited here.
Final Observation  The behavior modifications exemplified here, which were achieved by mixing in traits in various orders, could not be achieved via multiple inheritance. In the following hierarchy, \texttt{Id} is-a \texttt{Add} and \texttt{Id} is-a \texttt{Mult}. With multiple inheritance, a call to \texttt{super.fun} from within an \texttt{Id} object would have to be resolved in one way or another; either as addition or as multiplication. The \texttt{Id} object’s behavior could not be modified in both ways.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form:

\texttt{http://cs.brown.edu/courses/cs018/feedback}