Lecture 21: The Many Hats of Scala: OOP

10:00 AM, Mar 14, 2018

Contents

1 Mutation in the Doghouse .................................................. 1
  1.1 Aside: Access Modifiers ............................................. 2
2 Multiple Inheritance? ..................................................... 3
3 Traits as Interfaces ....................................................... 4
4 Traits as Mixins: Overcoming the Diamond Problem ............... 4

Objectives

By the end of these notes, you will know:

- Scala OOP features such as traits and how they work

By the end of these notes, you will be able to:

- Write and mix in traits in your Scala programs

The last lecture gave us a taste of imperative programming in Java. Now let’s have a go at object-oriented programming. First, we’ll revisit the issue of var vs. val in an OOP context.

1 Mutation in the Doghouse

Let’s write a class representing a very basic Dog. Dogs can each speak; by default they say “woof”. They also each have a height, and when they feed, they grow. Since we want a dog’s height to be mutable, we declare the height field as var:

```scala
class Dog(var height: Int) {
  def speak: String = "woof"
  def feed() {
    height += 1
  }
}
```

Now let’s create two dogs (either in the REPL or in our App object). We’ll make one val and one var:
val dog1 = new Dog(100)
var dog2 = new Dog(200)

Let’s look closely at what we’ve actually done. We’ve created two distinct Dog objects. Each of 
these resides at its own memory location. The identifiers dog1 and dog2 reference those addresses.

In this example, we’ll call the two addresses 0x1000 and 0x1010.

```
0x1000

val height = 100

0x1010

val height = 200
```

Now let’s feed the dogs and print out their height:

scala> dog1.feed
scala> dog2.feed
scala> println(dog1.height)
101
scala> println(dog2.height)
102

Notice that although dog1 is declared as a val, that doesn’t keep us from calling that dog’s feed  
method. Since all Dogs have a mutable height field, both dogs grow when fed!

So what good is val in this case? It’s making something immutable, but that something is clearly  
not a Dog. To answer this question, we’ll create a third dog.

```
val dog3 = new Dog(300)
```

Because dog2 is declared as a var, we can overwrite the address it currently holds:

```
dog2 = dog3
```

The name dog2 now references the latest, third dog, and the reference to the second dog at 
0x1010 is now lost. However, if we try to overwrite dog1 with the third dog’s location, Scala will 
complain—dog1 is a val and can’t be mutated!

Whether a data structure (like a dog or a database) is mutable depends on how its fields and 
methods are defined. Similarly, whether a reference is mutable depends on how that reference is 
declared. An immutable reference could point to a mutable object, and mutable references can point 
to immutable objects! Just make sure that you know what is and isn’t mutable—add mutability 
only where it makes sense!

Tim likes to think of these as “doghouses”.

---

1Tim likes to think of these as “doghouses”.

2
1.1 Aside: Access Modifiers

In Scala, there is no need to write getters and setters because all attributes of a class can be accessed via the dot operation, assuming those attributes are not explicitly declared to be private. Anything not explicitly declared to be private is public by default in Scala. Of course, if a field is declared as a val, it cannot be set!

If a field is declared without either val or var, Scala will make it immutable (i.e., a val) but not allow it to be read, either. (To prevent field reads on a field declared with val or var, use the private keyword.) One exception to this policy occurs in case classes, where all fields are vals and thus immutable and readable (which echoes the way we use case classes!)

2 Multiple Inheritance?

In Java, classes can only extend one other class, although they can implement any number of interfaces. Let’s take a look at why Java does this.

In object-oriented design, multiple inheritance raises a dangerous trap called the diamond problem. This problem arises 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 Type D perceive what has been inherited from Type A if both of Type B and Type C inherit from Type A?” In particular, “If Type B and Type C both provide implementations of a method defined in Type A, to which implementation are we referring when a Type D 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.

---

2 When a subtype inherits from multiple supertypes, this is called multiple inheritance.
3 Except Object, which sits at the root of the Java hierarchy and has no base class.
4 A Java class can implement multiple interfaces (as they provide only empty promises!). This is Java’s answer to multiple inheritance.
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 even more powerful than multiple inheritance, called traits. Traits can be used to do what Java interfaces do: provide a specification of methods that must be provided. But they can also be used for much more. In particular, something called stackable modifications. We’ll address both of these via the above whimsical canine example.

### 3 Traits as Interfaces

Look at the Dog class again. Is there any particular reason that the speak method should be unique to Dogs? There are many kinds of animals that might speak, although perhaps none so eager to do so. In Java, we could create an interface for the speak method, which Dog (and all other noisy animals) implement. In Scala, we will do the same thing—it’s just called a trait instead!

```scala
trait Speaker {
  def speak: String
}
```

Now all we have to do is modify the first line of Dog:

```scala
class Dog(var height: Int) extends Speaker {
  ...
}
```

Note that Dog didn’t implement the trait, but extended it. 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, traits are not pure: they can include implementation details—more on this soon!

Because Scala blurs the distinction 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.

### 4 Traits as Mixins: Overcoming the Diamond Problem

Let’s go a bit further and define two subclasses of Dog: Labrador and Mastiff. These each have a custom speak method. Labradors are so energetic that their speech comes out surrounded by stars, and Mastiffs are so loud that their speech is in uppercase. Furthermore, Labradors are always friendly—they can wag their tail at people.

```scala
class Labrador(height: Int) extends Dog(height) {
  override def speak: String = Console.RED + "woof" + Console.RESET
  def wag(name: String): String = "The dog wags its tail at " + name
}
class Mastiff(height: Int) extends Dog(height) {
  override def speak: String = "WOOF"
}
```
This doesn’t seem fair! Perhaps not all Mastiffs are friendly, but surely some of them are. Moreover, what if we decide to add more types of dog, some of which are as friendly as Labradors? This won’t do at all.

To solve our dilemma, we pull the notion of friendliness out of the Labrador class, making it a trait. Every Labrador ever created will mix in that trait:

```scala
trait Friendly extends Dog {
  def wag(name: String): String = "The dog wags its tail at " + name
}

class Labrador(height: Int) extends Dog(height) with Friendly {
  override def speak: String = "*woof*"
}
```

Notice how we made Friendly extend Dog. If traits were really “just interfaces”, this would be backwards! But traits can contain behavior, too. Traits like this are said to “mix in” behavior (and hence we call them mixins). (To avoid potential confusion, we enforce that, for now, only Dogs can exhibit friendliness by having the trait extend Dog.)

Now if we create a lab and a mastiff, both can speak, but only the lab can wag:

```scala
val lab = new Labrador(100)
val m = new Mastiff(200)

scala> println(lab.speak)
*woof*
scala> println(m.speak)
WOOF
scala> println(lab.wag("Tim"))
The dog wags its tail at Tim
```

This still isn’t fair! What about all the friendly mastiffs out there? Let’s pick one at random: Bob. Bob the friendly mastiff. When he was born, someone mixed in the Friendly trait. Yes, we don’t have to apply traits only at class definition—we can apply them at object instantiation, like this:

```scala
@scala val bobTheFriendlyMastiff = new Mastiff(200) with Friendly
```

```scala
scala> println(bobTheFriendlyMastiff.wag("Tim"))
The dog wags its tail at Tim
```

Great! But now that we’re adding traits, we should take another look at speak. After all, colorful or loud speech shouldn’t have to be reimplemented every time a type of dog uses them. We’ll make these traits, too. There’s only one problem: adding asterisks and capitalization are add-ons that modify an existing speech pattern. We don’t want to just say:

```scala
trait EmphaticSpeaker extends Dog {
  override def speak: String = "*woof*"
}
```

because we might be mixing in EmphaticSpeaker with a kind of dog that doesn’t say “woof”, but something else entirely. If we hard-code “woof” in the trait, it will override the speech of whatever

---

5Meow?
we mix it into.

Fortunately, Scala gives us a convenient way to address this problem: `super`. We can use `super.speak` to invoke the original, pre-mixin class’s `speak` method:

```scala
trait EmphaticSpeaker extends Dog {
  override def speak: String = "*" + super.speak + "*
}
trait LoudSpeaker extends Dog {
  override def speak: String = super.speak.toUpperCase
}
```

We can even have some kinds of dog elongate their lowercase vowels:

```scala
trait LongLowercaseVowelSpeaker extends Dog {
  override def speak: String =
    super.speak.toList.flatMap{
      // Note: not toString
      c => if("aeiou".indexOf(c) < 0) List(c) else List(c,c,c)).mkString
}
```

Now we can create dogs with whatever traits we might want, even a hypothetical dog that says “meow” emphatically.

```scala
class ConfusedDog(height: Int) extends Dog(height) with EmphaticSpeaker {
  override def speak: String = "meow"
}
```

It’s easy to mix in multiple traits. For instance:

```scala
val dog3 = new Dog(100) with LoudSpeaker with LongLowercaseVowelSpeaker
val dog4 = new Dog(100) with LongLowercaseVowelSpeaker with LoudSpeaker
val dog5 = new Dog(100) with LongLowercaseVowelSpeaker with LoudSpeaker with EmphaticSpeaker
```

```scala
scala> println(dog3.speak)
WOOF
scala> println(dog4.speak)
WOOOOOOF
scala> println(dog5.speak)
*WOOOOOOF*
```

Notice, however, that **mixins can interfere with one another**. If `LoudSpeaker` gets applied to the original string "woof" first, there will be no lowercase vowels for `LongLowercaseVowelSpeaker` to match and elongate. Scala resolves this issue via *linearization*: the rightmost mixin receives the initial method call. Its parent is the class with the next innermost mixin, and so on. This is all done via the class hierarchy: the rightmost mixin is just the last class-modification in the chain. This figure shows the inheritance for the mixins of `dog5`:
When we call `dog5.speak`, we are really calling the `speak` method in `EmphaticSpeaker`, in a situation where `super` is a `Dog with LongLowerCaseVowelSpeaker with LoudSpeaker`. So when that `EmphaticSpeaker`’s method calls `super.speak`, `LoudSpeaker` gets its turn, and so on. Hence the term “stackable modifications”: mixing in multiple traits implicitly builds a stack of superclasses.

`Dog3` doesn’t elongate its vowels since those vowels are converted to uppercase before `LongLowerCaseVowelSpeaker`’s `flatMap` operation applies. `Dog4` and `Dog5`, however, are alright.

Finally, a word of warning. Suppose we create another breed of dog:

```scala
class Terrier(height: Int) extends Dog(height) with EmphaticSpeaker {
  override def speak: String = "grrrrr"
}
```

The first line here **should not** be read as “Terrier extends Dog, and then mixes in EmphaticSpeaker”. The class that `Terrier` extends is `Dog with EmphaticSpeaker already mixed in`. Why does this matter? Just ask a `Terrier` to speak:

```scala
scala> val thomas = new Terrier(50)
scala> println(thomas.speak)
grrrrr
```

Not very emphatic at all. This is because we’ve overridden the entire chain of mixins. The `speak` method defined in `EmphaticSpeaker` simply never gets called: the `Terrier`’s redefinition overrides it.

**Final Observation**  Whereas a class can only inherit from one superclass, a class can mix in any number of traits. This allows Scala to work around the diamond problem. The flexible behaviors seen here, which were achieved by mixing in traits in various orders, could not be easily achieved via multiple inheritance.
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: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)