Lecture 20: Scala: Classes and Traits

11:00 AM, Mar 12, 2021

Contents

Motivating Question

How do we create a class hierarchy for a type of data with multiple optional features?

Objectives

By the end of these notes, you will know:

- The idea behind traits in Scala
- The difference between classes and traits in Scala
- The difference between Java interfaces/abstract classes and Scala traits

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

- Create classes and traits in Scala
- Use traits to tailor the behavior of objects

1 Scala Classes and Subclasses

This lecture centers around a design challenge, which we will explore in each of Scala and Java.

1.1 Design Challenge for this Lecture

Our goal is to create a class hierarchy for listings of hotel rooms, as might be found on a travel website. All Listings will have a property name, a daily rate, and the number of people it can sleep. Listings also have a list of detailed amenities. Suites are listings that have multiple rooms. Listings and Suites could be marked as *Premium*, meaning that they charge extra for some feature, like a fabulous view. Listings and Suites could also have a kitchenette (featuring a fridge and microwave) and/or a gym.

Listings will support three methods (in addition to `equals` and `toString`):

- `price`, which takes a number of nights and returns the cost of that many nights at the daily rate
- `descr`, which takes no arguments and returns a description of the listing

- `changeRate`, which takes a new daily rate and updates the rate in the listing

In addition, Suites have a method `roomsAtLeast` which takes a number of rooms and returns a Boolean indicating whether the number of rooms is at least the given number.

We're first going to create Listings and Suites in Scala, then we will turn to the features (premium, kitchen, and gym).

2 Scala Classes

Listings, with their concrete fields and methods, would have been a class in Java. Here's a Scala class for Listings:

```
class Listing(val name : String, var rate : Double, val sleeps : Int) {
  var amenities = List("hairdryer")
  def price(nights : Int) = nights * rate
  def descr() = "Lodging at " + name + " fits " + sleeps + " people"
  def changeRate(newRate : Double): Unit = {
    this.rate = newRate
  }
  override def toString() = name + " has " + amenities.toString()
  override def equals(that: Any): Boolean = that match {
    case that: Listing => that.name == this.name && that.sleeps == this.
        sleeps
    case _ => false
  }
}
```

To create an object from this class, we use `new`, as in Java:

```
val l1 = new Listing("hotel A", 100, 4)
```

Several things to note about this definition:

- The class header line defines the default constructor (the sequence of field declarations after the class name are the constructor parameters.

  Side note: you can define additional constructors within the class as needed. See https://alvinalexander.com/scala/how-to-create-multiple-class-constructors-in-scala-alternate-constructors

- The `val` and `var` annotations on the constructor parameters control the visibility and modifiability outside of the class. Any parameter marked as `val` can be accessed outside the class through the usual dot notation (like `l1.name`). Any parameter arked as `var` can be accessed through dot notation and modified (like `l1.rate = 125`). Put differently, these annotations automatically create getters and setters for fields.

- The `override` annotation is part of method annotations. It is required when overriding an existing method.

- Pattern matching is used to help define `equals` methods in Scala.

- Curly braces are used around multi-expression method definitions. If the body contains a single expression, the curly braces can be left off.

2.1 A Subclass for Suites

Suites would have been a subclass in Java: they share considerable code with Listings, but have an additional variable. Here's the Suites subclass written in Scala:

```
class Suite(name : String, rate : Double, sleeps : Int,
            val rooms: Int) extends Listing(name, rate, sleeps) {
  override def descr() = super.descr() + " in " + rooms + " rooms."
  def roomsAtLeast(min: Int) = rooms >= min
}
```

To create an object from this class, we again use `new`, as in Java:

```
val s1 = new Suite("hotel B", 150, 4, 2)
```

Several things to note about this definition:

- The call to `super` from the Java constructor is also embedded in the header line, as part of the `extends` clause.
- `override` is used to extend method definitions
- We omit the `val`/`var` declarations from the parent class variables in the subclass. Otherwise, Scala will require you to put `override` annotations on those field definitions. The subclass inherits the annotations of the parent class automatically.

3 Handling the Features of Listings

Now let's handle premium views, kitchens, and gyms. Each of these features should add to the description of a listing, with phrases like "it offers a superior view" or "it has a gym". Premium views must augment the computation of the daily rate (with an additional per-day charge). Kitchens and gyms add to the list of amenities that a room offers.

How would you have added these features to your class hierarchy in Java. How would you have created listings with different combinations of these features? Think about it and write down something concrete before reading on.

The key phrase here is "different combinations of these features". How can we define features to enable different combinations in Java?

One idea is to make a boolean for each feature in the `Listing` class, then set those to indicate whether an object has the features. This is poor design for two reasons: (1) it clutters objects with code that they don't need for features they don't have, and (2) it makes for a maintenance nightmare if more features get added later (which is common) – you'd have to edit the class each time an optional feature was desired.

Making subclasses won't work either. Imagine that we created (in Java):

```
class Premium extends Listing { ... }
class Kitchen extends Listing { ... }
class Gym extends Listing { ... }
```

How do we make a listing with both a Kitchen and a Gym? The class hierarchy doesn't allow this. With Java subclasses, we'd have to do something like the following:

```
class Premium extends Listing { ... }
class PremiumKitchen extends Listing { ... }
class PremiumKitchenGym extends Listing { ... }
```

With the code for each feature repeated in all of those classes. This does not scale at all.

Fortunately, Scala offers a solution to this problem.

4 Scala Traits

Scala offers a construct called *traits* that are designed for exactly this sort of problem. Let's see an example first, then explain how it works.

```
trait Kitchen extends Listing {
  this.amenities = this.amenities:::List("fridge", "microwave")
  override def descr() = super.descr + " It has a kitchen."
}
```

The `Kitchen` feature builds off of (`extends`) the fields and methods in `Listing`. It adds two new amenities, and extends (overrides) the description with additional text.

Lists in Scala are immutable, so we have to redefine the `amenities` list rather than add to it as in Java. Triple colon (:::) is the append operator in Scala. If you want cons/link, use a double colon. (We will talk about mutable lists later this week).

Here are the traits for the other features.

```
trait Premium extends Listing {
  override def price(nights : Int) = super.price(nights) + (30 * nights)
  override def descr() = super.descr + " It offers a superior view."
}

trait Gym extends Listing {
  this.amenities = this.amenities:::List("treadmill", "weights")
  override def descr() = super.descr + " It has a gym."
}
```

How do we use these traits to create objects with optional features? Here's an example:

```
val ps = new Suite("hotel B", 175, 4, 2) with Premium
val pks1 = new Suite("hotel B", 175.0, 4, 2) with Premium with Kitchen
val pks2 = new Suite("hotel B", 175, 4, 2) with Kitchen with Premium with
    Gym
```

When Scala creates one of these objects, it makes the initial object, then adds each of the traits in order from left to right. If you think of it as a class hierarchy, it's as if `pks2` were a hierarchy like:

```
 Suite
   |
Kitchen
   |
Premium
   |
  Gym
```

So this means that any method call is first sent to `Gym`, then to `Premium`, etc until an appropriate method is found. This same sequence explains how the gym description ends up on the end of the produced string.

4.1 Some Persepctive on Traits

In Scala, traits cover the role of interfaces in Java while being much more powerful. Traits can have their own fields and methods (as we will see as we go forward). They can be mixed into existing classes (though it is possible to constrain which kind of classes they can be added onto).

Traits are an implementation of a general programming languages concept called "mixins". Think about mixins as constructs that transform classes, adding fields and methods. Put differently:

A mixin is a function that takes a class and produces a class

We'll return to this point in the coming weeks. But still, functions from classes to classes to enable flexible design? How's that for a neat extension to functional programming!
