Lecture 8

Reading

Programming in Scala, Chapter 6.1—6.11.[1]

Object Oriented Scala

Although we've focused on functional programming so far, Scala supports Java-style object-oriented programming too. For example, fig. 14.1 defines a `Point` class (not a case class) with two private fields[2] and four methods.

Notice that we didn't have to declare a constructor for `Point`. Scala creates a constructor automatically that initializes the values of `x` and `y`.

We could make the fields public by writing:

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

We can make methods private by writing:

```
private def add(other: Point) = { ... }
```

If we wanted to compute the value of a field, e.g., if we wanted the magnitude to be a field whose value is calculated at construction, we could write:

```
class Point(x: Double, y: Double) {

  val magnitute = math.sqrt(x * x + y + y)

  ...
}
```

In fact, you can think of the entire body of the class as the constructor. For example, the following code prints each time a new point is constructed:

```
class Point(x: Double, y: Double) {

  println(s"Created a new Point with x = $x and y = $y.")

  ...
}
```

These and other features of Scala's classes are described in depth in the assigned reading. However, note that they are mostly syntactic differences. The semantics of classes in Scala and Java are exactly the same (for now).

Converting Functional Code to Object-Oriented Code

Figure 14.3 shows a library for shapes, written in a functional style. The library defines a sealed trait called `Shape` and two constructors for `Circle` and `Square` using case classes. The library defines a function called `inShape` to test whether an (x, y) coordinate is within the given shape. The function uses pattern-matching to handle the two cases for circles and squares.

[1] http://www.artima.com/pins1ed/

[2] Recall that case class fields are public, which is what allows us to use pattern matching. We cannot pattern match on ordinary classes, even if their fields are public.

```scala
class Point(x: Double, y: Double) {

  def getX() = this.x

  def getY() = this.y

  def add(other: Point) = {
    new Point(this.x + other.getX(),
              this.y + other.getY())
  }

  def magnitude() = math.sqrt(x * x + y + y)
}
```

Figure 14.1: A class for points.

```java
class Point {
  private Double x;
  private Double y;

  public Point(Double x, Double y) {
    this.x = x;
    this.y = y;
  }

  public Double getX() {
    return this.x;
  }

  public Double getY() {
    return this.y;
  }

  public Point add(Point other) {
    return new Point(this.x + other.getX(),
                     this.y + other.getY());
  }

  public Double magnitude() = {
    return math.sqrt(x * x + y + y);
  }
}
```

Figure 14.2: A similar class in Java.

```scala
sealed trait Shape
case class Circle(radius: Double) extends Shape // Center of the circle is (0, 0)
case class Square(side: Double) extends Shape // Bottom-left corner of the square is (0, 0)

object Shape {
  def inShape(s: Shape, x: Double, y: Double) = s match {
    case Circle(radius) => math.sqrt(x * x + y * y) <= radius
    case Square(side) => x >= 0 && x <= side && y >= 0 && y <= side
  }
}
```

Figure 14.3: A Functional Shape Library

```scala
trait Shape {
  def inShape(x: Double, y: Double) = {
    if (this.isInstanceOf[Circle]) {
      math.sqrt(x * x + y * y) <= this.asInstanceOf[Circle].radius
    }
    else if (this.isInstanceOf[Square]) {
      val side = this.asInstanceOf[Square].side
      x >= 0 && x <= side && y >= 0 && y <= side
    }
    else {
      sys.error("unknown shape!")
    }
  }
}
class Circle(val radius: Double) extends Shape
class Square(val side: Double) extends Shape
```

Figure 14.4: An object-oriented shape library: do not write code like this.

```
trait Shape {
  def inShape(x: Double, y: Double): Boolean
}

class Circle(radius: Double) extends Shape {
   def inShape(x: Double, y: Double) = math.sqrt(x * x + y * y) <= radius
}

class Square(side: Double) extends Shape {
  def inShape(x: Double, y: Double) = x >= 0 && x <= side && y >= 0 && y <= side
}
```

Figure 14.5: An object-oriented shape library done well.

Object-Oriented Refactoring Done Badly To refactor this code into an object-oriented style, we need to stop using case classes and pattern matching and turn `inShape` into a method instead of a function. A literal translation of the code would be to turn the case classes into classes with public fields, move the method into the `Shape` trait,[3], and rewrite the body to use type-tests and casts. Figure 14.4 shows this refactoring of the code.

Notation In Scala:

```
this.instanceOf[Circle]
```

is the same as `this instanceof Circle` in Java.
In Scala:

```
this.asInstanceOf[Square]
```

is the same as `(Square)this` in Java.

You should already know that this is terrible code! It is an extremely bad idea to use `instanceof` and type-casts in Java (and in Scala) because it *defeats the type system.* The point of the type system in Java (and Scala) is to help you catch errors. When you write code like this, the type system can't help you. **You are not permitted to use `isInstanceOf` and `asInstanceOf`, since writing high-quality code is a major goal of this class.**

Object Oriented Refactoring Done Right The right way to build an object-oriented shape library is shown in Figure 14.5. We've made `inShape` an abstract method in the trait `Shape`[4] and each shape provides an implementation. Note that we are not using any type-casts or type-tests and that the shapes' fields are private, which is typically good practice in object-oriented code. **Most significantly,** notice that the method bodies are exactly the same as the cases in the function from fig. 14.3.

Functional code that uses pattern matching can be converted to equivalent object-oriented code by following this recipe:

1. Functions that use pattern-matching become abstract methods on the type on which they pattern match.

   i.e., a function that looks like this:

   ```
   def f(x: T, args ...): R = x match { ... }
   ```

   Becomes an abstract method:

   ```
   trait T {
     def f(args ...): R
   }
   ```

   Think! Notice that the *function* f takes an argument caled x, but the *method* f does not. Why not?

2. Each case in a function turns into an implementation of the abstract method in the corresponding class.

   i.e., a function that has a case like this:

[3] Think of this trait as an abstract class in Java. Traits are actually more powerful, but we'll get to that later.
[4] Now, we're using the trait like an interface in Java.

```
def f(x: T, args ...): R = x match {
  case C(fields ...) => body
}
```

Becomes an implementation of the method f:

```
class C(fields ...) extends T {

  def f(args ...): R = body
}
```

Think! A key feature of pattern-matching is the *exhuastivity check*. If we forget to write a case in a function, Scala complains that pattern-matching may not be exhaustive and actually lists all the cases that we have not handled. Since we are no longer using pattern-matching, we no longer rely on exhuastivity-checking. Is this okay? How do similar bugs manifest in object-oriented code and can the Scala compiler catch them?

Using Libraries

Libraries and APIs are an important part of programming. Without libraries, we'd be writing a lot of basic functionality from scratch. The Scala and Java *standard libraries* define a variety of useful data structures and programming patterns and are available to all code that you write. SBT makes it easy to use libraries from the Web.

For example, uPickle is a handy library for converting Scala data structures to JSON, which is often used in systems where programs in several languages need to communicate. To use uPickle in a project, create a file called `build.sbt` in the root directory of the project and add the line:

```
libraryDependencies += "com.lihaoyi" %% "upickle" % "0.3.8"
```

After restarting SBT, you can use uPickle as documented on its website. Maven Central has thousands of libraries that you can use in a similar way.

Writing Libraries

Any SBT project can be packaged into a library by simply giving it a name, organization, and version number in the `build.sbt` file. For example, we could use the following `build.sbt` for our functional shape library:

```
name := "fshapes"
organization := "compsci220"
version := "1.0"
```

Before publishing a library on the Web, it is a good idea to test it by publishing it locally. We can run `sbt publish-local` to do so.

When it completes, we can test the library by using it from another project. To do so, we'd have to add this line to its `build.sbt` file:

```
libraryDependencies += "compsci220" %% "fshapes" % "1.0"
```

It takes a little more work to publish libararies online. If you want to learn how to do so, we recommend using Bintray SBT. (It is a lot simpler than using Maven Central directly.)

Reusing and Extending Libraries

Imagine that we wanted to use a library of shapes and we'd found both the functional and object-oriented shape library we wrote earlier on the Web. We can use either one by editing the `build.sbt` file:

```
libraryDependencies += "compsci220" %% "fshapes" % "1.0"
```

or

```
libraryDependencies += "compsci220" %% "jshapes" % "1.0"
```

```
sealed trait ShapeExt
case class Rectangle(width: Double, height: Double) extends ShapeExt
case class OtherShape(s: Shape) extends ShapeExt

object FunctionalExtension {
  def growShape(s: ShapeExt): ShapeExt = s match {
    case OtherShape(Circle(radius)) => OtherShape(Circle(radius * 2))
    case OtherShape(Square(side)) => OtherShape(Square(side * 2))
    case Rectangle(width, height) => Rectangle(width * 2, height * 2)
  }

  def inShape(s: ShapeExt, x: Double, y: Double): Boolean = s match {
    case OtherShape(shape) => Shape.inShape(shape, x, y)
    case Rectangle(width, height) => x >= 0 && y >= 0 && x <= width && y <= height
  }
}
```

Figure 14.6: Extending fshapes with rectangles.

So, which one should we use? Recall that both libraries implement the same features: only two shapes and just one method/function to calculate if a point is within a shape. So, perhaps the choice is just a matter of taste: i.e., whether we prefer to think functionally or in an object-oriented way.

But, let's consider the choices more carefully. In particular, suppose neither library was quite right and we needed to extend them in some way. In fact, our libraries are so small, it's easy to imagine that we'd want a larger variety of shapes and other functions or methods. So, let's see what it takes to extend each library.

Remember that these are meant to be libraries that someone else wrote that we are re-using in our own project. So, we do not want to read or modify their code. (Imagine that the library has millions of lines of code). In fact, it may be a closed-source library, so the original code may not even be available for us to read or modify. So, imagine that all we have to work with is the Scaladoc and some example code.

We'll consider two kinds of extensions:

- *Extending the data model* by adding support for rectangles.

- *Extending the feature set* by adding a routine to double the size of a shape.

Extending the Functional Shape Library

It is easy to write a new function to double the size of a shape. In our own project, we could simply write:

```
object FunctionalExtension {
  def growShape(s: Shape): Shape = s match {
    case Circle(radius) => Circle(radius * 2)
    case Square(side) => Square(side * 2)
  }
}
```

Now, let's add support for rectangles. The obvious thing to do is write:

```
import fshapes._
case class Rectangle(width: Double, height: Double) extends Shape
```

But, this code does not type-check, because `Shape` was *sealed* earlier. This may appear annoying now, but its a good thing that it doesn't work. Remember that the `inShape` function in the library doesn't have a case for `Rectangle`. How could it? Someone else wrote it and didn't anticipate that we wanted support for rectangles. Therefore, by preventing us from extending `Shape`, Scala is ensuring that the library code doesn't fail. (i.e., Scala is ensuring that the exhuastivity analysis it did on the library remains valid.)

However, all is not lost. We can add support for rectangles by *wrapping* the previously defined `Shape` type in a new type. Figure 14.6 shows how to do this. The idea is quite straightforward. We simply have to define a new `inShape` function that handles rectangles and invokes the original `inShape` function on other shapes.

Now, imagine doing this on a large library with hundreds of functions. It would be extremely tiresome. But, this illustrates a *fundamental limitation of functional programming*: it is easy to extends a library by adding new functions, but it is hard to extend a library to support new kinds of data.

```
object JShapesExtension {
  def growShape(shape: Shape) {
    if (this.isInstanceOf[Circle]) {
      new Circle(this.asInstanceOf[Circle].radius * 2)
    }
    else if (this.isInstanceOf[Square]) {
      new Square(this.asInstanceOf[Square].side * 2)
    }
    else if (this.isInstanceOf[Rectangle]) {
      new Rectangle(this.asInstanceOf[Rectangle].width * 2, this.asInstanceOf[Rectangle].height * 2)
    }
    else {
      sys.error("unknown shape!")
    }
  }
}
```

Figure 14.7: Using type-tests and type-casts to grow shows. Does not compile.

Think! Imagine that the original library had a `growShape` function and that we were trying to write the `inShape` function. Would wrapping still work?
Suppose the original library had a constructor for creating composite shapes:

```
case class Overlay(top: Shape, bottom: Shape) extends Shape
```

Would wrapping still work?

Extending the Object-Oriented Shape Library

It is easy to extend the object-oriented library with rectangles. All we have to do is create a new class that extends `Shape` and define its `inShape` method:

```
class Rectangle(width: Double, height: Double) extends Shape {
  def inShape(x: Double, y: Double): Boolean = x >= 0 && y >= 0 && x <= width && y <= height
}
```

Now, let's try to add a feature to double the size of shapes. The object-oriented way is to have a `growShape` method in the `Shape` trait. But, we can't do that because `Shape` was defined in a library that someone else wrote (and we don't have the source code).

Instead, we could write a `growShape` function using type-tests and type-casts, as shown in fig. 14.7. We've already discussed why this is terrible code. Moreover, it has a simpler problem: it does not compile.

Think! Why doesn't the code in fig. 14.7 compile?

An alternative approach uses inheritance, as shown in fig. 14.8. The key idea is to create new types for circles and squares that inherit from the existing libraries. Again, imagine doing this at scale: if the library had dozens of objects and dozens of methods, this would be extremely tedious. This illustrates a *fundamental limitation of object-oriented programming*: it is easy to extends a library with new kinds of data (new classes), but it is hard to extend a library to support new methods.

Think! Suppose the original library had a class that represented composite shapes:

```
class Overlay(top: Shape, bottom: Shape) extends Shape { ... }
```

Would wrapping work?

Perspective

These two examples illustrate that functional and object-oriented styles involve fundamental tradeoffs when it comes to extending and re-using code. There are certain kinds of extensions that can be done in object-oriented style that

```scala
trait Growable {
  def growShape(): Shape
}

class MyCircle(radius: Double) extends Circle(radius) with Growable {
  def growShape(): MyCircle = new MyCircle(radius * 2)
}

class MySquare(side: Double) extends Square(side) with Growable {
  def growShape(): MySquare = new MySquare(side * 2)
}

class Rectangle(width: Double, height: Double) extends Shape with Growable {
  def inShape(x: Double, y: Double: Boolean = x >= 0 && y >= 0 && x <= width && y <= height

  def growShape(): Rectangle = new Rectangle(width * 2, height * 2)
}
```

Figure 14.8: Using inheritance to extend the library.

cannot be done in functional style, and vice versa. It is misguided to argue that one style is better than the other. What matters is how you expect your code to be used (and extended). One could argue that a language that forces you to pick one style of programming limits the kinds of code re-use that are possible.