5 Scala Specifics

Last time, we saw the basics of implementing ADTs in Scala. Much of what we learned, aside from the syntax and the notion of companion objects, applies to most class-based object-oriented languages.

Today, we look at various Scala-specific aspects of the implementation. In particular, every Scala class needs to implement a few specifics methods to ensure that instances of the class "play well" with other aspects of the languages, such as libraries, and the interactive system. If you do not implement those methods — called *canonical methods* — then Scala provides a default implementation for you, which will generally *not* do what you want.

5.1 Canonical Methods

There are three methods that should be present in every class in Scala, which ensures that the class interacts well with the rest of the Scala environment. The point is, even if you don't define those methods, default will be provided, and those defaults most likely will not do exactly what you would like them to do.

5.1.1 The `equals()` Canonical Method

We defined a `isEqual()` operation in the POINT ADT which allowed us to compare points for equality. But it only works on `Point`s. Which makes sense.

Scala, however, has an equality operator `==` which allows it to compare any two values for equality. Intuitively, `a == b` is interpreted as `a.equals(b)`, delegating to an `equals()` method in the class of `a`.

If `a` does not define an `equals()` method, then the default is to use something called *object identity*: two objects are equal (under object identity) if they are *the same actual object*. The idea is that when an object is created, it gets allocated somewhere in memory. Two objects to be the same actual object if they live at the same address in memory. Object identity is very rarely what you want.

For example:

```
val obj1 : Point = Point.cartesian(1.0,2.0)
val obj2 : Point = obj1
```

then `obj1` and `obj2` are the same actual object, so the default `equals()` method for `Point` will say `true`. However, the slight variant:

```
val obj1 : Point = Point.cartesian(1.0,2.0)
val obj2 : Point = Point.cartesian(1.0,2.0)
```

makes `obj1` and `obj2` into two distinct objects, even though they "look the same" — every invocation of `Point.cartesian()` calls `new`, which creates a different object every time. So the defaults `equals()` method would say `false` to `obj1` and `obj2` being equal. Not good.

So the easiest way to take care of this is to redefine `equals()` in `Point` so that it uses our `isEqual()` operation. But we have a slight problem. Scala expects the `equals` method to have the following signature:

```
def equals (other :  Any):Boolean
```

Intuitively, type `Any` asys that `equals` can handle any kind of value, of any type. (We'll come back to type `Any` in Lecture 6.) Now, generally, if we compare a value that is not a value of the ADT to a value of the ADT for equality, the result should be false.

So what we would like is an `equals()` operation that essentially does the following:

```
override def equals (other : Any):Boolean =
  // if other is a Point, return the result of isEqual()
  // if other is not a point, return false
```

First off, note the `override` qualifier on the method, which indicates to Scala that we want to override the default. This is required. The compiler will complain if you don't supply it. All of the canonical methods require this `override` annotation.

So how do we do this check to see if `other` has the right type? There are a few ways to do that, the cleanest is to use *pattern matching* — we're going to do a match on the type of the value supplied as an argument to `equals()`. Technically, we're going to do a match on the *dynamic type* of the value — more on that in Lecture 7.

So here is the structure of our `equals()` method for `Point`:

```
override def equals (other : Any):Boolean =
  other match {
    case that : Point => this.isEqual(that)
    case _ => false
  }
```

The `match` primitive checks the type of `other` against the various cases, and if it finds a match it binds the value *at the appropriate type* within the corresponding branch, and returns

the value of the branch. Here, the first case matches when `other` is of type `Point`, and you can use `that` to refer to the value as a value of type `Point` — since `other` has type `Any` still. The second case is a catch-all, which always matches.

This is our first pass at an `equals()` method. For the time being, this is how we are going to implement `equals()`: do a match, then rely on an underlying `isEqual()` predicate defined in the ADT.

A few side notes. In order for `equals()` to truly behave like an equality, it has to satisfy the three main properties of equality:

Reflexivity: `obj1.equals(obj1)` = *true*

Symmetry: if `obj1.equals(obj2)` = *true*, then `obj2.equals(obj1)` = *true*

Transitivity: if `obj1.equals(obj2)` = *true* and `obj2.equals(obj3)` = *true*, then `obj1.equals(obj3)` = *true*.

These are the three properties that `equals()` must satisfy in order for it to behave like a "good" equality method. Programmers will often unconsciously take as a given that `equals()` satisfies the above properties. It is an implicit specification that `equals()` satisfies the three properties above. Because of this, we will require that `equals()` always satisfies these properties. As we shall see later on, it is difficult to get `equals()` to satisfy them in Scala — in fact, in any object-oriented language. In particular, most naive implementations of equality will fail to satisfy symmetry. My own implementation above, for instance, will not satisfy symmetry in the general case. That may surprise you, and we'll come back to that point later in the course, after we see subtyping.

Now, programming languages (Scala included) do not enforce any of those properties! It would be cool if they did, but it's a very hard problem. Think about it: you can write absolutely anything in an `equals()` method... so you need to be able to check properties of arbitrary code, something we know is hard to do after taking *Logic and Computation*.

5.1.2 The `hashCode()` Canonical Method

The next canonical method turns out to be related to `equals()`, and it is used to define a *hash code* for an object. Intuitively, the hash code of an object is an integer that cen be used to identify (not uniquely) an object. That hash codes are integers makes them useful in data structures such as hash tables.

Suppose you wanted to implement a data structure to represents sets of objects. The main operations you want to perform on sets is adding and removing objects from the set, and checking whether an object is in the set. The naive approach is to use a list, but of course, checking membership in a list is proportional to the size of the list, making the operation expensive when sets become large. A more efficient implement is to use a hash table. A hash

table is just an array of some size n, and each cell in the array is a list of objects. To insert an object in the hash table, you convert the object into an integer (this is what the hash code is used for), convert that integer into an integer i between 0 and $n-1$ using modular arithmetic (e.g., if $n = 100$, then 23440 is 40 (mod 100)) and use i as an index into the array. You attach the object at the beginning of the list at position i. To check for membership of an object, you again compute the hash code of the object, turn it into an integer i between 0 and $n-1$ using modular arithmetic, and look for the object in the list at index i. The hope is that the lists in each cell of the array are much shorter than an overall list of objects would be.

In order for the above to work, of course, we need some restrictions on what makes a good hash code. In particular, let's look again at hash tables. Generally, we will look for the object in the set using the object's `equals()` method — after all, we generally are interested in an object that is indistinguishable in the set, not for that exact same object.

This means that two equal objects must have the same hash code, to ensure that two equal objects end up in the same cell.[3] Thus, two equal objects must have the same hash code. Formally:

> For all objects `obj1` and `obj2`, if `obj1.equals(obj2)` = *true* then `obj1.hashCode()` = `obj2.hashCode()`.

The default implementation of `hashCode()`, if you do not write one, is to use a value based on the location of the object in memory. This works fine when `equals()` is object identity, since this satisfies the above property. (Two objects are equal under object identity if they live at the same address in memory, and therefore their hash codes are equal.)

But if you redefine `equals()`, then to satisfy the above property, you need to redefine `hashCode()`. Because equality is typically defined in terms of the data local to the object — that is, the value of its fields — the hash code will generally be computed from the fields of the object as well. Here is a general recipe for computing hash codes: if the fields of the object are $x_1, \ldots, x_n$, then define $h_0, \ldots, h_n$ as

$$h_0 = 1$$
$$h_i = 41(h_{i-1}) + x_i\texttt{.hashCode()} \qquad \text{for } i = 1, \ldots, n$$

and take the hash code of the object to be h_n.

This way of computing hash codes has the advantage of satisfying another interesting property of hashcodes, namely that the values are somewhat "spread out": given two unequal objects of the same class, their hash codes should be "different enough". To see why we want something like that, suppose an extreme case, that `hashCode()` returns always value 0. (Convince yourself that this is okay, that is, it satisfies the property given in the bullet above!)

[3] Try to think in the above example of a hash table what would happen if two equal objects have hash codes that end up being different mod n.

What happens in the hash table example above? Similarly, suppose that `hashCode()` always returns either 0 or 1. What happens then?

For `Point`, the above discussion yields the following implementation of `hashCode()`:

```
override def hashCode ():Int =
  41 * (
    41 + xpos.hashCode()
  ) + ypos.hashCode()
```

5.1.3 The `toString()` Canonical Method

The final canonical method is `toString()`, which is used to obtain a string representation of the object. This is mostly useful for debugging, or for reporting results to the user. (This is what interactive sessions use to display their results — see §5.2 below.) This method is also automatically called by methods in the Scala library, but figuring out exactly which can be tricky.[4]

By default, `toString()` returns a string made up of the class name of the object, an `@` sign, and an hexadecimal number that may or may not be related to the address of the object in memory. Again, hardly useful. Override this method to get some nice outputs.

Here is the `toString()` method for `Point`:

```
override def toString ():String =
  "cart(" + xpos + "," + ypos + ")"
```

Thus, executing:

```
val p = Point.cartesian(1.0,2.0)
println("result = " + p)
```

— where the `p` will be interpreted as `p.toString()` — will print

```
result = cart(1.0,2.0)
```

on the console.

Here is the final complete code for `Point`, taking everything we've said in this lecture into account:

```
object Point {
```

[4] In reality, `toString()` seems to be invoked from the `valueOf()` method in the `String` class, which is used among other places in the concatenation operation on strings, as well as the `print`/`println` operations.

```
  def cartesian (x:Double,y:Double):Point =
     new Point(x,y)

  def polar (r:Double,theta:Double):Point =
     if (r<0)
       throw new Error("r negative")
     else
       new Point(r*math.cos(theta),r*math.sin(theta))
}

class Point (xpos : Double, ypos : Double) {

  // OPERATIONS

  def xCoord ():Double = xpos

  def yCoord ():Double = ypos

  def distanceFromOrigin ():Double = math.sqrt(xpos*xpos+ypos*ypos)

  def angleWithXAxis ():Double = math.atan2(ypos,xpos)

  def distance (q:Point):Double = math.sqrt(math.pow(xpos-q.xCoord(),2)+
                                            math.pow(ypos-q.yCoord(),2))

  def move (dx:Double, dy:Double):Point = new Point(xpos+dx,ypos+dy)

  def add (q:Point):Point = this.move(q.xCoord(),q.yCoord())

  def rotate (t:Double):Point =
     new Point(xpos*math.cos(t)-ypos*math.sin(t),
               xpos*math.sin(t)+ypos*math.cos(t))

  def isEqual (q:Point):Boolean = (xpos == q.xCoord()) &&
                                  (ypos == q.yCoord())

  def isOrigin ():Boolean = (xpos == 0) && (ypos == 0)


  // CANONICAL METHODS

  override def equals (other : Any):Boolean =
```

```
    other match {
      case that : Point => this.isEqual(that)
      case _ => false
    }

  override def hashCode ():Int =
    41 * (
      41 + xpos.hashCode()
    ) + ypos.hashCode()

  override def toString ():String =
    "cart(" + xpos + "," + ypos + ")"
}
```

Just so that you have more code to look at, here is a second implementation of points that uses the second representation we talked about in class.

```
object Point {

  def cartesian (x:Double, y:Double):Point =
     new Point(true,x,y)

  def polar (r:Double, theta:Double):Point =
     new Point(false,r,theta)
}

class Point (isCart:Boolean, first:Double, second:Double) {
    // if isCart is true, then (first,second) are cartesian coord
    // if isCart is false, then (first,second) are polar coord

    // OPERATIONS

    def xCoord ():Double =
       if (isCart)
         first
       else
         first * math.cos(second)

    def yCoord ():Double =
       if (isCart)
         second
       else
         first * math.sin(second)
```

```
def angleWithXAxis ():Double =
   if (isCart)
     math.atan2(second,first)
   else
     second

def distanceFromOrigin ():Double =
   if (isCart)
     distance(new Point(true,0,0))
   else
     first

def distance (q:Point):Double =
   math.sqrt(math.pow(xCoord() - q.xCoord(),2) +
             math.pow(yCoord() - q.yCoord(),2))

def move (dx:Double,dy:Double):Point =
   new Point(true, xCoord()+dx, yCoord()+dy)

def add (q:Point):Point =
   move(q.xCoord(), q.yCoord())

def rotate (theta:Double):Point =
   if (isCart)
     new Point(true,
               first * math.cos(theta) - second * math.sin(theta),
               first * math.sin(theta) + second * math.cos(theta))
   else
     new Point(false, first, second+theta)

private def normalize (angle:Double):Double =
  if (angle >= 2*math.Pi)
    normalize(angle-2*math.Pi)
  else if (angle < 0)
    normalize(angle+2*math.Pi)
  else
    angle

def isEqual (q:Point):Boolean =
   if (isCart)
     (first==q.xCoord()) && (second==q.yCoord())
```

```
      else
        (first==q.distanceFromOrigin()) &&
           (normalize(second)==normalize(q.angleWithXAxis()))

  def isOrigin ():Boolean =
     if (isCart)
       first==0 && second==0
     else
       first==0

  // CANONICAL METHODS

  override def equals (other : Any):Boolean =
    other match {
      case that : Point => this.isEqual(that)
      case _ => false
    }

  override def hashCode ():Int =
    41 * (
      41 * (
        41 + isCart.hashCode()
      ) + first.hashCode()
    ) + second.hashCode()


  override def toString ():String =
    if (isCart)
      "cart(" + first + "," + second + ")"
    else
      "pol("+ first + "," + second + ")"
}
```

Note that the helper method `normalize()` is private.

5.2 Executing Scala Code

This is the pattern that I'll want you to use to define implementations of ADTs in Scala — at least to a first approximation. This is not ideal — in particular, one can still access the representation directly, that is create instances of the representation without going through the creators. We'll take care of that later.

I have followed a couple of conventions for naming, which you should follow as well. The Scala compiler will not enforce them, but your brain will learn to recognize them and use them to spot some errors some times. Class names are capitalized, like `Point`, or an hypothetical `ColoredPoint`. Method names and variable names are capitalized but for the first letter, like `distance`, or `isEqual`.

All code should be commented. Not putting any comments is a sin that I will not permit you to indulge in. Every class should have a comment at the top indicating the purpose of the class, and every method and variable should have a comment indicating the role of the method or variable.

How do you execute your code? Before executing your code, you must *compile* it. Compilation is the process of taking source code and producing code that the computer can execute more efficiently. The Scala compiler is called `scalac`, and can be invoked from the terminal.

Compiling a file `foo.scala` produces a bunch of `.class` files in the same directory as the source file, one (or more) for every class defined in your source. Once your code has been compiled, you can run it in two ways: as an executable, or within an interactive session.

First, running your code as an executable. To do that, your code must contain a module that implements a `main()` method of the appropriate type. The name of the module is completely irrelevant. Here is such a module that could be used with the `Point` implementation.

```
object Main {

  def main (args:Array[String]):Unit = {
     val point1 = Point.polar(10, math.Pi/2)
     val point2 = Point.cartesian(50,-50)
     val point3 = Point.polar(100, math.Pi/4)

     println(point1.distance(point1.add(point2).rotate(math.Pi/2)
                             .add(point3.rotate(math.Pi/8))))
  }
}
```

Note that the expression for the body of `main()` uses braces, which is a way to have *local definitions* inside a method. Intuitively,

{
definition 1
⋮
definition k
expression
}

is an expression that creates a bunch of local definitions and then evaluates *expression* (which may use the local definitions), returning its result. Definitions can be values (using `val`) or even functions (using `def`). Note also that the return type `Unit` for `main` indicates that there is no result to the function — `Unit` is a type with only one value, `()`.

Once you have such a module in your source code, and after you've compiled your source code, you can run the code by calling the `main()` method of module `Main` from the terminal using `scala Main`. You can have multiple modules with a `main()` function, you just need to specify which one you want to execute when you call `scala`.

The other way to run code is to run it in an interactive session. Suppose you've compiled your code. In that same directory, run `scala` (without any arguments). You'll get something like:

```
Welcome to Scala version 2.8.0.final (Java HotSpot(TM) 64-Bit Server VM, ...
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

At that point, you are in an interactive session, and have access to all the classes compiled in the current directory. This interactive session is very much like the interaction window in DrRacket: you can evaluate expressions such as

```
1 + 1
```

you can define new values such as

```
val p = Point.cartesian(3.0,4.0)
```

and even define new functions such as

```
def square (x:Int):Int = x * x
```

Running code as an executable is usually what you do to run an application when it is done, while interactive sessions are useful for debugging and testing your code interactively.