4 Object-Oriented ADT Implementations

Last time, we talked about implementing ADTs, and we have looked at sample implementations in Scheme (as well as in ACL2).

Those implementations, while they do provide all the operations in the signature and satisfy the specification, are not perfect, however. The main problem is that they expose the representation of the elements of the ADT. This makes it too easy for clients to rely on representation details, such as the "bad client" we saw last time. A client relying on implementation details means that we, as implementers of the ADT, cannot just change the implementation without breaking client code.

Many languages have facilities to prevent the exposure of data representation, thereby ensuring that clients cannot take advantage of it.

One approach, called *signature ascription*, is to "wall off" the data representation and the operations that operate on it, and by only exposing the operations themselves.

Another approach, which is similar in its results but philosophically different, is to push the operations inside the data representation (and hide that data representatino). Because the operations live inside the data representation, they know the data representation (which they need in order to actually implement their required functionality), but clients cannot access that representation. This approach is the basis for object orientation.

We will be using Scala as an implementation language for this course. Scala is similar to Java, and you can think of it as the next step in the evolution of Java. Some remarks on the language: Scala is a class-based object-oriented language. "Object-oriented" here means that every value in the language is an object. "Class-based" means that objects are created from classes. A class is essentially a template, a description of how objects of the class are created.

It helps to think of a program has having two parts. One part, the part that correspond to the source code, is static. (Static means non-moving, which we take to mean non-executing). It represents what information about the program we have before anything executes. In Scala, the only thing we know before a program executes are what classes are defined. Thus, the classes are static. Classes exist, in some sense, even before programs start executing. The other part is the dynamic part, which corresponds to program execution. During execution, instances of the classes, that is, objects, get created, updated, destroyed. Thus, objects are dynamic.

You should probably be aware that not every object-oriented language is class-based. Self,

for example, has no concept of class, but still has objects. Scala is also statically typed — types are associated with variables, and before a program is run those types are checked, to make sure that values of the right kind are stored in variables, or passed to methods. Not every object-oriented language is statically typed. Smalltalk, for example, checks types dynamically, like Scheme does. We'll talk more about types next lecture.

4.1 Object-Oriented Signatures and Specifications

To help us devise implementation for ADTs in object-oriented languages, we consider a slightly different way of writing signatures and specifications. Recall the POINT ADT signature from last time (except I've replaced all `Float` with `Double` for convenience):

```
CREATORS
  cartesian : (Double, Double) -> Point
  polar :     (Double, Double) -> Point

OPERATIONS
  xCoord :              (Point) -> Double
  yCoord :              (Point) -> Double
  angleWithXAxis :      (Point) -> Double
  distanceFromOrigin : (Point) -> Double
  distance :            (Point, Point) -> Double
  move :                (Point, Double, Double) -> Point
  add :                 (Point, Point) -> Point
  rotate :              (Point, Double) -> Point

  isEqual :             (Point, Point) -> Boolean
  isOrigin :            (Point) -> Boolean
```

An object-oriented signature, to a first approximation, consider that operations (but not the creators), which must take at least one argument of the type of the ADT[1] take that value on which they act as an *implicit* argument, as opposed to an *explicit* argument that appears in the argument list. Implicit arguments are meant to capture the idea that the operations live inside an object (an element of the ADT) and therefore have access to that element as an implicit argument. For instance, if `p` and `q` are `Point`s, while we would write `add(p,q)` to add `p` and `q` in a conventional language, in an object-oriented setting we would call the `add` operation inside `p`, usually written `p.add(q)`, and `p` here is considered the implicit argument to `add`, while `q` is an explicit argument.

Thus, here is the object-oriented signature for the POINT ADT:

[1] if not, such an operation probably has no business being part of the ADT.

```
CREATORS
  cartesian : (Double, Double) -> Point
  polar :     (Double, Double) -> Point

OPERATIONS
  xCoord :             () -> Double
  yCoord :             () -> Double
  angleWithXAxis :     () -> Double
  distanceFromOrigin : () -> Double
  distance :           (Point) -> Double
  move :               (Double, Double) -> Point
  add :                (Point) -> Point
  rotate :             (Double) -> Point

  isEqual :            (Point) -> Boolean
  isOrigin :           () -> Boolean
```

Let's change how those operations are used, to understand exactly where the implicit argument to operations is coming from. Creators are invoked as before, e.g., `cartesian(10,20)`. Operations, on the other hand, are invoked on an expression yielding a `Point` value, e.g., `p.xCoord()`, where `p` is a `Point` value, or `p.rotate(2.0).move(3,4)`, again where `p` is a `Point` value.

We can easily adapt the specification of points to this new way of invoking operations — here is the specification from Lecture 2, adapted to the signature above, and presented in such a way that we have exactly two equations per operation, once for each creator.

$$\texttt{cartesian}(x,y)\texttt{.xCoord()} = x$$
$$\texttt{polar}(r,\theta)\texttt{.xCoord()} = r\cos\theta$$

$$\texttt{cartesian}(x,y)\texttt{.yCoord()} = y$$
$$\texttt{polar}(r,\theta)\texttt{.yCoord()} = r\sin\theta$$

$$\texttt{cartesian}(x,y)\texttt{.distanceFromOrigin()} = \sqrt{x^2+y^2}$$
$$\texttt{polar}(r,\theta)\texttt{.distanceFromOrigin()} = r$$

$$\texttt{cartesian}(x,y)\texttt{.angleWithXAxis()} = \begin{cases} \tan^{-1}(y/x) & \text{if } x \neq 0 \\ \pi/2 & \text{if } y \geq 0 \text{ and } x = 0 \\ -\pi/2 & \text{if } y < 0 \text{ and } x = 0 \end{cases}$$
$$\texttt{polar}(r,\theta)\texttt{.angleWithXAxis()} = \theta$$

$$\texttt{cartesian}(x,y)\texttt{.distance}(q) = \sqrt{(x - q\texttt{.xCoord()})^2 + (y - q\texttt{.yCoord()})^2}$$

$$\texttt{polar}(r,\theta)\texttt{.distance}(q) = \sqrt{(p\texttt{.xCoord()} - q\texttt{.xCoord()})^2 + (p\texttt{.yCoord()} - q\texttt{.yCoord()})^2}$$

$$\texttt{cartesian}(x,y)\texttt{.move}(dx,dy) = \texttt{cartesian}(x+dx, y+dy)$$
$$\texttt{polar}(r,\theta)\texttt{.move}(dx,dy) = \texttt{cartesian}(r\cos\theta + dx, r\sin\theta + dy)$$

$$\texttt{cartesian}(x,y)\texttt{.add}(q) = \texttt{cartesian}(x + q\texttt{.xCoord()}, y + q\texttt{.yCoord()})$$
$$\texttt{polar}(r,\theta)\texttt{.add}(q) = \texttt{cartesian}(r\cos\theta + q\texttt{.xCoord()}, r\sin\theta + q\texttt{.yCoord()})$$

$$\texttt{cartesian}(x,y)\texttt{.rotate}(\rho) = \texttt{cartesian}(x\cos\rho - y\sin\rho, x\sin\rho + y\cos\rho)$$
$$\texttt{polar}(r,\theta)\texttt{.rotate}(\rho) = \texttt{polar}(r, \theta+\rho)$$

$$\texttt{cartesian}(x,y)\texttt{.isEqual}(q) = \begin{cases} \textit{true} & \text{if } x = q\texttt{.xCoord()} \text{ and } y = q\texttt{.yCoord()} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{polar}(r,\theta)\texttt{.isEqual}(q) = \begin{cases} \textit{true} & \text{if } r = q\texttt{.distanceFromOrigin()} \text{ and} \\ & \quad \theta \equiv q\texttt{.angleWithXAxis()} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{cartesian}(x,y)\texttt{.isOrigin()} = \begin{cases} \textit{true} & \text{if } x = 0 \text{ and } y = 0 \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{polar}(r,\theta)\texttt{.isOrigin()} = \begin{cases} \textit{true} & \text{if } r = 0 \\ \textit{false} & \text{otherwise} \end{cases}$$

(Where $\equiv$ for angles is defined in Lecture 2.)

4.2 Implementation in Scala

Let's implement the above signature, then. The one decision we have to make, at this point, is how to represent points. The same decision was had to make when we were thinking of implementing points in Scheme last time. Just like in that case, we have two natural representation for points — as a pair of cartesian coordinates, or as a triple where the first element of the triple is a flag indicating whether the next two elements are the coordinates of the point in cartesian coordinates or in polar coordinates. Let's do the first representation first. We define a class `Point` to hold the representation of points. The definition of the class specifies the values that must be supplied to the class to construct an instance of the class:

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

  // OPERATIONS
```

```
  ...

  // CANONICAL METHODS

  ...

}
```

To construct an instance of a point, we will use an expression `new Point(10.2,20.4)` or somesuch, where `xpos` will be bound to `10.2` in the newly created instance, and `ypos` will be bound to `20.4` in the newly created instance. Both `xpos` and `ypos` are available as fields in the instance created. (They are not accessible from outside the instance, though — they are *private*.)

Let's fill in the body of this class.

There are several rules that we will follow when writing classes in this course. Four, to be precise, to be introduced throughout this example. Here is the first one:

(1) The only methods in the class that we should be able to invoke are those corresponding to the operations in the signature, as well as the canonical methods.

Canonical methods will be defined in §4.3 below. Now, the class can define other methods, we just have to make sure they are not accessible from outside the class.

Scala lets us restrict accessibility to methods (and to fields) using the `private` keyword. (Much more can and will be said about `private`.) By default, methods and fields without a qualifier are public.

We will not use fields much in this course. (Most of the time, they will be hidden as arguments to the class, as we did above for `xpos` and `ypos`.) When we do use fields, though, they will always be private.

(2) All fields are private.

Fields are not part of the signature, so the spirit of rule (1) says that they should indeed be private. This is not a big restrictions, as we can always use methods (if the signature tells us to) to read and update fields. Mostly, this is to make sure that the rest of the code does not depend on there being a particular field in the object, so that we can, for instance, change the representation of an ADT without worrying about breaking code elsewhere in our program.

Let's implement the operations:

```
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

  ...
}
```

This is all completely straightforward. A method is defined using

`def` *name* `(`*argname*`:`*argtype*`,...):`*resulttype* `=` *body*

Here, *body* is an expression that returns a value of type *resulttype*. We can use the keyword `this` to stand for the implicit argument in any method body, as in Java. I will often use `this` explicitly to emphasize when I'm invoking a method on the same object.

Classes can define not only methods but fields as well. In the above class, `xpos` and `ypos` are fields, albeit implicit fields, appearing only in the signature of the class. Explicit fields can be defined using

`val` *name* `:` *type* `=` *initvalue*

or

`var` *name* `:` *type* `=` *initvalue*

A `val` field is a field that cannot be updated, while a `var` field can be updated. For us, for the time being, fields are never updated. This is important enough that I will make it a rule that we will only break towards the end of the course:

(3) Fields, once initialized, are never updated.

In combination with rule (2) that makes every field private, this makes every instance of the class *immutable*—once created, it cannot be changed. Immutable instances have a host of advantages: the code is easier to reason about, it is easier to replace the code or debug it, etc. As we will see when we look at mutation, understanding what actually happens when a field is updated can get very tricky when a program uses all the features of Scala. Because of this, and other reasons that we will return to in the course of the semester, we will restrict our attention to immutable instances.

Okay, so we have class `Point`, that lets us create a representation of a point in cartesian coordinates. The only thing missing are the creators. Now, we cannot put the creators within the representation of points, because, intuitively, when we invoke the creators, we may not have any point around. So what do we do with them?

Really, we would like to define two functions `cartesian` and `polar` that live outside the the `Point` class, and that look like:

```
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))
```

But Scala doesn't let us define "free-floating" functions like that. They need to live inside something. It seems silly to define a class *just* to have those two functions live inside it, so we'll use a special kind of class called a *singleton class* — also known as a *module* — that is, a class that has only one instance, and that instance is created automatically for you when the program starts. Here is a possible definition:

```
Object Creators {

  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))
}
```

This creates an instance (called `Creators`) of the class `Creators`, and doesn't let you define new instances of that class. To invoke methods in the `Creators` module, you would call them like you would any other methods, that is, as `Creators.cartesian(10.2,20.4)`, or `Creators.polar(10.0,math.Pi/2)`.

Calling the module `Creators` is not very mnemonic, especially if we have other ADTs around with their own creators. So we shall define a module called `Point` in which the creators for the POINT ADT live — that the module has the same name as the class used for representation of a point is something that Scala allows.[2]

This gives us our fourth rule:

(4) Creators live in a module (singleton class) of the same name as the ADT.

Here is the code for the `Point` module and the `Point` class, which we can put in a file `Point.scala`:

```
object Point {

  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
```

[2]In that situation, the resulting module is sometimes called a *companion object to the class*. The details are actually not that important.

```
  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

  ...

}
```

We are almost done. There's just one bit left to do, namely taking care of the last `...` at the bottom there.

4.3 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.

4.3.1 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()` ACLLSnew, 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*.

4.3.2 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,

[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.

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!) 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()
```

4.3.3 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 the interactive loop uses to display its results.) 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 + ")"
```

[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.

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 {

  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 + ")"
}
```

4.4 Concluding Remarks

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? You can either use the interactive loop, as demo-ed in class, or (and this is more reliable), do like you do in Java and compile your code against a a module that implements a `main` method of the appropriate type. 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))))
  }
}
```

Return type `Unit` for `main` indicates that there is essentially no result to the function.

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, derived using the above rules.

```
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.