6 Code Reuse: Subtyping

Last lecture, we saw several different kind of errors, and I pointed out that some of those errors can be detected at statically, that is, before executing the program, and some can only be detected at dynamically.

Scala, for instance, tries to catch several kind of errors statically. The Scala type system is an algorithm (defined by a set of rules) that takes the source code of a program P, and without actually running P will try to establish that:

> when P executes it will not try to invoke a method on an object that does not define that method.[1]

Recall that we called a program P *safe* when P satisfies this property. The Scala type system ensures that if a program type-checks, then it is safe.

Safety is pretty important. It is the first line of defense, generally, to make sure your code is correct. Of course, a safe program is not necessarily correct — since it may now behave as you want it to behave — but an unsafe program is definitely not correct.

What I want to look at today is a code reuse technique that lets us maximize the applicability of functions we write while still preserving the safety of a program.

Consider the following example. Take (a simplified form of) the POINT ADT we have been playing with:

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

OPERATIONS
  xCoord :              () -> Double
  yCoord :              () -> Double
  add :                 (Point) -> Point
  rotate :              (Double) -> Point
```

[1] As I pointed out last time, in Scala every value is an instance of a class, and thus every type error takes the form of trying to invoke a non-existent method on some object. Other languages would likely add that executing P will not lead to primitive operations being invoked on arguments of the wrong type.

with spec:

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

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

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

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

This is easy to implement in Scala as we saw in Lecture 4 — we choose a representaton that we implement using a `Point` class, and put the creators in a `Point` module.

Now, suppose we write a function[2] `RotateAroundPoint()` to rotate a point a certain angle counter-clockwise around another point. Even though we do not have such an operation in our ADT, it is easy enough to implement using the operations that are provided:

```
// Given point (x,y) returns point (-x,-y)
def negatePoint (p : Point):Point =
  p.add(Point.cartesian(-2*p.xCoord(), -2*p.yCoord()))

// Rotate 'p' by 'a' radians around point 'c'
def rotateAroundPoint (p : Point, a : angle, c : Point):Point =
  p.add(negatePoint(c)).rotate(a).add(c)
```

The idea is simple once you understand change the coordinates frame: we move the origin of the plane so it it is now at point c, rotate around the origin by the given angle, and then we restore the origin to its original location.

Now, clearly, if we call `rotateAroundPoint()` with a `Point` as a first and third arguments, then the result is safe, since the function only relies on the fact that the objects passed as arguments have methods `xCoord()`, `yCoord()`, `add()`, and `rotate()`. And instances of `Point` do have those methods. And indeed, the type checker does not complain if we call `rotateAroundPoint()` with two instances of `Point`.

But notice that we should be able to call `rotateAroundPoint()` using objects that implement `xCoord()`, `yCoord()`, `add()`, and `rotate()` methods, and the result would still be safe. Let's make this a bit more precise:

[2] I'm generally going to call methods inside modules *functions*, because they really do play the role of functions more than methods — in particular, they rarely if ever use their implicit argument.

Definition: D is a *structural subtype* of C if every instance of D implements all the (public) methods defined in C, as well as all the (public) fields defined in C.[3]

(If D is a subtype of C, we call C a suptertype of D.)

So, if we give `rotateAroundPoint()` two instances of a structural subtype of `Point`, the resulting function call should still be safe, at least as the definition of safety we're using is concerned.

Now, note that the only thing that subtyping says is that the subtype should implement all the public methods of its supertype. The implementation of those methods, however, can be completely different — they are only required to be defined.

Consider the following ADT for colored point, the CPoint ADT (assuming we have a class `Color` implementing some Color ADT, the details of which are irrelevant for our purpose):

```
CREATORS
  cartesian : (Double, Double, Color) -> CPoint
  polar :     (Double, Double, Color) -> CPoint

OPERATIONS
  xCoord :              () -> Double
  yCoord :              () -> Double
  color :               () -> Color
  add :                 (CPoint) -> CPoint
  rotate :              (Double) -> CPoint
```

with specification:

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

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

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

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

[3] This is still somewhat imprecise because I should say something about the argument and return types of the methods in question. To a first approximation, you can think of those methods as needing to have the same signature in C and D.

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

$\texttt{polar}(r,\theta,c).\texttt{rotate}(\rho) = \texttt{polar}(r,\theta+\rho,c)$

This ADT is also easy to implement in Scala, using a class `CPoint` and module `CPoint` for the creators.

Now, every instance of `CPoint` implements methods `xCoord()`, `yCoord()`, `add()`, and `rotate()`, so we should be able to pass instances of `CPoint` to `rotateAroundPoint()` and have the function compute a result, without trying to invoke an non-existent method.

And in many languages with structural subtyping, you can do just that. Not in Scala however (or Java, for that matter). If you try the above in Scala, you get a type error when you call `rotateAroundPoint()` passing in instances of `CPoint`.

6.1 Subtyping in Scala

In Scala, subtyping is not automatic, but needs to be declared. Type D is a *nominal subtype* of C if D is a structural subtype of C and is *declared as such*. Scala implements nominal subtyping.

There are several ways to define subtypes in Scala. The most direct is to take a given class C and *subclass* it to create a derived class D, using the `extends` keyword. There are some added subtleties to the use of subclasisng, because subclassing also introduces *inheritane*, that is, the possibility for a subclass to reuse methods in its parent class. We will not use inheritance for the time being, simply because I want to emphasize that subtyping is distinct from inheritance.

Let `Point` be an immutable implementation of the Point ADT, such as we saw in Lecture 4. For instance:

```
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 move (dx:Double, dy:Double):Point =

  def add (q:Point):Point = new Point(xpos+q.xCoord(),ypos+q.yCoord())dy

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

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

  // 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 + ")"
}
```

Assume further an implementation of a Color ADT:

```
object Color {
  def yellow():Color =
    new Color(0)

  def red():Color =
    new Color(1)

  def blue():Color =
```

```
    new Color(2)
}

class Color (color:Int) {

  private val names : Array[String] = Array("yellow","red","blue")

  def code ():Int = color

  private def isEqual (c:Color):Boolean =
    (this.color == c.code())

  // CANONICAL METHODS

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

  override def hashCode ():Int = 41+color.hashCode()

  override def toString ():String =
    if (color<0 || color>2)
      "unknown"
    else
      names(color)
}
```

Here is how we could define an implementation of the CPoint ADT in Scala in such a way that the resulting `CPoint` class is a subtype of `Point` — it relies on the fact that the `CPoint` class extends the `Point` class.

```
object CPoint {

  def cartesian(x:Double,y:Double,c:Color):CPoint =
    new CPoint(x,y,c)

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

```
}

class CPoint (xpos : Double, ypos : Double, col:Color)
  extends Point(xpos,ypos) {

  override def xCoord ():Double = xpos

  override def yCoord ():Double = ypos

  def color ():Color = col

  def add (q:CPoint):CPoint =
    new CPoint(xpos+q.xCoord(),ypos+q.yCoord(),col)

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

  private def isEqual (q:CPoint):Boolean =
    (xpos == q.xCoord()) && (ypos == q.yCoord()) && (col == q.color())

  // CANONICAL METHODS

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

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

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

The only difficulty in the above code is figuring out that because of the way subclassing works, every method we define in a subclass that also happens to be defined in the parent class needs

to be annotated with `override`. As I said, this has to do with inheritance, to which we'll come back later. The other bit of difficulty is figuring out that `CPoint(xpos,ypos,col)` extends `Point(xpos,ypos)` — the best way to understand this is to think that whenever we use a `CPoint(xpos,ypos,col)` in a context that expects a `Point`, then we should treat the `CPoint(xpos,ypos,col)` as a `Point(xpos,ypos)`. Roughly.

This code compiles perfectly fine. In particular, the following code can now be compiled and run:

```
def test ():Unit = {
  val cp1 : CPoint = CPoint.cartesian(1.0,2.0,Color.red())
  val cp2 : CPoint = CPoint.cartesian(1.0,1.0,Color.blue())
  val result = rotateAroundPoint(cp1,math.Pi/2,cp2)

  println("New point = " + result)
}
```

This illustrates how subtyping enables code reuse:

Subtyping lets you reuse client code.

The `rotateAroundPoint()` function, which is client code with respect to the `Point` class, this *same piece of code* can be used to work on both points and colored points, because the class `CPoint` is a subtype of `Point`.

Now, this seems like awfully redundant code. And it is. There is a lot of functionality in common between points and color points that we are not taking advantage of right now. That's what inheritance will be useful for. But that's later. By doing things this way I mean to emphasize that inheritance is something different. **Inheritance is not subtyping.** The two are related, of course, but it is quite possible to have subtyping without inheritance. The above code uses subtyping, and does not rely on inheritance. (It is also possible to have inheritance without subtyping, but that's far less common.) Inheritance brings its own backpack of problems with it, that subtyping by itself does not have.

6.2 The Specification Design Pattern

We saw that once we have an ADT given as a signature and a specification, the first step in coming up with an implementation is to decide on a representation. That can be easy, or it can be difficult, depending on the ADT.

It turns out that subtyping, in conjunction with the way in which we have described our specificatins, provides us with an easy and natural way to represent essentially any ADT. The advantage is that there is no thinking required — we obtain an implementation that satisfies the specification pretty much immediately, without having to torture ourselves coming up

with a representation. One just falls out of the woodwork. The disadvantage is that this natural representation is rarely the most efficient one, so the resulting code may be slow. Now, we are not really focusing on efficiency in this course — we are focusing on correctness — so this won't bother us too too much, but it's good to keep in mind. In particular, this means that we can develop code using this natural implementation, and later, if we decide to make our code more efficient, we can replace this implementation of the ADT with another one that works better, one that is based on a more carefully reasoned representation, and if that new implementation satisfies the specification of the ADT, then we can simply swap it in for the natural implementation and our code should still work.

So what is this natural representation for an ADT. Recall that an ADT is given by a set of creators telling you how to creates elements of the ADT, and operations that are specified by describing what they produce when applied to the elements of the ADT created by the various creators.

The idea is to define a representation class corresponding to each creator, so that each creator creates an instance of the corresponding representation class. The implementation of the operations for the representation class corresponding to creator c will be driven by the equation describing what each operation produces when applied by an element created using c.

These representation classes should be usable wherever we expect an element of the ADT, so we will define a type T corresponding to the ADT in such a way that all the representation classes will be a subtype of T — this will ensure that whenever we have a function that can take a T as an argument, we can give it an instance of one of the representation classes and by subtyping the function should work.

I will call this way of deriving an implementation from the description of ADT the the *Specification Design Pattern*. A design pattern is just a "recipe" for how to write code to achieve a certain objective. Here, the objective is to implement an ADT.

Let me illustrate the Specification Design Patter with the POINT ADT, in full this time. To remind you of the full signature and specification:

```
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
```

and the specification:

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

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

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

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

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

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

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

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

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

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

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

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

The first step is to define a representation class for each of the creators. The names are unimportant, but chosing a combination ofthe ADT name and the creator name is a good choice. Each of the representation classes is parameterized by the arguments with which their corresponding creator is called.

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

  // OPERATIONS
  ...

  // CANONICAL METHODS
  ...
}

class PolarPoint (r:Double, theta:Double) {

  // OPERATIONS
  ...

  // CANONICAL METHODS
  ...
}
```

The second step is to define the methods in those representation classes, one for every operation in the ADT, and where the implementation is taken from the corresponding equation in the specification — that is, the implementation of operation *op* in representation class T_c corrsponding to creator c is read off the equation for operation *op* applied to creator c in the specification. Doing this for the representation classes above, and adding in the canonical methods, yields the following:

```
class CartesianPoint (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 CartesianPoint(xpos+dx,ypos+dy)

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

def rotate (t:Double):Point =
  new CartesianPoint(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 toString ():String =
  "cartesian(" + xpos + "," + ypos + ")"

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()
}


class PolarPoint (r:Double, theta:Double) {

  // OPERATIONS

  def xCoord ():Double = r * math.cos(theta)

  def yCoord ():Double = r * math.sin(theta)

  def angleWithXAxis ():Double = theta

  def distanceFromOrigin ():Double = r

  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 CartesianPoint(xCoord()+dx, yCoord()+dy)

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

  def rotate (angle:Double):Point =
    new PolarPoint(r, theta+angle)

  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 =
    (r == q.distanceFromOrigin()) &&
      (normalize(theta) == normalize(q.angleWithXAxis()))

  def isOrigin ():Boolean = (r == 0)
```

```
  // CANONICAL METHODS

  override def toString ():String =
    "polar(" + r + "," + theta + ")"

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

  override def hashCode ():Int =
    41 *
      (41 + r.hashCode()
    ) + theta.hashCode()
}
```

Of course, in order for all of this to make sense, we need to have a class `Point` to act as a supertype for those representation classes. The interesting thing of course is that the only purpose for `Point` to exist is to act a convenient supertype for the representation classes. It has no real functionality of its own, though, since all the action is taken care of in the representation classes. So we can define `Point` as an *abstract class*, that is, an incomplete class that cannot be instantiated because it is missing pieces. (In contrast, a class that *can* be instantiated is sometimes called a *concrete class*.) Here is the abstract class `Point`:

```
abstract class Point {
  def xCoord ():Double
  def yCoord ():Double
  def angleWithXAxis ():Double
  def distanceFromOrigin ():Double
  def distance (q:Point):Double
  def move (dx:Double,dy:Double):Point
  def add (q:Point):Point
  def rotate (theta:Double):Point
  def isEqual (q:Point):Boolean
  def isOrigin ():Boolean
```

Notice that the methods are declared but not implemented. You can think of them as a promise that they will be implemented in subclasses. (The type system will check that for you, so that if you forget to implement one of them, the system will complain.)

Now the only thing remaining to do is to tell the system that the two representation classes `CartesianPoint` and `PolarPoint` should subclass `Point` (so that they can be considered

subtypes of `Point`), and to define a module `Point` containing our creators, each in charge of creating a new instance of the appropriate representation class, and *voilà*. Here is the final code:

```
object Point {

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

  def polar (r:Double,theta:Double):Point =
    if (r<0)
      throw new Error("r negative")
    else
      new PolarPoint(r,theta)
}


abstract class Point {
  def xCoord ():Double
  def yCoord ():Double
  def angleWithXAxis ():Double
  def distanceFromOrigin ():Double
  def distance (q:Point):Double
  def move (dx:Double,dy:Double):Point
  def add (q:Point):Point
  def rotate (theta:Double):Point
  def isEqual (q:Point):Boolean
  def isOrigin ():Boolean
}


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

  // 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 CartesianPoint(xpos+dx,ypos+dy)

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

  def rotate (t:Double):Point =
    new CartesianPoint(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 toString ():String =
    "cartesian(" + xpos + "," + ypos + ")"

  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()
}


class PolarPoint (r:Double, theta:Double) extends Point {
```

```
// OPERATIONS

def xCoord ():Double = r * math.cos(theta)

def yCoord ():Double = r * math.sin(theta)

def angleWithXAxis ():Double = theta

def distanceFromOrigin ():Double = r

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 CartesianPoint(xCoord()+dx, yCoord()+dy)

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

def rotate (angle:Double):Point =
  new PolarPoint(r, theta+angle)

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 =
  (r == q.distanceFromOrigin()) &&
    (normalize(theta) == normalize(q.angleWithXAxis()))

def isOrigin ():Boolean = (r == 0)

// CANONICAL METHODS

override def toString ():String =
  "polar(" + r + "," + theta + ")"
```

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

  override def hashCode ():Int =
    41 *
      (41 + r.hashCode()
    ) + theta.hashCode()
}
```

Here is a formal description of the Specification Design Pattern for implementing an immutable ADT that is specified using an algebraic specification. Let T be the name of the ADT.

(1) Define a module (singleton class) named T.

(2) Define an abstract class named T.

(3) For each basic creator c, define a concrete subclass T_c of abstract class T whose parameters correspond to the arguments that are passed to c.

(4) For each creator c of the ADT, define a method in module T that creates and returns a new instance of the subclass that corresponds to c.

(5) For each operation f of the ADT, declare a method f in abstract class T.

(6) For each operation f in the ADT and for each concrete class T_c, define f as a method in T_c that takes the arguments given by the signature of f and returns the value specified by the algebraic specification for operation f acting on creator c. If the algebraic specification does not specify this case, then the code for f should throw a `RuntimeException` such as an `IllegalArgumentException`.

(7) Implement the canonical methods in each of the concrete classes T_c.