

16 Code Reuse: Inheritance and Delegation

In this lecture, we revisit points and colored points, and see the extent to which we can reuse code *in the implementations* of those ADTs. Until now, most of our reuse has been client-centric. But there are lots of opportunities for code reuse on the implementation side. Unfortunately, reuse is more subtle and problem-prone on the implementation side.

Before we start, recall our POINT 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
```

and our CPOINT ADT:

CREATORS

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

OPERATIONS

```
xCoord :        () -> Double
yCoord :        () -> Double
angleWithXAxis : () -> Double
distanceFromOrigin : () -> Double
distance :      (CPoint) -> Double
```

```

move :           (Double,Double) -> CPoint
add :           (CPoint) -> CPoint
rotate :        (Double) -> CPoint
isEqual :       (CPoint) -> Boolean
isOrigin :      () -> Boolean
color :         () -> Color
updateColor :   (Color) -> CPoint

```

16.1 Innocuous Inheritance

Most object-oriented languages, including Scala, make a code reuse technique available to you: *inheritance*. Inheritance lets you *reuse implementation code*. (Contrast this with subclassing, which lets you reuse client code.) Inheritance is an implementation technique — a client generally couldn’t care less if you implement something via inheritance or not. Inheritance is related to subtyping, but it is a different notion. Unfortunately, most languages conflate the two, which tends to confuse matters somewhat.

Inheritance is powerful, and like any powerful tool, its power must be wielded wisely. At its core, inheritance is a very simple idea: when you create a subtype, instead of reimplementing every method in the subtype, you can just say that you will reuse methods that are already defined in the supertype. This is often called *extending* a class — and the source of the keyword `extends` that Scala and Java use to define subtypes. Inheritance basically lets us only write the “new” stuff when defining a subtype. Everything else comes from the definition of the supertype.

In my view, there are really two kinds of inheritance: innocuous, and non-innocuous. Innocuous inheritance is completely unproblematic and a nice way to reuse code. Non-innocuous inheritance is trickier and may introduce subtle bugs.

Innocuous inheritance is the idea that we have, say two class A and B that are subtypes of another class C . Both A and B have a method m implemented in exactly the same way — sometimes a helper function. The idea is that we can take that definition of m in A and B and hoist it up into C . Then inheritance says that m is now available in A and B (just like before), but it needs only be defined once. This is innocuous because m was defined to work with A and B in the first place (since that’s where they lived before the move up the subtyping hierarchy). We just changed the place where m was defined.

Non-innocuous inheritance is the idea that we have a class A that is a subtype of B , and B has a method m that A wants to reuse, and therefore simply inherits it from B instead of redefining it. The reason why this is different than the situation described in the previous paragraph is that m is a method meant to work with B . That we are reusing it in A may make sense, or may not, or may work slightly differently, because A makes some changes to B , of which m is unaware. So we have to be more careful, and make sure that the behavior that we expect from m is in fact the one we get when we use m in A .

I will return to non-innocuous inheritance in upcoming lectures, but for now, let's use innocuous inheritance to get us some code reuse. Consider the implementation of the POINT ADT we obtain if we apply the usual Interpreter Design Pattern:

```
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)

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

    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 =
      Point.cartesian(xpos+dx, ypos+dy)

    def add (q:Point):Point =
      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()
}

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

  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 =
    Point.cartesian(xCoord()+dx, yCoord()+dy)
}

```

```

def add (q:Point):Point =
  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()
}
}

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
}

```

We see that there is a lot of redundant code in classes `CartesianPoint` and `PolarPoint` — specifically, methods `distance()`, `move()`, `add()`, and `equals()` are all essentially the same, modulo some uses of `xpos` and `ypos` in `CartesianPoint` instead of the equivalent `xCoord()` and `yCoord()`.

We can hoist those common definitions into a common supertype. Now, `Point` would seem a reasonable candidate here, and indeed, we can lift the common methods into `Point`, and inheritance would make those available to its subtypes. But `Point` is abstract, and is accessible to the rest of the world, and any code we put in there will also be accessible to the rest of the world — for instance, other classes that are subtypes of `Point`, such as `CPoint`, will also inherit those definitions, which may not really make sense for them. To keep `Point` as abstract as possible, let me instead define a new abstract class `Common` sitting midway between `CartesianPoint` and `PolarPoint`, and `Point`. The sole purpose of `Common` is to serve as a repository of the common methods between `CartesianPoint` and `PolarPoint`, and since it will be a supertype of both, the common methods will be available to the subtypes by inheritance. Here's one way to do it:

```

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)

  private abstract class Common extends Point {

```

```

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 =
  Point.cartesian(xCoord()+dx, yCoord()+dy)

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

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

private class CartesianPoint (xpos:Double, ypos:Double) extends Common
{

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

private class PolarPoint (r:Double, theta:Double) extends Common {

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

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

    def angleWithXAxis ():Double = theta

    def distanceFromOrigin ():Double = r

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

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
}

```

This trick can be pulled off pretty much any time you have two identical methods in two subtypes.

Before turning to non-innocuous forms of inheritance, let's take a detour via another, simpler, form of code reuse in implementations, that happens to provide a nice path to understanding inheritance in general.

16.2 Delegation

Consider the implementation of ADT POINT above — either one, the one with the **Common** class or the one without.

Now also consider the implementation of ADT CPOINT, again via the Interpreter Design Pattern:

```

object CPoint {

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

  def polar(r:Double,theta:Double,c:Color):CPoint =
    if (r<0)
      throw new Error("r negative")
    else
      new PolarCPoint(r,theta,c)

  private class CartesianCPoint (xpos:Double, ypos:Double, c:Color)
                                extends CPoint {

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

    def move (dx:Double,dy:Double):CPoint =
      new CartesianCPoint(xpos+dx, ypos+dy,c)

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

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

```

```

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

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

def color ():Color = c

def updateColor (nc:Color):CPoint =
  new CartesianCPoint(xpos,ypos,nc)

// BRIDGE METHODS

def distance (q:Point):Double =
  Point.cartesian(xpos,ypos).distance(q)

def add (q:Point):Point =
  Point.cartesian(xpos,ypos).add(q)

def isEqual (q:Point):Boolean = q match {
  case cq:CPoint => isEqual(cq)
  case _ => false
}

// CANONICAL METHODS

override def toString ():String =
  "cartesian(" + xpos + "," + ypos + "," + c + ")"

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()
  )

```

```

    ) + c.hashCode()
}

private class PolarCPoint (r:Double, theta:Double, c:Color)
                                extends CPoint {

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

    def move (dx:Double,dy:Double):CPoint =
        new CartesianCPoint(xCoord()+dx, yCoord()+dy,c)

    def add (q:CPoint):CPoint =
        new CartesianCPoint(xCoord()+q.xCoord(),
            yCoord()+q.yCoord(),
            q.color())

    def rotate (angle:Double):CPoint =
        new PolarCPoint(r, theta+angle,c)

    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:CPoint):Boolean = {
        (r == q.distanceFromOrigin()) &&
        (normalize(theta) == normalize(q.angleWithXAxis())) &&

```

```

    c==q.color()
  }

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

  def color ():Color = c

  def updateColor (nc:Color):CPoint =
    new PolarCPoint(r,theta,nc)

  // BRIDGE METHODS

  def distance (q:Point):Double =
    Point.polar(r,theta).distance(q)

  def add (q:Point):Point =
    Point.polar(r,theta).add(q)

  def isEqual (q:Point):Boolean = q match {
    case cq:CPoint => isEqual(cq)
    case _ => false
  }

  // CANONICAL METHODS

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

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

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

```

```

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

  def color ():Color
  def updateColor (nc:Color):CPoint

  // bridge methods
  def distance (q:Point):Double
  def add (q:Point):Point
  def isEqual (q:Point):Boolean
}

```

As usual, because we want `CPoint` to be a subtype of `Point`, we need to implement bridge methods. I've done them in a different way here — have a look.

Clearly, we could pull the same trick as we did in `Point` and hoist some of the common definitions of methods from the implementation classes of `CPoint` into an abstract class. But we can do better. Notice that there is a *lot* of code redundancy between `CPoints` and `Points`. In fact, most of the methods for `CPoints` have been lifted from `Point`, and modified slightly in a few cases.

So can we implement `CPoint` in such a way that we can reuse the effort we've put into `Point`? The answer is yes, up to a point. The big problem right now is that `Point` is pretty well sealed off. That was the point of the course until now: hide implementation details as much as possible. But hiding implementation details makes it difficult to reuse implementation code.

Thankfully, in this case, we can reuse implementation code without actually knowing how `Points` are implemented. The idea is to consider a colored point to be simply a point with a color attached. We know how to deal with points (we have a class `Point` to work with them), so most of the work can be done there. So how we're going to do this is as follows: in each of the representation classes of `CPoints`, we will create a little underlying point by calling the appropriate creator of `Point` with the same arguments as the representation. Methods of

CPoints that do not involve colors or do not need to construct CPoints will simply *delegate* to the underlying Point we've created — which is called a *delegate*. This approach is called *delegation*. Here's the code:

```
object CPoint {

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

  def polar(r:Double,theta:Double,c:Color):CPoint =
    if (r<0)
      throw new Error("r negative")
    else
      new PolarCPoint(r,theta,c)

  private class CartesianCPoint (xpos:Double, ypos:Double, c:Color)
    extends CPoint {

    // delegate -- takes care of point-related operations

    val del:Point = Point.cartesian(xpos,ypos)

    // these methods can all be delegated

    def xCoord ():Double = del.xCoord()
    def yCoord ():Double = del.yCoord()
    def distanceFromOrigin ():Double = del.distanceFromOrigin()
    def angleWithXAxis ():Double = del.angleWithXAxis()
    def distance (q:CPoint):Double = del.distance(q)
    def isOrigin ():Boolean = del.isOrigin()

    def distance (q:Point):Double = del.distance(q)
    def add (q:Point):Point = del.add(q)

    // these method cannot be easily delegated
    // (they create new CPoints, or they rely on colors)

    def move (dx:Double,dy:Double):CPoint =
      new CartesianCPoint(xpos+dx, ypos+dy,c)

    def add (q:CPoint):CPoint =
```

```

    new CartesianCPoint(xpos+q.xCoord(),ypos+q.yCoord(),q.color())

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

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

def color ():Color = c

def updateColor (nc:Color):CPoint =
    new CartesianCPoint(xpos,ypos,nc)

// BRIDGE METHODS

def isEqual (q:Point):Boolean = q match {
    case cq:CPoint => isEqual(cq)
    case _ => false
}

// CANONICAL METHODS

override def toString ():String =
    "cartesian(" + xpos + "," + ypos + "," + c + ")"

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()
    ) + c.hashCode()
}

```



```

private class PolarCPoint (r:Double, theta:Double, c:Color) extends
  CPoint {

  // delegate

  val del:Point = Point.polar(r,theta)

  def xCoord ():Double = del.xCoord()
  def yCoord ():Double = del.yCoord()
  def distanceFromOrigin ():Double = del.distanceFromOrigin()
  def angleWithXAxis ():Double = del.angleWithXAxis()
  def distance (q:CPoint):Double = del.distance(q)
  def isOrigin ():Boolean = del.isOrigin()

  def distance (q:Point):Double = del.distance(q)
  def add (q:Point):Point = del.add(q)

  def move (dx:Double,dy:Double):CPoint =
    new CartesianCPoint(xCoord()+dx, yCoord()+dy,c)

  def add (q:CPoint):CPoint =
    new CartesianCPoint(xCoord()+q.xCoord(),yCoord()+q.yCoord(),q.color
  ())

  def rotate (angle:Double):CPoint =
    new PolarCPoint(r, theta+angle,c)

  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:CPoint):Boolean = {
    (r == q.distanceFromOrigin()) &&
    (normalize(theta) == normalize(q.angleWithXAxis())) &&

```

```

    c==q.color()
  }

  def color ():Color = c

  def updateColor (nc:Color):CPoint =
    new PolarCPoint(r,theta,nc)

  // BRIDGE METHODS

  def isEqual (q:Point):Boolean = q match {
    case cq:CPoint => isEqual(cq)
    case _ => false
  }

  // CANONICAL METHODS

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

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

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

```

```

abstract class CPoint extends Point {

  def xCoord ():Double
  def yCoord ():Double
  def angleWithXAxis ():Double
  def distanceFromOrigin ():Double

```

```

def distance (q:CPoint):Double
def move (dx:Double,dy:Double):CPoint
def add (q:CPoint):CPoint
def rotate (theta:Double):CPoint
def isEqual (q:CPoint):Boolean
def isOrigin ():Boolean

def color ():Color
def updateColor (nc:Color):CPoint

// bridge methods
def distance (q:Point):Double
def add (q:Point):Point
def isEqual (q:Point):Boolean
}

```

Note what is going on — we implement all methods in the representation classes `CartesianCPoint` and `PolarCPoint`, except that for many of these methods, we delegate to the underlying point that we constructed when we instantiated the representation. In this way, we get to reuse code from the `Point` class. Note that this does not rely on inheritance, or in fact in even knowing how `Points` are implemented. The reason why the code above is important is that we can think of (non-innocuous) inheritance as a form of *implicit delegation*, and the above code will correspond pretty closely to what our code using inheritance will do, except where everything is spelled out completely.

The main purpose of the code above is to serve as a starting point to understand non-innocuous inheritance. However, if we truly wanted to use the code above, we could improve it by noticing that there are a lot of methods in common between `CartesianCPoint` and `PolarCPoint` — all of those involving delegation, for instance — especially after we replace a few `xpos` and `ypos` in `CartesianCPoint` by `xCoord()` and `yCoord()`, respectively. There, we could pull off the trick from §16.1 and define an abstract class `Common` sitting above `CartesianCPoint` and `PolarCPoint` and implementing the methods in common between the two classes and making them available by (innocuous inheritance). Here is the code:

```

object CPoint {

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

  def polar(r:Double,theta:Double,c:Color):CPoint =
    if (r<0)
      throw new Error("r negative")
    else

```

```

    new PolarCPoint(r,theta,c)

private abstract class Common extends CPoint {

    // need to delclare that there will be a 'del' field in
    // the subtypes for this class to compile

    val del : Point

    def xCoord ():Double = del.xCoord()
    def yCoord ():Double = del.yCoord()
    def distanceFromOrigin ():Double = del.distanceFromOrigin()
    def angleWithXAxis ():Double = del.angleWithXAxis()
    def distance (q:CPoint):Double = del.distance(q)
    def isOrigin ():Boolean = del.isOrigin()

    def distance (q:Point):Double = del.distance(q)
    def add (q:Point):Point = del.add(q)

    def move (dx:Double,dy:Double):CPoint =
        new CartesianCPoint(xCoord()+dx, yCoord()+dy,c)

    def add (q:CPoint):CPoint =
        new CartesianCPoint(xCoord()+q.xCoord(),yCoord()+q.yCoord(),q.color
        ())

    def color ():Color = c

    def isEqual (q:Point):Boolean = q match {
        case cq:CPoint => isEqual(cq)
        case _ => false
    }

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

```

```

private class CartesianCPoint (xpos:Double, ypos:Double, c:Color)
  extends CPoint {

  // delegate -- takes care of point-related operations

  val del:Point = Point.cartesian(xpos,ypos)

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

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

  def updateColor (nc:Color):CPoint =
    new CartesianCPoint(xpos,ypos,nc)

  // CANONICAL METHODS

  override def toString ():String =
    "cartesian(" + xpos + "," + ypos + "," + c + ")"

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

private class PolarCPoint (r:Double, theta:Double, c:Color) extends
  CPoint {

  // delegate

  val del:Point = Point.polar(r,theta)

  def rotate (angle:Double):CPoint =
    new PolarCPoint(r, theta+angle,c)

```

```

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:CPoint):Boolean = {
  (r == q.distanceFromOrigin()) &&
  (normalize(theta) == normalize(q.angleWithXAxis())) &&
  c==q.color()
}

def updateColor (nc:Color):CPoint =
  new PolarCPoint(r,theta,nc)

// CANONICAL METHODS

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

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

abstract class CPoint extends Point {

  def xCoord ():Double
  def yCoord ():Double
  def angleWithXAxis ():Double
  def distanceFromOrigin ():Double
  def distance (q:CPoint):Double
  def move (dx:Double,dy:Double):CPoint
  def add (q:CPoint):CPoint

```

```

def rotate (theta:Double):CPoint
def isEqual (q:CPoint):Boolean
def isOrigin ():Boolean

def color ():Color
def updateColor (nc:Color):CPoint

// bridge methods
def distance (q:Point):Double
def add (q:Point):Point
def isEqual (q:Point):Boolean
}

```

This is already a big improvement. The representation classes now truly only hold what is specific to the representations — everything else is pushed to the `Common` superclass.

16.3 Another Example of Delegation

The code in §16.2 was written in such a way to allow us to make the jump to the corresponding code that uses inheritance (which we will see next).

However, delegation by itself has its uses, and if we wanted to use delegation seriously, we could have structured our code to take advantage of it much more than we have above.

To illustrate another way in which you can set up delegation to reuse code from the implementation of another ADT, here is another implementation of `CPoint`. Here, we push the idea that a `CPoint` is just a `Point` with a color attached to its limit, by simply representing a `CPoint` as a combination of a `Point` and a color. The code has the same structure as that obtained from the Interpreter Design Pattern, but instead of there being two representation classes, there is a single representation class `CPointImpl` that wraps a color around a `Point`. Most methods delegate to the underlying `Point`, involving color only when needed.

```

object CPoint {

  def cartesian(x:Double,y:Double,c:Color):CPoint = {
    // create delegate
    val del:Point = Point.cartesian(x,y)
    new CPointImpl(del,c)
  }

  def polar(r:Double,theta:Double,c:Color):CPoint = {
    // create delegate
    val del:Point = Point.polar(r,theta)
    new CPointImpl(del,c)
  }
}

```

```

}

private class CPointImpl (del:Point, c:Color) extends CPoint {

  // easy delegations

  def xCoord ():Double = del.xCoord()
  def yCoord ():Double = del.yCoord()
  def distanceFromOrigin ():Double = del.distanceFromOrigin()
  def angleWithXAxis ():Double = del.angleWithXAxis()
  def distance (q:CPoint):Double = del.distance(q)
  def isOrigin ():Boolean = del.isOrigin()

  // delegations where we do something extra

  def isEqual (q:CPoint):Boolean =
    (del.isEqual(q) && c==q.color())

  // delegations where we need to "rebuild" a CPoint

  def move (dx:Double,dy:Double):CPoint =
    new CPointImpl(del.move(dx,dy),c)

  def add (q:CPoint):CPoint =
    new CPointImpl(del.add(q),q.color())

  def rotate (t:Double):CPoint =
    new CPointImpl(del.rotate(t),c)

  // methods specific to colored points

  def color ():Color = c

  def updateColor (nc:Color):CPoint =
    new CPointImpl(del,nc)

  // BRIDGE METHODS (some by delegation):

  def distance (q:Point):Double = del.distance(q)

  def add (q:Point):Point = del.add(q)

```



```

def isEqual (q:Point):Boolean = q match {
  case cq:CPoint => this.isEqual(cq)
  case _ => false
}

// CANONICAL METHODS

override def toString ():String =
  "cpoint("+del+", "+c+")"

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

override def hashCode ():Int =
  41 * (
    41 + del.hashCode()
  ) + c.hashCode()
}
}

abstract class CPoint extends Point {
  def xCoord ():Double
  def yCoord ():Double
  def angleWithXAxis ():Double
  def distanceFromOrigin ():Double
  def distance (q:CPoint):Double
  def move (dx:Double,dy:Double):CPoint
  def add (q:CPoint):CPoint
  def rotate (theta:Double):CPoint
  def isEqual (q:CPoint):Boolean
  def isOrigin ():Boolean
  def color ():Color
  def updateColor (nc:Color):CPoint

  // bridge methods
  def distance (q:Point):Double
  def add (q:Point):Point
  def isEqual (q:Point):Boolean
}

```