17 From Delegation to Inheritance

Last time, we saw how to reuse code from the implementation of Point in CPoint via delegation. Recall the setup: we have an implementation of the POINT ADT using the Interpreter Design Pattern, using two representation classes CartesianPoint and PolarPoint, and an implementation of the CPOINT ADT using the Interpreter Design Pattern, using two representation classes CartesianCPoint and PolarCPoint. The idea is that the code for CartesianCPoint shares a lot of similarity with that of CartesianPoint, and similarly for PolarCPoint and PolarPoint. Delegation gives us a way to reuse the code from CartesianPoint in CartesianCPoint, and from PolarPoint in PolarCPoint.

17.1 A Simple Example of Inheritance

Let's start simple. Here is the code from last lecture — or at least, a minor variant of it — where we delegate the easy methods, and simply recode the ones that we cannot easily delegate.

```scala
object CP {  
  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 CP {  
      // 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 isOrigin ():Boolean = del.isOrigin()
def distance (q:Point):Double = del.distance(q)
def add (q:Point):Point = del.add(q)

// special: uses an upcast
def distance (q:CPoint):Double = del.distance(q)

// these method cannot be easily delegated

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())

// Specific to color points
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 PolarCPPoint (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 isOrigin ():Boolean = del.isOrigin()
    def distance (q:Point):Double = del.distance(q)
    def add (q:Point):Point = del.add(q)

    // special: uses an upcast
    def distance (q:CPPoint):Double = del.distance(q)

    // these method cannot be easily delegated

    def move (dx:Double,dy:Double):CPPoint =
      new CartesianCPPoint(xCoord()+dx, yCoord()+dy,c)
```scala
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()
}

// Specific to color points

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 o:CPoint => normalize(o).isEqual(normalize(this))
        case o => false
    }
```
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
}

An alternative is to directly reuse code from CartesianPoint and PolarPoint in CartesianCPoint and PolarCPoint, using inheritance. This would be a case of noninnocuous inheritance, since the methods are developed to work in CartesianPoint and PolarPoint, not CartesianCPoint and PolarCPoint. So we will have to be careful, and make sure that the methods we inherit do what we expect them to do.
The first problem is that in Scala (like in Java), you can only set up inheritance if you also have subtyping. So we need to make sure that \texttt{CartesianCPoint} is a subtype of both \texttt{CartesianPoint} and \texttt{CPoint}, and similarly that \texttt{PolarCPoint} is a subtype of both \texttt{PolarPoint} and \texttt{CPoint}.

Since we can only subtype one actual class, the other supertype better be a trait. Since we have no choice in \texttt{CartesianPoint} being a class (since it is concrete), we have to make \texttt{CPoint} a trait. Not a problem, since traits and abstract classes are pretty much interchangeable.

Here is the resulting code that using inheritance instead of delegation:

```scala
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 CartesianPoint(xpos,ypos) with CPoint {

            def distance (q:CPoint):Double = super[CartesianPoint].distance(q)

            // these method cannot be inherited

            override 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())

            override 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())

```
// Specific to color points

def color ():Color = c

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

// BRIDGE METHODS

override 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 PolarPoint(r,theta) with CPoint {

    def distance (q:CPoint):Double = super[PolarPoint].distance(q)

    // these method cannot be inherited
override 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())

override 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()
}

// Specific to color points

def color():Color = c

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

// BRIDGE METHODS

override 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()
}

trait 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
}

The idea here is that inheritance acts as a form of implicit delegation. Rather than us defining a delegate and having methods in CartesianCPoint call the methods in the delegate, inheritance creates a delegate for us and the methods in this implicit delegate are
automatically made available to us, unless we *override* them.

Note the declaration of the `CartesianCPPoint`:

```java
private class CartesianCPPoint (xpos:Double, ypos:Double, c:Color)
  extends CartesianPoint(xpos, ypos) with CPoint {

It declares not only that `CartesianCPPoint` is a subtype of `CartesianPoint` and of `CPoint`, but also that the implicit delegate for `CartesianPoint` is created with arguments `xpos` and `ypos` — contrast with the explicit definition of the delegate in the original code.

With this definition, the methods that used to be direct delegations are now omitted, since they are directly inherited from the delegate, and therefore directly available. All that remains to deal with are the ones that were not directly delegated. This includes `move()`, `add()` (with a `CPoint` argument), `rotate()`, `isEqual()` (both with a `CPoint` argument and with a `Point` argument), `color()`, `updateColor()`, `toString()`, `equals()`, and `hashCode()`. For some of these, we need to indicate that we are overriding (since there is a similarly-defined method in `CartesianPoint` that does not do the right thing for us), and for others we do not need to override because there is no similarly-defined method in `CartesianPoint`. Note in particular that one version of `isEqual()` is overriding the one existing in `CartesianPoint`, while the other one does not. Recall that we may have multiply-defined methods in the same class, which different as far as their arguments, and those are considered distinct methods.

The one method that is problematic is `distance()` taking a `CPoint` as an argument. In the explicit delegation code, this looks like a direct delegation to the `distance()` method in `CartesianPoint`, but in reality there is an upcast that is inserted by the compiler, and that upcast is a problem here. Roughly, the system does not automatically insert upcasts to resolve inheritance. In other words, if we omit the definition of `distance()` taking a `CPoint` as an argument, the system complains that we have not defined the method (which is required in order for `CartesianCPPoint` to be a subtype of `CPoint`), and so we need to give it a definition. But really, we want the definition of `distance()` to simply call the inherited version of `distance()`, giving the compiler an opportunity to insert the appropriate upcast to make the call go through.

The fix is to use the special keyword `super`, which refers to the implicit delegate. Because (as we will see), there may be more than one implicit delegate, we also explicitly state which implicit delegate we want to invoke, using `super[CartesianPoint]` to say that it is the implicit delegate from `CartesianPoint` that we want to use. Thus, we get the definition:

```java
def distance (q:CPPoint):Double = super[CartesianPoint].distance(q)
```

### 17.2 A More Complex Example of Inheritance

Once we see the “trick” of being able to refer explicitly to an inherited method, as in `distance()` above, then we explore how to get even more reuse out of our methods.
To see how we can do that, let’s go back to our explicit delegation code. The methods that we could not easily delegate all essentially had the property that they needed to reconstruct a new `CPoint`, something that the delegate could not do (since a delegate could only reconstruct a `Point`). But if we took the result of the delegate, and converted that `Point` into a `CPoint`, then we’d be in business. What we need are two helper functions to do the conversion for us:

```scala
def reconstructCart (p:Point,c:Color):CPoint =
  new CartesianCPoint(p.xCoord(), p.yCoord(), c)

def reconstructPolar (p:Point,c:Color):CPoint =
  new PolarCPoint(p.distanceFromOrigin(), p.angleWithXAxis(), c)
```

Once we have such helper functions, we can reuse more code by delegating the core functionality of some of those methods we could not delegate easily. Here is the resulting code:

```scala
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 def reconstructCart (p:Point,c:Color):CPoint =
  new CartesianCPoint(p.xCoord(),p.yCoord(),c)

private def reconstructPolar (p:Point,c:Color):CPoint =
  new PolarCPoint(p.distanceFromOrigin(),
  p.angleWithXAxis(),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 =
  reconstructCart(del.move(dx,dy),c)

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

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

def isEqual (q:CPoint):Boolean =
  del.isEqual(q) && (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 =
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 PolarCPPoint (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 =
      reconstructCart(del.move(dx,dy),c)

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

    def rotate (angle:Double):CPoint =
      reconstructPolar(del.rotate(angle),c)
def isEqual (q:CPoint):Boolean = {
    del.isEqual(q) \&\& 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 that we have put the helper functions in the module, marking them private so that they are not available from outside the module. However, they are available from all the code within the module, including the two representation classes defined there.

Note the delegation and reconstruction that occurs in methods move(), add(), and rotate(). Also, note that we can use some delegation in isEqual() as well. And this is the case for both CartesianCPoint and PolarCPoint.

We can now look at a similar version but using inheritance, as we have done before, and now using an explicit call to inherited methods to turn those methods that call the delegate and reconstruct the result into methods that explicitly call inherited methods and reconstruct the result.

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 def reconstructCart (p:Point,c:Color):CPoint =
    new CartesianCPoint(p.xCoord(),p.yCoord(),c)
private def reconstructPolar (p:Point, c:Color):Point =
    new PolarPoint(p.distanceFromOrigin(),
                   p.angleWithXAxis(), c)

private class CartesianPoint (xpos:Double, ypos:Double, c:Color)
    extends CartesianPoint with CartesianPoint {

    def distance (q:Point):Double =
        super[CartesianPoint].distance(q)

    override def move (dx:Double, dy:Double):Point =
        reconstructCarte(sup[CartesianPoint].move(dx, dy), c)

    def add (q:Point):Point =
        reconstructCarte(sup[CartesianPoint].add(q), q.color())

    override def rotate (t:Double):Point =
        reconstructCarte(sup[CartesianPoint].rotate(t), c)

    def isEqual (q:Point):Boolean =
        super[CartesianPoint].isEqual(q) && (c==q.color())

    def color ():Color = c

    def updateColor (nc:Color):Point =
        new CartesianPoint(xpos, ypos, nc)

    // BRIDGE METHODS

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

    // CANONICAL METHODS

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

    override def equals (other : Any):Boolean =
        other match {

223
case that : CPPoint => this.isEqual(that)
case _ => false
}

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

private class PolarCPPoint (r:Double, theta:Double, c:Color)
  extends PolarPoint(r,theta) with CPPoint {

  def distance (q:CPPoint):Double = 
    super[PolarPoint].distance(q)

  override def move (dx:Double,dy:Double):CPPoint = 
    reconstructCart(super[PolarPoint].move(dx,dy),c)

  def add (q:CPPoint):CPPoint = 
    reconstructCart(super[PolarPoint].add(q),q.color())

  override def rotate (angle:Double):CPPoint = 
    reconstructPolar(super[PolarPoint].rotate(angle),c)

  def isEqual (q:CPPoint):Boolean = {
    super[PolarPoint].isEqual(q) & c==q.color()
  }

  def color ():Color = c

  def updateColor (nc:Color):CPPoint = 
    new PolarCPPoint(r,theta,nc)

  // BRIDGE METHODS

  override def isEqual (q:Point):Boolean = q match {
    case cq:CPPoint => 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()
def isEqual (q:Point):Boolean
}

An interesting question for you to think about, suggested by a colleague: if you could change the code for CartesianPoint and PolarPoint to maximize code reuse by inheritance, could you do so in such a way that move(), add(), and rotate(), in particular, could be directly inherited in CartesianCPoint and PolarCPoint? The answer is yes, if you’re curious, and it doesn’t require anything that we haven’t seen already.