13 Subtyping Multiple Types

The goal in this lecture is to look at creating types that are subtypes of multiple types at the same time, and why that might be useful. Intuitively, this will enable even more code reuse.

Let's start with the implementation of the POINT and CPOINT ADT from last lecture. I won't give you the whole code, just remind you of the abstract classes `Point` and `CPoint`:

```
abstract class Point {

  def xCoord ():Double
  def yCoord ():Double
  def move (dx:Double,dy:Double):Point
  def rotate(t:Double):Point
  def add (p:Point):Point
  def isEqual (p:Point):Boolean
}


abstract class CPoint extends Point {

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

  def xCoord ():Double
  def yCoord ():Double
  def move (dx:Double,dy:Double):CPoint
  def rotate (t:Double):CPoint
  def add (cp:CPoint):CPoint
  def isEqual (cp:CPoint):Boolean

  // bridge methods
  def add (p:Point):Point
  def isEqual (p:Point):Boolean
}
```

We have implementations of these.

Suppose we do something similar for rectangles, defining both rectangles and colored rectangles. Here's the RECT ADT:

```
CREATORS
  create :   (Point, Point) -> Rect

OPERATIONS
  upperLeft :   () -> Point
  lowerRight :  () -> Point
  move :        (Double, Double) -> Rect
  within :      (Point) -> Rect
  isEqual :     (Rect) -> Boolean
```

with specification:

$$\texttt{create(}ul\texttt{,}lr\texttt{).upperLeft()} = ul$$

$$\texttt{create(}ul\texttt{,}lr\texttt{).lowerRight()} = lr$$

$$\texttt{create(}ul\texttt{,}lr\texttt{).move(}dx\texttt{,}dy\texttt{)} = \texttt{create(}ul\texttt{.move(}dx\texttt{,}dy\texttt{), }lr\texttt{.move(}dx\texttt{,}dy\texttt{))}$$

$$\texttt{create(}ul\texttt{,}lr\texttt{).within(}p\texttt{)}$$

$$= \begin{cases} \mathit{true} & \text{if } ul\texttt{.xCoord()} \leq p\texttt{.xCoord()} \leq lr\texttt{.xCoord()} \\ & \text{and } lr\texttt{.yCoord()} \leq p\texttt{.yCoord()} \leq ul\texttt{.yCoord()} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{create(}ul\texttt{,}lr\texttt{).isEqual(}r\texttt{)}$$

$$= \begin{cases} \mathit{true} & \text{if } ul\texttt{.isEqual(}r\texttt{.upperLeft())} = \mathit{true} \\ & \text{and } lr\texttt{.isEqual(}r\texttt{.lowerRight())} = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

It's a straightforward exercise to implement this ADT using the Specification Design Pattern:

```
object Rect {

  def create (p:Point, q:Point):Rect =
    if (p.xCoord() <= q.xCoord() &&
        p.yCoord() <= q.yCoord())
      new RectImpl(p,q)
    else
      throw new IllegalArgumentException("Rect.create()")

  private class RectImpl (ul:Point, lr:Point) extends Rect {
    def upperLeft ():Point = ul
```

```
    def lowerRight ():Point = lr

    def move (dx:Double,dy:Double):Rect =
      new RectImpl(ul.move(dx,dy), lr.move(dx,dy))

    def within (p:Point):Boolean = {
      ul.xCoord() <= p.xCoord() && p.xCoord() <= lr.xCoord() &&
      ul.yCoord() <= p.yCoord() && p.yCoord() <= lr.yCoord()
    }

    def isEqual (r:Rect):Boolean = {
      ul==r.upperLeft() && lr==r.lowerRight()
    }

    // CANONICAL

    override def toString ():String =
      "rect(" + ul + "," + lr + ")"

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

    override def hashCode ():Int =
      41 * (
        41 + ul.hashCode()
      ) + lr.hashCode()
  }
}


abstract class Rect {

  def upperLeft ():Point
  def lowerRight ():Point
  def move (dx:Double,dy:Double):Rect
  def within (p:Point):Boolean
  def isEqual (r:Rect):Boolean
}
```

What about colored rectangles? The CRect ADT is what you would expect:

```
CREATORS
  create :   (Point, Point, Color) -> CRect

OPERATIONS
  upperLeft :     () -> Point
  lowerRight :    () -> Point
  move :          (Double, Double) -> CRect
  within :        (Point) -> CRect
  isEqual :       (CRect) -> Boolean
  color :         () -> Color
  updateColor :   (Color) -> CRect
```

The specification I will leave as an exercise — it is a simple variation on the specification for ADT Rect.

Implementing the CRect ADT so that it is a subtype of Rect is straightforward using the Specification Design Pattern and a bridge method for `isEqual()`.

```
object CRect {

  def create (p:Point, q:Point, c:Color):CRect =
    if (p.xCoord() <= q.xCoord() &&
        p.yCoord() <= q.yCoord())
      new CRectImpl(p,q,c)
    else
      throw new IllegalArgumentException("CRect.create()")

  private class CRectImpl (ul:Point, lr:Point, col:Color) extends CRect {
    def color ():Color = col

    def updateColor (c:Color):CRect =
      new CRectImpl(ul,lr,c)

    def upperLeft ():Point = ul
    def lowerRight ():Point = lr

    def move (dx:Double,dy:Double):CRect =
      new CRectImpl(ul.move(dx,dy), lr.move(dx,dy), col)

    def within (p:Point):Boolean = {
      ul.xCoord() <= p.xCoord() && p.xCoord() <= lr.xCoord() &&
      ul.yCoord() <= p.yCoord() && p.yCoord() <= lr.yCoord()
```

```
    }

    def isEqual (cr:CRect):Boolean = {
      ul==cr.upperLeft() && lr==cr.lowerRight() && color==cr.color()
    }

    def isEqual (r:Rect):Boolean =
      r match {
        case cr:CRect => this.isEqual(cr)
        case _ => false
      }

    // CANONICAL

    override def toString ():String =
      "crect(" + ul + "," + lr + "," + color + ")"

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

    override def hashCode ():Int =
      41 * (
        41 * (
          41 + ul.hashCode()
        ) + lr.hashCode()
      ) + col.hashCode()
  }
}


abstract class CRect extends Rect {

  def color ():Color
  def updateColor (c:Color):CRect

  def upperLeft ():Point
  def lowerRight ():Point
  def move (dx:Double,dy:Double):CRect
  def within (p:Point):Boolean
```

```
  def isEqual (r:CRect):Boolean

  // bridge method
  def isEqual (r:Rect):Boolean
}
```

So `CPoint` is a subtype of `Point`, and `CRect` is a subtype of `Rect`. We already know we can get some code reuse out of those relationships — any function that works on `Point`s will work on `CPoint`s, and any function that works on `Rect`s will work on `CRect`s.

Now, suppose we wanted to write a function that extracted the color out of a colored "shape" and complemented it. (ADT Color has an operation `complement()` that returns the complement of a color on the color wheel.) Right now, given our definition, we would have to write two functions:

```
def colorComplementPoint (c:CPoint):Color =
  c.color().complement()

def colorComplementRect (c:CRect):Color =
  c.color().complement()
```

The fact that both of those functions look exactly the same except for the type suggest that there might be a way to write a single function to work with both `CPoints` and `CRects`. Unfortunately, there is no type that is both a supertype of `CPoint` and `CRect` and that has a `color()` operation.

So how about we introduce one, call it `Colored`, and make sure that `CPoint` and `CRect` are both subtypes of `Colored`, on top of being subtypes of `Point` and `Rect`, respectively.

What we would like to define is something like an abstract class `Colored`:

```
abstract class Colored {
  def color ():Color
}
```

and when we define, say, `CPoint`, we would say:

```
abstract class CPoint extends Point,Colored {

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

  def xCoord ():Double
  def yCoord ():Double
```

```
  def move (dx:Double,dy:Double):CPoint
  def rotate (t:Double):CPoint
  def add (cp:CPoint):CPoint
  def isEqual (cp:CPoint):Boolean

  // bridge methods
  def add (p:Point):Point
  def isEqual (p:Point):Boolean
}
```

Unfortunately, this doesn't work. (It works in some languages, just not the ones we're using.) We can technically only *extend* one other class. If we want to be a subtype of other types, we have to make those types *traits*. Traits are reminiscent of Java interfaces, except that they let you do more. We'll see what that "more" denotes later. For the time being, think of traits as simply abstract classes. Traits are easy to define:

```
  trait Colored {
    def color ():Color
  }
```

To use the `Colored` trait, redefine both abstract classes `CPoint` and `CRect`:

```
abstract class CPoint extends Point with Colored {

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

  def xCoord ():Double
  def yCoord ():Double
  def move (dx:Double,dy:Double):CPoint
  def rotate (t:Double):CPoint
  def add (cp:CPoint):CPoint
  def isEqual (cp:CPoint):Boolean

  // bridge methods
  def add (p:Point):Point
  def isEqual (p:Point):Boolean
}


abstract class CRect extends Rect with Colored {

  def color ():Color
```

```
  def updateColor (c:Color):CRect

  def upperLeft ():Point
  def lowerRight ():Point
  def move (dx:Double,dy:Double):CRect
  def within (p:Point):Boolean
  def isEqual (r:CRect):Boolean

  // bridge methods
  def isEqual (r:Rect):Boolean
}
```

Now, I have both `CPoint` and `CRect` being subtypes of `Colored`, so I can replace my two functions above with a single function that can work with any value of type `Colored`:

```
  def colorComplement (c:Colored):Color =
    c.color().complement()
```

We can call `colorComplement()` with a `CPoint` or a `CRect`, because the type checker will insert an upcast automatically, since both `CPoint` and `CRect` are subtypes of `Colored`.

That takes care of `color()`. Now, what about `updateColor()`? Suppose we wanted to create a shape that looked just like some other shape but colored with the complement of that other shape? This is easy to do for `CPoint`:

```
  def makeComplementPoint (c:CPoint):CPoint =
    c.updateColor(c.color().complement())
```

And we can write a similar function for `CRect`:

```
  def makeComplementRect (c:CRect):CRect =
    c.updateColor(c.color().complement())
```

Again, the same code occurs in both function, so maybe we can write a single function instead. As before, we need to make sure we have a supertype for both `CPoint` and `CRect` with a suitable `updateColor()` method declared.

The easiest might just be to add `updateColor()` to trait`Colored`. But we hit a bit of a snag — `updateColor()` returns a result of the same type as the class in which it lives. So we cannot easily abstract it away in `Colored`.

The solution is to *parameterize* `Colored` by the result type of the `updateColor()` operation:

```
trait Colored[A] {
  def color ():Color
```

```
  def updateColor (c:Color):A
}
```

Think of the `A` in `trait Colored[A]` as a parameter like a parameter in a method. When we *use* `Colored`, we get to choose the exact type we want to instantiate the parameter `[A]` to. Such a parameterized trait is sometimes called a `generic trait`.

With this change, the abstract classes for `CPoint` and `CRect` look like:

```
abstract class CPoint extends Point with Colored[CPoint] {

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

  def xCoord ():Double
  def yCoord ():Double
  def move (dx:Double,dy:Double):CPoint
  def rotate(t:Double):CPoint
  def add (cp:CPoint):CPoint
  def isEqual (cp:CPoint):Boolean

  // bridge methods
  def add (p:Point):Point
  def isEqual (p:Point):Boolean
}


abstract class CRect extends Rect with Colored[CRect] {

  def color ():Color
  def updateColor (c:Color):CRect

  def upperLeft ():Point
  def lowerRight ():Point
  def move (dx:Double,dy:Double):CRect
  def within (p:Point):Boolean
  def isEqual (r:CRect):Boolean

  // bridge methods
  def isEqual (r:Rect):Boolean
}
```

Think about it, in `CPoint`, the `updateColor()` method should take a `Color` and return a `CPoint`, so we instantiate `Colored` to `Colored[CPoint]`, and similarly for `CRect`.

Now we can write a single function `makeComplement()` that creates a shape of the same kind as the argument, but with its color replaced by its complement:

```
def makeComplement[A] (c:Colored[A]):A =
  c.updateColor(c.color().complement())
```

Note that because we want this function to work on `Colored[A]` instances for any kind of `A`, we need to use a generic method.

Here's some sample code that illustrates this:

```
val q2 : CPoint = CPoint.cartesian(1,2,Color.red())
println("q2 = " + q2)
println("Complementing q2 = " + makeComplement[CPoint](q2))

val p3:Point = Point.cartesian(20,30)
val q3:Point = CPoint.cartesian(40,60,Color.red())
val r2:CRect = CRect.create(p3,q3,Color.blue())
println("r2 = " + r2)
println("Complementing r2 = " + makeComplement[CRect](r2))
```

which yiels the result:

```
q2 = cpoint(1.0,2.0,red)
Complementing q2 = cpoint(1.0,2.0,green)
r2 = crect(point(20.0,30.0),cpoint(40.0,60.0,red),blue)
Complementing r2 = crect(point(20.0,30.0),cpoint(40.0,60.0,red),orange)
```

Of course, now that we have changed the definition of `Colored` by giving it a parameter, our old `colorComplement()` method doesn't compile anymore — the system complains that `Colored` needs a parameter. Here is the updated version of `colorComplement()`:

```
def colorComplement[A] (c:Colored[A]):Color =
  c.color().complement()
```