20  More on Mutation

Deciding whether to use mutable structures is all about trade-off. On the plus side, mutable structures let you implement some things much more simply than a corresponding implementation with immutable structures — the moveable anchors from last time being a perfect example of something that is straightforward to implement using mutation but somewhat more painful to implement with immutable structures. On the minus side, mutable structures force you to think about sharing, and require you to keep in mind explicit diagrams like we saw last time, in order to predict what happens when you actually end up mutating a field of a mutable structure.

Let’s suppose that you have understood the trade-offs, and that you accept the added complexity of mutable structures. In other words, you decided to have some classes with mutable state, which generally means having mutable fields.

Even if you are okay with having mutable fields, I strongly suggest you make your fields private, and provide getters and setters for each field that can mutate – that is, methods to get the value of those fields, and methods to set the value of those fields. Why? Because this lets us enforce invariants. Suppose we only want to work with anchors in the positive quadrant, that is, points whose coordinates are nonnegative. Consider the following implementation of anchors (which for simplicity does not define creators or an abstract class):

```
class Anchor (p:Point) {

    // var indicates that the field can be reassigned a new value
    var pos = p

    def position ():Point = pos

    def move (dx:Double, dy:Double):Unit =
        pos = pos.move(dx,dy) // move point and reassign
        // pos to be that point
    override def toString ():String = "anchor(" + pos + ")"
}
```

Enforcing that only positive points can be created is easy by changing the constructor of the class, as well as the `move` method:

```
class Anchor (p:Point) {

    def Anchor (p:Point) {
        // only create anchors in the positive quadrant
        if (p.x < 0 || p.y < 0) throw new IllegalArgumentException("Anchor must be in the positive quadrant:
```
if (p.xCoord() < 0 || p.yCoord() < 0)
    throw new IllegalArgumentException("Point not positive")

// var indicates that the field can be reassigned a new value
var pos = p

def position ():Point = pos

def move (dx:Double, dy:Double):Unit = {
    newPos = pos.move(dx,dy)  // move point and reassign
    // pos to be that point
    if (newPos.xCoord() < 0 || newPos.yCoord() < 0)
        throw new RuntimeException("Point moved to negative position")
    pos = newPos
}

override def toString ():String = "anchor(" + pos + ")"

(Note that the code to check the parameters is free floating in the class — any code that does not belong to a method in a class is executed when an instance of a class is created. If we had used a creator instead, that check would probably have gone into the creator.)

If we allow unrestricted field access, however, then anyone can just change the position and break the invariant. Which, in this case, can lead to points having negative coordinates, invalidating the invariant we want to preserve.

If you really want to give access to the fields to users (as opposed to forcing them to use an operation like move), it is better to define setters, where we can check that the invariant is maintained whenever state is changed.

class Anchor (p:Point) {

    if (p.xCoord() < 0 || p.yCoord() < 0)
        throw new IllegalArgumentException("Point not positive")

    // var indicates that the field can be reassigned a new value
    var pos = p

    def position ():Point = pos

    def setPosition (newPos:Point):Unit =
        if (newPos.xCoord() < 0 || newPos.yCoord() < 0)
throw new RuntimeException("Point moved to negative position")

pos = newPos

def move (dx:Double, dy:Double):Unit =
setPosition(pos.move(dx,dy))

override def toString ():String = "anchor(" + pos + ")"

Therefore, I will expect you all to keep fields private and use explicit setters, instead of making fields public when you want a class to be mutable.

(Exercise: I did not bother making my field pos private in the implementation of Anchor in last lecture. Still, I claim one cannot change the value of the field pos there from outside anyways. Why?)

20.1 Shallow and Deep Copies

As we saw last time, if we pass an instance to a method, we are really passing the address of the instance to the method. And if the method just takes those values and store them somewhere, then we get sharing, which may or may not be what we want.

Recall the code we had last time, slightly modified to make Line an explicitly mutable structure. (We saw that Line is already mutable by virtue of Anchor being mutable, we just make Line explicitly mutable by making the start and end points mutable fields, and adding setters.)

object Anchor {

  def create (p:Point):Anchor = new AnchorRepr(p)

  private class AnchorRepr (p:Point) extends Anchor {

    private var pos = p

    def position ():Point = pos

    def move (dx:Double, dy:Double):Unit =
      pos = pos.move(dx,dy)  // move point and reassign
      // pos to be that point

    override def toString ():String = "anchor(" + pos + ")"

  }

}
abstract class Anchor {
    def position ():Point
    def move (dx:Double, dy:Double):Unit
}

object Line {
    def create (s:Anchor, e:Anchor):Line = new LineRepr(s,e)

    private class LineRepr (initS:Anchor, initE:Anchor) extends Line {
        private var s:Anchor = initS
        private var e:Anchor = initE

        def start ():Anchor = s
        def end ():Anchor = e

        def setStart (a:Anchor):Unit = { s = a }
        def setEnd (a:Anchor):Unit = { e = a }

        override def toString ():String = s + " <-> " + e
    }
}

abstract class Line {
    def start ():Anchor
    def end ():Anchor
    def setStart (a:Anchor):Unit
    def setEnd (a:Anchor):Unit
}

An example of sharing was the Line.create(a3,a3) example from last time:

val a3 = Point.create(Point.cartesian(0,10))
val 12 = Line.create(a3,a3)
The two fields s and e of the newly created Line instance end up pointing to the same instance of Anchor, so that modifying that Anchor instance is reflected in both the start and end position of the line.

To make our examples more interesting, consider an additional class to represent triangles using a base line and another anchor::

```java
object Triangle {
    def create (b:Line, a:Anchor):Triangle = new TriangleRepr(b,a)

    private class TriangleRepr (b:Line, a:Anchor) extends Triangle {

        private var base:Line = b
        private var tip:Anchor = a

        def side1 ():Line = base
        def side2 ():Line = Line.create(base.start(),tip)
        def side3 ():Line = Line.create(base.end(),tip)

        def setBase (l:Line):Unit = { base=l }
        def setTip (a:Anchor):Unit = { tip=a }

        override def toString ():String = side1() + " " + side2() + " " + side3()
    }
}
```
abstract class Triangle {
  def side1 ():Line
  def side2 ():Line
  def side3 ():Line
  def setBase (l:Line):Unit
  def setTip (a:Anchor):Unit
}

Now we can get even more interesting sharing going between lines and points.
Consider the following definitions:

scala> val a = Anchor.create(Point.cartesian(0,0))
a: Anchor = [0.0,0.0]

scala> val b = Anchor.create(Point.cartesian(100,0))
b: Anchor = [100.0,0.0]

scala> val c = Anchor.create(Point.cartesian(50,50))
c: Anchor = [50.0,50.0]

scala> val l = Line.create(a,b)
l: Line = [0.0,0.0] <-> [100.0,0.0]

scala> val t = Triangle.create(l,c)
t: Triangle = [0.0,0.0] <-> [100.0,0.0]  [0.0,0.0] <-> [50.0,50.0]  [100.0,0.0] <-> [50.0,50.0]

Suppose we wanted to create another triangle that looks just like t. An easy way to do that is to simply define:

scala> val u1 = t
u1: Triangle = [0.0,0.0] <-> [100.0,0.0]  [0.0,0.0] <-> [50.0,50.0]  [100.0,0.0] <-> [50.0,50.0]

But of course, these is maximal sharing between t and u1:
In particular, changing any part of one changes the other:

```scala
scala> t.setTip(Anchor.create(Point.cartesian(9,9)))
scala> u1
```
res2: Triangle = [0.0,0.0] <-> [100.0,0.0]  [0.0,0.0] <-> [9.0,9.0]  
[100.0,0.0] <-> [9.0,9.0]

What if we wanted the new triangle to look just like t but not share as much with t? Let’s create a new method in Triangle that creates a copy of the current triangle that does not share its fields.

```java
object Triangle {

    def create (b:Line, a:Anchor):Triangle = new TriangleRepr(b,a)

    private class TriangleRepr (b:Line, a:Anchor) extends Triangle {

        private var base:Line = b
        private var tip:Anchor = a

        def side1 ():Line = base

        def side2 ():Line = Line.create(base.start(),tip)

        def side3 ():Line = Line.create(base.end(),tip)

        def setBase (l:Line):Unit = { base=l }
        def setTip (a:Anchor):Unit = { tip=a }

        def copy ():Triangle = new TriangleRepr(base,tip)

        override def toString ():String =
            side1() + "  " + side2() + "  " + side3()
    }

    abstract class Triangle {

        def side1 ():Line
        def side2 ():Line
        def side3 ():Line
        def setBase (l:Line):Unit
        def setTip (a:Anchor):Unit
        def copy ():Triangle
    }
```

261
Now, if we redo the above example with a, b, c, l, t, and u1 as above and say:

```scala
cscala> val u2 = t.copy()
u2: Triangle = [0.0,0.0] <> [100.0,0.0] [0.0,0.0] <> [9.0,9.0] [100.0,0.0] <> [9.0,9.0]...
```

we end up with less sharing between t and u2:

```
+-------------------+      +-------------------+
 b *----------> | Anchor | <>--  |      | Anchor | <>--  |
 |-----------------|      |-----------------|
 | pos = *-----------> | Point |
 +-------------------+      +-------------------+      | xpos = 100 |      | xpos = 0 |
 +-------------------+      +-------------------+      | ypos = 0 |      | ypos = 0 |
 a *----------> | Anchor | <>-- |      | Anchor | <>-- |      | Anchor | <>-- |
 |-----------------|      |-----------------|
 | pos = *-----------> | Point |
 +-------------------+      +-------------------+      | xpos = 0 |      | xpos = 0 |
 +-------------------+      +-------------------+      | ypos = 0 |      | ypos = 0 |
 l *----------> | Line | <>-- |      | Line | <>-- |      | Line | <>-- |
 |-----------------|      |-----------------|
 | s = *----------+ |      | s = *----------+ |      | s = *----------+ |
 | e = *----------+ |      | e = *----------+ |      | e = *----------+ |
 +-------------------+      +-------------------+      | xpos = 50 |      | xpos = 50 |
 +-------------------+      +-------------------+      | ypos = 50 |      | ypos = 50 |
 c *----------> | Anchor | <>--+ |      | Anchor | <>--+ |      | Anchor | <>--+ |
 |-----------------|      |-----------------|
 | pos = *-----------> | Point |
 +-------------------+      +-------------------+      | xpos = 50 |      | xpos = 50 |
 +-------------------+      +-------------------+      | ypos = 50 |      | ypos = 50 |
```
But there is still some sharing. In fact, if we change either of 1, a, b, or c, we see that change both in t and in u2:

```scala
scala> t.side1().end().move(99,99)
```

```scala
scala> u2
res2: Triangle = [0.0,0.0] <-> [199.0,99.0] [0.0,0.0] <-> [9.0,9.0] [199.0,99.0] <-> [9.0,9.0]
```

So copy() may not be quite what we want. It does create a fresh copy of t, but because base and tip are simply given the address of the instance pointed to by base and by tip in t, the value of the fields end up the same in both t and u2.

So while u2 is a copy of t, they are not fully disjoint. Rather, u2 is what we call a shallow copy of t. The “top level” of the instances are disjoint (in the sense that their fields live in different places), but any sharing within the values held in the fields is preserved.

If we wanted a truly disjoint new triangle, then we need to make what is called a deep copy, that is, a copy that recursively deep copies (creating new instances) for every instance held in every variable, all the way down. Thus:

```scala
object Triangle {

  def create (b:Line, a:Anchor):Triangle = new TriangleRepr(b,a)

  private class TriangleRepr (b:Line, a:Anchor) extends Triangle {

    private var base:Line = b
    private var tip:Anchor = a

    def side1 ():Line = base

    def side2 ():Line = Line.create(base.start(),tip)

    def side3 ():Line = Line.create(base.end(),tip)

    def setBase (l:Line):Unit = { base=l }

  }

}
```

263
def setTip (a:Anchor):Unit = { tip=a }

def copy ():Triangle = new TriangleRepr(base,tip)

def deepCopy ():Triangle =
    new TriangleRepr(base.deepCopy(),tip.deepCopy())

    override def toString ():String =
        side1() + " " + side2() + " " + side3()
  }
}

abstract class Triangle {

def side1 ():Line
def side2 ():Line
def side3 ():Line
def setBase (l:Line):Unit
def setTip (a:Anchor):Unit
def copy ():Triangle
def deepCopy ():Triangle
}

So, you see, to create a deep copy of a triangle, we recursively deep copy all the values of all the relevant fields, and create a new triangle with those new values. This means that we need a deepCopy() method in Anchor and in Line — let’s do that, and add some shallow copy() methods as well, for completeness:

object Anchor {

    def create (p:Point):Anchor = new AnchorRepr(p)

private class AnchorRepr (p:Point) extends Anchor {

    private var pos = p

    def position ():Point = pos

    def move (dx:Double, dy:Double):Unit =
        pos = pos.move(dx,dy) // move point and reassign
        // pos to be that point

264
def copy ():Anchor =
    new AnchorRepr(p)

def deepCopy ():Anchor =
    new AnchorRepr(p)

    override def toString ():String =
        "[" + pos.xCoord() + "," + pos.yCoord() + "]"
    }
}

abstract class Anchor {

    def position ():Point
    def move (dx:Double, dy:Double):Unit
    def copy ():Anchor
    def deepCopy ():Anchor
}

object Line {

    def create (s:Anchor, e:Anchor):Line = new LineRepr(s,e)

    private class LineRepr (initS:Anchor, initE:Anchor) extends Line {

        private var s:Anchor = initS
        private var e:Anchor = initE

        def start ():Anchor = s
        def end ():Anchor = e

        def setStart (a:Anchor):Unit = { s = a }
        def setEnd (a:Anchor):Unit = { e = a }

        def copy ():Line =
            new LineRepr(s,e)

        def deepCopy ():Line =
            new LineRepr(s.deepCopy(),e.deepCopy())

    }

    override def toString ():String =
        "[" + pos.xCoord() + "," + pos.yCoord() + "]"
    }
}
override def toString (): String = s + " <-> " + e
}
}

abstract class Line {
  def start (): Anchor
  def end (): Anchor
  def setStart (a: Anchor): Unit
  def setEnd (a: Anchor): Unit
  def copy (): Line
  def deepCopy (): Line
}

Note that copy() and deepCopy() for Anchor are the same. That’s because Points are immutable — copying a Point is the same as creating a new Point. So for some classes, a deep copy is going to look the same as a shallow copy. For the sake of clarity, let’s keep the two methods distinct, just to make clear that they have different roles, even though they do the same thing.

Now, if we continue with our example: redefine a,b,c,l,t as before and say:

scala> t
res2: Triangle = [0.0,0.0] <-> [199.0,99.0] [0.0,0.0] <-> [9.0,9.0]
       [199.0,99.0] <-> [9.0,9.0]

scala> val u3 = t.deepCopy()
u3: Triangle = [0.0,0.0] <-> [199.0,99.0] [0.0,0.0] <-> [9.0,9.0]
       [199.0,99.0] <-> [9.0,9.0]

You get that t and u3 are truly disjoint: each of their start and end fields point to different instances. Thus:

scala> t
res4: Triangle = [0.0,0.0] <-> [1199.0,1099.0] [0.0,0.0] <-> [9.0,9.0]
       [1199.0,1099.0] <-> [9.0,9.0]

scala> u2
res5: Triangle = [0.0,0.0] <-> [1199.0,1099.0] [0.0,0.0] <-> [9.0,9.0]
       [1199.0,1099.0] <-> [9.0,9.0]

scala> u3
res6: Triangle = [0.0,0.0] <-> [199.0,99.0]  [0.0,0.0] <-> [9.0,9.0]  
[199.0,99.0] <-> [9.0,9.0]

I will let you draw the resulting diagram.