Object-Oriented Design	**Lecture 19**
CS 3500 Spring 2011 (Pucella)	**Friday, Mar 25, 2011**

19 Mutation

We have not talked about mutation until now. But mutation exists in most languages, including Java and Scala. Many libraries rely on it. So we need to know how to deal with it.

Recall that a class is *immutable* if an instance of the class never changes after it has been created (that is, observations made on the class, such as results of operations, are always the same). In contrast, a class is *mutable* if its instances may change during their lifetime.

So let's try to get an understanding of mutation. Mutation can be problematic because an instance can change under you without you noticing, leading to hard to find bugs. This is the root of my advocacy for immutability — it is just plain easier to reason about immutable classes. Immutable classes are also easier to parallelize, as well as easier to debug. The flip side is that there are some algorithms that are much easier to implement using mutation than without.

Consider a scenario that is common in diagram-drawing programs. You can create a drawing with lines, where each line as a starting point and an end point, and some of those starting points may be shared. Often you are also able to click on those so-called *anchor points* and drag them, and all the lines connected to that anchor point move with them.

Now, doing this with our immutable implementation of points, where a line has a starting point and an end point, is difficult. Not impossible, but difficult. The problem is that because points are immutable, when you move a point, you actually create a new instance of a point at the new (moved) location. But the lines are still connected to the original point, which hasn't moved (it is immutable!) So there needs to be a way for that newly created point to somehow tell all the lines that are connected to the original point to now instead connect to it. And because lines are immutable, this requires creating a new set of lines that are now connected to the new point. It is possible to set up something like this (and in fact, it's a good exercise to do so...) but an easier way is simply to avoid creating a new point when we move the point, and instead just mutate it. Then every line connected to the point is automatically aware of the change, and there is no need to reconstruct new lines.

Rather than making points mutable, I will instead define a new ADT for anchor points, `Anchor`, with the following interface:

```
CREATORS     create :   Point -> Anchor

OPERATIONS   position : () -> Point
             move :     (Double,Double) -> Unit
```

Note that `move()` returns `Unit`, which is an indication that it actually returns no useful value. The only reason why we call `move()` is for its side-effect, that is, changing the current instance. I will skip the specification, because it turns out that giving algebraic specifications for mutable ADTs is nearly impossible. Try it, and you'll see where things break down.

Here is an implementation:

```
object Anchor {

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

  private class AnchorRepr (p:Point) extends Anchor {

    // 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 + ")"
  }
}


abstract class Anchor {

  def position ():Point
  def move (dx:Double, dy:Double):Unit
}
```

A class can be made mutable (to a first approximation) by first making some of its fields mutable – defining them with `var` instead of `val`. (In Java, all fields are always mutable) and then making sure that those fields can be updated. This can be achieved in two ways: either by making those fields public (so that instances of the class can be modified by clients), or by having methods change the value of fields.

Having the ability to mutate fields means that an instance may yield different observations at different point in times. For instance:

```
scala> val a = Anchor.create(Point.cartesian(1,2))
a: Anchor = anchor(cartesian(1.0,2.0))
```

```
scala> a.position().xCoord()
res0: Double = 1.0

scala> a.move(50,50)

scala> a
res3: Anchor = anchor(cartesian(51.0,52.0))

scala> a.position().xCoord()
res2: Double = 51.0
```

Here, the call to `toString()` returns different results, even though it is invoked on the same value `p`. One difficulty with mutation is that an update may be hidden away in some random method that provides no clue that it is mutating something.

```
scala> a
res4: Anchor = anchor(cartesian(51.0,52.0))

scala> def someFunction (anchor:Anchor):Unit = { anchor.move(100,100); }
someFunction: (anchor: Anchor)Unit

scala> someFunction(a)

scala> someFunction(a)

scala> a
res7: Anchor = anchor(cartesian(251.0,252.0))
```

Again, this mutates anchor `a`, but there is no indication that the call to `someFunction()` does a mutation. If this mutation is unintended, that can lead to bugs that are difficult to track down.

Let's consider the Line ADT where the end points of the line are anchors:

```
  CREATORS     create :   (Anchor, Anchor) -> Line

  OPERATIONS   start :    () -> Anchor
               end :      () -> Anchor
```

with specification:

$$\texttt{create}(a_1,a_2)\texttt{.start()} = a_1$$

$$\texttt{create}(a_1,a_2)\texttt{.end()} = a_2$$

and implementation:

```
object Line {

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

  private class LineRepr (s:Anchor, e:Anchor) extends Line {

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

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


abstract class Line {

  def start ():Anchor
  def end ():Anchor
}
```

This implementation of the Line ADT suggests another difficulty with mutability. Mutability is a *contagious property*: a class that looks immutable may in fact be mutable if it relies on classes that are themselves mutable. `Line` looks like it is an immutable implementation of the Line ADT: it has no variable fields (aside from the implicit fields holding the values passed as parameters) and it cannot change the value of those fields. Unfortunately, the following snippet of code shows that an instance of `Line` can indeed change:

```
scala> val a1 = Anchor.create(Point.cartesian(0,0))
a1: Anchor = anchor(cartesian(0.0,0.0))

scala> val a2 = Anchor.create(Point.cartesian(50,50))
a2: Anchor = anchor(cartesian(50.0,50.0))

scala> val l = Line.create(a1,a2)
l: Line = anchor(cartesian(0.0,0.0)) <-> anchor(cartesian(50.0,50.0))

scala> l.start().position().xCoord()
res8: Double = 0.0
```

```
scala> a1.move(100,100)

scala> l
res10: Line = anchor(cartesian(100.0,100.0)) <-> anchor(cartesian(50.0,50.0))

scala> l.start().position().xCoord()
res11: Double = 100.0
```

So `l` was originally a line between an anchor at $(0,0)$ and an anchor at $(50,50)$, but after mutating `a`, `l` is now a line between an anchor at $(100,100)$ and an anchor at $(50,50)$. So `l` has changed. That's something to keep in mind: a class may be mutable even though it looks like it is not. As soon as part of your program is mutable, it will have a tendency to make the rest of your program mutable as well.

Part of the point of this lecture is to provide a model of mutation so that you can understand exactly what is happening in the examples above.

To work with mutation correctly, and help you track down bugs, you need to have a good understanding of what actually gets update when you make an update! You update some field in some instance of a class, what actually gets updated, and who else can see it? The problem is that when the state of an instance can change, it becomes very important to understand when instances are *shared* between different instances, so that we can track when a change can be seen from another instance.

To pick a silly example, if we write:

```
scala> val a1 = Anchor.create(Point.cartesian(0,0))
a1: Anchor = anchor(cartesian(0.0,0.0))

scala> val a2 = Anchor.create(Point.cartesian(0,0))
a2: Anchor = anchor(cartesian(0.0,0.0))
```

Then computing with `a1` and `a2` both give the same result, and if we mutate `a1` by calling `move()`, `a2` is unaffected:

```
scala> a1.move(20,20)

scala> a1
res13: Anchor = anchor(cartesian(20.0,20.0))

scala> a2
res14: Anchor = anchor(cartesian(0.0,0.0))
```

Contrast that to:

```
scala> val a3 = Anchor.create(Point.cartesian(0,0))
a3: Anchor = anchor(cartesian(0.0,0.0))

scala> val a4 = a3
a4: Anchor = anchor(cartesian(0.0,0.0))
```

where again computing with `a3` and `a4` both give the same result, but if we mutate `a3` anywhere, then `a4` is changed as well:

```
scala> a3.move(20,20)

scala> a3
res16: Anchor = anchor(cartesian(20.0,20.0))

scala> a4
res17: Anchor = anchor(cartesian(20.0,20.0))
```

That's because `a3` and `a4` hold the same instance — they share that instance. In contrast, `a1` and `a2` both hold different instances (even though those instances look the same). Tracking this kind of sharing is what makes working with mutation error prone.

My claim is that to understand mutation, you need to have a working model of how a language represents instances internally. It does not need to be an accurate model; it just needs to have good predictive power. Let me describe a basic model that answers the question: where do variables and fields live? (The question 'where do methods live?' is less interesting because methods are not mutable.)

Recall that when you create an instance with a `new` statement, a block of memory is allocated in memory (in the heap) representing the new instance. Here is how we represent an instance in the heap:

```
  +------------+
  | class name |
  |------------|
  |            |
  | fields     |
  |            |
  +------------+
```

This representation does not have include the methods, because that's not what I want to focus on right now. You can think of the value returned by a `new` as the address in memory where the instance lives. (This is sometimes called an instance reference, or a pointer.) Thus, for instance, when you write

```
val a = Anchor.create(Point.cartesian(0,10))
```

a new instance of `Point`[19] is created in memory, and its instance reference is passed to the creator `Anchor.create()`, which creates an instance of `Anchor` and stores the passed argument (the instance reference to the `Point` instance) in the `pos` field of the `Anchor` instance. What is returned (and stored in `a`) is the instance reference for the `Anchor` instance. We can represent this as follows:

```
                   +-------------+          +------------+
a1 *----------> | Anchor      |     +---> | Point      |
                   |-------------|     |     |------------|
                   | pos = *-----------+     | xpos = 0   |
                   +-------------+           | ypos = 10  |
                                             +------------+
```

When you pass an instance as an argument to a method, what you end up passing is the instance reference of that instance (that is, the value returned by the `new` that create the instance in the first place). That is how instances get manipulated.

Consider lines. When you create a `Line` instance, passing in two anchors (that is, two instance references to anchors), you store those instance reference in the fields `s` and `e` in the created line. Thus:

```
val a1 = Anchor.create(Point.cartesian(0,10))
val a2 = Anchor.create(Point.cartesian(0,20))
val l1 = Line.create(a2,a1)
```

[19]Well, in reality, if you look at the code for `Point.cartesian()`, you see that a new instance of `Point.PointCartesian` is in fact created, but for the sake of discussion here we can just simply things and say that an instance of `Point` is created. We'll do something similar for `Anchor`, where `Anchor.create()` actually creates an instance of `Anchor.AnchorRepr`, but we will just call it an instance of `Anchor`.

```
                  +-------------+
  a1 *----------> | Anchor      | <-----+
                  |-------------|       |  +------------+
                  | pos = *--------------> | Point      |
                  +-------------+       |  |------------|
                                        |  | xpos = 0   |
                                        |  | ypos = 10  |
                                        |  +------------+
                  +-------------+       |
  a2 *----------> | Anchor      | <--+  |
                  |-------------|    |  |  +------------+
                  | pos = *--------------> | Point      |
                  +-------------+    |  |  |------------|
                                     |  |  | xpos = 0   |
                                     |  |  | ypos = 20  |
                                     |  |  +------------+
                                     |  |
                  +-------------+    |  |
  l1 *----------> | Line        |    |  |
                  |-------------|    |  |
                  | s = *------------+  |
                  | e = *---------------+
                  +-------------+
```

To find the value of a field, you follow the arrows to the instance that holds the field you are trying to access. Thus, `a1.position().xCoord()` looks up the `pos` field in the instance pointed to by `a1`, and then the `xpos` field in the instance pointed to by that `pos` field. Similarly, `l1.end().position().yCoord()` accesses the `ypos` field of the instance pointed to by `l1.end().position()`, which is the content of the `pos` field in the instance pointed to by `l1.end()`, which is the instance pointed to by the content of the `e` field of `l1`.

In particular, you see why if after creating the above we write

```
scala> a1.move(0,5)

scala> l1
res19: Line = anchor(cartesian(0.0,20.0)) <-> anchor(cartesian(0.0,15.0))
```

we get that `l1` is now a line between an anchor at $(0, 20)$ and an anchor at $(0, 15)$; intuitively, that's because `a1` is the same instance as the anchor stored as the end point in `l1`. Here is the diagram of the result of the call to `a1.move(0,5)`. Note that `move()` in `Anchor` replaces the content of the `pos` field with a new point.

```
                                           +------------+
                                    +----> | Point      |
                                    |      |------------|
                                    |      | xpos = 0   |
                                    |      | ypos = 15  |
                 +-------------+    |      +------------+
a1 *---------->  | Anchor      | <--|--+
                 |-------------|    |  |   +------------+
                 | pos = *----------+  |   | Point      |
                 +-------------+       |   |------------|
                                       |   | xpos = 0   |
                                       |   | ypos = 10  |
                                       |   +------------+
                 +-------------+       |
a2 *---------->  | Anchor      | <--+  |
                 |-------------|    |  |   +------------+
                 | pos = *---------------> | Point      |
                 +-------------+    |  |   |------------|
                                    |  |   | xpos = 0   |
                                    |  |   | ypos = 20  |
                                    |  |   +------------+
                                    |  |
                 +-------------+    |  |
l1 *---------->  | Line        |    |  |
                 |-------------|    |  |
                 | s = *------------+  |
                 | e = *---------------+
                 +-------------+
```

We call this phenomenon *sharing*. It is reflected by the fact that there are two arrows pointing to the same instance in the above diagram.

Compare the above by what gets constructed if we write:

```
val a3 = Point.create(Point.cartesian(0,10))
val l2 = Line.create(a3,a3)
```

```
                +-------------+ <-----+
a3 *---------->  | Anchor      | <--+  |
                |-------------|    |  |  +------------+
                | pos = *---------------> | Point      |
                +-------------+    |  |  |------------|
                                   |  |  | xpos = 0   |
                                   |  |  | ypos = 10  |
                +-------------+    |  |  +------------+
l2 *---------->  | Line        |    |  |
                |-------------|    |  |
                | s = *------------+  |
                | e = *---------------+
                +-------------+
```

Here, the *same* anchor is used as start and end points of `l2`. Meaning, in particular, that if we update field `pos` of `a3`, both the start and end points of `l2` change.