16 Mutation

We have been avoiding mutations until now. But they exist in most languages, including Java and Scala. Some of the libraries rely on them. So we need to know how to deal with them.

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.

For the purpose of this lecture, we shall consider the following two classes. First, a class `Point` representing two-dimensional points:

```
class Point (x:Int, y:Int)  {

  def xCoord ():Int = x
  def yCoord ():Int = y

  override def toString ():String = "(" + xCoord() + "," + yCoord() + ")"
}
```

Second, a class `Line` representing lines:

```
class Line (s:Point,e:Point) {

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

  override def toString ():String = "[" + start() + " <-> " + end() + "]"
}
```

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.

For the time being, let's make `Point` mutable by creating mutable fields to holds the coordinates of the point, and illustrate why mutation is hard to reason about:

```
class Point (x:Int, y:Int)  {

  var xpos:Int = x
  var ypos:Int = y

  def xCoord ():Int = xpos
  def yCoord ():Int = ypos

  override def toString ():String = "(" + xCoord() + "," + yCoord() + ")"
}
```

Having the ability to mutate fields means that an instance may yield different observations at different point in times. For instance: (I will be using the scala interactive interpreter for these examples)

```
scala> val p = new Point(1,2)
p: Point = (1,2)

scala> p.toString()
res0: String = (1,2)

scala> p.xpos = 99

scala> p.toString()
res2: String = (99,2)
```

Here, the call to `toString()` returns different results, even though it is invoked on the same value `p`. One difficulty with mutation is an update may be hidden in some other method, which provides no indication that it is changing the instance. Suppose that we have a function

```
scala> def someFunction (q:Point):Unit = { q.xpos = 999 }
someFunction: (q: Point)Unit

scala> val p = new Point(1,2)
p: Point = (1,2)
```

```
scala> p.toString()
res2: String = (1,2)

scala> someFunction(p)

scala> p.toString()
res4: String = (999,2)
```

Again, this mutates point `p`, but there is no indication that the call to `someFunction()` does a mutation.

Another difficulty with mutability is that it is a *contagious property*: a class that looks immutable may in fact be mutable if it relies on classes that are themselves mutable. Consider the mutable implementation of `Point`s above. What about `Line` though? It has no fields (aside from the implicit fields holding the values passed as parameters) and it cannot change the value of those fields, so as far as one can tell by looking at the class definition, it is an immutable class. Unfortunately, the following snippet of code shows that an instance of `Line` can indeed change:

```
scala> val p = new Point(1,2)
p: Point = (1,2)

scala> val q = new Point(100,200)
q: Point = (100,200)

scala> val l = new Line(p,q)
l: Line = [(1,2) <-> (100,200)]

scala> p.xpos = 99

scala> l
res6: Line = [(99,2) <-> (100,200)]
```

So `l` was originally `[(1,2) <-> (100,200)]`, but after mutating `p`, `l` is now `[(99,2) <-> (100,200)]` — `l` looks different. 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.

16.1 Detour: Setters for Fields

So let's suppose that you have understood this issue about mutability, and that you accept the ensuing risks. 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 points in the positive quadrant, that is, points whose coordinates are nonnegative. That's easy to enforce with the above `Point` class by adding some code that checks that the coordinates are positive when an instance of the class is created:

```
class Point (x:Int, y:Int)  {

  // executed whenever you create a Point instance
  if (x < 0 || y < 0)
    throw new IllegalArgumentException("Point not positive")

  var xpos:Int = x
  var ypos:Int = y

  def xCoord ():Int = xpos
  def yCoord ():Int = ypos

  override def toString ():String = "(" + xCoord() + "," + yCoord() + ")"
}
```

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

By forcing users to use setters, we can check that the invariant is maintained whenever state is changed.:

```
class Point (x:Int, y:Int)  {

  // executed whenever you create a Point instance
  if (x < 0 || y < 0)
    throw new IllegalArgumentException("Point not positive")

  private var xpos:Int = x
```

```
  private var ypos:Int = y

  def xCoord ():Int = xpos
  def yCoord ():Int = ypos

  def setXCoord (x:Int):Unit =
    if (x >= 0)
      xpos = x
    else
      throw new IllegalArgumentException("Point not positive")

  def setYCoord (y:Int):Unit =
    if (y >= 0)
      ypos = y
    else
      throw new IllegalArgumentException("Point not positive")

  override def toString ():String = "(" + xCoord() + "," + yCoord() + ")"
}
```

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. For completeness, even though we saw that `Line` is already mutable by virtue of `Point` being mutable, let's make `Line` directly mutable by making the start and end points mutable fields, and adding setters.

```
class Line (s:Point,e:Point) {

  private var sp : Point = s
  private var ep : Point = e

  def start ():Point = sp
  def end ():Point = ep

  def setStart (p:Point):Unit = (sp=p)
  def setEnd (p:Point):Unit = (ep=p)

  override def toString ():String = "[" + start() + " <-> " + end() + "]"
}
```

16.2 Instance Creation and Manipulation

To work with mutation correctly, and help you track down bugs, you need to have a good understanding of what changes when something changes! You update some field in some instance of a class, what actually gets changed, 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:

```
val p1 = new Point(0,0)
p1: Point = (0,0)

scala> val p2 = new Point(0,0)
p2: Point = (0,0)
```

Then computing with `p1` and `p2` both give the same result, and if we mutate `p1` by calling `setXCoord()`, `p2` is unaffected:

```
scala> p1.setXCoord(99)

scala> p1
res10: Point = (99,0)

scala> p2
res11: Point = (0,0)
```

Contrast that to:

```
scala> val p3 = new Point(0,0)
p3: Point = (0,0)

scala> val p4 = p3
p4: Point = (0,0)
```

where again computing with `p3` and `p4` both give the same result, but if we mutate `p3` anywhere, then `p4` is changed as well:

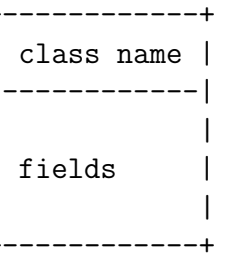
```
scala> p3.setXCoord(99)

scala> p3
```

```
res1: Point = (99,0)

scala> p4
res2: Point = (99,0)
```

That's because `p3` and `p4` point to the same instance — they share that instance. In contrast, `p1` and `p2` both point to 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 live? (The question 'where do methods live?' is less interesting because methods are not mutable.)

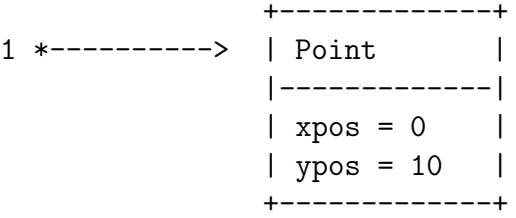
Recall that when you create an instance with a `new` statement, a block of memory is allocated in memory (in the heap) representing the new instance. Here is how we represent an instance in the heap:



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

```
val p1 = new Point(0,10)
```

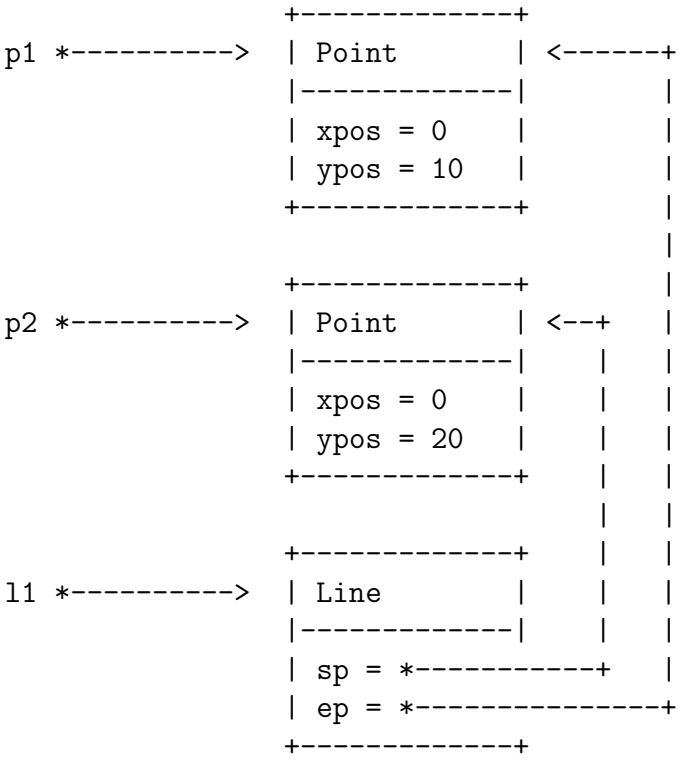
a new instance is created in memory, say living at some address *addr*, and variable `p1` holds value *addr*. We can represent this as follows:



Now, when you pass an instance as an argument to a method, what you end up passing is the *address* 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 points, you get as values of the fields `sp` and `ep` in the created line the addresses of the two points that were used to create the line in the first place.

```
val p1 = new Point(0,10)
val p2 = new Point(0,20)
val l1 = new Line(p2,p1)
```



To find the value of a field, you follow the arrows to the instance that holds the field you are trying to access. Thus, `p1.xCoord()` looks up the `xpos` field in the instance pointed to by `p1`. Similarly, `l1.end().yCoord()` accesses the `ypos` field of the instance returned by `l1.end()`, which itself can be found as field `ep` in the instance pointed to by `l1`.

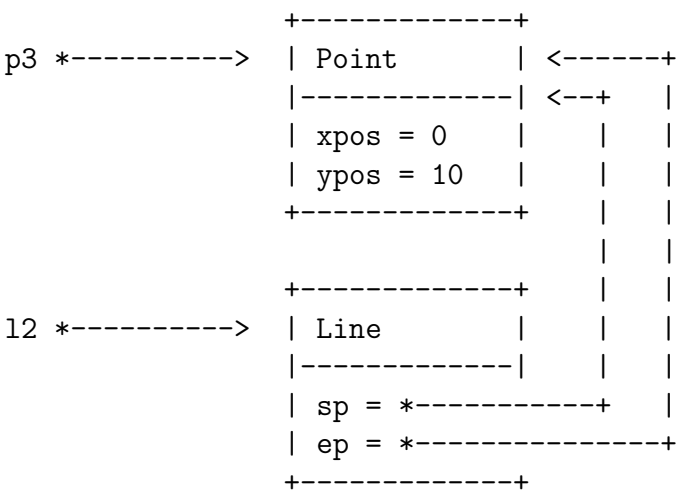
In particular, you see why if after creating the above we write

```
scala> p1.setYCoord(5)

scala> l1
res10: Line = [(0,20) <-> (0,5)]
```

we get that `l1` is now `[(0,20) <-> (0,5)]`; intuitively, because `p1` is the same instance as the point stored as the end point in `l1`. We call this phenomenon *sharing*. It is reflected by the fact that there are two arrows pointing to the same instance in a diagram.

Compare the above by what gets constructed if we write:

```
val p3 = new Point(0,10)
val l2 = new Line(p3,p3)
```



Here, the *same* point is used as start and end points of `l2`. Meaning, in particular, that if we update field `xpos` of `p3`, both the start and end points of `l2` look different.

16.3 Inheritance [Not Covered in Class]

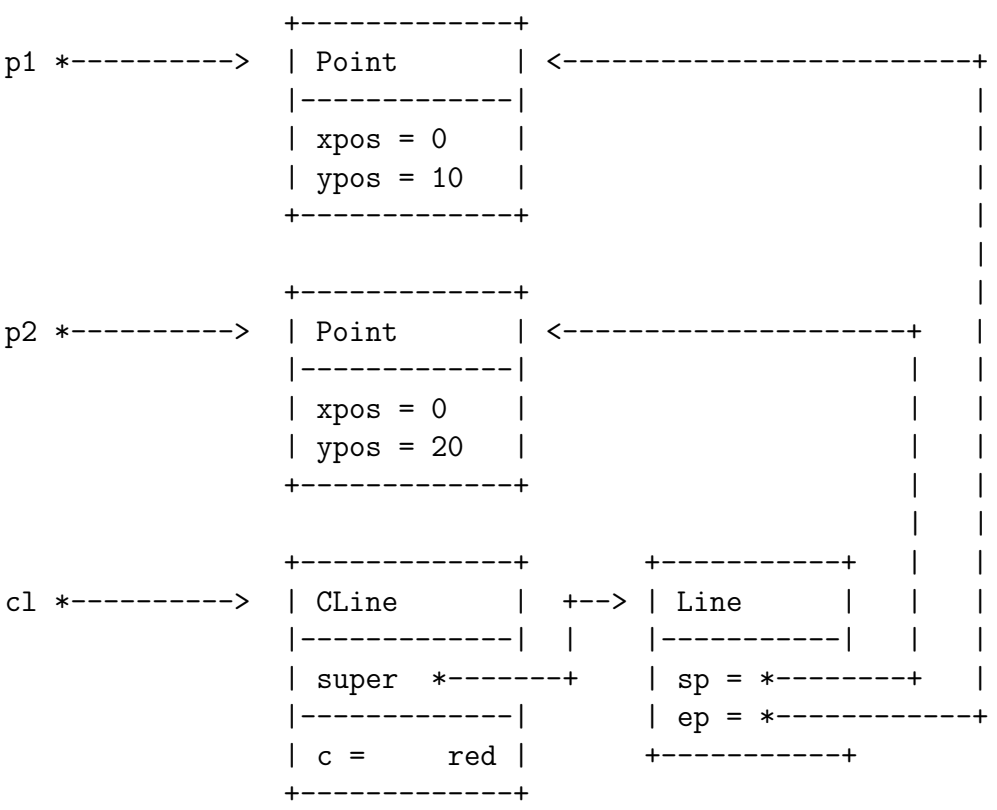
What about inheritance? Well, when a class (say `A`) extends another class (say `B`), it allocates not only a chunk of memory to hold the new instance of type `A`, but also allocates a chunk of memory for an instance of type `B` that will hold the fields that are inherited from `B`. (This is the implicit delegation model I presented when I talked about inheritance — the instance of type `B` created is the implicit delegate for `B`.) Thus, in the presence of inheritance, creating an instance actually ends up creating a chain of instances. Consider the following example, where we define a subtype `CLine` of `Line` and inherit the `Line` methods:

```
class CLine (s:Point,e:Point,c:String) extends Line(s,e) {

  // We inherit Line's methods:
  // start(), end(), setStart(), setEnd()

  def color ():String = c
```

```
  override def toString ():String = "[" + start() + " <-> " + end() + "]@
    " + c
}
```

Here's a sample execution with resulting diagram:

```
val p1 = new Point(0,10)
val p2 = new Point(0,20)
val cl = new CLine(p2,p1,"red")
```



Thus, when we invoke `cl.end().yPos()`, we follow the arrow from `cl` to find the `CLine` instance; it does not contain the `ep` field, so we follow the `super` arrow that points to the inherited instance, which does contain the `ep` field, and to get the y-coordinate of that point, we follow the arrow to find the appropriate point instance, and extract its `ypos` value.

The above diagram is grossly simplified, because, in particular, every instance in Scala extends at least the `Any` class. Thus, if you *really* wanted to be accurate, you would have to create inheritance arrows to instances of class `Any`, and so on, for every instance. The above model is sufficient for quick back-of-the-envelope computations, though. In particular, it lets you predict the obvious result:

```
scala> p1.setYCoord(99)

scala> cl
res1: CLine = [(0,20) <-> (0,99)]@red
```

16.4 Shallow and Deep Copies

As we saw in the first example, 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.

An example of sharing was the `new Line(p1,p1)` example earlier. The two fields `sp` and `ep` of the newly created `Line` instance end up pointing to the same instance of `Point`, so that modifying that instance is reflect 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 point:

```
class Triangle (b:Line, p:Point) {

  private var base:Line = b
  private var tip:Point = p

  def side1 ():Line = base

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

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

  def setBase (l:Line):Unit = (base=l)
  def setTip (p:Point):Unit = (tip=p)

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

Now we can get even more interesting sharing going between lines and points.

Consider the following definitions:

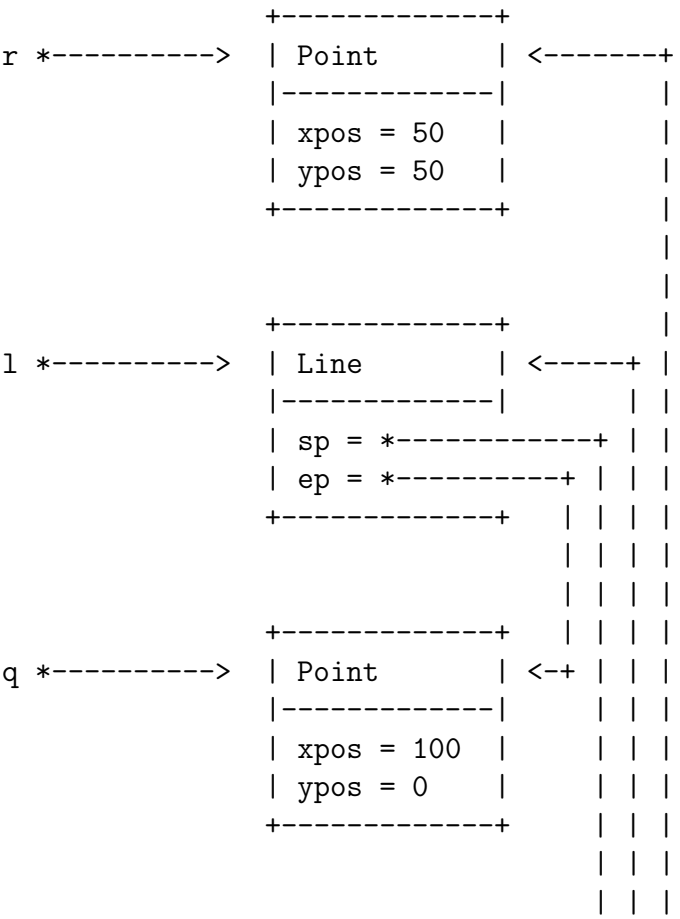
```
scala> val p = new Point(0,0)
p: Point = (0,0)

scala> val q = new Point(100,0)
```

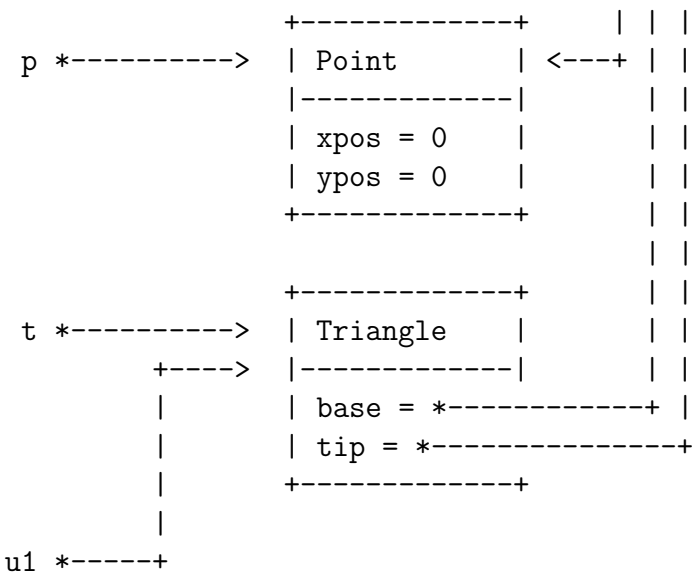
```
q: Point = (100,0)

scala> val r = new Point(50,50)
r: Point = (50,50)

scala> val l = new Line(p,q)
l: Line = [(0,0) <-> (100,0)]

scala> val t = new Triangle(l,r)
t: Triangle = [(0,0) <-> (100,0)]  [(0,0) <-> (50,50)]  [(100,0) <-> (50,50)]
```

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) <-> (100,0)]  [(0,0) <-> (50,50)]  [(100,0) <-> (50,50)]
```

But of course, these is *maximal* sharing between `t` and `u1`:





In particular, changing any part of one changes the other:

```
scala> t.setTip(new Point(9,9))

scala> u1
res7: Triangle = [(0,0) <-> (100,0)]  [(0,0) <-> (9,9)]  [(100,0) <-> (9,9)]
```

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.

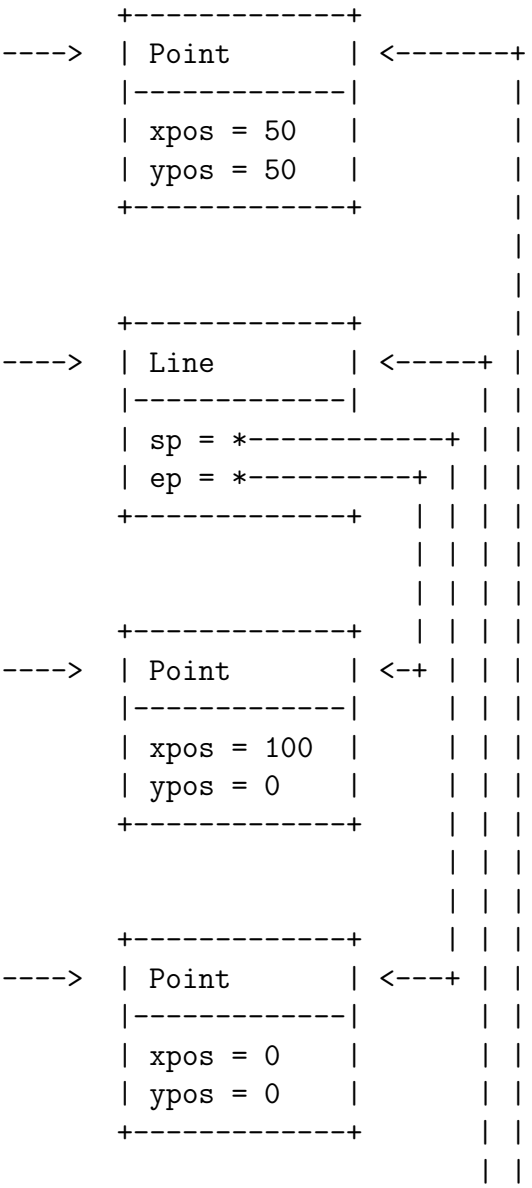
```
class Triangle (b:Line, p:Point) {

  private var base:Line = b
  private var tip:Point = p

  def side1 ():Line = base

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

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

  def setBase (l:Line):Unit = (base=l)
  def setTip (p:Point):Unit = (tip=p)

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

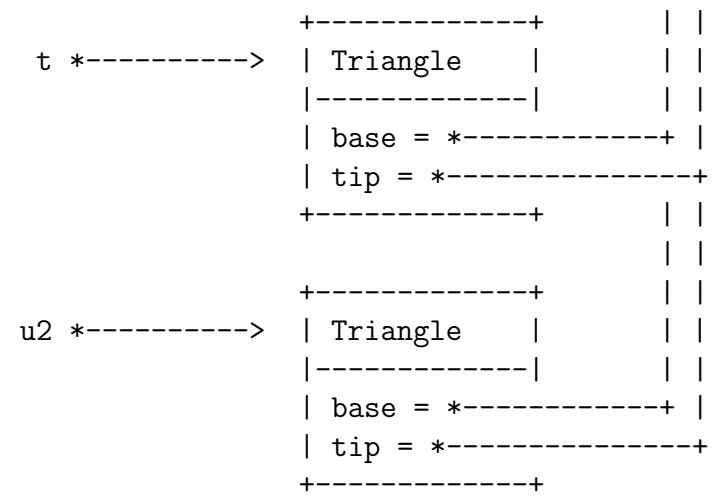
```
  override def toString ():String = side1() + "  " + side2() + "  " +
   side3()
}
```

Now, if we redefine p,q,r,l,t as above and say:

```
scala> val u2 = t.copy()
u2: Triangle = [(0,0) <-> (100,0)]  [(0,0) <-> (9,9)]  [(100,0) <-> (9,9)]
```

we end up with less sharing between t and u2:





But there is still some sharing. In fact, if we change either of `l`, `p`, `q`, or `r`, we see that change both in `t` and in `u2`:

```
scala> t.side1().end().setXCoord(999)

scala> u2
res12: Triangle = [(0,0) <-> (999,0)]  [(0,0) <-> (50,50)]  [(999,0) <-> (50,50)]
```

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 line, 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:

```
class Triangle (b:Line, p:Point) {

  private var base:Line = b
  private var tip:Point = p

  def side1 ():Line = base

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

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

```
  def setBase (l:Line):Unit = (base=l)
  def setTip (p:Point):Unit = (tip=p)

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

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

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

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 `Point` and in `Line` — let's do that, and add some shallow `copy()` methods as well:

```
class Point (x:Int, y:Int)  {

  // executed whenever you create a Point instance
  if (x < 0 || y < 0)
    throw new IllegalArgumentException("Point not positive")

  private var xpos:Int = x
  private var ypos:Int = y

  def xCoord ():Int = xpos
  def yCoord ():Int = ypos

  def setXCoord (x:Int):Unit =
    if (x >= 0)
      xpos = x
    else
      throw new IllegalArgumentException("Point not positive")

  def setYCoord (y:Int):Unit =
    if (y >= 0)
      ypos = y
    else
      throw new IllegalArgumentException("Point not positive")

  def copy ():Point = new Point(xCoord(),yCoord())
```

```
  def deepCopy ():Point = new Point(xCoord(),yCoord())

  override def toString ():String = "(" + xCoord() + "," + yCoord() + ")"
}


class Line (s:Point,e:Point) {

  private var sp : Point = s
  private var ep : Point = e

  def start ():Point = sp
  def end ():Point = ep

  def setStart (p:Point):Unit = (sp=p)
  def setEnd (p:Point):Unit = (ep=p)

  def copy ():Line = new Line(start(),end())

  def deepCopy ():Line = new Line (start().deepCopy(),
                                   end().deepCopy())

  override def toString ():String = "[" + start() + " <-> " + end() + "]"
}
```

Note that `copy()` and `deepCopy()` for `Point` are the same. That's because copying an integer is the same as creating a new integer. (Because `Int` is an immutable class!) 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 redefine `p`,`q`,`r`,`l`,`t` as before and say:

```
scala> val u3 = t.deepCopy()
u3: Triangle = [(0,0) <-> (100,0)]  [(0,0) <-> (9,9)]  [(100,0) <-> (9,9)]
```

You get that `t` and `u3` are truly disjoint: each of their `start` and `end` fields point to *different* instances. I will let you draw the resulting diagram.