17 A Model of Mutation in Java

We have been avoiding mutations until now; but there are there, in the Java system, for better or for worse, especially in the libraries.

Recall that a class is *immutable* is the state of an instance of the class never changes after it's been created (where the state of an instance includes the value of all its fields). In contrast, a class is *mutable* if its instances may change state during their lifetime.

So let's move towards understanding mutation. Mutation can be problematic because an object 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 code is also easier to parallelize, but that's a bit beyond what we're looking at here.

For the purpose of this lecture, we shall consider the following two classes, a class `Point` representing two-dimensional points, and a class `Rect` representing rectangles, defined by two points.

```
public class Point {
    private int xpos;
    private int ypos;

    private Point (int x, int y) {
      this.xpos = x;
      this.ypos = y;
    }

    public static Point create (int x, int y) { return new Point(x,y); }

    public int xPos () { return this.xpos; }

    public int yPos () { return this.ypos; }

    public String toString () { return "(" + xpos + "," + ypos + ")"; }
}


public class Rect {
```

```
    private Point fst;
    private Point snd;

    protected Rect (Point f, Point s) {
      this.fst = f;
      this.snd = s;
    }

    public static Rect create (Point f, Point s) {
      return new Rect(f,s);
    }

    public Point first () { return this.fst; }

    public Point second () { return this.snd; }

    public String toString () {
      return "[" + fst.toString() + "," + snd.toString() + "]";
    }
}
```

17.1 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, you still want to keep your fields private, and provide selectors and setters for each field that can mutate. 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 changing the creator or constructor:

```
  public static Point create (int x, int y) {
    if (x < 0 || y < 0) throw new IllegalArgumentException ("negative");
    return new Point(x,y);
  }
```

The creator enforces the invariant that the coordinates of a point are always nonnegative.

If we allow unrestricted field access, however, then anyone can just change one of the bounds 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:

```
public void setXPos (int x) {
  if (x < 0) throw new IllegalArgumentException("negative");
  this.xpos = x;
}

public void setYPos (int y) {
  if (y<0) throw new IllegalArgumentException("negative");
  this.ypos = y;
}
```

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

Now, one difficulty with mutability is that mutability is contagious: a class that looks immutable can be mutable if it relies on classes that are mutable. Consider the mutable form of `Point`, with setters `setXPos()` and `setYPos()`. Clearly, `Point` is a mutable class. What about `Rect` though, that does not have setters, and never changes the value of its fields? I claim it is mutable:

```
Point p1 = Point.create(0,0);
Point p2 = Point.create(10,10);
Rect r = Rect.create(p1,p2);
System.out.println(r.toString());
p1.setXPos(5);
System.out.println(r.toString());
```

which outputs:

```
[(0,0),(10,10)]
[(5,0),(10,10)]
```

and note that the state of `r` has changed — thus the class is mutable. That's something to keep in mind: a class may be mutable even though it looks like it is not.

17.2 Object 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 class, what actually gets changed, and who else can see it? The problem is that when the

state of an object can change, it becomes very important to understand when objects are *shared* between other objects, so that we can track when a change can be seen from another object.

To pick a silly example, if we write:

```
Point p1 = Point.create(0,0);
Point p2 = Point.create(0,0);
```

Then computing with `p1` and `p2` both give the same result, and if we mutate `p1` (by calling `setXPos()`), `p2` is unaffected. As opposed to:

```
Point p1 = Point.create(0,0);
Point p2 = p1;
```

Where again computing with `p1` and `p2` both give the same result, but if we mutate `p1` anywhere, then `p2` is changed as well. 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 Java represents objects 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 state.)

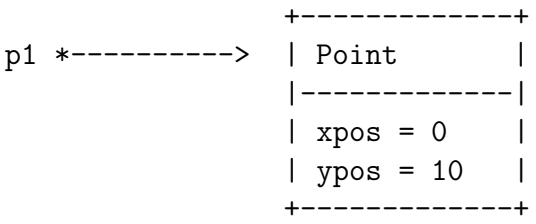
When you create an object with a `new` statement, a block of memory is allocated in memory (in the heap) representing the new object, which gets filled by the constructor. Here is how we represent an object in the heap:

```
+------------+
| class      |
|------------|
| non-static |
|  fields    |
+------------+
```

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

```
Point p1 = Point.create(0,10);
```

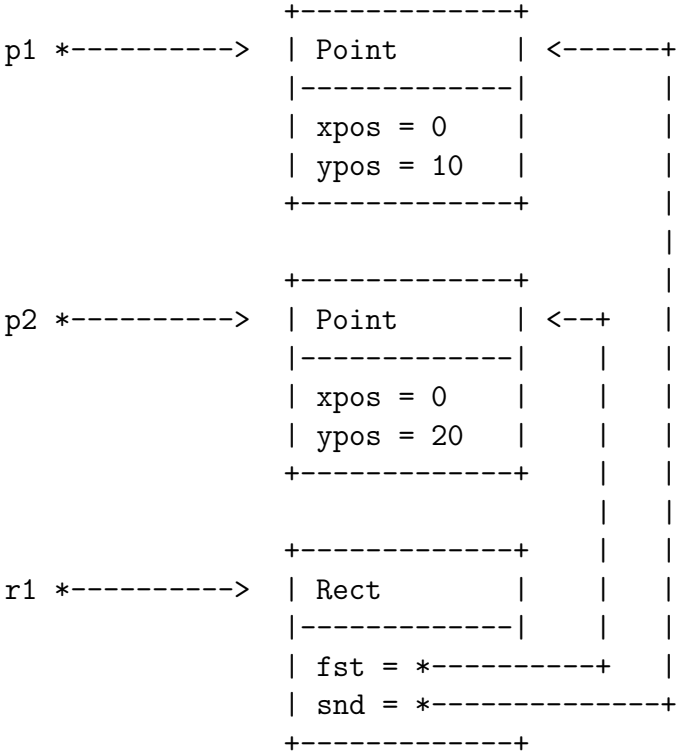
a new object is created in memory, say living at some address *addr*, and variable `p1` holds value *addr*. We can represent this as follows:



Now, when you pass an object as an argument to a method, what you end up passing is the *address* of that object (that is, the value returned by the constructor). That is how objects get manipulated.

Consider the class `Rect` above, representing rectangles. When you create a `Rect` object, passing in two points, you get as values of the fields `fst` and `snd` in the created rectangle the addresses of the two points that were used to create the rectangle in the first place.

```
Point p1 = Point.create(0,10);
Point p2 = Point.create(0,20);
Rect r1 = Rect.create(p2,p1);
```



To find the value of a field, you follow the arrows to the object that holds the field you are trying to access. Thus, `p1.xPos()` looks up the `xpos` field in the object pointed to by `p1`. Similarly, `r1.second().yPos()` access the `ypos` field of object returned by `r1.snd()`, which itself can be found as field `snd` in the object pointed to by `r1`.

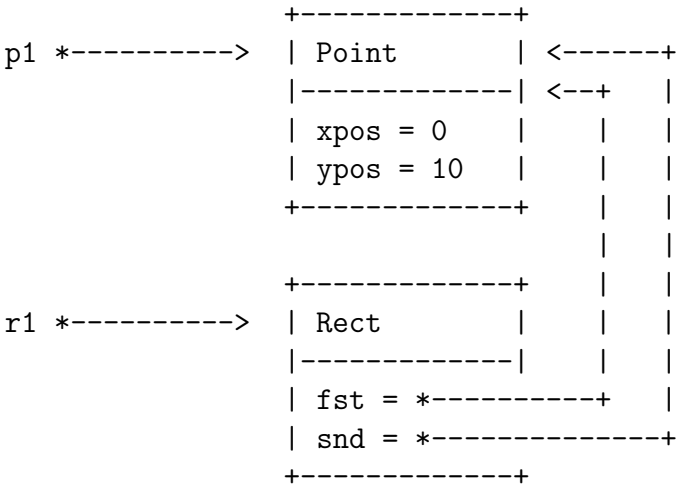
In particular, you see why if after creating the above we write

```
p1.setYPos (5);
System.out.println(r1.toString());
```

we get `[(0,20),(0,5)]`; intuitively, because `p1` is the same object as the point stored as the second point in `r1`. We call this phenomenon *sharing*. It is reflected by the fact that there are two arrows pointing to an object in a diagram.

Compare the above by what gets constructed if we write:

```
Point p1 = Point.create(0,10);
Rect r1 = Rect.create(p1,p1);
```



Here, the *same* point is used as first and second. Meaning, in particular, that if we update field `xpos` of `p1`, both the first and the second points of `r1` are also changed.

17.3 Static Fields

Static fields are fields annotated with the `static` qualifier. We haven't discussed them until now, because frankly they make no sense unless we have mutability. Intuitively, a static field is associated with a class, as opposed to the instances of a class. A static field is a field that is shared amongst all the instances of a class. So if one instance of the class updates the fields, that update is seen by all other instances.

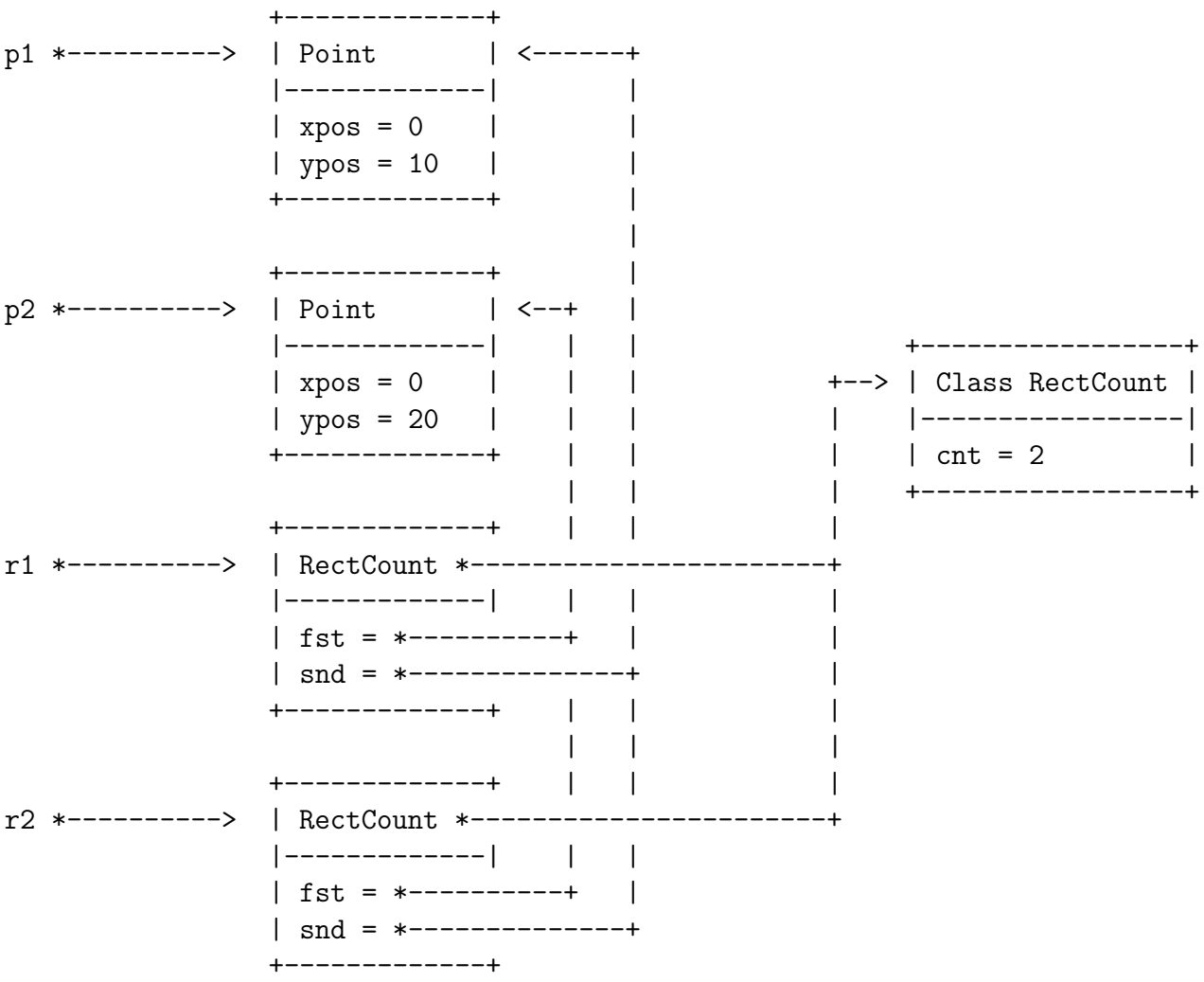
Suppose we create an alternate class `RectCount` instead of `Rect` that keeps track of the number of rectangles that has been created. The code is straightforward:

```
public class RectCount {
```

```
  private Point fst;
  private Point snd;
  private static int cnt = 0;

  private RectCount (Point f, Point s) {
    this.fst = f;
    this.snd = s;
    this.cnt = this.cnt + 1;
  }

  public static RectCount create (Point f, Point s) {
    return new RectCount(f,s);
  }

  public Point first () { return this.fst; }

  public Point second () { return this.snd; }

  public int count () { return this.cnt; }

  public String toString () {
    return "[" + fst.toString() + "," + snd.toString() + "] @ " + cnt;
  }
}
```

So what happens when this executes? Think of the static field `cnt` as somehow associated with the class rather than with instances of the class. In other words, there is a single copy of that field around, and all the instances of `RectCount` refer to that field. We are going to model this by having the class field of an object contain a point to a structure holding all the static fields of a class. This structure, which is shared amongst all the objects of the class, is created automatically by Java when you execute your program, before it yields control to your `main` method.

```
Point p1 = Point.create(0,10);
Point p2 = Point.create(0,20);
RectCount r1 = RectCount.create(p2,p1);
RectCount r2 = RectCount.create(p2,p1);
```



Note that both `r1` and `r2` have a link to the structure corresponding to class `RectCount`, which holds the static variable `cnt`. So when you have a `RectCount` object and you access its `cnt` field, Java sees it is not a variable in the object itself, so it will follow the "static" pointer to the structure holding the statics, and look for the field there.

17.4 Inheritance [Not Covered in Class]

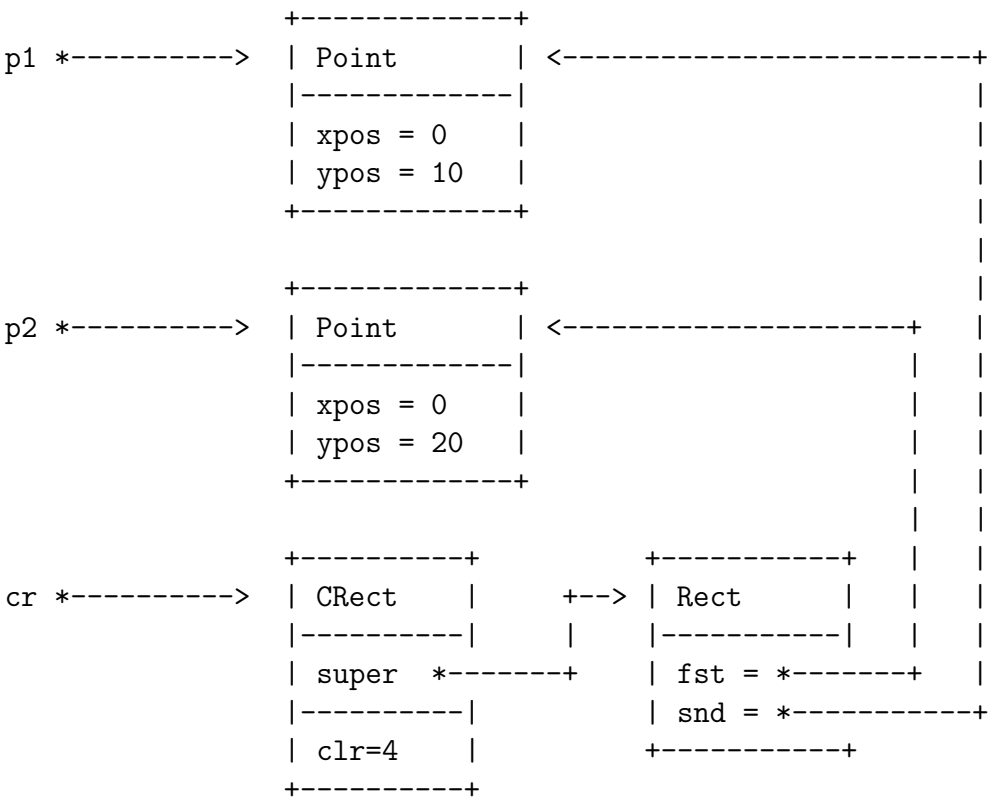
So that takes care of static fields. What about inheritance? Well, when a class (say `A`) extends another class (say `B`), it allocates not only a chunk of memory to hold the new object of type `A`, but also allocates chunk of memory for an object of type `B` that will hold the fields that are inherited from `B`. Thus, in the presence of inheritance, creating an object actually ends up creating a chain of objects. Consider the following example:

```
public class CRect extends Rect {
  private int clr;

  protected CRect (Point f, Point s, int c) {
    super(f,s);
    this.clr = c;
  }

  public static CRect create (Point f, Point s, int c) {
    return new CRect(f,s,c);
  }

  public int color () {
    return this.clr;
  }
}
```

Executing the following:

```
Point p1 = Point.create(0,10);
Point p2 = Point.create(0,20);
CRect cr = CRect.create(p2,p1,4);
```

yields the following:



Thus, when we invoke `cr.second().yPos()`, we follow the arrow from `cr` to find the `CRect` object; it does not contain the `snd` field, so we follow the `super` arrow that points to the inherited object, which does contain the `snd` field, and to get the y-coordinate point, we follow the arrow to find the appropriate point object, and extract its `ypos` value.

The above diagram is grossly simplified, because, in particular, every object in Java extends at least the `Object` class. Thus, if you *really* wanted to be accurate, you would have to create inheritance arrows to objects of class `Object`, and so on, for every object. The above model is sufficient for quick back-of-the-envelope computations, though.

17.5 Shallow and Deep Copies

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

An example of sharing was the `Rect.create(p1,p1)` example earlier. The two fields `fst` and `snd` of the newly created `Rect` object end up pointing to the same object, so that modifying one of course leads to modifying the other.

It makes sense sometimes to create a new object that is an exact copy of an object, but does not have any of its sharing. Of course, the best place to put this is as a method to the object itself: "Object, copy thyself!"

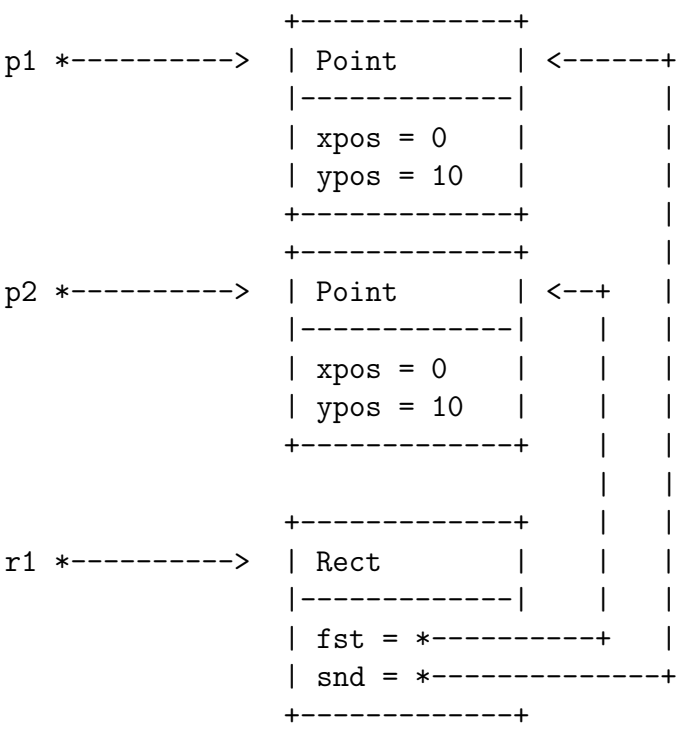
Let's write a method for `Point` that creates a new copy of a point:

```
public Point copy () {
  return new Point(this.xpos, this.ypos);
}
```

Simple enough. And indeed, if we draw what's happening when we write, say,

```
Point p1 = Point.create(0,10);
Point p2 = p1.copy();
Rect r1 = Rect.create(p2,p1);
```

we get what we expect, two distinct copies of the same object with the same field values:



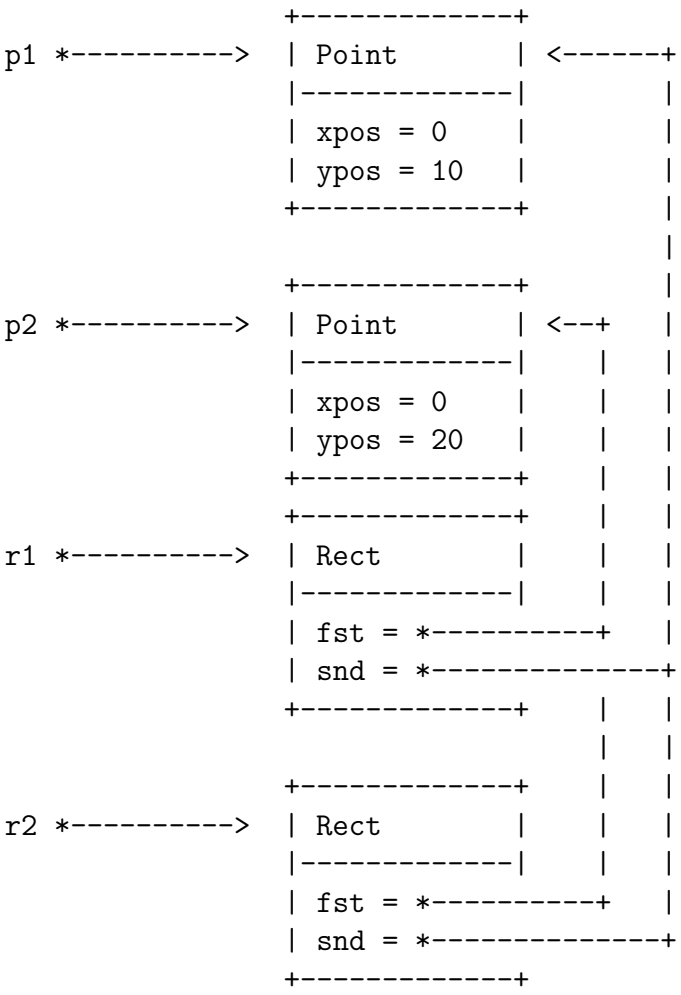
However, if we implement a similar method in `Rect`, we have a problem:

```
public Rect copy () {
  return new Rect(this.fst,this.snd);
}
```

And let's execute the following code:

```
Point p1 = Point.create(0,10);
Point p2 = Point.create(0,20);
Rect r1 = Rect.create(p2,p1);
Rect r2 = r1.copy();
```

If you work through carefully what happens, you get the following:



Which may not be *quite* what we want. It does create a fresh copy of `r1`, but since `fst` and `snd` are simply given the address of the object pointed to by `fst` and by `snd` are the same in both `r1` and `r2`.

So while `r2` is a copy of `r1`, they are not fully disjoint. Rather, it is what we call a *shallow copy* of `r1`. The "top level" of the objects 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 rectangle, then we need to make what is called a *deep copy*, that is, a copy that recursively deep copies (creating new objects) for every object held in every variable, all the way down. Let's implement a method `deepCopy`, in `Rect`:

```
public Rect deepCopy () {
  Point newfst = this.fst.deepCopy();
  Point newsnd = this.snd.deepCopy();
  return new Rect(newfst, newsnd);
}
```

So, you see, to create a deep copy of a rectangle, we recursively deep copy all the values of all the relevant fields, and create a new rectangle with those new values.

This also means that we need a `deepCopy` function in `Point`:

```
public Point deepCopy () {
  return new Point(this.xpos, this.ypos;
}
```

Pleasantly, this is exactly the same as a shallow copy. That's because copying an integer does require us to create a new integer. (That's a nice property of primitive types.) Let's keep the two methods around anyways, just to make clear that they have different roles, even though they do the same thing.

Now, if we write:

```
Point p1 = Point.create(0,10);
Point p2 = Point.create(0,20);
Rect r1 = Rect.create(p2,p1);
Rect r2 = r1.deepCopy();
```

You get that `r1` and `r2` are truly disjoint: each of their `fst` and `snd` fields point to *different* objects. I will let you draw the pictures, as a simple exercise.