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.

So let's move towards understanding mutation. We already saw why mutation was problematic: because an object can change under you, which can lead to hard-to-find bugs. Hence, my advocacy of immutability.

Now let's suppose that you have understood this issue about mutability, and that you accept the ensuring risks. In other words, you decided to have some classes with some mutable state (in particular, mutable fields).

First off, let me settle a minor detail. Even if you are okay with having mutable state, you still want to keep your fields private, and provide selectors (and setters) for each field that can mutate Why? Consider a class Interval that defines intervals of integers:

```
public class Interval {
  public int lower;
  public int higher;

  public Interval (low, high) {
    if (low > high) throw new IllegalArgumentException ("...");
    this.lower = low;
    this.higher = high;
  }

  public int getLower () {
    return this.lower;
  }

  public int getHigher () {
    return this.higher;
  }

  public int size () {
    return this.getHigher() - this.getLower();
  }
}
```

The constructor enforces the invariant that the lower bound is less (or equal) to the upper bound. And the methods in the body can use that invariant. In particular, the `size` method is always guaranteed to return a non-negative integer when the invariant is true.

However, if you allow unrestricted field access, then anyone can just change one of the bounds and break the invariant. Which, in this case, can lead the class to return a negative integer for the size of the interval, which may or may not be a problem..

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

```
public void setLower (int low) {
  if (low > this.higher) throw new IllegalArgumentException("...");
  this.lower = low;
}

public void setHigher (int high) {
  if (this.lower > high) throw new IllegalArgumentException("...");
  this.higher = high;
}
```

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

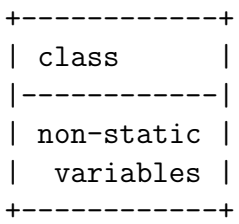
To work with mutation correctly, and help you track down bugs, you need to have a good understanding of what changes when something changes! You update some field in some class, what actually gets changed, and who else can see it?

My claim: 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.)

Object Creation and Manipulation

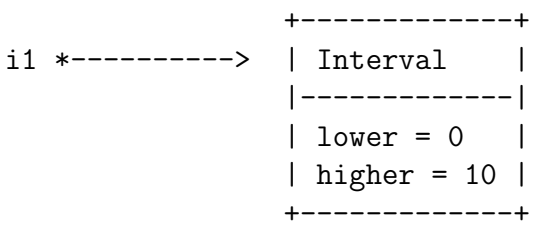
When you create an object with a `new` statement, a block of memory is allocated in memory (in the heap) representing the new object, which gets filled by the constructor. Here is how we represent an object in the heap:



(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. Thus, for instance, when you write

```
Interval i1 = new Interval(0,10);
```

a new object is created in memory, say living at some address *addr*, and variable `i1` 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.

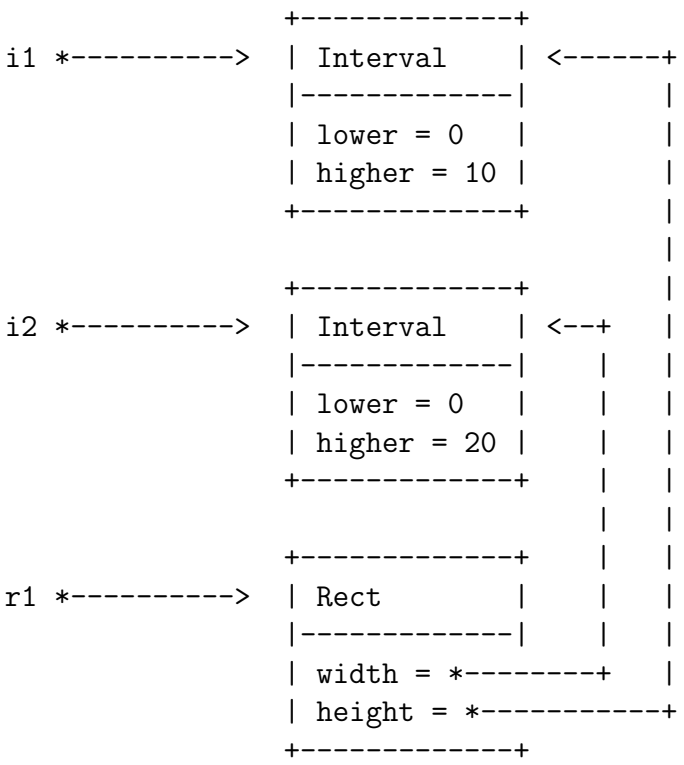
As an example, consider a class `Rect` to represent rectangles. A rectangle is made up of two intervals, representing the width and the height of the rectable, respectively, in the plane.

```
public class Rect {
  private Interval width;
  private Interval height;

  public Rect (Interval width, Interval height);
    this.width = width;
    this.height = height;
```

```
  }

  public Interval getWidth () {
    return this.width;
  }

  public Interval getHeight () {
    return this.height;
  }

  public int area () {
    return this.width.size() * this.height.size();
  }
}
```

Thus, when you create a `Rect` object, passing in two intervals, you get as values of the fields `width` and `height` in the created rectable the addresses of the two intervals that were used to create the rectangle in the first place.

```
Interval i1 = new Interval(0,10);
Interval i2 = new Interval(0,20);
Rect r1 = new Rect(i2,i1);
```



To find the value of a field, you follow the arrows to the object that holds the field you are trying to access. Thus, `i1.getHigher()` looks up the `higher` field in the object pointed to by `i1`. Similarly, `r1.height.getLower()` access the `lower` field of object pointed out by `r1.height`, which itself can be found as field `height` in the object pointed to by `r1`.

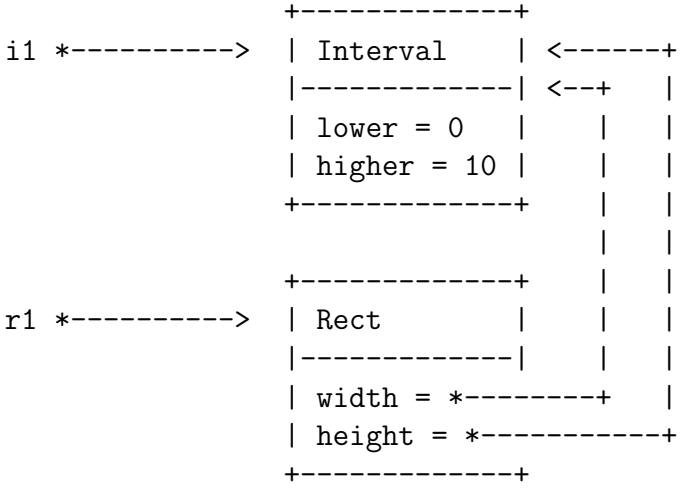
In particular, you see why if after creating the above we write

```
i1.setLower (5);
System.out.println("Lower width of r1 = " + r1.getWidth().getLower());
```

we get `5`; intuitively, because `i1` is the same object as the interval stored as the width 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:

```
Interval i1 = new Interval(0,10);
Rect r1 = new Rect(i1,i1);
```



Here, the *same* interval is used as width and height. Meaning, in particular, that if we update field `higher` of `i1`, both the width and the height of `r1` are also changed.

Static Variables

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:

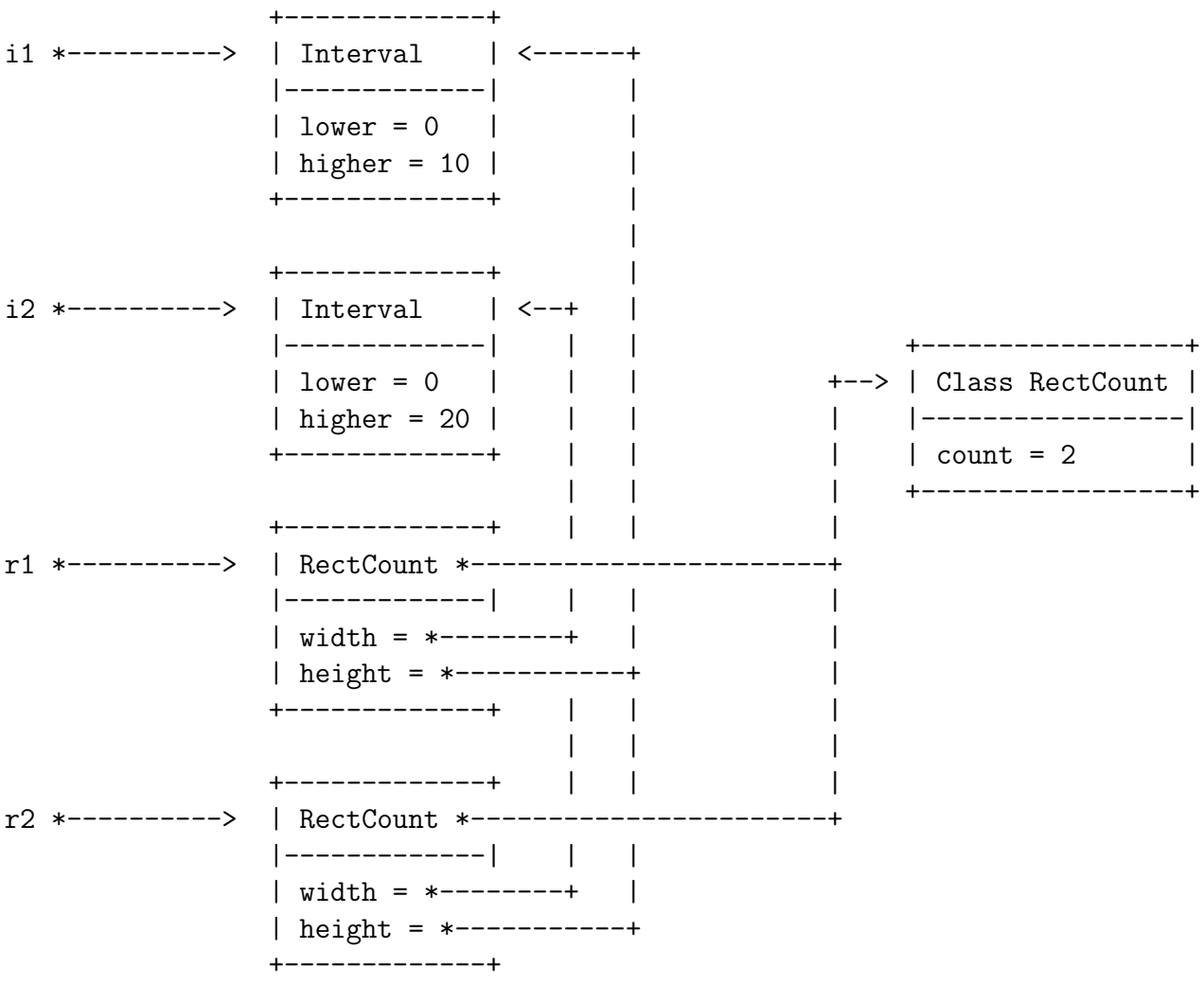
```
public class RectCount {
  private Interval width;
```

```
  private Interval height;
  private static int count = 0;

  public Rect (Interval width, Interval height);
    this.width = width;
    this.height = height;
    this.count = this.count + 1;
  }

  public Interval getWidth () {
    return this.width;
  }

  public Interval getHeight () {
    return this.height;
  }

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

  public int area () {
    return this.width.size() * this.height.size();
  }
}
```

This should be a review of something we saw way back in the second week of classes.

So what happens when this executes. Well, as I mentioned in the second week of classes, the static variable `count` is somehow associated with the class, rather than with instances of the class. In other words, there is a single copy of that variable around, and all the instances of `RectCount` refer to that variable. 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.

```
Interval i1 = new Interval(0,10);
Interval i2 = new Interval(0,20);
RectCount r1 = new RectCount(i2,i1);
RectCount r2 = new RectCount(i2,i1);
```



Note that both `r1` and `r2` have a link to the structure corresponding to class `RectCount`, which holds the static variable `count`. So when you have a `RectCount` object and you access its `count` field, Java sees it's 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.

Inheritance

So that takes care of statics. 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 variables 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:

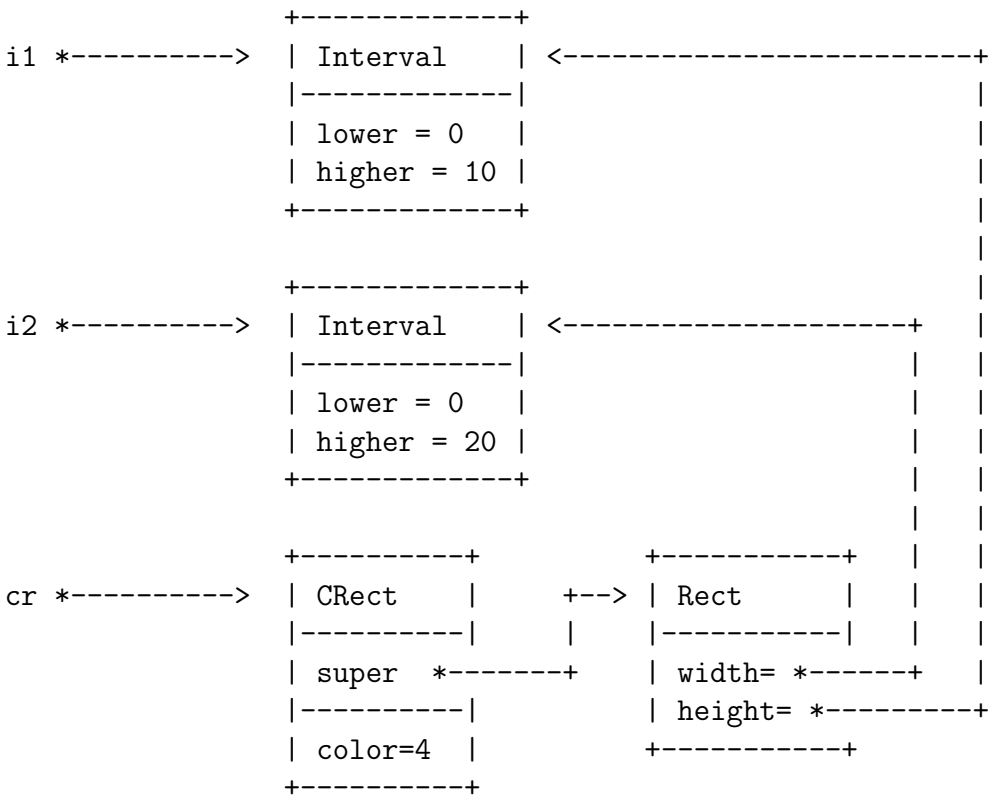
```
public class CRect extends Rect {
```

```
  int color;

  public CRect (Interval width, Interval height, int color) {
    super(width,height);
    this.color = color;
  }

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

Executing the following:

```
Interval i1 = new Interval(0,10);
Interval i2 = new Interval(0,20);
Rect cr = new CRect(i2,i1,4);
```

yields the following:



Thus, when we invoke `cr.getWidth().getLower()`, we follow the arrow from `cr` to find the `CRect` object; it does not contain the `width` field, so we follow the `super` arrow that points to the inherited object, which does contain the `width` field, and to get the lower bound of that interval, we follow the arrow to find the appropriate interval object, and extract its `lower` value.

(As was pointed out in class, the above diagram is 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.)

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 `new Rect(i1,i1)` example earlier. The two fields `width` and `height` of the newly created `Rect` object end up pointing to the same object, so that modifying one of course leads to modifying the other (duh, 'cause it's the same object!).

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!"

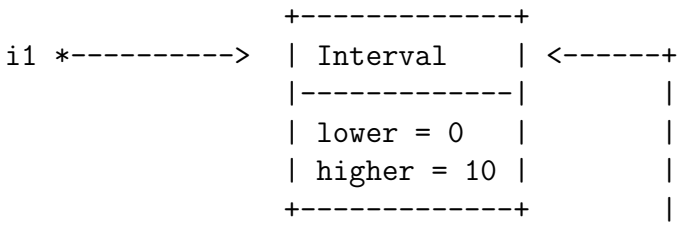
Let's write a method for `Interval` that creates a new copy of an interval.

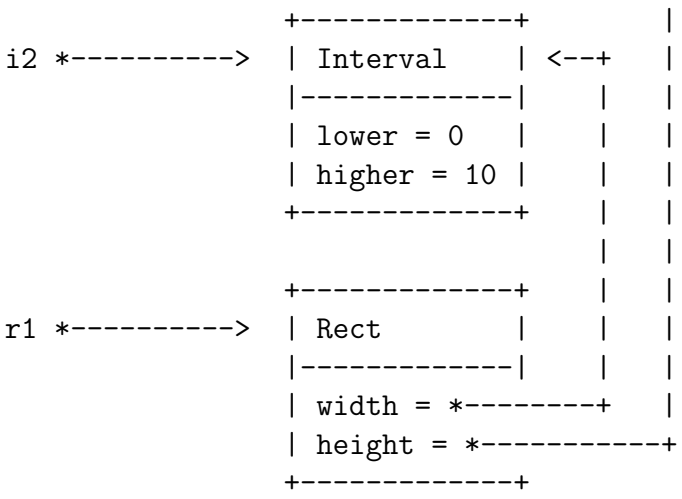
```
public Interval makeCopy () {
  return new Interval(this.lower, this.higher);
}
```

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

```
Interval i1 = new Interval(0,10);
Interval i2 = i1.makeCopy();
Rect r1 = new Rect(i2,i1);
```

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





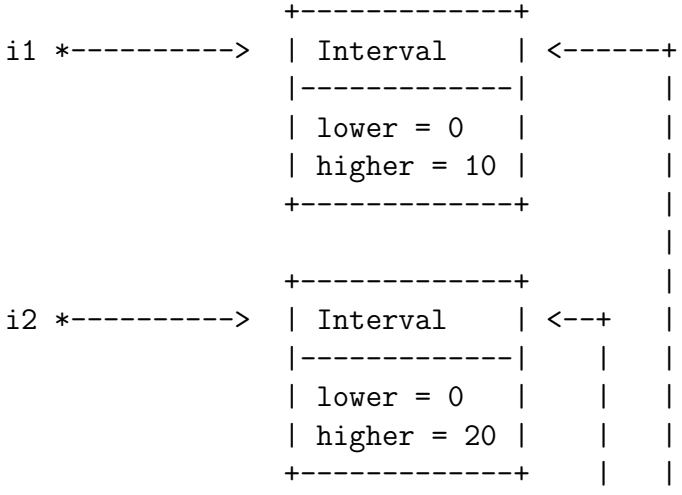
However, if we implement a similar method in `Rect`, we have a problem:

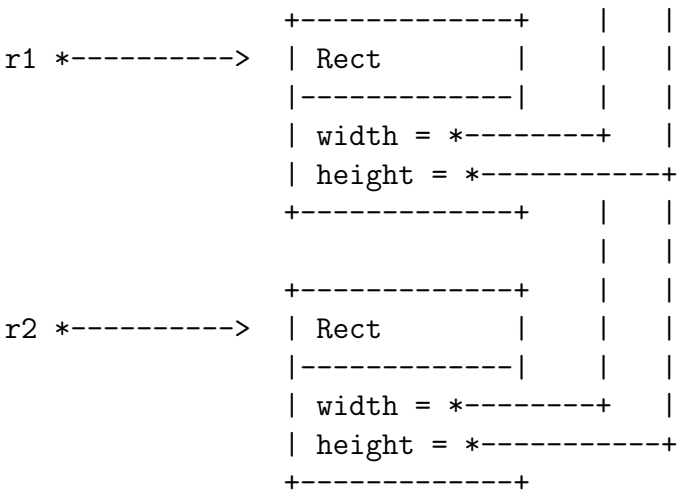
```
public Rect makeCopy () {
  return new Rect(this.width,this.height);
}
```

And let's execute the following code:

```
Interval i1 = new Interval(0,10);
Interval i2 = new Interval(0,20);
Rect r1 = new Rect(i2,i1);
Rect r2 = r1.makeCopy();
```

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





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

So while `r2` is a copy of `r1`, it is 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 variables live in different places), but any sharing within the values held in the variables 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 `makeDeepCopy`, in `Rect`:

```
public Rect makeDeepCopy () {
  Interval newWidth = this.width.makeDeepCopy();
  Interval newHeight = this.height.makeDeepCopy();
  return new Rect(newWidth, newHeight);
```

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

This also means that we need a `makeDeepCopy` function in `Interval`:

```
public Interval makeDeepCopy () {
  return new Interval(this.lower, this.higher);
}
```

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:

```
Interval i1 = new Interval(0,10);
Interval i2 = new Interval(0,20);
Rect r1 = new Rect(i2,i1);
Rect r2 = r1.makeDeepCopy();
```

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