Code Reuse: Inheritance

Recall the Point ADT we talked about in Lecture 8: The Point ADT:

```
public static Point make (int, int);
public int xPos ();
public int yPos ();

Point.nake(x,y).xPos() = x
Point.nake(x,y).yPos() = y
```

and its subclass, the CPoint ADT:

```
public static CPoint make (int, int, Color);
public int xPos ();
public int yPos ();
public Color color ();

CPoint.nake(x,y,c).xPos() = x
CPoint.nake(x,y,c).yPos() = y
CPoint.make(x,y,c).color() = c
```

From Lecture 8, we know that we can implement two classes defining the ADTs, and use extends to indicate to Java that there is subclassing going on.

However, if you remember how we did it back then, there was a lot of code redundancy between our `Point` and `CPoint` classes. Much of what `CPoint` did is the same thing that `Point` did. In fact, much of the code in `CPoint` I just copy-pasted from the `Point` class. That can be considered bad style. First, it is error-prone: suppose we find a bug in the `Point` class implementation, and correct it; it is quite easy to forget that we should also reflect the fix in the `CPoint` class.

So Java makes a code reuse technique available to you: *inheritance*. Inheritance lets you *reuse implementation code.* (Contrast to subclassing, which lets you reuse client code.) It is *not* subclassing, but it is related. Unfortunately, Java conflates the two, which will force us to jump through hoops at times.

Inheritance is incredibly powerful, and like any powerful tool, its power must be wielded wisely. Inheritance basically lets us only write the "new" stuff when defining a subclass.

Everything else comes from the definition of the superclass. Here is an alternate definition of the `CPoint` class using inheritance:

```
public class CPoint extends Point {
  private Color col;

  private CPoint (int x, int y, Color c) {
    super(x,y);
    col = c;
  }

  public static CPoint make (int x, int y, Color c) {
    return new CPoint(x,y,c);
  }

  public Color color () { return this.col; }
}
```

This is much nicer.

- Note that we have invoked the superclass constructor in `CPoint`'s constructor using `super(x,y)`.
- We get to reuse the fields holding the position, and reuse the position selector methods.

Unfortunately, the above code fails miserably to compile.

What's the problem? The problem is that we have made the constructor `Point` and the fields `xpos` and `ypos` private in the `Point` class. By definition, a private method and private fields are not accessible from outside the class in which they are defined. But the `CPoint` class must invoke the `Point` constructor.

One solution would be to make the constructor public in `Point`, but that goes against our philosophy, that everything which is not in the interface should be made private.

To get around this problem, Java introduces a new protection level, midway between private and public: protected. Roughly, when a method or a field is qualified as protected, then the method or the fields is not accessible from outside the class, *except* a subclass of the class.

Therefore, the complete code that works is as follows:

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

```
  protected Point (int x, int y) {
    xpos = x;
    ypos = y;
  }

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

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

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


public class CPoint extends Point {
  private Color col;

  private CPoint (int x, int y, Color c) {
    super(x,y);
    col = c;
  }

  public static CPoint make (int x, int y, Color c) {
    return new CPoint(x,y,c);
  }

  public Color color () {
    return this.col;
  }
}
```

There are some general rules for what is accessible from an inheriting subclass, and what is not. These are Java-specific, but every OO language will have similar kind of restrictions. Given an object A of a class T inheriting from S. The basic idea is that object A has all the fields and methods of T, as well as all the public and protected fields and methods of S.

- The constructor of T, when constructing A, will invoke the constructor of S, meaning

that the latter has to be protected or public. This invocation can be explicit by using the syntax `super(`arg_1`,...,`arg_k`);` as the first line in the constructor body in T. If no such call is made, then the constructor of S is invoked automatically by the compiler, with no arguments. (*Meaning that S must implement a protected or public constructor taking no arguments for this to compile.*)

- Remember dynamic dispatch: Every time you invoke a method m on A, the method code is looked up in the definition of T, the actual class from which object A was created. If there is no definition of m in T, then method m is looked for in S, and it is found only if it is protected or public. It is important that the *first* definition found is executed. This lets you overwrite a definition of a method in a subclass. (This is what happens for the canonical methods; the defaults are defined in class `Object`, but you are welcome to overwrite them.) The overwriting defintion can invoke the superclass's method by invoking `super.`method(...).

- Fields are more complicated. An object of class T can refer to a field defined in S, as long as that field is protected or public. Field shadowing—defining a field in a subclass that is also defined in the superclass—is the field-equivalent of method overwriting, except that the rules are much more painful to remember. Don't shadow fields unless you know what you are doing.[1]

There are some subtleties with how inheritance works in general, and in Java in particular. We already saw the issues with method and field access, requiring the need for a protected qualifier, and the difficulty with field shadowing.

For another subtlety, let's build up another example. Recall the stack signature we had in Lecture 11. Consider the dirty implementation we had before we learned about the recipe.

```
public class Stack {
  private int topVal;
  private Stack rest;

  protected Stack (Stack s, int i) {
    topVal = i;
    rest = s;
  }

  public static emptyStack () {
    return new Stack(null, null);
  }

  public static push (Stack s, int i) {
```

[1] See `http://articles.techrepublic.com.com/5100-22_11-5031837.html`, for instance.

```
    return new Stack(s,i);
  }

  public boolean isEmpty () { return (this.topVal==null); }

  public int top () {
    if (this.isEmpty ())
      throw new RuntimeException ("top of empty stack");
    return this.topVal;
  }

  public Stack pop () {
    if (this.isEmpty ())
      throw new RuntimeException ("pop of empty stack");
    return this.rest;
  }
}
```

Let's extend the Stack ADT. Suppose we really cared about keeping track of length of stacks in some application. A measurable stack has the following interface, an extension of the Stack ADT.

```
public static MStack emptyStack ();
public static MStack push (MStack, int);
public boolean isEmpty ();
public int top ();
public MStack pop ();
public int length ();

MStack.push(s,i).top() = i
MStack.push(s,i).pop() = s
MStack.push(s,i).isEmpty() = false
MStack.emptyStack().isEmpty() = true
MStack.emptyStack().length() = 0
MStack.push(s,i).length() = 1 + s.length()
```

The naive implementation of a length method by simply counting how many elements are in the stack can be inefficient if the stack is large. There is no way to make the "count how many elements are in the stack" algorithm more efficient, but there is a way to implement measurable stacks to make the `length` method more efficient: keep a count of the current stack size alongside the stack content, and simply increment the count upon a `push`. The `length` method now simply returns the current stack size, a constant-time field lookup

operation. This is an example of an *augmented data structure*, a data structure augmented with information that make some operations more efficient.

It makes sense to want to implement measurable stacks, which clearly should be a subclass of stacks, by inheriting from stacks:

```
public class MStack extends Stack {
  private int count;

  private MStack (MStack s, int i, int c) {
    super(i,s);
    count = c;
  }

  public static MStack emptyStack () {
    return new MStack(null,null,0);
  }

  public static MStack push (MStack s, int i) {
    return new MStack(s,i,1+s.length());
  }

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

  public MStack pop () { return (MStack) super.pop(); }
}
```

It all goes as expected, except for the `pop` method. That method does nothing special in the `MStack` class—all the action is in the `Stack` class. But the types are not right. The `pop` method should return an `MStack`. But we know by construction that what gets stored in the `rest` field when constructing an `MStack` is a measurable stack (although the field it is stored in has type `Stack`), so we need to re-implement the `pop` method in `MStack` to simply "correct" the type of the returned value.

This is a limitation of Java, and a subtlety to be aware of, that it requires you to jump through such hoops when inheriting from classes that have a method returning an object of the class itself.

Question: how would you go about using inheritance to implement ***`MStack`*** *when* ***`Stack`*** *is implemented using the recipe?*