15 Code Reuse: Inheritance and Delegation

Recall (a variant of) the Point ADT we talked about in lecture 7:

```
static Point create (int, int)
int xPos ()
int yPos ()
double distance (Point)

create(x,y).xPos() = x
create(x,y).yPos() = y
create(x,y).distance(p) = Math.sqrt(Math.pow(x-p.xPos(),2)+
                                    Math.pow(y-p.yPos(),2))
```

and its subclass, the CPoint ADT, where we're using `String` to represent colors:

```
static CPoint create (int, int, String)
int xPos ()
int yPos ()
double distance (CPoint)
String color ()

create(x,y,c).xPos() = x
create(x,y,c).yPos() = y
create(x,y).distance(p) = Math.sqrt(Math.pow(x-p.xPos(),2)+
                                    Math.pow(y-p.yPos(),2))
create(x,y,c).color() = c
```

From lecture 7, 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 String col;

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

  public static CPoint create (int x, int y, String c) {
    return new CPoint(x,y,c);
  }

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

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

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

  public double distance (Point p) {
      return Math.sqrt(Math.pow(this.xPos()-p.xPos(),2)+
                       Math.pow(this.yPos()-p.yPos(),2));
  }

}


public class CPoint extends Point {
  private String col;

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

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

```
  public String 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 object-oriented 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. One issue with inheritance is that the superclass needs to be set up so that it can be inherited from — in particular, methods and fields need to be protected and not private. This is not always possible if you do not have access to the source code of a library, for instance.n

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

There is an alternative to inheritance, available even in languages that do not have inheritance. The idea is that instead of inheriting methods (and fields) from a superclass, we shove away an instance of the superclass inside an object of the new class we are creating, and *delegate* method execution to that object when appropriate. Here is an example, again using the Point and CPoint ADT.

```
public class Point {
    // needed for subclassing in Java
    protected 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 double distance (Point p) {
        return Math.sqrt(Math.pow(this.xPos()-p.xPos(),2)+
                         Math.pow(this.yPos()-p.yPos(),2));
    }
}


public class CPoint extends Point {
    // 'extends' here really means subclassing, not inheritance

    private Point sup;         // delegate
    private String color;
```

```
    private CPoint (int x, int y, String c) {
        this.sup = Point.create(x,y);   // create the delegate
        this.color = c;
    }

    public static CPoint create (int x, int y, String c) {
        return new CPoint(x,y,c);
    }

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

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

    public double distance (Point p) {
        return this.sup.distance(p);
    }

    public String color () {
        return this.color;
    }
}
```

Note what is going on — we implement all methods in the subclass, except for many of these methods, we delegate to the underlying point that we created in the constructor.

In the Point/CPoint example, inheritance and delegation do not make a big difference. Let's look at an example where there is a difference. Consider the List ADT from several lectures ago, and let's extend it. Suppose we really cared about keeping track of length of lists in some application. A measurable list has the following interface, an extension of the List ADT.

```
static MList empty ()
static MList cons (int, MList)
boolean isEmpty ()
int first ()
MList rest ()
int length ()
```

```
empty().isEmpty() = true
cons(i,s).isEmpty() = false
cons(i,s).first() = i
cons(i,s).rest() = s
empty().length() = 0
cons(i,s).length() = 1 + s.length()
```

The naive implementation of a length method by simply counting how many elements are in the list can be inefficient if the list is large. There is no way to make the "count how many elements are in the list" algorithm more efficient, but there is a way to implement measurable list to make the `length` method more efficient: keep a count of the current list size alongside the list content, and simply increment the count upon a `cons`. The `length` method now simply returns the current list 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 lists, which clearly should be a subclass of lists, by inheriting from lists. The problem, if we use the implementation of lists derived from the design pattern we have, is that there is nothing to inherit from in the `List` abstract class! Yet, there is a lot of code that we would expect to be able to reuse. But inheritance is not helpful here. Delegation is. Here is an implementation of `MList` using delegation to reuse code from *any* list implementation:

```
public class MList extends List {
   // 'extends' ere really means subclassing, not inheritance

    private List del;       // delegate list
    private int length;

    private MList (int length, List del) {
        this.length = length;
        this.del = del;
    }

    public static MList empty () {
        return new MList(0,List.empty());
    }

    public static MList cons (int v, MList l) {
        return new MList(1+l.length(), List.cons(v,l));
    }

    public boolean isEmpty () {
```

```
        return del.isEmpty();
    }

    public int first () {
        return del.first();
    }

    public MList rest () {
        return (MList) del.rest();   // cast needed because Java doesn't
                                     // know that del.rest() really
                                     // gives you back an MList
    }

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

It is all as you expect, except for the `rest` method. That method does nothing special in the `MList` class — all the action is in the `List` class. But the types are not right. The `rest` method should return an `MList`. But we know by construction that what gets stored in the *rest* of the list when constructing an `MList` is a really a measurable list, even though it is statically treated as a `List`. We lose information here, treating an `MList` as a `List`, which we need to recover using a cast to "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 delegating to or inheriting from classes that have a method returning an object of the class itself.

The above implementation of `MList` relies on us knowing exactly what the relationship is between lists and measurable lists, and also us knowing properties of the `length` operation, and is somewhat ad hoc. Next lecture, we shall consider uniform ways of extending ADTs implemented using the design pattern we have seen in class.