Object-Oriented Design — Lecture 16
CS 3500 Spring 2010 (Pucella) — Tuesday, Mar 16, 2010

# 16 Code Reuse: Inheritance and Delegation

Way back in Lecture 7, we saw two ADTs, Point and CPoint, with a variant of the following for Point:

```
CREATORS     Point create (int, int)
OPERATIONS   int xPos ()
             int yPos ()
             double distance (Point)

SPECIFICATIONS
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 a variant of the following for CPoint (where we use `String` for colors):

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

SPECIFICATIONS
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
```

In Lecture 7, we saw that we could implement two classes `Point` and `CPoint` defining the ADTs and use `extends` to indicate to Java that there is subclassing going on. That subclassing let us reuse client code: if we write a function that expects `Point`s, we can pass it `CPoint`s and it will work (or at least, not complain that we are trying to invoke methods that do not exist).

Remember the classes we obtain as a result:

```
public class Point {
    private int xPos;
    private int yPos;

    // magic invocation to allow Point to have subclasses
    protected Point () {} ;

    private 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 xPos;
    }

    public int yPos () {
        return yPos;
    }

    public double distance (Point p) {
        return Math.sqrt(Math.pow(xPos−p.xPos(),2.0)+Math.pow(yPos−p.yPos(),2.0));
    }
}


public class CPoint extends Point {
    private int xPos;
    ptivate int yPos;
    private String color;

    private CPoint (int x, int y, String c) {
        xPos = x;
        yPos = y;
        color = carg;
    }
```

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

  public int xPos () {
    return xPos;
  }

  public int yPos () {
    return yPos;
  }

  public double distance (CPoint p) {
    return Math.sqrt(Math.pow(xPos-p.xPos(),2.0)+Math.pow(yPos-p.yPos(),2.0));
  }

  public String color () {
    return color;
  }

  public CPoint newColor (String c) {
    return CPoint.create(xPos,yPos,c);
  }
}
```

As we noted already back in Lecture 7, there is a lot of code redundancy between `Point` and `CPoint`: much of what `CPoint` does is the same thing that `Point` does. In fact, much of the code in `CPoint` I just copy-pasted from the `Point` class. That can be considered bad style, because it is error prone — were we to find a bug in the `Point` class implementation, it is easy to imagine correcting it and forgetting that we should also reflect the fix in the `CPoint` class. So how can we reuse code from `Point` in `CPoint`?

## 16.1 Inheritance

Java and most object-oriented languages makes a code reuse technique available to you: *inheritance.* Inheritance lets you *reuse implementation code.* (Contrast this with subclassing, which lets you reuse client code.) Inheritance is an implementation technique — a client generally couldn't care less if you implement something via inheritance or not. Inheritance is related to subclassing, but it is a different notion. Unfortunately, Java conflates the two, which will force us to jump through hoops at times.

Inheritance is 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 (we keep `Point` the same):

```
public class CPoint extends Point {
  private String color;

  private CPoint (int x, int y, String c) {
    super(x,y);   // construct an implicit instance of Point
    color = c;
  }

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

  public String color () {
    return color;
  }

  public CPoint newColor (String c) {
    return CPoint.create(xPos(),yPos(),c);
}
```

This is much nicer. Or at least, shorter. The intuition here is that when Java constructs an instance of `CPoint`, it creates also constructs an instance of the class it is inheriting from (here, `Point`) to which you do not have access explicitly, but that has fields and methods that can be accessed by `CPoint` implicitly. A couple of things to note:

- We invoke 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.

That’s the idea. Unfortunately, the code above fails miserably to compile.

What’s the problem? The problem is that we have made the constructor `Point(x,y)` private in the `Point` class. By definition, a private method is not accessible from outside the class in which it is defined. But the `CPoint` class must invoke the `Point` constructor in its own 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 annoyance, 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 field is not accessible from outside the class, *except* from another class that extends it.

Therefore, the complete code that works is as follows:

```
public class Point {
   private int xPos;
   private int yPos;

   // magic invocation to allow Point to have subclasses
   protected Point () {} ;

   private 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 xPos;
   }

   public int yPos () {
      return yPos;
   }

   public double distance (Point p) {
      return Math.sqrt(Math.pow(xPos−p.xPos(),2.0) +
                       Math.pow(yPos−p.yPos(),2.0));
   }
}


public class CPoint extends Point {
   private String color;

   private CPoint (int x, int y, String c) {
     super(x,y);   // construct an implicit instance of Point
     color = c;
   }
```

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

  public String color () {
    return color;
  }

  public CPoint newColor (String c) {
    return CPoint.create(xPos(),yPos(),c);
}
```

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 instance $A$ of a class $D$ that inherits from $C$. Roughly, instance $A$ has all the fields and methods defined in $D$, as well as all the public and protected fields and methods of $C$.

- The constructor of $D$, when constructing $A$, will invoke the constructor of $C$, 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 $D$. If no such call is made, then the constructor of $C$ is invoked automatically by the compiler, with no arguments. (*Meaning that $C$ must implement a protected or public constructor taking no arguments for this to compile.*[1])

- Remember dynamic dispatch: Every time you invoke a method $m$ on $A$, the method code is looked up in the definition of the run-time type of $A$, say $D$. If there is no definition of $m$ in $D$, then method $m$ is looked for in $C$ (the class from which $D$ inherits), and it is found only if it is protected or public there. 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 $D$ can refer to a field defined in $C$, 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

[1]This is the explanation for the mysterious magic invocation needed for subclassing to work in Java. Because `extends` represents both subclassing and inheritance, even if you don't use inheritance, Java sets things up so that inheritance would work, meaning that it will invoke the constructor for the superclass, which requires it to be available.

overwriting, except that the rules are much more painful to remember. Don't shadow fields unless you know what you are doing.[2]

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 the case.

## 16.2 What About `move()`?

There is a phenomenon that occurs with inheritance (among others) that we need to understand and know how to address. Consider adding an operation such as `move()` to the Point and CPoint ADTs. Operation `move()` is interesting because it returns a value of the same type as the type of the ADT. Here are the new ADTs. First, Point:

```
CREATORS     Point create (int, int)
OPERATIONS   int xPos ()
             int yPos ()
             double distance (Point)
             Point move (int, int)

SPECIFICATIONS
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))
create(x,y).move(dx,dy) = create(x+dx,y+dy)
```

and CPoint:

```
CREATORS     CPoint create (int, int, String)
OPERATIONS   int xPos ()
             int yPos ()
             double distance (CPoint)
             Point move (int, int)
             String color ()
             CPoint newColor (String)

SPECIFICATIONS
```

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

```
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).move(dx,dy) = create(x+dx,y+dy,c)
create(x,y,c).color() = c
create(x,y,c).newColor(c’) = create(x,y,c’)
```

Implementing this using two classes directly without relying on inheritance is nothing special. It all works. But we don’t reuse code. What if try to do this with inheritance?

To a first approximation, we get something like this:

```
public class Point {
  private int xPos;
  private int yPos;

  // magic invocation to allow Point to have subclasses
  protected Point () {} ;

  private 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 xPos;
  }

  public int yPos () {
    return yPos;
  }

  public double distance (Point p) {
    return Math.sqrt(Math.pow(xPos−p.xPos(),2.0) +
                     Math.pow(yPos−p.yPos(),2.0));
  }

  public Point move (int dx, int dy) {
    return Point.create(xPos+dx,yPos+dy);
  }
}


public class CPoint extends Point {
  private String color;

  private CPoint (int x, int y, String c) {
    super(x,y);   // construct an implicit instance of Point
    color = c;
  }

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

  public String color () {
    return color;
  }

  public CPoint newColor (String c) {
    return CPoint.create(xPos(),yPos(),c);
  }
}
```

The problem is that this does not satisfy the specification: `CPoint` inherits a `move()` method from `Point` that returns a `Point`, whereas we want a `move()` method that returns a `CPoint`. If we try the following code, for instance, Java will reject the program:

```
  CPoint cp = CPoint.create(0,0,"Red");
  CPoint newcp = cp1.move(1,1);
```

And it's easy to see why: it's not safe! If you tried to access the `color()` method of the resulting `newcp`, it would fail to find it. (Make sure you understand why! This is the basics of the idea of run-time types, without which you can't really understand object-oriented programs.)

Therefore, we cannot inherit `move()`, and must instead define it in `CPoint` — here is the new `CPoint`:

```
public class CPoint extends Point {
  private String color;
  private CPoint (int x, int y, String c) {
    super(x,y);   // construct an implicit instance of Point
    color = c;
  }

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

  public CPoint move (int dx, int dy) {
    return CPoint.create(xPos()+dx,yPos()+dy,color);
  }

  public String color () {
    return color;
  }

  public CPoint newColor (String c) {
    return CPoint.create(xPos(),yPos(),c);
  }
}
```

As we will see later, every method in an ADT that returns a value of the same type as the ADT will need to have something special done if you try to inherit that method in another class. In the example above, we cannot actually inherit it, we need to redefine. In other cases, we will need to do something else. More examples of this below.

## 16.3 Delegation

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 explicitly create and shove away an instance of the superclass inside the instance of the new class we are creating, and explicitly *delegate* method execution to that instance of the superclass whenever appropriate. Here is an example, again using the Point and CPoint ADT (with the `move()` operation).

```
public class Point {
  private int xPos;
  private int yPos;
  // needed for subclassing in Java
  protected Point () {}

  private 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 xPos;
  }

  public int yPos () {
    return yPos;
  }

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

  public Point move (int dx, int dy) {
    return Point.create(xPos+dx,yPos+dy);
  }
}


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

  private Point del;          // delegate
  private String color;

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

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

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

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

  public double distance (CPoint p) {
    return del.distance(p);
  }

  public CPoint move (int dx, int dy) {
   return CPoint.create(xPos()+dx,yPos()+dy,color);
  }

  public String color () {
    return color;
  }

  public CPoint newColor (String c) {
    return CPoint.create(xPos(),yPos(),c);
  }
}
```

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. We get to reuse code from the `Point` class, but without relying on inheritance. You can think of delegation as an explicit form of inheritance. Note also that the `move()` method, because it returns a value of the type of the ADT, here just like in the inheritance example cannot be delegated, but must be redefined.

Of course, in the Point/CPoint example, delegation does not buy you much, because inheritance is quite usable. But as we will see below and next time, code reuse by delegation is handy in cases where inheritance is not available.

## 16.4 Delegation Example: Measurable Lists

Let's look at an example where delegation is nearly necessary. Consider the List ADT from several lectures ago:

```
CREATORS     List<A> empty ()
             List<A> cons (A, List<A>)
OPERATIONS   boolean isEmpty ()
             A first ()
             List<A> rest ()
```

with the obvious specification. (We only consider the above operations for simplicity.)

Suppose we want to extend that ADT with a `length()` operation that computes the length of a list. Clearly, we could add that operation to the implementation of lists, but suppose that we don't want to for whatever reason. (For instance, the code is part of a library and we don't have access to it, or can't modify it.)

So lets define a new ADT, called Measurable List, and figure out the kind of code reuse we can apply to the situation to reuse the implementation of lists we already have. Here is the Measurable List ADT:

```
CREATORS     MList<A> empty ()
             MList<A> cons (A, MList<A>)
OPERATIONS   boolean isEmpty ()
             A first ()
             MList<A> rest ()
             int length ()
```

The specification for `length()` is the obvious one:

```
empty().length() = 0
cons(v,l).length() = 1 + l.length()
```

Clearly, we could simply obtain an implementation of the Measurable List ADT by applying the Specification design pattern. That's easy. (Do it if you're shaky on the basics.) But my goal now is to try to implement a class `MList<A>` that is a subclass of `List<A>` (because everywhere you want to use a `List<A>` you should be able to use an `MList<A>` and things should still work — measurable lists are still lists after all) and moreover that reuses as much code as possible from `List<A>`. So we could imagine that have `MList<A>` inherit from `List<A>` is the way to go. But that does not make any sense. Why? Because if you recall our implementation of `List<A>` obtained from the Specification design pattern, `List<A>` is an abstract class, *and therefore contains no method definition.* There is nothing to inherit. Which makes inheritance useless. However, even though inheritance is unusable here, we can still use delegation. The idea is that a measurable list contains a delegate which is a normal list — representing the content of the measurable list — and operations delegate to that list, except for `length()` that computes the length.

Here is an implementation — I discuss some of the details below.

```
public class MList<A> extends List<A> {
   private List<A> del;          // delegate

   private MList (List<A> l) {
      del = l;
   }

   public static <B> MList<B> empty () {
      return new MList<B>(List.<B>empty());
   }
   public static <B> MList<B> cons (B val, MList<B> l) {
      return new MList<B>(List.cons(val,l));
   }

   public boolean isEmpty () {
      return del.isEmpty();
   }

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

   public MList<A> rest () {
      return (MList<A>) del.rest();
   }

   public int length () {
      if (isEmpty())
         return 0;
      else
         return 1 + rest().length();
   }
}
```

Everything is pretty much as you'd expect. The constructor simply records the delegate — the creators are in charge of creating the appropriate delegates. `MList.empty()` creates simply creates an empty-list delegate, while `Mlist.cons()` creates a cons-list delegate. All operations delegate to the list delegate, except for `length()` which is actually defined in `MList`.

The `rest()` operation is interesting. First off, notice that in the ADT, `rest()` returns a value of the type of the ADT. So already, based on what I've told you earlier, we know that something funny may be going on. And indeed: we cannot simply have `rest()` in `MList<A>` return the result of calling `rest()` on the delegate list, because the result is a `List<A>`, and we're expected to return an `MList<A>` from `rest()` in `Mlist<A>`. It's the same problem as earlier in the `move()` operation of Point/CPoint. Except here, we cannot just redefine `rest()` in `MList<A>`, because we want `rest()` to return the rest of the list stored as the deleguate.

We are saved by noticing one important fact: when we create a non-empty list delegate, it's in the `MList.cons()` creator. And in that creator, we are passed an `MList<A>` as a "rest" argument. So we know that the run-time type of the "rest" argument is at least an `MList<A>`. So when we get that "rest" back by calling `rest()` on the delegate, while what we get has compile-time type `List<A>` (because that's what the source code declaration says), we *know* that its run-time type is at least `MList<A>`. The Java type system isn't smart enough to track that information down, but because we understand what the code is doing, we know that's the case. So we can simply use a cast to "remind" the type system that the result of calling `rest()` on the delegate is in fact an `MList<A>`. (The cast does a check at run time because it doesn't completely trust you, but the type system is happy.)

This need for a cast here is a limitation of Java, which doesn't handle methods with result type the same type as the class too well (like many other object-oriented languages, to be fair) and is a subtlety to be aware of.