Subclassing and Inheritance

A Detour: Mutability versus Immutability

Before starting this lecture, let me say something about an important point in class design.

All of the classes we have worked with until now have been *immutable* classes, as oppose to *mutable* classes.

A class is *immutable* if instances of the class, once created, cannot change. In particular, field names cannot change value once they have been assigned in a constructor. The main way to make a class immutable is to make all fields private (so that no one from outside the class can change the value of a field), and to not provide any method that can change the value of a field.

Thus, any method that one would think should change a field of an object should in fact make a new copy of the object with the field value changed. Consider the following example. Suppose a Point class, with the following interface.

```
make : int x int -> Point
move : Point x int x int -> Point
getX : Point -> int
getY : Point -> int

getX(make(x,y)) = x
getY(make(x,y)) = y
getX(move(p,dx,dy)) = getX(p)+dx
getY(move(p,dx,dy)) = getY(p)+dy
```

Here is an immutable implementation:

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

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

```
  }

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

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

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

  public Point move (int dx, int dy) {
    return make (this.xpos + dx, this.ypos + dy);
  }
}
```

Compare it to the following mutable implementation (which may seem a bit artificial because of the signature of the ADT that requires `move` to return a Point object):

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

  private MPoint (int x, int y) {
    xpos = x;
    ypos = y;
  }

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

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

  public int getY () {
    return this.ypos;
  }
```

```
  public MPoint move (int dx, int dy) {
    this.xpos = this.xpos + dx;
    this.ypos = this.ypos + dy;
    return this;
  }
}
```

The difference may appear minor. However, I claim that the second implementation is much harder to reason about in a larger program.

Very roughly speaking, immutability lets us freely pass objects around without worrying about who exactly own a copy of the object. If a class is mutable, this means that an object stored in some other object can presumably change "under the hood", so to speak, by someone else changing that object through another name somewhere else in the system.

Suppose that the Point ADT is used in a larger windowing system, that includes a class Window with partial implementation:

```
public class Window {
  private MPoint topLeftCorner;
  private int width;
  private int height;

  private Window (MPoint tl, int w, int h) {
    topLeftCorner = tl;
    width = w;
    height = h;
  }

  public static make (MPoint tl, int w, int h) {
    return new Window (tl, w, h);
  }

  ...
}
```

What happens when we execute the following code?

```
...
MPoint origin = MPoint.make (20,20);
Window main = Window.make (origin,100,100);
MPoint moved_origin = origin.move (origin,-20,-20);
...
```

The `origin` object is passed to `Window.make` and stored in the `main` window as its top-left corner, but it is still accessible as the `origin` object in the above blurb. (In fact, what happens is that what gets passed to the `Window.make` method is really a pointer to the object definition, and that pointer just points to the original object.) So when it is moved in the `origin.move` method call, the object changes, and this change is actually visible from the object in the `main` window. In other words, by changing the object `origin` somewhere in the code, we managed to change the location of the top-left corner of the window `main`, without the window being actually made aware of it. This can easily lead to an extremely difficult to track down bug. Making the class immutable solves this problem. In particular, using `Point` in the above windowing code solves the problem.

Most classes in the Java libraries are immutable for that very reason.

Whether a class is immutable or not must be part of the specification. Sometimes, a class is said to have "no side-effects" to indicate that it is immutable. (Although having "no side-effects" means something more than just immutability.)

Extending Classes

Suppose that we extended the Point ADT into a Colored Point ADT, with the following interface:

```
make : int x int x Color -> CPoint
move : CPoint x int x int -> CPoint
getX : CPoint -> int
getY : CPoint -> int
getColor : CPoint -> Color

getX(make(x,y,c)) = x
getY(make(x,y,c)) = y
getColor(make(x,y,c)) = c
getX(move(p,dx,dy)) = getX(p)+dx
getY(move(p,dx,dy)) = getY(p)+dy
getColor(move(p,dx,dy)) = getColor(p)
```

I claim that this is an extension of the Point ADT simply because every colored point is really a point: everything you can do to a point, you can in fact do to a colored point. This is reflected by the algebraic specification of the two classes, which agree on the operations that the ADT have in common. More specifically, we can identify a colored point `CPoint.make (x,y,c)` as a `Point.make(x,y)`, and `getX(CPoint.make(x,y,c)) = getX(Point.make(x,y))`, and similarly for `getY` and the `move` operation. It is possible to make this notion of extension completely formal, but let me not do so here, and instead rely on your intuition.

What does this mean? It means, among other things, that if we implement the CPoint ADT as a class, we should really make it so that it is a *subclass* of the Point class. This means that an object of class of CPoint can be passed to any method or stored in any variable expecting an object of class Point. The main requirement, as I pointed out some lectures ago, is that every public method of Point should also be a public method of CPoint. (Of course, the implementation of the method can be quite different, but the method should exist, and have the same types.) Similarly, every public field of Point should be a public field of CPoint. Since we will never use public fields in this course (recall immutability!), we only have to worry about methods.

Subclassing is expressed in Java using an `extends` annotation on the class. Let `Point` be the above immutable implementation of the Point ADT.[1] Assume that we have a class `Color` already available, the details of which are completely irrelevant.

```
public class CPoint extends Point {
  private int xpos;
  private int ypos;
  private Color color;

  private CPoint (int x, int y, Color c) {
    xpos = x;
    ypos = y;
    color = c;
  }

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

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

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

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

[1] Why should the `CPoint` class, which is immutable, not be considered a subclass of the mutable `Point` class? Note that Java does not (in fact, cannot) enforce this.

```
    public CPoint move (int dx, int dy) {
      return make (this.xpos + dx, this.ypos + dy, this.color);
    }
  }
```

This works perfectly fine, and Java will work with this quite happily.

But there is a lot of code redundancy between `Point` and `CPoint`. Much of what `CPoint` does is the same thing that `Point` does. Case in point, much of the code above I just copy-and-pasted from the `Point` class. That's very bad style. Partly because 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 is an implementation technique that lets you reuse code when implementing subclasses. It is *not* subclassing, but it is related. (Unfortunately, Java conflates the two, which will force us to jump through some hoops later on.)

Inheritance is incredibly useful. It 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 CPoint2 extends Point {
    private Color color;

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

    public static CPoint2 make (int x, int y, Color c) {
      return new CPoint2 (x,y,c);
    }

    public Color getColor () {
      return this.color;
    }

    public CPoint2 move (int dx, int dy) {
      return make (this.xpos + dx, this.ypos + dy, this.color);
    }
  }
```

This is much nicer.

- Note that we have invoked the superclass constructor in `CPoint2`'s constructor using `super (x,y)`.
- Note also that every method that requires a `CPoint2` in the type must be rewritten, and cannot be inherited.
- Still, 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 `CPoint2` class must invoke the `Point` constructor, and in the `move` method, it access the `xpos` and `ypos` fields.

One solution would be to make the constructor and the fields public in `Point`, but that goes against our philosophy, that everything which is not in the interface should be made private. In particular, making the fields public makes the class mutable, which I argued is bad.

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 Point2 {
  protected int xpos;
  protected int ypos;

  protected Point2 (int x, int y) {
    xpos = x;
    ypos = y;
  }

  public static Point2 make (int x, int y) {
    return new Point2 (x,y);
  }

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

  public int getY () {
    return this.ypos;
```

```
  }

  public Point2 move (int dx, int dy) {
    return make (this.xpos + dx, this.ypos + dy);
  }
}

public class CPoint2 extends Point2 {
  private Color color;

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

  public static CPoint2 make (int x, int y, Color c) {
    return new CPoint2 (x,y,c);
  }

  public Color getColor () {
    return this.color;
  }

  public CPoint2 move (int dx, int dy) {
    return make (this.xpos + dx, this.ypos + dy, this.color);
  }
}
```