Inheritance (IS – A Relationship)

We've talked about the basic idea of inheritance before, but we haven't yet seen how to implement it.

Inheritance encapsulates the IS – A Relationship.
A String IS – A Object.
A Corvette IS – A Card.
A ThreeDimensionalPoint IS – A Point.
A Clarinet IS – A MusicalInstrument.

In this lecture we'll look at two examples of inheritance:

(1) A Coordinate and ColorCoordinate class
(2) An extension of the Fraction class, the MixedFraction class.

When defining an inherited class, we must explicitly state that we are *extending* another class as follows:

public class ColorCoordinate extends Coordinate { … }

In this situation, we refer to Coordinate as the base class. We can also refer to it as the superclass.

ColorCoordinate is known as the derived class, or subclass.

Beyond that, there are quite a few rules that we must discuss.

First, there are some changes that we must make to our original class if we had not created it with the intention of inheriting from it.

Let's take a look at the Coordinate class to see these changes.

Protected Visibility Modifier

In a typical class, we make our instance variables private. However, if we did this and we created a derived class that inherited from our original class, then we would NOT have access to the instance variables of the base class in our derived class.

This could prove to be problematic if we want access to these instance variables. (In some instances we won't need it, because the methods in the base class can adequately manipulate these variables.)

Instead, if we declare our instance variables to be *protected*, then we have access to them BOTH in the current class AND all inherited classes.

Here is the beginning of the Coordinate class:

```
public class Coordinate {

  protected int num;
  protected char c;

}
```

The rest of the Coordinate class looks like other examples of simple classes we've seen. The goal of this class is to manage a Coordinate object that is indexed by a number and a letter, much like a location in the game of Battleship.

Constructors in a subclass

We might think that a ColorCoordinate constructor might look like this:

```
public ColorCoordinate(int num, char c, String color) {
  this.num = num;
  this.c = c;
  this.color = color;
}
```

But, if you really think about it, this is redundant!

The reason this is redundant is that we ALREADY have a constructor in the Coordinate class that takes care of initializing both num and c.

The whole point of inheritance is to UTILIZE the code from the base class!!!

Thus, we have an explicit way of calling the constructor from a super class so that we can REUSE this code. So our constructor will ACTUALLY look like this:

```
public ColorCoordinate(int num, char c, String color) {
  super(num, c);
  this.color = color;
}
```

The super call (without any object before it), automatically makes a call to the appropriate constructor from the direct base class of ColorCoordinate, which is Coordinate. This call will properly assign num and c. When it finishes, all we have to do is assign color.