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.

Default Constructors in a subclass

If we make NO reference to super in a constructor of our subclass, then a call to the DEFAULT constructor of the base class is made anyway!!!

Thus, when we see the following code in the ColorCoordinate class:

```
public ColorCoordinate(String color) { this.color = color; }
```

What is REALLY executed by the computer is the following:

```
public ColorCoordinate(String color) {
  super();
  this.color = color;
}
```

So, our whole object gets initialized, with Random instance variables as is specified in the default constructor in the Coordinate class.

In summary, in all super class constructors, we do NOT need to initialize ALL instance variables explicitly.

Instead, we can reuse code from the base class constructors in two ways:

1) Explicitly calling super (which will invoke the constructor of your choice)

2) Omitting the call to super (which will invoke the default constructor of the base class anyway)

Other Instance Methods in a Subclass

When we design methods for a subclass, remember that we MUST look at the functionality already provided to us from the base class. In particular, there's no need to reinvent the wheel. We already have access to all of these methods, so there's NO need to redefine any of these methods if we want to use them exactly as they are.

Here are the types of design decisions we are free to make:

1) Keep methods from the base class and don't write a similar method in the subclass.

2) Redefine a method in the subclass because you want it working differently for an object of the subclass than an object of the base class.

3) Define a new method that is specific to the subclass that isn't defined in the base class at all.

In our example, we have examples of all three choices:

(1) getNum and getC exist in Coordinate only because they are ALSO adequate for a ColorCoordinate object.

(2) The toString() method is redefined for ColorCoordinates, because it works a bit differently for ColorCoordinates than it works for Coordinates.

(3) The getColor method only makes sense for the ColorCoordinate class. It makes no sense for the Coordinate class, so it's not included in that class at all.

Redefining Methods in a Subclass

To redefine methods in a subclass, you use the same method signature as the base class, but place the method in the subclass. From here, you are free to define the method as you see fit.

Even though you are redefining the method, you may find it useful to CALL the method of the base class of the same name. To do this, you must invoke the call using the super keyword. Unlike constructors where super is automatically invoked, in other methods it is NOT. You have to EXPLICITLY call super:

```
public String toString() {
        return super.toString() + " Color = " + color;
}
```

Technically speaking, the reason the equals method is NOT redefined in the ColorCoordinate class is because its method signature:

public boolean equals(ColorCoordinate sample);

IS different than the equals method in the Coordinate class:

public boolean equals(Coordinate sample);

Furthermore, note that you are not REQUIRED to make a call to super in a method in a subclass that is redefining a preexisting method in a base class.

Defining New Methods in a Subclass

If you want MORE functionality in your subclass, you have the right to define new methods in it that DON'T already exist at all in the base class. (For example, for the Primate class, you might NOT define a method readNewspaper(), but you WOULD define it for the Human class, that inherits from Primate.)

In our example, there are two newly defined methods:

public boolean equals(ColorCoordinate sample);

which was just discussed. This is newly defined because it takes in a ColorCoordinate object, something that is NOT done in the Coordinate class method.

The other example is the following:

```
public String getColor()
  return color;
}
```

This simply doesn't make sense for a regular Coordinate object!

Using an Object of a Subclass

Luckily, this part is easy. You use an object of a subclass exactly as you use any object. You declare a reference and then create an object by calling the constructor.

Then you can call methods on that object as desired.

What is tricky is polymorphism, which is what we'll look at in a future lecture. In particular, this deals with what method gets called when there are multiple methods whose signatures match the called method.

Secondly, it's important to focus on two details when determining what method gets called when there's ambiguity:

1) The type of each reference involved.
2) The type of each corresponding object involved.