COP 3330 – Object Oriented Programming in Java
Introduction to Inheritance (IS-A relationship)

Object-Oriented Languages separate themselves from non-object-oriented languages further by allowing for an entirely different type of code reuse beyond typical methods: inheritance.

The original word comes from the world of biology and taxonomy. Many news articles state that humans derive or inherited x% of their DNA from their primate predecessors. This process of inheritance is an accurate description of what occurs in an object-oriented programming language. Our human blueprint (class) has the recipe for arms, legs, a head, many internal organs, etc. The vast majority of these blueprints are derived from the blueprints of our ancestors. Instead of humans springing up from thin air with all of our complexity, we started with the complexity of our ancestors but then ADDED characteristics and specialized behaviors compared to our ancestors. (Our ancestors still ate with their mouths and could run, but we most definitely eat differently and run differently; our actions are modified versions of our ancestors' versions.) Other characteristics between us and our ancestors have changed very little. For example, the way in which humans see the color red and chimpanzees see the color red is identical. Both species inherited this particular trait from a common ancestor (https://www.amnh.org/exhibitions/permanent/human-origins/understanding-our-past/dna-comparing-humans-and-chimps).

Inheritance in Java encapsulates the IS-A relationship.

A String IS – A Object.
A Corvette IS – A Car.
A ThreeDimensionalPoint IS – A Point.
A Clarinet IS – A MusicalInstrument.

When defining a class, we can make it "inherit" from another class. In practice, when class B inherits from class A it means the following:

1. Any object of class B IS an object of class A.

2. By default, any method not defined in class B is inherited from class A.

3. If we want a method to work differently for an object of type class B than it does for class A, we can do the following: call the corresponding method for class A, then ADD our own code, OR completely redefine the behavior. (So if we want to define run for a human differently than for a primate, we could first execute the primate version of run and then ADD some steps to it, OR, we could completely abandon the default primate version of run and replace it our new version.

Java's Object class.

Before we dive into the real content of this lecture, let's talk about the Object class in Java.

It's an actual class that acts as a base class for all classes in Java. A base class is the class from which other classes inherit from. Every class one creates inherits from the Object class automatically.

Many of the methods in the Object class deal specifically with threads which we won't cover in this class. The methods in the Object class (not dealing with threads) that we will touch upon are:

```
// Creates and returns a copy of this object.
Object clone()

// Returns true if and only if this equals obj.
boolean equals(Object obj)

// Returns the runtime class of this Object.
Class<?> getClass()

// Returns the hash code value for this Object.
int hashCode()

// Returns a string representation of this Object.
String toString()
```

When we looked at our first class, GiftCard, a few lectures ago, we wrote two of these methods: toString() and equals(). If we hadn't written these methods, we would STILL have been able to call them on GiftCard objects because any GiftCard object IS-A Object object, and there's a definition for both of these methods in the Object class. To see how the Object class defines toString(), just comment out the toString() method in the GiftCard class and then try to print out a GiftCard object. (It prints out the name of the class, followed by the @ symbol, then a hexadecimal code.) Since this isn't what we want our code to do, we have to **override** the corresponding methods in the Object class. Thus, when we inherit from a class, we must do the following:

1. Decide which behaviors we want to keep the same as the class we are inheriting from.
2. Decide which behaviors we want to change/adapt/revise.

For #1, we do nothing, the code will naturally get reused. For #2, we'll define the method in our new class.

Before we start looking at our first actual examples of inheritance that we've fully defined, it's important to note that the class of a reference need not be identical to the type of the object it's referencing. A Car reference can point to a Corvette object. However, a Corvette reference is not allowed to point to a Car object. This is because all Corvettes are Cars but not all Cars are Corvettes.

Our Examples of Inheritance

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 is because private access means you can only access the instance variables within the class and not within any other 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.) But it seems to make sense that we ought to be able to access all instance variables that comprise our object.

Instead, if we declare our instance variables to be *protected*, then we have access to them BOTH in the current class AND all of its inherited classes. (Note that if class C can NOT inherit from more than one class directly. But what we mean here is a situation where class C inherits from class B and class B inherits from class A. In this case, class C should have access to all instance variables originally defined in classes A and B.)

Here is the beginning of the Coordinate class:

```
public class Coordinate {

    protected int num;
    protected char c;
    // rest of the class...
}
```

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.

For completeness, here is the code. Since the purpose of this lecture is to elucidate the mechanics of inheritance, some extra print statements will be inserted into methods that wouldn't normally have them. The purpose of these print statements is to prove which of the several methods with the same name are getting called whenever we execute them. Keep in mind that the Object class has an equals method defined as well.

```
public class Coordinate {

    protected int num; // row of chart, these are protected for use in
                       // ColorCoordinate class.
    protected char c; // column of chart
    protected static Random r = new Random();

    public Coordinate() {
        num = 1 + r.nextInt(10);
        c = (char)('a' + r.nextInt(10));
    }

    public Coordinate(int n, char ch) {
        num = n;
        c = ch;
    }

    public int getNum() {
        return num;
    }

    public char getC() {
        return c;
    }

    public boolean equals(Coordinate c2) {
        System.out.println("Method #1");
        return num == c2.num && c == c2.c;
    }

    public boolean equals(ColorCoordinate c2) {
        System.out.println("Method #2");
        return num == c2.num && c == c2.c;
    }

    public String toString() {
        return "Row = " + num + " Column = " + c;
    }

    public void printCoordinate() {
        System.out.print("Row = " + num + " Column = " + c);
    }
}
```

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. Our constructor will ACTUALLY look like this:

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

The super call (which is implicitly called on this), 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;
}
```

This means that if we were to write the Coordinate class without a default constructor and not explicitly call super() from our ColorCoordinate constructor, our code would not compile. But, if you look at the default constructor in Coordinate, it just sets the number and letter to randomly generated values within the valid range (1-10, 'a'-'j').

In summary, in all sub 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.

instanceof operator

Since a reference in Java can be of a direct or indirect base class of the object its referencing, it makes sense to have the ability to test at run time whether or not the actual object that a reference is pointing to is of a particular class.

In particular, if sample is a Coordinate reference, it's possible it could be pointing to a ColorCoordinate. The manner in which we can check this is via the instanceof operator. The syntax of its use is as follows:

```
referenceName instanceof className
```

This operator returns a boolean: true if the corresponding reference is currently referencing an object of the type className, and false otherwise. Here is the equals method in the ColorCoordinate in which it's used:

```
public boolean equals(Coordinate sample) {
    System.out.println("Method #3");
        if (sample instanceof ColorCoordinate)
            return (color.equals(((ColorCoordinate)sample).color) &&
                    super.equals(sample));
        else
            return false;
}
```

Basically, here we are saying that if a Coordinate is not a ColorCoordinate, it can not be equal to this ColorCoordinate object. If a Coordinate is a ColorCoordinate, then we make sure to compare all components for equality.

Tracing Through Some Tests

Now, let's take a look at a few tests of the Coordinate and ColorCoordinate classes to see which methods get called in which situations. All of the code shown below is in the main method of the ColorCoordinate class. (For ease we have not written a third class with our tests.)

We will generate four objects in our tests, three of them will be of class ColorCoordinate, one will be of class Coordinate:

```
Coordinate test = new ColorCoordinate(3,'a',"Green");
ColorCoordinate red = new ColorCoordinate(3,'a',"Red");
ColorCoordinate blue = new ColorCoordinate(3,'a',"Blue");
Coordinate nocolor = new Coordinate(3,'a');
```

Notice that the reference pointing to the first ColorCoordinate object (the green one) is a Coordinate reference. We can not have a ColorCoordinate reference pointing to a Coordinate object, because a Coordinate IS NOT A ColorCoordinate. (Mathematically speaking, the inheritance relationship is anti-symmetric.)

Let's print these out:

```
System.out.println("test is "+test);
System.out.println("red is "+red);
System.out.println("blue is "+blue);
System.out.println("nocolor is "+nocolor);
```

Here is the output:

```
test is Row = 3 Column = a Color = Green
red is Row = 3 Column = a Color = Red
blue is Row = 3 Column = a Color = Blue
nocolor is Row = 3 Column = a
```

Notice that at run-time Java is smart enough to realize that test is referencing a ColorCoordinate object. (Essentially, it runs the instanceof test.) This ability is called **polymorphism.** When a Java program is running, it dynamically uses the actual type of the objects, not the reference type, to determine which method to call. When looking for the appropriate method definition, Java first looks in the code for the class that the object actually IS. If the method is there, it calls it. If it is not, then it looks in the class that it directly inherits from. If it's not there, then it goes to the class that one inherits from, and so forth. So if a method isn't defined in Human, then we look in Primate. If it's not defined in Primate, we look in Mammal, if it's not in Mammal we look in Vertebrate, etc. In Java, the end of the line is the Object class. If we still haven't found the method, then the code won't compile.

Here is our first equals test:

```
if (nocolor.equals(red))
    System.out.println("These two are the same.");
```

This call runs Method #2, since red is a ColorCoordinate (this matches over the equal method that takes in a Coordinate). Here the method choice is made based on the type of the **reference** of the parameter, not the type of the object that reference is pointing to. (So, we get polymorphism for the object on which a method is called for binding the method call, but we use the reference type of a parameter to determine the binding of a method call, not the corresponding object type.) This method call will return true and the println occurs.

Now, let's flip it:

```
if (red.equals(nocolor))
    System.out.println("Equal - called Coordinate");
else
    System.out.println("Not Equal - called Color");
```

Since red is referencing an object of type ColorCoordinate, a method from ColorCoordinate will be called if it exists. Of the methods listed, only the one taking in a Coordinate matches based on the reference type of nocolor, thus, Method #3 is the one called and the output is "Not Equal – called Color"

Here is our next test.

```
if (red.equals(blue))
    System.out.println("Red and blue make purple.");
```

This one runs as expected. It's going to call a method from the ColorCoordinate class if it exists since red is referencing a ColorCoordinate object. Since blue is a ColorCoordinate reference, it will called Method #4 and not print anything out, since that method will return false.

Now, let's try calling the equals method on test:

```
if (test.equals(blue))
    System.out.println("Green and Blue are the same.");
else
    System.out.println("Green and Blue are different.");
```

Since the object that test is referencing is actually of type ColorCoordinate, if a ColorCoordinate equals method exists (which it does), it will get called. Of the two equals methods in the class, since blue is a ColorCoordinate reference, that is the method that will get called. Thus, Method #4 is called and "Green and Blue are different." gets printed out.

Here is our final test:

```
if (blue.equals(test))
    System.out.println("Blue matched test.");
else
    System.out.println("Blue didn't match test.");
```

Since blue is referencing a ColorCoordinate object, either Method #3 or Method #4 is called. Since the **reference type** of test is Coordinate, this means that Method #3 will be called. Since this method automatically rejects all Coordinate references, "Blue didn't match test." will be displayed.

mixedfraction class

Now, let's take a quick look at the mixedfraction class. It inherits from fraction. To enable this, we had to change fraction's instance variables to protected status. Then, we just needed one extra instance variable (for the whole number part) and three constructors:

```
public class mixedfraction extends fraction {

    private int whole;

    public mixedfraction(int w, int n, int d) {
        super(n%d, d);
        whole = n/d + w;
    }

    public mixedfraction(int n, int d) {
        super(n%d, d);
        whole = n/d;
    }

    public mixedfraction() {
        super(0,1);
        whole = 0;
    }

    // Rest of class
}
```

Notice our use of super isn't trivial since we're trying to force our fractional portion to be less than 1.

Now, let's look at our two add methods, one that takes in a fraction, the other a mixedfraction. This is similar in design to the two equals methods in the ColorCoordinate class:

```
public mixedfraction add(fraction other) {
    fraction tmp = super.add(other);
    return new mixedfraction(whole, tmp.numerator, tmp.denominator);
}

public mixedfraction add(mixedfraction other) {
    int wholeTotal = whole + other.whole;
    fraction tmp = super.add(other);
    return new mixedfraction(wholeTotal, tmp.numerator, tmp.denominator);
}
```

The key is to see how we leveraged both the use of super and the constructor which forces our fractional parts to be less than 1.

Finally, let's take a look at the toString method in the mixedfraction class:

```
public String toString() {
    return whole == 0 ? super.toString() : whole + " " + super.toString();
}
```

Essentially, we are trying to omit a whole part of 0. (Note: This code probably won't work nicely with negatives. It was written with non-negative fractions and mixedfractions in mind.)

Now, we can run a couple basic tests on fractions and mixedfractions:

```
fraction a = new fraction(2, 3);
mixedfraction b = new mixedfraction(14, 5);
mixedfraction c = new mixedfraction(2, 3, 5);

mixedfraction d = b.add(a);
System.out.println(b+" + "+a+" = "+d);

mixedfraction e = b.add(c);
System.out.println(b+" + "+c+" = "+e);

mixedfraction f = new mixedfraction(1, 3);
mixedfraction g = new mixedfraction(2, 5);
mixedfraction h = f.add(g);
System.out.println(f+" + "+g+" = "+h);
```

The corresponding output is what we'd expect:

```
2 4/5 + 2/3 = 3 7/15
2 4/5 + 2 3/5 = 5 2/5
1/3 + 2/5 = 11/15
```