CECS 277 – Lecture 3 – Polymorphism

Polymorphism - is the ability to take on multiple forms. In programming, it is the idea that a variable of the type of the superclass can hold a reference to an object that is of the subclass. Here is an example where Animal is the superclass and Dog extends Animal.

```
Animal a = new Dog( "Spot" );
```

The Dog is constructed and the reference to that object is stored in the variable of type Animal. This does not cause any compiler errors since Dog is an Animal.

Unfortunately though, by creating a subclass object and storing it in a superclass variable, you no longer have access to the methods that are specific to your subclass. However, you do have access to any methods that were overridden from the superclass itself.

You may wonder why anyone would ever do this if you lose functionality. It turns out that polymorphism can be very handy in cases where you don't necessarily care about the functionality of the subclass. Examples: An array of People, where some of the objects are Employees, some are Customers, but all you want to do is print out their names. Or passing an Animal object to a method when you don't care whether the animal is a Cat, a Dog, or a Bird.

There are two remedies for when you do need the functionality of the subclass:

Downcasting – Temporarily typecasting the object into a variable of the subclass. That way you have access to the methods that are only available to the subclass (This method is usually frowned upon for breaking encapsulation and occurs because of bad object oriented design).

```
Dog d = (Dog) a;
d.bark();
```

Dynamic Binding – Create the needed methods in the superclass, and then override those methods in any subclasses that require them. The method call will always use the subclass' version of the method. Dynamic Binding should be used whenever possible (ie. when it makes sense), otherwise use downcasting.

```
a.speak();
```

instanceof – There may be times when you do not know what type of object is stored in a variable. Luckily, there is the keyword instanceof that can test the class type of an object.

Here, we do not know if the Animal being passed in is a Cat, a Dog, or a Bird:

```
public static void printName( Animal a ) {
```

In these cases we can ask the object itself what class it was made from.

```
if( a instanceof Dog ) { ...
```

- or -

```
if( Dog.class.isInstance(a) ) { ...
```

Which means that you might need to check each different subclass type to find out which one it is. Be careful not to test if it is of the type of the superclass, because if the subclass was inherited from the superclass, then it will obviously return true.

Abstract – refers to an item that exists as an idea but does not physically exist.

Abstract Methods – An abstract method is defined, but not created (ie. does not have a body). All subclasses created from this class must override this method. If a class has an abstract method in it, then the class itself must be abstract as well.

```
public abstract void speak();
```

Abstract Classes – An abstract class can be fully defined and then extended, but objects cannot be created from it. Classes that are created as abstract are usually used as a generic superclass from which subclasses are made.

```
public abstract class Animal {
      ...
}
```

Final – We have seen the keyword final used to create constants (variables that cannot be modified), but final can also be applied to classes and methods.

Final Methods – A final method cannot be overridden.

```
public final void sit( ) {
      ...
}
```

Final Classes – A final class cannot be inherited. If a class is declared as final, all of its methods are implicitly final as well.

```
public final class Chihuahua extends Dog {
      ...
}
```

Example – Suppose we need to implement a Mug class where we can add any type of liquid to it. On a first pass we might try something like the following:

```
public class Mug {
      private Coffee coffee;
      public void addCoffee( Coffee c ) {
            coffee = c;
            coffee.stirCoffee();
      }
}

public class Coffee {
      public void stirCoffee() {
            System.out.println("Stirring coffee...");
      }
}

public class MugTest {
      public static void main( String[] args ) {
            Mug mug = new Mug();
            Coffee coffee = new Coffee();
            mug.addCoffee( coffee );
      }
}
```

Notice how restrictive the previous code is. The Mug can only handle adding Coffee to it. If we wanted to try adding Tea to the mug, we would have to change the data member and function so that it would instead use Tea. The same would apply for every new liquid we might ever need to add (ex. Milk, Cocoa, etc.). This would force us to alter the Mug class every single time, which can lead to errors.

If we think about the mug in an object-oriented way, we would realize that the mug doesn't care what type of liquid we add to it. We could create an abstract class Liquid that could then be extended by any number of classes that were liquids, each having their own functionality for the stir method. This allows us to never have to touch the code of the Mug class again, but makes it infinitely expandable to any type of liquid.

```
/**
 * Representation of a generic type of liquid.
 */
public abstract class Liquid {
	/** Stirs the liquid. */
	public abstract void stir();
}
```

```
/**
 * Representation of some Coffee, which is a liquid.
 */
public class Coffee extends Liquid {
	/** Stirs the coffee. */
	@Override
	public void stir() {
		System.out.println("Stirring in coffee...");
	}
}
```

```
/**
 * Representation of some Milk, which is a liquid.
 */
public class Milk extends Liquid {
	/** Stirs the milk. */
	@Override
	public void stir() {
		System.out.println("Stirring in milk...");
	}
}
```

```
/**
 * Representation of a Mug
 */
public class Mug {

      /** The liquids the mug is holding */
      private ArrayList<Liquid> liquids = new ArrayList<Liquid>();

      /**
       * Adds a Liquid to the mug and then stirs it.
       * @param l the liquid to add
       */
      public void addLiquid( Liquid l ) {
            liquids.add( l );
            l.stir();
      }
}



/**
 * Test implementation for the Mug class
*/
public class MugTest {
      public static void main( String[] args ) {
            Mug mug = new Mug();

            Coffee coffee = new Coffee();
            Milk milk = new Milk();

            mug.addLiquid( coffee );
            mug.addLiquid( milk );
      }
}
/* Output:
Stirring in coffee...
Stirring in milk...
*/
```