

Inheritance and Polymorphism

(CISS-111 William Jojo)

Inheritance

Recall from our discussion on creating a GUI that we chose to use the phrase `extends JFrame` for the class containing our program to obtain the ability to display a window as well as to place text fields and buttons within the content pane. The discussion then proceeded about the benefits of extending the `JFrame` class in such a way that our program was part of that class, an extension of that class.

Of course we did not add anything new to the existing `JFrame` class, or what we called the *superclass* (also known as the *base class*). Our program became the *subclass* (or *derived class*) by way of *inheritance*. The subclass inherits all of the properties of the superclass and as a result we can create instance variables of both the superclass and the subclass types in the same object which also reduces the complexity of programming since any change to the superclass design is automatically inherited by the subclass the next time we compile our program.

The details of inheritance are fairly simple. There exists *single inheritance* where a subclass inherits the properties of a single superclass. There also exists *multiple inheritance* where a subclass inherits the properties of more than one superclass. The single and multiple inheritance principles are created to describe how a subclass has come into being, but *Java only supports single inheritance*. In other words, we can only extend one class at a time. With inheritance there are some strict rules which are:

- The `private` members of the superclass cannot be accessed by the subclass.
- The `public` member of the superclass are directly accessible to the subclass.
- The subclass can contain additional data or method members.
- All data members of the superclass are also data members of the subclass (because of *inheritance*).
- The subclass can overload public methods of the superclass.
- The subclass can *redefine* or *override* public methods of the superclass.

We will use the code from *Example 1*, *Example 2* to demonstrate our itemized list of limitations and capabilities.

```

public class MyClass
{
    private int mcInt;
    private String mcString;

    public MyClass()
    {
        mcInt = 0;
        mcString = "NOTSET";
    }

    public MyClass(int i, String s)
    {
        mcInt = i;
        mcString = s;
    }

    public int getInt()
    {
        return mcInt;
    }

    public String getString()
    {
        return mcString;
    }

    public void setInt(int i)
    {
        mcInt = i;
    }

    public void setString(String s)
    {
        mcString = s;
    }

    public String toString()
    {
        return "mcInt = " + mcInt + " mcString = " + mcString;
    }
}

```

Example 1: Source code for MyClass.

The code for `MyClass` in *Example 1* has two private instance variables `mcInt` and `mcString`. There are two constructors, two mutator methods and two accessor methods. Included is a `toString()` method for details of the instance variable values. This class is very basic and conforms to all of the user-defined class components we have seen previously. In fact there is nothing new in this code whatsoever that requires any further discussion on the concepts of user-defined methods.

```
public class MySubClass extends MyClass
{
    private double mscDouble;
    private char mscChar;

    public MySubClass()
    {
        super();
        mscDouble = 0.0;
        mscChar = '\0';
    }

    public MySubClass(double d, char c, int i, String s)
    {
        super(i, s);
        mscDouble = d;
        mscChar = c;
    }

    public double getDouble()
    {
        return mscDouble;
    }

    public char getChar()
    {
        return mscChar;
    }
}
```

Example 2: Source code for MySubClass which extends MyClass.

```

    public void setDouble(double d)
    {
        mscDouble = d;
    }

    public void setChar(char c)
    {
        mscChar = c;
    }

    public void setString(String s)
    {
        super.setString(s.toUpperCase());
    }

    public String toString()
    {
        return super.toString() + " mscDouble = " + mscDouble +
            " mscChar = " + mscChar;
    }
}

```

Example 2: MySubClass continued.

In *Example 2*, `MySubClass` is defined as a class that extends `MyClass`. Clearly we are indicating that `MySubClass` is to inherit all of the properties of `MyClass`. In addition to this inheritance, we are defining two additional private instance variables `mscDouble` and `mscChar`.

Similar to `MyClass`, `MySubClass` also has two constructors, two accessor and two mutator methods. The methods `toString()` and `setString()` have been redefined – not overloaded. Recall that overloading is the act of creating a method of the same name, but with a *different* parameter list.

Our subclass constructors are invoking the superclass constructors by calling `super()` with an appropriate number of parameters (shown in bold). Whenever we call the superclass constructors they *must* be the first line of the subclass constructor body and must use the `super` reserved word.

We can also use the `super` reserved word to access methods within the superclass. This is especially necessary when our subclass has methods that override ones of the superclass. These forms are shown in the `setString()` and `toString()` methods.

It is important to note that we are indeed calling the mutator methods of the superclass from within our subclass and not attempting to modify the instance variables of the superclass. This direct manipulation would not be possible since the instance variables of the superclass are `private` and therefore cannot be accessed from the subclass.

If, however, we did want a subclass to have direct access to an instance variable of the superclass while still disallowing direct access outside the class, we could use a different modifier called `protected`. A `protected` instance variable of the superclass can be directly accessed from a subclass and still require all other access outside the class to be done using the accessor and mutator methods. So if we wanted the `mcInt` instance variable of `MyClass` to be accessed from `MySubClass`, we could simply modify its declaration as follows:

```
protected int mcInt;
```

No matter how many *generations* of class are derived from a superclass, all of the `public` and `protected` members of the superclass will be continue to be accessible to all the subsequent generations. This is also true for any intermediate generation forward in the inheritance hierarchy.

One must also be careful that all subclasses make appropriate calls to the `super()` constructors and that the superclass initializes the instance variables correctly.

Now let us take a look at our exerciser program in *Example 3*. We have declared reference variable `mcVar` of class type `MyClass` and reference variable `mScVar` of class type `MySubClass`. There is an additional reference variable `mcRefVar` that we will talk about in just a bit.

First let us look at the very commonplace appearance of `mcVar` and `mScVar`. They are populated with objects from their respective constructors and then made part of a call to `println()`. This of course implies a call to the `toString()` methods for each reference variable respective of their class. Recall that `MySubClass` redefined `toString()` to include the additional instance variables after a call to the superclass version of `toString()`.

```

public class mytester
{
    public static void main(String[] args)
    {
        // new MyClass variable
        MyClass mcVar = new MyClass(56, "MyClass string");
        // new MySubClass variable
        MySubClass mscVar = new MySubClass();
        // MyClass ref variable.
        MyClass mcRefVar;

        // Display contents of mcVar. implicit call to toString()
        System.out.println(mcVar);
        // Display contents of mscVar. implicit call to toString()
        System.out.println(mscVar);

        mscVar.setString("MySubClass string");
        mscVar.setDouble(3.1415926);
        mscVar.setChar('K');

        mcRefVar = mcVar;
        if ( mcRefVar instanceof MyClass )
            System.out.println("\nmcRefVar refers to a MyClass object.");
        System.out.println(mcRefVar);

        // Polymorphic reference invokes toString() of MySubClass
        // due to late binding.
        mcRefVar = mscVar;
        if ( mcRefVar instanceof MySubClass )
            System.out.println("\nmcRefVar refers to a MySubClass
object.");
        System.out.println(mcRefVar);
    }
}

```

Example 3: Program to test MyClass and MySubClass objects.

We then call all of the subclass mutator methods to alter the values of the instance variables for `mscVar`. Now for something very interesting. `mcRefVar` is assigned the value of `mcVar`, then we print `mcRefVar` which calls the `toString()` method of `MyClass` which is exactly as we would expect to happen.

However, we then assign `mScVar` to `mcRefVar`. This is not an error, on the contrary, any superclass reference variable may reference an object of any of its subclasses. What is really interesting is when we print `mcRefVar` the second time, the `toString()` method of `MySubClass` is the method that is invoked!

This requires a bit of explaining. First let us look at the output of the `mytester.java` program shown here:

```
mcInt = 56 mcString = MyClass string
mcInt = 0 mcString = NOTSET mscDouble = 0.0 mscChar =

mcRefVar refers to a MyClass object.
mcInt = 56 mcString = MyClass string

mcRefVar refers to a MySubClass object.
mcInt = 0 mcString = MYSUBCLASS STRING mscDouble = 3.1415926 mscChar = K
```

The Java runtime environment knows that `mcRefVar` is pointing to an object of `MySubClass` and not to an object of `MyClass`. As a result we call the `toString()` method of the referred object, not the method of the reference variable class type. This particular phenomenon is called *late binding* (or *dynamic binding* or sometimes *run-time binding*).

Essentially, the decision of which method to call is determined at run-time and not at compile time. We are essentially assigning multiple meanings to the `toString()` method and the meaning of the call to `toString()` is based on the object and not the reference variable. The ability for us to assign multiple meanings to a method is called *polymorphism*. Polymorphism is implemented by using late binding.

The reference variable `mcRefVar` can point to any object of class type `MyClass` or the class `MySubClass`. Because `mcRefVar` can have many forms, we say that `mcRefVar` is a *polymorphic reference variable*.

In fact, look closely at the conditional tests that make use of another reserved word called `instanceof`. This reserved word is an operator and we can use this operator at run-time to determine to what a reference variable actually refers. In this way, we can be certain that actions taken are on an appropriate object.

If polymorphism is something you wish to stop you can prohibit the overriding of a method by declaring the method of the superclass as `final`. You can also stop the ability to create a

subclass of a given class by declaring the class as `final`. There are some exceptions to late binding as well; Java does *not* use late binding on `static`, `final` or `private` methods.

The class `Object`

Recall that every class has a default `toString()` method even if you do not provide one. Also recall that the default value of a class printed with the `toString()` method provided is the class name and a hash value (not an address).

So where did this default `toString()` method come from? It comes from the standard class `Object`. As it turns out this class is defined in the package `java.lang`, which is the package that every Java program imports even when you have no import statements. In addition, when you define a class that has no `extends` keyword, your class automatically becomes a subclass of the class `Object`. One way or another, your class will be a subclass of the ultimate superclass `Object`. *Table 1* shows some members of the class `Object`.

Useful members of the class <code>Object</code>
public <code>Object()</code> Object constructor.
protected <code>Object clone()</code> Returns a object that is a copy of this object.
public <code>boolean equals(Object obj)</code> Returns <code>true</code> if <code>obj</code> and this object refer to the same memory space.
protected <code>void finalize()</code> This method is called when an object goes out of scope.
public <code>int hashCode()</code> Returns the integer representation of the object. Especially useful for hash tables supported by <code>java.util.Hashtable</code> .
public <code>String toString()</code> Returns a string representation of the object.

Since every class is a subclass of `Object`, we already know that all of the methods of `Object` are available to our classes when we define them. We now have a greater understanding of object hierarchies and should endeavor to be aware of this henceforth.

Abstract Methods and Classes

A method that has only a heading and no body is known as an *abstract method*. The heading ends in semicolon and includes the reserved word `abstract`. An *abstract class* is also declared with the reserved word `abstract`. Below are some rules about abstract classes:

- An abstract class may contain abstract methods.
- An abstract class may contain non-abstract methods, constructors, finalizers and instance variables.
- Any class that contains abstract methods must itself be declared abstract.
- Abstract class object cannot be instantiated, although you can have a reference variable of an abstract class type.
- You may instantiate an object of a subclass of an abstract class provided the subclass defined all of the abstract methods of the superclass.

The primary purpose of abstract classes is to force the definition of subclasses as direct descendants of the superclass as well as to force these subclasses to support the abstract methods the superclass declares.

Interfaces

Recall that the `ActionListener` class is a special form of class called an *interface*. We will discover later that there are other interesting interfaces defined within Java. Recall, too, that Java only supports single inheritance. Therefore we can only create classes that are derivatives of *one* superclass.

This is precisely the reason we have *interfaces*. Java will allow use to define a class that can use more than one *interface*. Our GUI programs implemented an event handling mechanism using what is known as inner classes. We will discover later that we can also use an anonymous class or have the class that contains the application implement the interface.

Recall that interfaces are attached to your program by using the `implements` reserved word. Instead of using the inner classes in our GUI examples we could have built our program like the following:

```
public class MyGUIProgram extends JFrame implements
                                   ActionListener
{
    // rest of programming
}
```

Essentially an interface is a class that declares only abstract methods. They may also contain named constants. Recall the definition of `ActionListener` shown here:

```
public interface ActionListener
{
    public void actionPerformed(ActionEvent e);
}
```

If a class implements an `interface`, it must provide definitions for all of the methods of the interface or you will not be able to instantiate an object of that class.

Inheritance Versus Composition

We are basically trying to tell the difference between an *is-a* relationship and a *has-a* relationship. With inheritance we are saying that the subclass *is-a* superclass, or using our examples from earlier, `MySubClass` *is-a* `MyClass` since it is a derivative class.

What comes about later is we define classes that have other classes as members within the class. For example in `MyClass` there is a `String` instance variable. So we can say that `MyClass` *has-a* `String`. Just as *every person has a name*. If our class examples were less generic and `MyClass` represented a `Person`, we could rename the `mcString` instance variable to `name`. For example:

```
public class Person
{
    String name;
    // rest of class definition
}
```

In this way we show *composition* or the relationship of two classes. `String` is a class defined inside of `Person`.