Polymorphism

Source	http://www.developer.com/tech/article.php/966001

What is polymorphism?

The meaning of the word polymorphism is something like *one name, many forms*.

How does Java implement polymorphism?

Polymorphism manifests itself in Java in the form of multiple methods having the same name.

In some cases, multiple methods have the same name, but different formal argument lists *(overloaded methods)*.

In other cases, multiple methods have the same name, same return type, and same formal argument list *(overridden methods)*.

Three distinct forms of polymorphism

From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

Method overloading

I will begin the discussion of polymorphism with method overloading, which is the simpler of the three. I will cover method overloading in this lesson and will cover polymorphism based on overridden methods and interfaces in subsequent lessons.

Method overloading versus method overriding

Don't confuse method *overloading* with method *overriding*.

Java allows you to have two or more method definitions in the same scope with the same name, provided that they have different formal argument lists.

More specifically, here is what Roberts, Heller, and Ernest have to say about overloading methods in their excellent book entitled The Complete Java 2 Certification Study Guide:

"A valid overload differs in the number or type of its arguments. Differences in argument names are not significant. A different return type is permitted, but is not sufficient by itself to distinguish an overloading method."

Similarly, here is what they have to say about method overriding:
"A valid override has identical argument types and order, identical return type, and is not less accessible than the original method. The overriding method must not throw any checked exceptions that were not declared for the original method."
You should read these two descriptions carefully and make certain that you recognize the differences.

Compile-time polymorphism

Some authors refer to method overloading as a form of *compile-time polymorphism,* as distinguished from *run-time polymorphism.*

This distinction comes from the fact that, for each method invocation, the compiler determines which method *(from a group of overloaded methods)* will be executed, and this decision is made when the program is compiled. *(In contrast, I will tell you later that the determination of which overridden method to execute isn't made until runtime.)*

Selection based on the argument list

In practice, the compiler simply examines the types, number, and order of the parameters being passed in a method invocation, and selects the overloaded method having a matching formal argument list.

A sample program

I will discuss a sample program named **Poly01** to illustrate method overloading. A complete listing of the program can be viewed in Listing 4 near the end of the lesson.

Within the class and the hierarchy

Method overloading can occur both within a class definition, and vertically within the class inheritance hierarchy. *(In other words, an overloaded method can be inherited into a class that defines other overloaded versions of the method.)* The program named **Poly01** illustrates both aspects of method overloading.

Class B extends class A, which extends Object

Upon examination of the program, you will see that the class named **A** extends the class named **Object**. You will also see that the class named **B** extends the class named **A**.

The class named **Poly01** is a driver class whose **main** method exercises the methods defined in the classes named **A** and **B**.

Once again, this program is not intended to correspond to any particular real-world scenario. Rather, it is a very simple program designed specifically to illustrate method overloading.

Will discuss in fragments

As is my usual approach, I will discuss this program in fragments.

The code in Listing 1 defines the class named **A**, which explicitly extends **Object**.

```
class A extends Object{
  public void m(){
    System.out.println("m()");
  }//end method m()
}//end class A

Listing 1
```

Redundant code

Recall that explicitly extending **Object** is not required *(but it also doesn't hurt anything)*.

By default, the class named **A** would extend the class named **Object** automatically, unless the class named **A** explicitly extends some other class.

The method named m()

The code in Listing 1 defines a method named **m()**. Note that this version of the method has an empty argument list *(it doesn't receive any parameters when it is executed)*. The behavior of the method is simply to display a message indicating that it has been invoked.

The class named B

Listing 2 contains the definition for the class named **B**. The class named **B** extends the class named **A**, and inherits the method named **m** defined in the class named **A**.

```
class B extends A{
  public void m(int x){
    System.out.println("m(int x)");
  }//end method m(int x)
  //-------------------------------//

  public void m(String y){
    System.out.println("m(String y)");
  }//end method m(String y)
}//end class B

Listing 2
```

Overloaded methods

In addition to the inherited method named **m**, the class named **B** defines two overloaded versions of the method named **m**:

- m(int x)
- m(String y)

(Note that each of these two versions of the method receives a single parameter, and the type of the parameter is different in each case.)

As with the version of the method having the same name defined in the class named **A**, the behavior of each of these two methods is simply to display a message indicating that it has been invoked.

The driver class

Listing 3 contains the definition of the driver class named **Poly01**.

```
public class Poly01{
  public static void main(
                        String[] args){
    B var = new B();
    var.m();
    var.m(3);
    var.m("String");
  }//end main
}//end class Poly01
```

Listing 3

Invoke all three overloaded methods

The code in the **main** method

- Instantiates a new object of the class named **B**
- Successively invokes each of the three overloaded versions of the method named **m** on the reference to that object.

One version is inherited

Note that the overloaded version of the method named **m**, defined in the class named **A**, is inherited into the class named **B**. Hence, it can be invoked on a reference to an object instantiated from the class named **B**.

Two versions defined in class B

The other two versions of the method named **m** are defined in the class named **B**. Thus, they also can be invoked on a reference to an object instantiated from the class named **B**.

The output

As you would expect, the output produced by sending messages to the object asking it to execute each of the three overloaded versions of the method named **m** is:

```
m()
m(int x)
m(String y)
```

Note that the values of the parameters passed to the methods do not appear in the output. Rather, the parameters are used solely to make it possible for the compiler to select the correct version of the overloaded method to execute in each case.

This output confirms that each overloaded version of the method is properly selected for execution based on the matching of method parameters to the formal argument list of each method.

~~~ End of Article ~~~