Week – 09

POLYMORPHISM

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real-life Illustration: Polymorphism

A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behaviour in different situations. This is called polymorphism.

The word "*poly*" means *many* and "*morphs*" means *forms*

TYPES OF POLYMORPHISM

In Java polymorphism is mainly divided into two types:

1. Compile-time Polymorphism
2. Runtime Polymorphism

Type 1: Compile-time polymorphism
It is also known as static polymorphism. This type of polymorphism is achieved by function overloading.

Type 2: Runtime polymorphism
It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. **Method overriding**, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

METHOD OVERLOADING

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.

> ***If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.***

Method overloading is one of the ways that Java supports polymorphism.

As you can see, method( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes

one double parameter. The fact that the fourth version of method( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

Example:

```
// Demonstrate method overloading.
class OverloadDemo
{
        void method()  // Overload method with no arguments
         {
        System.out.println("No parameters");
        }
        void method(int a)   // Overload method for one integer parameter.
        {
        System.out.println("a: " + a);
        }
        void method(int a, int b)  // Overload method for two integer parameters.
        {
        System.out.println("a and b: " + a + " " + b);
        }
        double method(double a) // overload method for a double parameter

        {
        System.out.println("double a: " + a);
        return a*a;
        }
}
class OverloadExample
{
        public static void main(String args[]) {
                OverloadDemo ob = new OverloadDemo();
                double result;
                // call all versions of method()
                ob.method();
                ob.method(10);
                ob.method(10, 20);
                result = ob.method(123.25);
                System.out.println("Result of ob.method(123.25): " + result);
        }
}
```

OVERRIDING METHODS

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

Example:

```
// Method overriding.
class A
{
        int i, j;
        A(int a, int b) { i = a; j = b; }
        void show() { System.out.println("i and j: " + i + " " + j); }
}
class B extends A
{
        int k;
        B(int a, int b, int c)
        {
        super(a, b);
        k = c;
        }
        // display k – this overrides show() in A
        void show()
        {
        //super.show();   this calls A's show()
        System.out.println("k: " + k);
        }
}
class Override
{
        public static void main(String args[])
        {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
        }
}
```

The Program above illustrates the method overriding. When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A** .

If you wish to access the superclass version of an overridden function, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

Here, super.show( ) calls the superclass version of show( ).
Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.
- Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time.

- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Example:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class DynamicMethodDispatch {
public static void main(String args[]) {
A = new A(); // object of type A
B = new B(); // object of type B
C = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

- Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.