**Polymorphism in Java**

The word ‘polymorphism’ literally means ‘a state of having many shapes’ or ‘the capacity to take on different forms’. When applied to object oriented programming languages like Java, it describes a language’s ability to process objects of various types and classes through a single, uniform interface.

Polymorphism in Java has two types: Compile time polymorphism (static binding) and Runtime polymorphism (dynamic binding). Method overloading is an example of static polymorphism, while method overriding is an example of dynamic polymorphism.

An important example of polymorphism is how a parent class refers to a child class object.  In fact, any object that satisfies more than one IS-A relationship is polymorphic in nature.

For instance, let’s consider a class Animal and let Cat be a subclass of Animal. So, any cat **IS** animal. Here, Cat satisfies the IS-A relationship for its own type as well as its super class Animal.

**Note:** It’s also legal to say every object in Java is polymorphic in nature, as each one passes an IS-A test for itself and also for Object class.

## Static Polymorphism:

In Java, static polymorphism is achieved through method overloading. Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.

At compile time, Java knows which method to invoke by checking the method signatures.  So, this is called **compile time polymorphism** or **static binding**. The concept will be clear from the following example:

class DemoOverload{

public int add(int x, int y){  //method 1

return x+y;

}

public int add(int x, int y, int z){ //method 2

return x+y+z;

}

public int add(double x, int y){ //method 3

return (int)x+y;

}

public int add(int x, double y){ //method 4

return x+(int)y;

}

}

class Test{

public static void main(String[] args){

DemoOverload demo=new DemoOverload();

System.out.println(demo.add(2,3));      //method 1 called

System.out.println(demo.add(2,3,4));    //method 2 called

System.out.println(demo.add(2,3.4));    //method 4 called

System.out.println(demo.add(2.5,3));    //method 3 called

}

}

In the above example, there are four versions of add methods. The first method takes two parameters while the second one takes three. For the third and fourth methods there is a change of order of parameters.  The compiler looks at the method signature and decides which method to invoke for a particular method call at compile time.

## Dynamic Polymorphism:

Suppose a sub class overrides a particular method of the super class. Let’s say, in the program we create an object of the subclass and assign it to the super class reference. Now, if we call the overridden method on the super class reference then the sub class version of the method will be called.

Have a look at the following example.

class Vehicle{

public void move(){

System.out.println(“Vehicles can move!!”);

}

}

class MotorBike extends Vehicle{

public void move(){

System.out.println(“MotorBike can move and accelerate too!!”);

}

}

class Test{

public static void main(String[] args){

Vehicle vh=new MotorBike();

vh.move();    // prints MotorBike can move and accelerate too!!

vh=new Vehicle();

vh.move();    // prints Vehicles can move!!

}

}

It should be noted that in the first call to move(), the reference type is Vehicle and the object being referenced is MotorBike. So, when a call to move() is made, Java waits until runtime to determine which object is actually being pointed to by the reference.  In this case, the object is of the class MotorBike. So, the move() method of MotorBike class will be called. In the second call to move(), the object is of the class Vehicle. So, the move() method of Vehicle will be called.

As the method to call is determined at runtime, this is called **dynamic binding** or **late binding**.

## Summary:

An object in Java that passes more than one IS-A tests is polymorphic in nature

Every object in Java passes a minimum of two IS-A tests: one for itself and one for Object class

Static polymorphism in Java is achieved by method overloading

Dynamic polymorphism in Java is achieved by method overriding