Week-9: Polymorphism

Polymorphism in Java

- 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

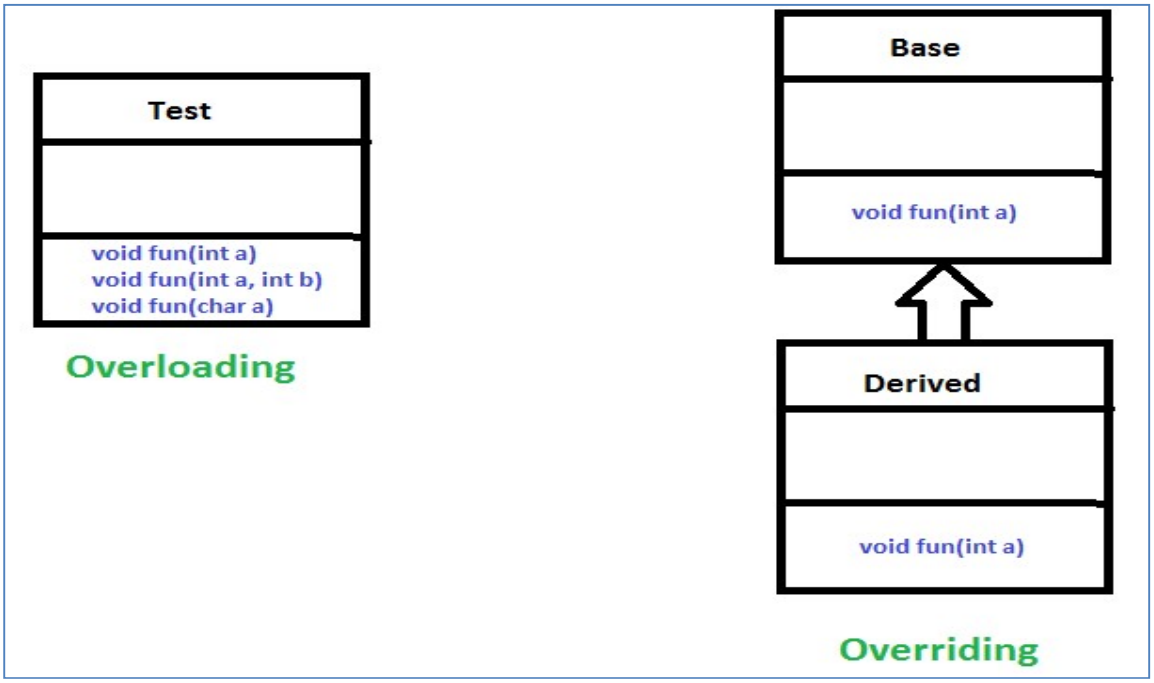
- A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.
- Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows us to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.
- To know whether an object is polymorphic, we can perform a simple test. If the object successfully passes multiple **is-a** or **instanceof** tests, it's polymorphic. In inheritance, all Java classes extend the class *Object*. Due to this, all objects in Java are polymorphic because they pass at least two **instanceof** checks.

Types of polymorphism

- In Java polymorphism is mainly divided into two types:
  - Compile-time Polymorphism
  - Runtime Polymorphism

Compile-time polymorphism

- It is also known as static polymorphism. This type of polymorphism is achieved by method overloading or operator overloading.
- ***Note:*** *But Java doesn't support the Operator Overloading.*
- **Method Overloading:** When there are multiple methods with the same name but different parameters then these methods are said to be **overloaded**. Methods can be overloaded by change in the number of arguments or/and a change in the type of arguments.
  - The parameter sets have to differ in at least one of the following three criteria:
  - They need to have a different number of parameters, one method accepting 2 and another one accepting 3 parameters
  - The types of the parameters need to be different, one method accepting a String and another one accepting a Long
  - They need to expect the parameters in a different order. For example, one method accepts a String and a Long and another one accepts a Long and a String. This kind of overloading is not recommended because it makes the API difficult to understand



Example 1

```
// Java Program for Method overloading by using Different Types of Arguments
class Helper
{
  static int Multiply(int a, int b)   // Method with 2 integer parameters
    {
       return a * b;      // Returns product of integer numbers


    }
static double Multiply(double a, double b)  // Method 2 with same name but with 2 double parameters
  {
      return a * b;    // Returns product of double numbers
  }
}
class Overloading
{
  public static void main(String[] args)
   {
     // Calling method by passing  input as in arguments
     System.out.println(Helper.Multiply(2, 4));
     System.out.println(Helper.Multiply(5.5, 6.3));
}
}
```

Output: 8
34.65

Runtime polymorphism

- It is also known as Dynamic Method Dispatch. It is a process in which a method 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 methods of the base class. That base method is said to be **overridden**.
- That is, Within an inheritance hierarchy, a subclass can override a method of its superclass, enabling the developer of the subclass to customize or completely replace the behavior of that method.
- Doing so also creates a form of polymorphism. Both methods implemented by the super- and subclasses share the same name and parameters. However, they provide different methodality.

Example:-

```
class Override
{
   int x;
   Override(int x)
   {
     this.x=x;
   }
  void display()
   {
     System.out.println("Super x=" +x);
    }
}
class Sub extends Override
{
  int y;
  Sub (int x, int y)
  {
    super (x);
    this.y=y;
   }
   void display()
   {
     System.out.println("Super x="+x);
     System.out.println("Super y="+y);
    }
}
class OverrideTest
{
     public static void main(String args[])
        {
             Sub s1 = new Sub(100,200);
```

```
            s1.displya();
        }
    }
```

Output

Super x = 100
Sub y = 200

- The **display()** method defined in the **Sub** class is invoked & executed instead of the one in the **Super** class.

Static Binding And Dynamic Binding :

Understanding Type

Let's understand the type of instance. Each variable has a type, it may be primitive and non-primitive.

int data=30;

Here data variable is a type of int.

References have a type

```
class Dog
 {
    public static void main(String args[])
      {
         Dog d1;//Here d1 is a type of Dog
    }
 }
```

Objects have a type An object is an instance of particular java class,but it is also an instance of its superclass.

```
 class Animal{ }
class Dog extends Animal
 {
  public static void main(String args[])
   {
      Dog d1=new Dog();
   }
 }
```

Static binding :

- When type of the object is determined at compiled time(by the compiler), it is known as static binding. If there is any private, final or static method in a class, there is static binding.

```
class Dog
{
    private void eat()
     {
        System.out.println("dog is eating...");
```

```
    }
  public static void main(String args[])
    {
      Dog d1=new Dog();
     d1.eat();
    }
  }
```

Dynamic binding :

- When type of the object is determined at run-time, it is known as dynamic binding.

```
class Animal
{
    void eat()
      {
        System.out.println("animal is eating...");
      }
}
class Dog extends Animal
  {
    void eat()
      {
         System.out.println("dog is eating...");
       }
    public static void main(String args[])
      {
        Animal a=new Dog();
        a.eat();
      }
 }
```