CA 105(A)
Java Programming (Core Java)

Unit-III Inheritance, Polymorphism and interfaces:

Polymorphism

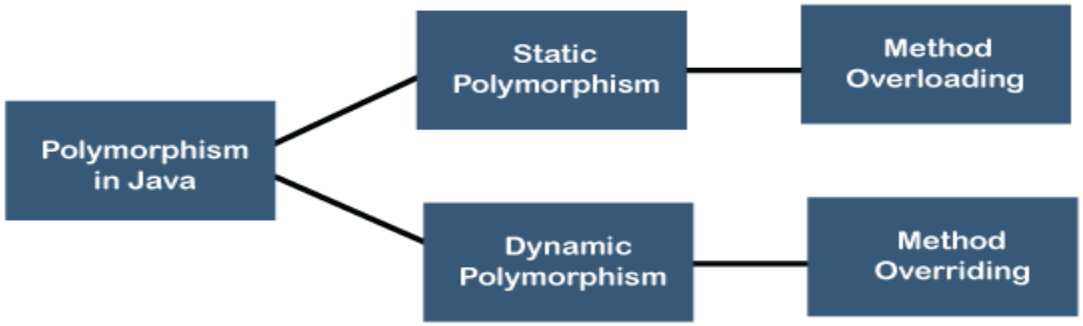
- The word polymorphism is a combination of two words i.e. **ploy** and **morphs**.
- The word poly means **many** and morphs means **different forms**.
- In short, a mechanism by which we can perform a single action in different ways.
- Let's understand the meaning of polymorphism with a real-world example.

A person in a shop is a customer, in an office, he is an employee, in the home he is husband/ father/son, in a party he is guest.

So, the same person possesses different roles in different places. It is called polymorphism.

- **Types of Polymorphism**
- There are **two** types of polymorphism in Java:

1. Static Polymorphism (Compile Time Polymorphism)
2. Dynamic Polymorphism (Run Time Polymorphism)



Dynamic Polymorphism

- **Dynamic polymorphism** is a process or mechanism in which a call to an overridden method is to resolve at runtime rather than compile-time. It is also known as **runtime polymorphism** or **dynamic method dispatch**.
- We can achieve dynamic polymorphism by using the **method overriding**.
- In this process, an overridden method is called through a reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.

- **Properties of Dynamic Polymorphism**

1. It decides which method is to execute at runtime.
2. It can be achieved through dynamic binding.
3. It happens between different classes.
4. It is required where a subclass object is assigned to a super-class object for dynamic polymorphism.
5. Inheritance involved in dynamic polymorphism.

Method Overriding

- It provides a specific implementation to a method that is already present in the parent class.
- it is used to achieve run-time polymorphism. Remember that, it is not possible to override the static method.
- Hence, we cannot override the main() method also because it is a static method.

- **Rules for Method Overriding**

1. The name of the method must be the same as the name of the parent class method.
2. The number of parameters and the types of parameters must be the same as in the parent class.
3. There must exist an IS-A relationship (inheritance).

We call an overridden method through a reference of the parent class. The type of object decides which method is to be executed and it is decided by the JVM at runtime.

➢ **Example of Dynamic Polymorphism:-**

In the following example, we have created two classes named **Sample** and **Demo**. The Sample class is a **parent class** and the Demo class is a **child** or **derived** class. The child class is overriding the **dispaly()** method of the parent class.

```
1.  //parent class
2.  class Sample
3.  {
4.  //method of the parent class
5.  public void display()
6.  {
7.  System.out.println("Overridden Method");
8.  }
9.  }
10. //derived or child class
11. public class Demo extends Sample
12. {
13. //method of child class
14. public void display()
15. {
16. System.out.println("Overriding Method");
17. }
18. public static void main(String args[])
19. {
20. //assigning a child class object to parent class reference
21. Sample obj = new Demo();
22. //invoking display() method
23. obj.display();
24. } }
```

Output:

Overriding Method

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

Method Overloading

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are **two** ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments

- In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

22 33

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

22 24.9

Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

1. For Method Overriding (so runtime polymorphism can be achieved).
2. For Code Reusability.

Terms used in Inheritance

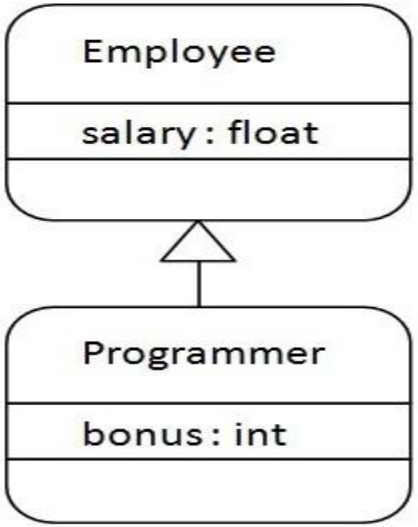
1. **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
2. **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
3. **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
4. **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

- The **syntax** of Java Inheritance
- **class** Subclass-name **extends** Superclass-name
- {
- //methods and fields
- }

The **extends keyword** indicates that you are making a new class that derives from an existing class.

The meaning of "extends" is to increase the functionality.

Java Inheritance Example



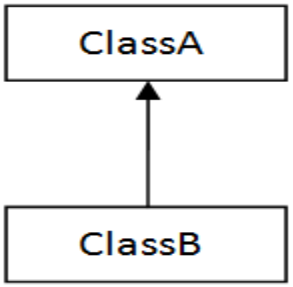

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Output:-

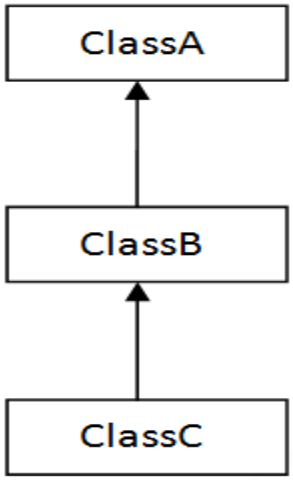
Programmer salary is:40000.0

Bonus of programmer is:10000

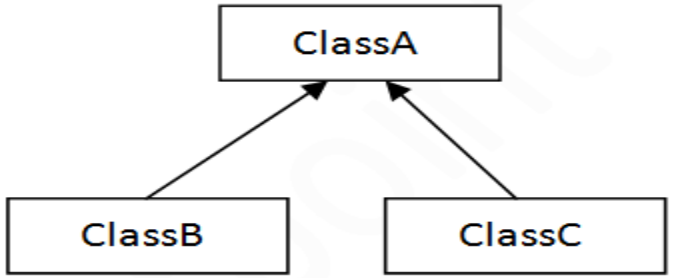
Types of inheritance in java



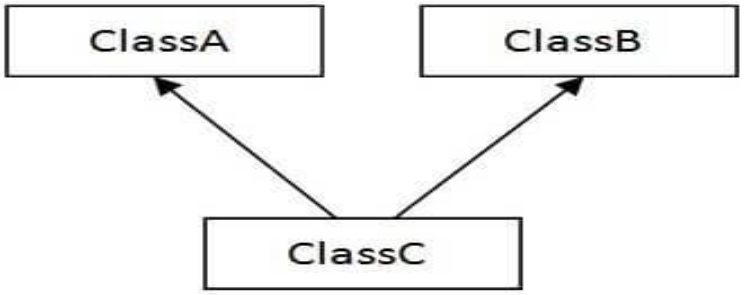
1) Single



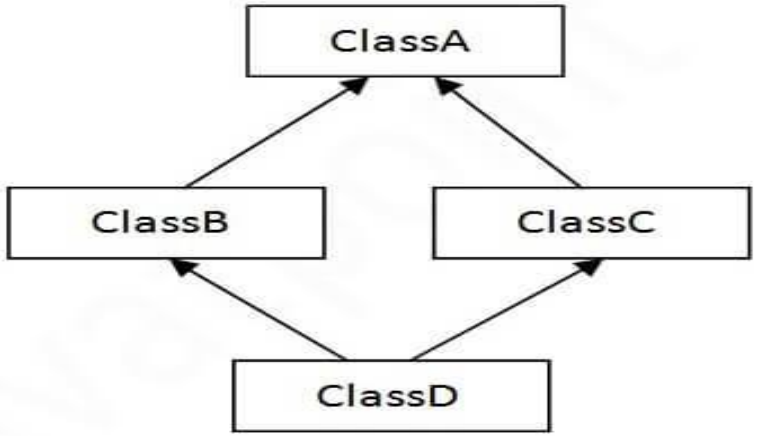
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

Note: Multiple inheritance is not supported in Java through class.

- **Single Inheritance Example**
- When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10.d.bark();
11.d.eat();
12.}}
```

- Output:
- barking...
- eating...

- **Multilevel Inheritance Example**
- When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

- Output:
- weeping...
- barking...
- eating...

- **Hierarchical Inheritance Example**
- When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10.class TestInheritance3{
11.public static void main(String args[]){
12.Cat c=new Cat();
13.c.meow();
14.c.eat();
15.//c.bark();//C.T.Error
16.}}
```

Output:

```
meowing...
eating...
```

Abstract classes

- An abstract class in Java is one that is declared with the abstract keyword. It may have both abstract and non-abstract methods(methods with bodies).
- An abstract is a java modifier applicable for classes and methods in java but *not for Variables*. In this article, we will learn the use of abstract class in java.

- **What is Abstract class in Java?**
- Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties.
- An abstract class is declared using the “abstract” keyword in its class definition.
- abstract class Shape
- {
- int color; // An abstract function abstract
- void draw();
- }
- The abstract keyword is a non-access modifier, used for classes and methods:
- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Abstract classes

- In Java, the following some ***important observations*** about abstract classes are as follows:

- An instance of an abstract class can not be created.
- Constructors are allowed.
- We can have an abstract class without any abstract method.
- There can be a **final method** in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”
- We can define static methods in an abstract class
- We can use the **abstract keyword** for declaring ***top-level classes (Outer class) as well as inner classes*** as abstract
- If a **class** contains at least **one abstract method** then compulsory should declare a class as abstract
- If the **Child class** is unable to provide implementation to all abstract methods of the **Parent class** then we should declare that **Child class as abstract** so that the next level Child class should provide implementation to the remaining abstract method

Abstract classes

```
abstract class Language {

  // method of abstract class
  public void display() {
    System.out.println("This is Java Programming");
  }
}

class Main extends Language {

  public static void main(String[] args) {

    // create an object of Main
    Main obj = new Main();

    // access method of abstract class
    // using object of Main class
    obj.display();
  }
}
```

Output

This is Java programming

- **Implementing Abstract Methods**
- If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

```
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

Output

```
Bark bark
I can eat.
```

Methods with a Variable Number of Parameters

Method:-A Java method is a collection of statements that are grouped together to perform an operation.

```
class Add
{
        int c;
        void addition(int x,int y)
        {
         c=x+y;
        }
public static void main(String[] arg)
{
int a,b;
Scanner sc=new Scanner(System.in);
System.out.println("Enter first number");
a=sc.nextInt();
System.out.println("Enter second number");
b=sc.nextInt();
Add r=new Add();
r.addition(a,b);
System.out.println("Addition of two numbers is : "+r.c);
}
}
```

Methods with a Variable Number of Parameters

```
publicclassJavaExample
{
int add(int num1, int num2)
    {
return num1+num2;
    }
int add(int num1, int num2, int num3)
    {
return num1+num2+num3;
    }
int add(int num1, int num2, int num3, int num4)
    {
return num1+num2+num3+num4;
    }
publicstaticvoid main(String[] args)
    {
        JavaExample obj = newJavaExample();
        //This will call the first add method
System.out.println("Sum of two numbers: "+obj.add(10, 20));
//This will call second add method
System.out.println("Sum of three numbers: "+obj.add(10, 20, 30));
//This will call third add method
System.out.println("Sum of four numbers: "+obj.add(1,  2, 3, 4));
    }
}
```

Enumeration Classes

- The Enum in Java is a data type which contains a fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.
- According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.
- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enumeration Classes

- Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more powerful.
- Here, we can define an enum either inside the class or outside the class.
- Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces.
- We can have fields, constructors, methods, and main methods in Java enum.

Enumeration Classes

Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Enumeration Classes

Simple Example of Java Enum

```
class EnumExample1{
//defining the enum inside the class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//main method
public static void main(String[] args) {
//traversing the enum
for (Season s : Season.values())
System.out.println(s);
}}
```

Test it Now

Output:

```
WINTER
SPRING
SUMMER
FALL
```

Enumeration Classes

Let us see another example of Java enum where we are using value(), valueOf(), and ordinal() methods of Java enum.

```
class EnumExample1{
//defining enum within class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//creating the main method
public static void main(String[] args) {
//printing all enum
for (Season s : Season.values()){
System.out.println(s);
}
System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());


}}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2
```

Enumeration Classes

What is the purpose of the values() method in the enum?

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

What is the purpose of the valueOf() method in the enum?

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

What is the purpose of the ordinal() method in the enum?

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

Interfaces

- An **interface in java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*.
- There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Interfaces

- **Declaring Interfaces:-**
- The **interface** keyword is used to declare an interface
- A class uses the **implements** keyword to implement an interface.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.
- Syntax:
- **interface** <interface_name>{
- // declare constant fields
- // declare methods that abstract
- // by default.
- }

Interfaces

- **Java Interface Example**
- In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.
- **interface** printable{
- **void** print();
- }
- **class** A6 **implements** printable{
- **public void** print(){System.out.println("Hello");}
-
- **public static void** main(String args[]){
- A6 obj = **new** A6();
- obj.print();
- }
- }
- Output:
- Hello

Interfaces

Example 1: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Run Code »

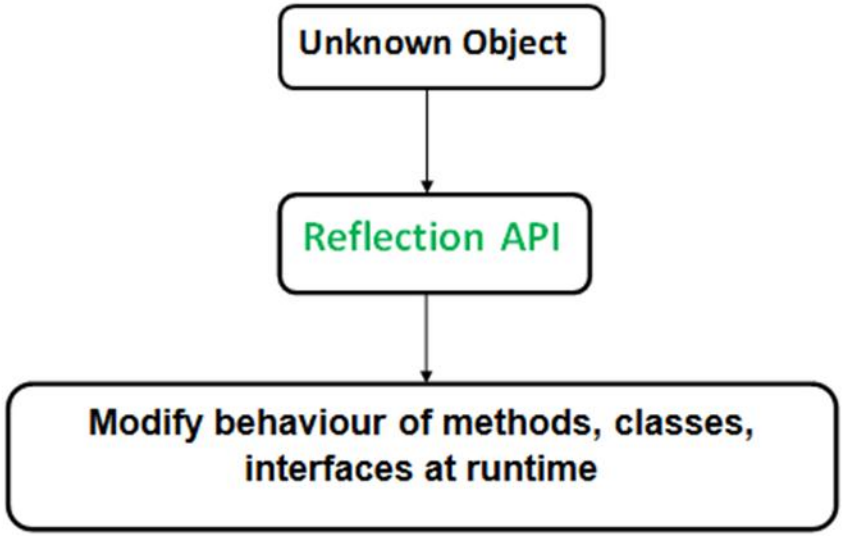
- **Output**
- **The area of the rectangle is 30**

Multiple Interfaces

```
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Reflection

- **What is Reflection in Java?**
- Java Reflection is the process of analyzing and modifying all the capabilities of a class at runtime. Reflection API in Java is used to manipulate class and its members which include fields, methods, constructor, etc. at runtime.
- One advantage of reflection API in Java is, it can manipulate private members of the class too.
- The java.lang.reflect package provides many classes to implement reflection java.Methods of the java.lang.Class class is used to gather the complete metadata of a particular class.



Reflection

- Reflection can be used to get information about –
- **Class** The getClass() method is used to get the name of the class to which an object belongs.
- **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
- **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.