Unit 2

Inheritance: concept of super and sub class, types of inheritance. Polymorphism: method overloading, method overriding; abstract class, constructor in multilevel inheritance, using final with inheritance. **Interface:** defining and implementing interface. extending interface, nested interface, importance of interface in java. **Package:** defining package. rules for creating a new package, concept of class-path, access protection, importing package.

Concept of Super and Sub Class: Inheritance is one of the most powerful features of object-oriented programming (OOP) in Java. It is a technique which allows us to create a new class by extending a previously declared class. That is, we can create a new class from the existing class.

By using the inheritance concept, we can acquire all the features (members) of a class and use them in another class by relating the objects of two classes.

A class that is used to create a new class is called superclass in Java. In the words, the class from where a subclass inherits the features is called superclass. It is also called a base class or parent class.

A class that inherits all the members (fields, method, and nested classes) from the other class is called subclass in Java. In simple words, a newly created class is called subclass. It is also called a derived class, child class, or extended class.

Thus, the process of creating a subclass from a superclass is called *inheritance in Java*. Let’s take some important example programs based on superclass and subclass in Java that will help to you to clear concepts of inheritance.

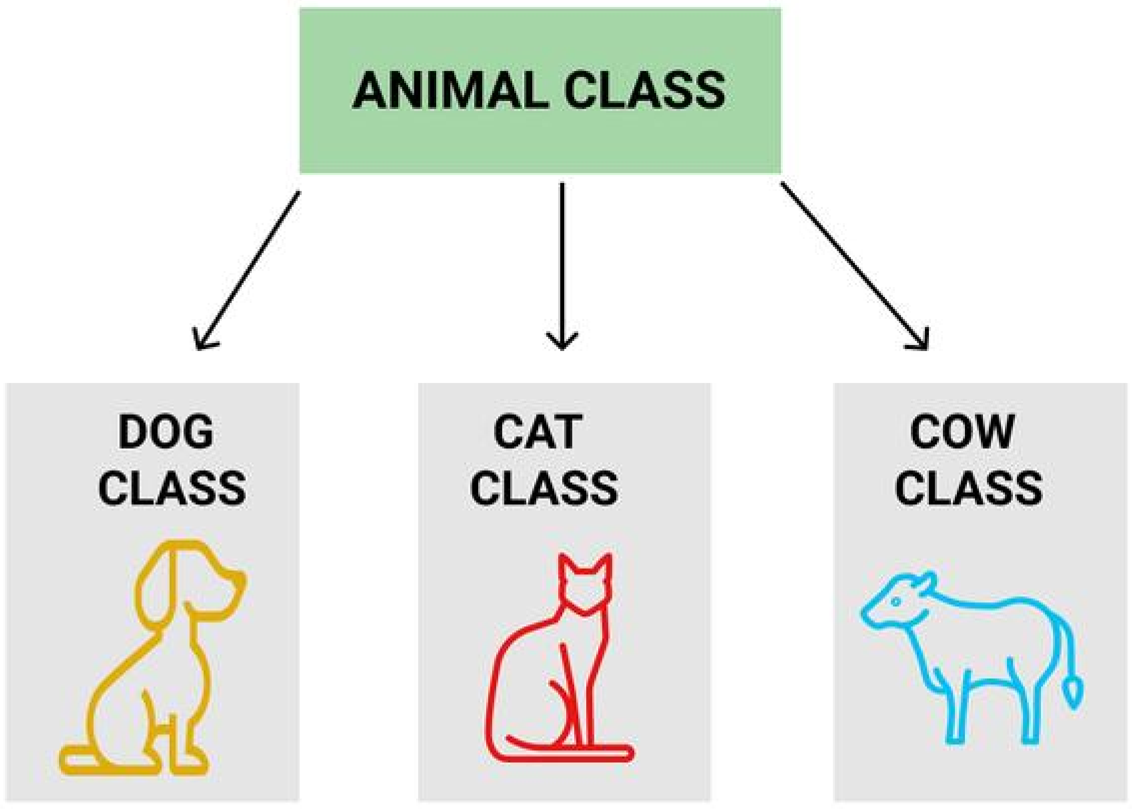
Example:-

```
// Superclass
class Animal {
    void eat() {
        System.out.println("This animal eats
food.");
    }
}
// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}


// Main class to test
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();


        // Calling method from superclass
```

Inheritance in Java

Java Inheritance is a fundamental concept in OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class.

Example: In the following example, Animal is the base class and Dog, Cat and Cow are derived classes that extend the Animal class.



Implementation:

```
// Parent class

class Animal {

   void sound() {

      System.out.println("Animal makes a sound");

   }
```

```
}

// Child class
class Dog extends Animal {
   void sound() {
      System.out.println("Dog barks");
   }
}

// Child class
class Cat extends Animal {
   void sound() {
      System.out.println("Cat meows");
   }
}

// Child class
class Cow extends Animal {
   void sound() {
      System.out.println("Cow moos");
   }
}

// Main class
public class Geeks {
   public static void main(String[] args) {
      Animal a;
```

```
        a = new Dog();
        a.sound();


        a = new Cat();
        a.sound();


        a = new Cow();
        a.sound();
    }
}
```

Output

```
Dog barks
Cat meows
Cow moos
```

Explanation:

- Animal is the base class.
- Dog, Cat and Cow are derived classes that extend Animal class and provide specific implementations of the sound() method.
- The Geeks class is the driver class that creates objects and demonstrates runtime polymorphism using method overriding.

Note: *In practice, inheritance and polymorphism are used together in Java to achieve fast performance and readability of code.*

Syntax

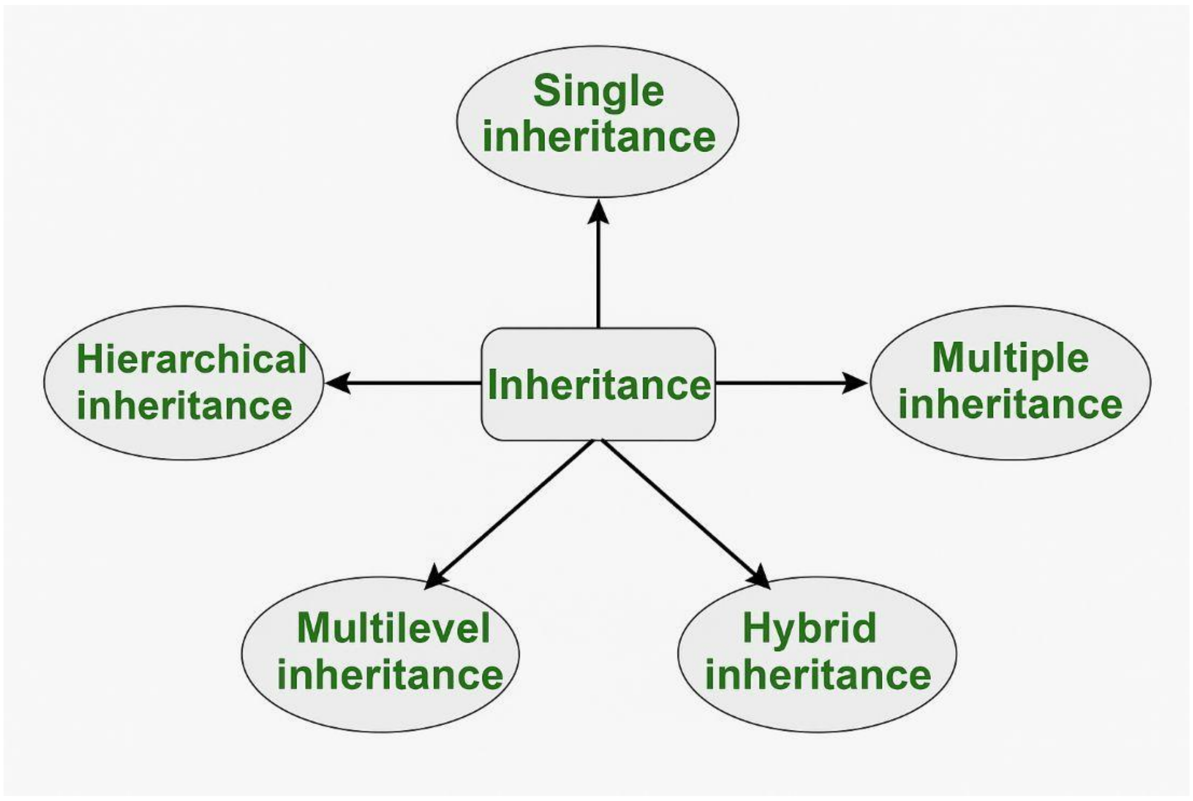
```
class ChildClass extends ParentClass {


    // Additional fields and methods
}
```

Note: *In Java, inheritance is implemented using the extends keyword. The class that inherits is called the subclass (child class) and the class being inherited from is called the superclass (parent class).*

Why Use Inheritance in Java?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstraction where we do not have to provide all details, is achieved through inheritance. Abstraction only shows the functionality to the user.

Types of Inheritance in Java



Types of Inheritance in Java

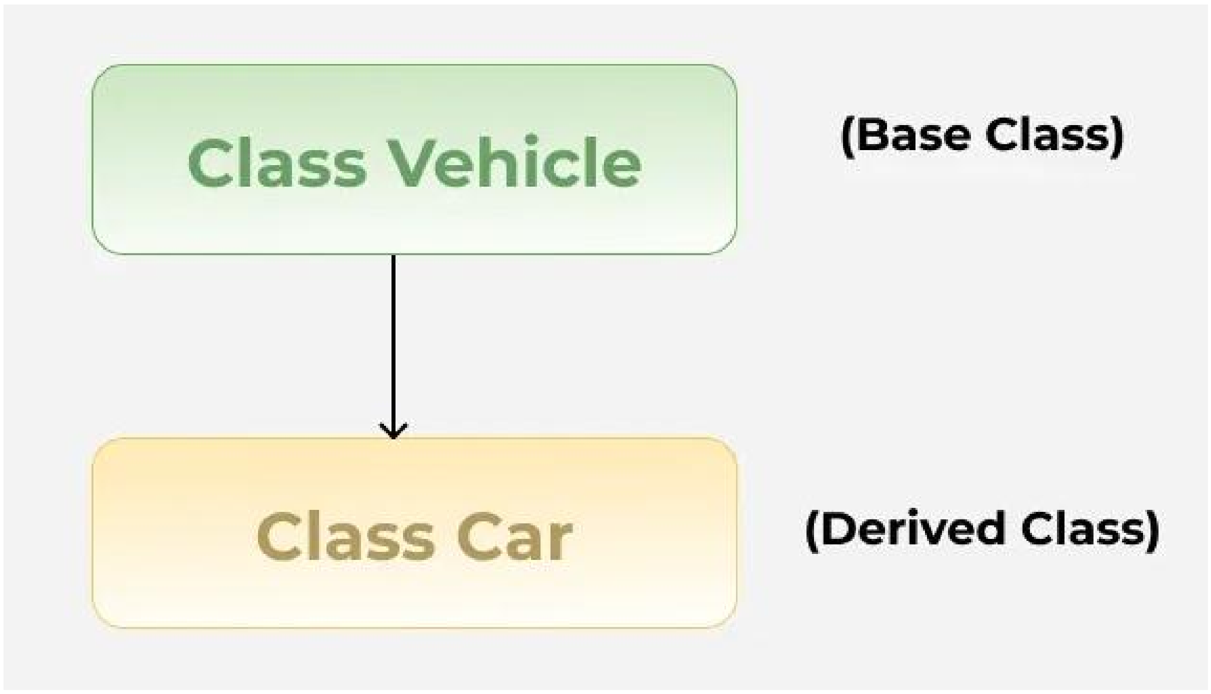
Below are the different types of inheritance which are supported by Java.

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance

- Hybrid Inheritance

1. Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance.



Example:

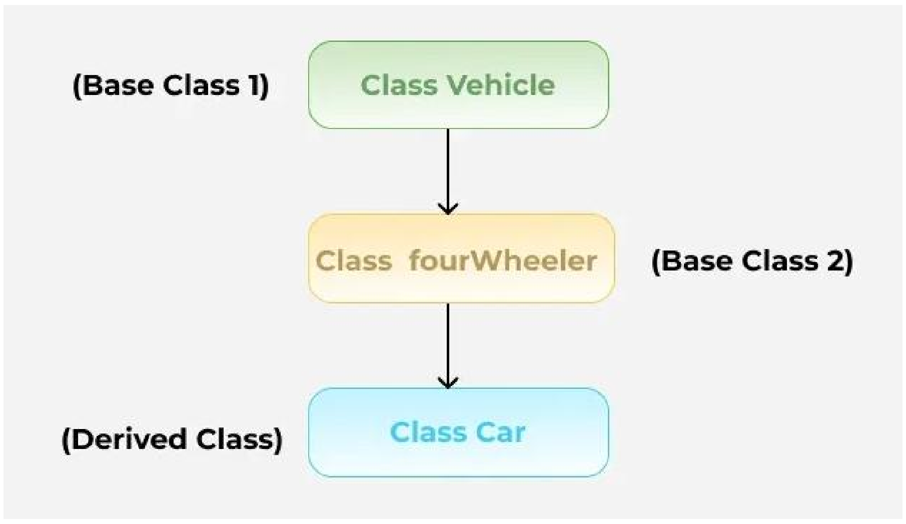
```
//Super class
class Vehicle {
   Vehicle() {
      System.out.println("This is a Vehicle");
   }
}


// Subclass
class Car extends Vehicle {
   Car() {
        System.out.println("This Vehicle is Car");
    }
}

public class Test {
    public static void main(String[] args) {
        // Creating object of subclass invokes base class constructor
        Car obj = new Car();
    }
}
```

Output

```
This is a Vehicle
This Vehicle is Car
```

2. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also acts as the base class for other classes.



Example:

```
class Vehicle {
   Vehicle() {
      System.out.println("This is a Vehicle");
   }
}
class FourWheeler extends Vehicle {
   FourWheeler() {
      System.out.println("4 Wheeler Vehicles");
   }
}
class Car extends FourWheeler {
   Car() {
      System.out.println("This 4 Wheeler Vehicle is a Car");
   }
}
public class Geeks {
   public static void main(String[] args) {
      Car obj = new Car(); // Triggers all constructors in order
   }
}
```

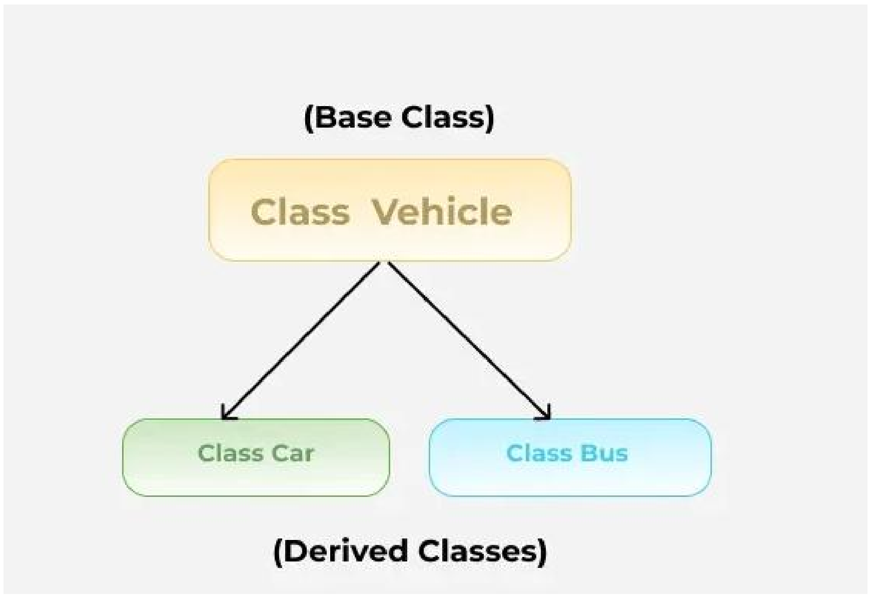
Output

```
This is a Vehicle
4 Wheeler Vehicles
```

This 4 Wheeler Vehicle is a Car

3. Hierarchical Inheritance

In hierarchical inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class. For example, cars and buses both are vehicle



Example:

```
class Vehicle {
   Vehicle() {
      System.out.println("This is a Vehicle");
   }
}


class Car extends Vehicle {
   Car() {
      System.out.println("This Vehicle is Car");
   }
}
```

```
class Bus extends Vehicle {
    Bus() {
        System.out.println("This Vehicle is Bus");
    }
}

public class Test {
    public static void main(String[] args) {
        Car obj1 = new Car();
        Bus obj2 = new Bus();
    }
}
```

Output

```
This is a Vehicle
This Vehicle is Car
This is a Vehicle
This Vehicle is Bus
```

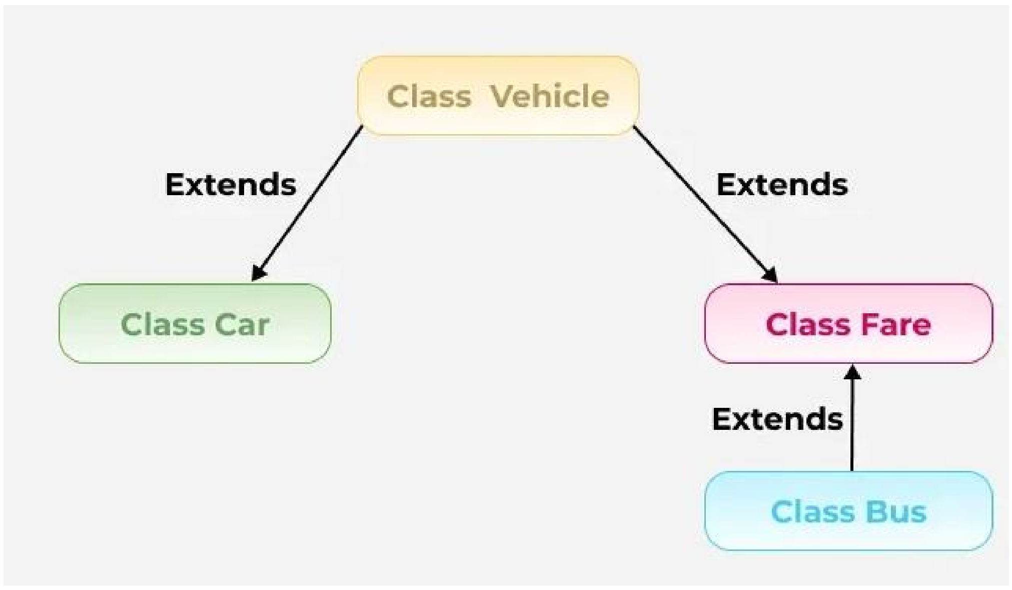
4. Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.

Note: *that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces.*

5. Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.



hybrid

Explanation:

- class Car extends Vehicle->Single Inheritance
- class Bus extends Vehicle and class Bus extends Fare->Hybrid Inheritance (since Bus inherits from two sources, forming a combination of single + multiple inheritance).

Java IS-A type of Relationship

IS-A represents an inheritance relationship in Java, meaning this object is a type of that object.

public class SolarSystem {

}

public class Earth extends SolarSystem {

}

public class Mars extends SolarSystem {

}

public class Moon extends Earth {

}

Now, based on the above example, in Object-Oriented terms, the following are true:

- SolarSystem is the superclass of Earth class.
- SolarSystem is the superclass of Mars class.
- Earth and Mars are subclasses of SolarSystem class.
- Moon is the subclass of both Earth and SolarSystem classes. class SolarSystem {

```
}

class Earth extends SolarSystem {

}

class Mars extends SolarSystem {

}

public class Moon extends Earth {

   public static void main(String args[])

   {

      SolarSystem s = new SolarSystem();

      Earth e = new Earth();

      Mars m = new Mars();

      System.out.println(s instanceof SolarSystem);

      System.out.println(e instanceof Earth);

      System.out.println(m instanceof SolarSystem);

   }

}
```

Polymorphism in Java

Polymorphism in Java is one of the core concepts in object-oriented programming (OOP) that allows objects to behave differently based on their specific class type. The word **polymorphism** means having **many forms**, and it comes from the Greek words **poly (many)** and **morph (forms)**, this means one entity can take many forms. In Java, polymorphism allows the same method or object to behave differently based on the context, specially on the project's actual runtime class.

Key features of polymorphism:

- **Multiple Behaviors**: The same method can behave differently depending on the object that calls this method.
- **Method Overriding**: A child class can redefine a method of its parent class.
- **Method Overloading**: We can define multiple methods with the same name but different parameters.
- **Runtime Decision:** At runtime, Java determines which method to call depending on the object's actual class.

Real-Life Illustration of Polymorphism

Consider a person who plays different roles in life, like a father, a husband, and an employee. Each of these roles defines different behaviors of the person depending on the object calling it.

Example: Different Roles of a Person

```
// Base class Person

class Person {


    // Method that displays the

    // role of a person

    void role() {
        System.out.println("I am a person.");
    }
}

// Derived class Father that
// overrides the role method
class Father extends Person {

    // Overridden method to show
    // the role of a father
    @Override
    void role() {
        System.out.println("I am a father.");
    }
}

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

        // Creating a reference of type Person
        // but initializing it with Father class object
        Person p = new Father();

        // Calling the role method. It calls the
        // overridden version in Father class
        p.role();
    }
}
```

Output

```
I am a father.
```

Explanation: In the above example, the Person class has a method role() that prints a general message. The Father class overrides role() to print a specific message. The reference of type Person is used to point to an object of type Father, demonstrating polymorphism at runtime. The overridden method in Father is invoked when role() is called.

Why Use Polymorphism In Java?

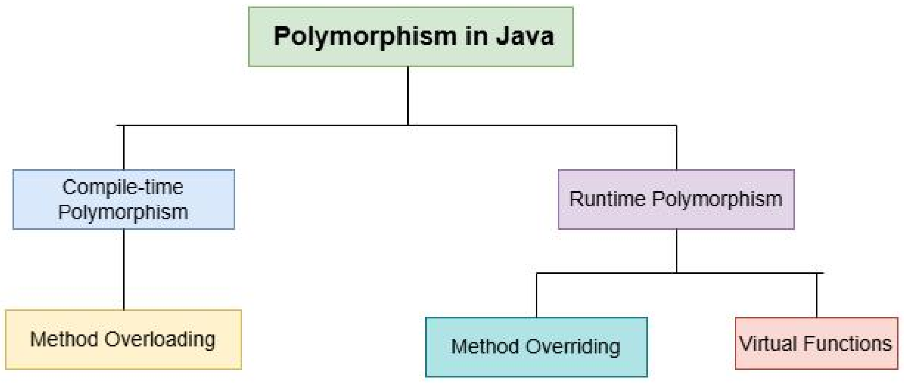
Using polymorphism in Java has many benefits which are listed below:

- **Code Reusability:** Polymorphism allows the same method or class to be used with different types of objects, which makes the code more useable.
- **Flexibility**: Polymorphism enables object of different classes to be treated as objects of a common superclass, which provides flexibility in method execution and object interaction.
- **Abstraction**: It allows the use of abstract classes or interfaces, enabling you to work with general types (like a superclass or interface) instead of concrete types (like specific subclasses), thus simplifying the interaction with objects.
- **Dynamic Behavior**: With polymorphism, Java can select the appropriate method to call at runtime, giving the program dynamic behavior based on the actual object type rather than the reference type, which enhances flexibility.

Types of Polymorphism in Java

In Java Polymorphism is mainly divided into two types:

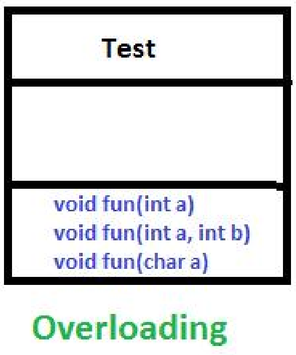
1. Compile-Time Polymorphism (Static)
2. Runtime Polymorphism (Dynamic)

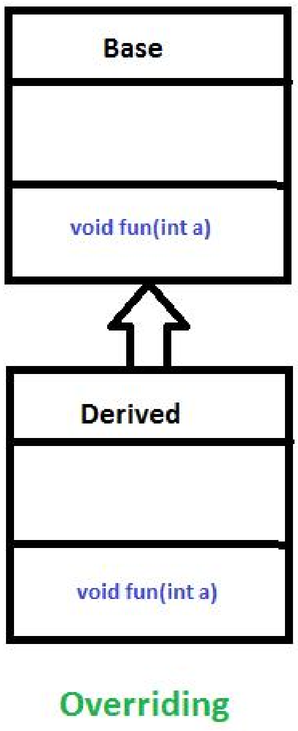


1. Compile-Time Polymorphism

Compile-Time Polymorphism in Java is also known as static polymorphism and also known as method overloading. This happens when multiple methods in the same class have the same name but different parameters.

***Note**: But Java doesn't support the Operator Overloading.*





Method Overloading

As we discussed above, Method overloading in Java means when there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

Example: Method overloading by changing the number of arguments

```
// Method overloading By using

// Different Types of Arguments


// Class 1

// Helper class

class Helper {


   // Method with 2 integer parameters

   static int Multiply(int a, int b)

   {

      // Returns product of integer numbers

      return a * b;

   }


   // Method 2

   // With same name but with 2 double parameters

   static double Multiply(double a, double b)

   {

      // Returns product of double numbers
```

```
        return a * b;
    }
}

// Class 2
// Main class
class Geeks
{
    // Main driver method
    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
```

Explanation: The **Multiply** method is overloaded with different parameter types. The compiler picks the correct method during compile time based on the arguments.

2. Runtime Polymorphism

Runtime Polymorphism in Java 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 Overriding

Method overriding in Java means when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass. Method overriding allows a subclass to modify or extend the behavior of an existing method in the parent class. This enables dynamic method dispatch, where the method that gets executed is determined at runtime based on the object's actual type.

Example: This program demonstrates method overriding in Java, where the Print() method is redefined in the subclasses (subclass1 and subclass2) to provide specific implementations.

```
// Java Program for Method Overriding


// Class 1

// Helper class

class Parent {


    // Method of parent class

    void Print() {

        System.out.println("parent class");

    }

}
```

```
// Class 2
// Helper class
class subclass1 extends Parent {

    // Method
    void Print() {
      System.out.println("subclass1");
    }
}

// Class 3
// Helper class
class subclass2 extends Parent {

    // Method
    void Print() {
        System.out.println("subclass2");
    }
}

// Class 4
// Main class
class Geeks {
    // Main driver method
    public static void main(String[] args) {

        // Creating object of class 1
        Parent a;

        // Now we will be calling print methods
        // inside main() method
        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Output

```
subclass1
subclass2
```

Explanation: In the above example, when an object of a child class is created, then the method inside the child class is called. This is because the method in the parent class is overridden by the child class. This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

Advantages of Polymorphism

- Encourages code reuse.
- Simplifies maintenance.
- Enables dynamic method dispatch.
- Helps in writing generic code that works with many types.

Disadvantages of Polymorphism

- It can make more difficult to understand the behavior of an object.
- This may cause performance issues, as polymorphic behavior may require additional computations at runtime.

Abstract Class

What is an Abstract Class?

An **abstract class** in Java is a class that is **declared with the abstract keyword**. It **cannot be instantiated** (i.e., you cannot create objects of it directly). It is designed to be **inherited by subclasses**, which then **provide implementations** for its **abstract methods**.

Think of an abstract class as a **blueprint** for other classes.

Why Use Abstract Classes?

- To provide **common functionality** (methods, variables) to all subclasses.
- To **force subclasses** to implement certain methods.
- To achieve **partial abstraction** (some methods implemented, some not).

Syntax:

```
abstract class ClassName {
    // Fields or variables
    // Constructors (optional)

    abstract void abstractMethod(); // method without body
    void concreteMethod() {
        // method with body
    }
}
```

Key Points:

Feature	**Description**
abstract keyword	Used to declare an abstract class or method
Object creation	Not allowed directly (e.g., new AbstractClass() is invalid)
Abstract method	Method without a body, must be overridden in subclass
Concrete method	Regular method with body, can be inherited or overridden
Constructors	Abstract classes can have constructors
Fields/variables	Can have variables like a normal class
Subclass responsibilities	Must override all abstract methods unless it's also abstract

Example 1: Basic Abstract Class

```
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Concrete method (with body)
    void eat() {
        System.out.println("This animal eats food.");
    }
}
// Subclass must implement abstract method
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound(); // implemented in Dog
        d.eat();   // inherited from Animal
    }
}
```

Output:

```
Dog barks.
This animal eats food.
```

Example 2: Abstract Class with Constructor and Fields

```
abstract class Shape {
    String color;
    // Constructor
    Shape(String color) {
        this.color = color;
    }

    // Abstract method
    abstract double area();

    // Concrete method
    void displayColor() {
        System.out.println("Color: " + color);
    }
}

class Circle extends Shape {
    double radius;

    Circle(String color, double radius) {
        super(color); // calling abstract class constructor
        this.radius = radius;
    }

    // Implementing abstract method
    double area() {
        return Math.PI * radius * radius;
    }
}

public class Main {
    public static void main(String[] args) {
        Circle c = new Circle("Red", 5.0);
        c.displayColor(); // inherited
        System.out.println("Area: " + c.area()); // implemented
    }
}
```

Output:

```
Color: Red
Area: 78.53981633974483
```

When to Use Abstract Classes?

Use an abstract class when:

- You want to share **code** among several closely related classes.
- You want some **methods defined**, and some **methods to be implemented by subclasses**.
- You want to **enforce a contract** (structure) for subclasses but still provide common functionality.

Constructors in a multilevel inheritance

Multilevel inheritance is when a class inherits a class which inherits another class. An example of this is class C inherits class B and class B in turn inherits class A.

A program that demonstrates constructors in a Multilevel Hierarchy in Java is given as follows:

Example

```
class A {
   A() {
      System.out.println("This is constructor of class A");
   }
}
class B extends A {
   B() {
      System.out.println("This is constructor of class B");
   }
}
class C extends B {
   C() {
      System.out.println("This is constructor of class C");
   }
}
public class Demo {
   public static void main(String args[]) {
      C obj = new C();
   }
}
```

Output

```
This is constructor of class A

This is constructor of class B

This is constructor of class C
```

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java’s most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {

final void meth() {

System.out.println("This is a final method.");

}

}

class B extends A {

void meth() { // ERROR! Can't override.
System.out.println("Illegal!");

}

}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A { //...

}

// The following class is illegal.

class B extends A { // ERROR! Can't subclass A //...
```

}

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

What is an Interface?

In Java, an **interface** is a blueprint of a class. It is a **completely abstract class** that can contain:

- **Abstract methods** (methods without a body)
- **Constants** (variables declared as `public static final`)
- **Default and static methods** (from Java 8 onward)

An **interface defines a contract** that any class implementing the interface must fulfill.

1. Defining and Implementing an Interface

Syntax:

```
interface Animal {
    void sound();  // abstract method
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

Key Points:

- The **`interface`** keyword is used to declare an interface.
- The **`implements`** keyword is used by a class to implement an interface.
- All methods in an interface are **public** and **abstract** by default (before Java 8).

2. Extending an Interface

An interface can **extend one or more other interfaces** using the `extends` keyword.

Syntax:

```
interface A {
    void methodA();
}

interface B extends A {
    void methodB();
}

class MyClass implements B {
    public void methodA() {
        System.out.println("Method A");
    }
    public void methodB() {
        System.out.println("Method B");
    }
}
```

Key Points:

- Interfaces can extend **multiple interfaces**.
- This is how **Java supports multiple inheritance** (which classes alone cannot do).

3. Nested Interface

A **nested interface** is an interface declared **within another class or interface**.

Syntax:

```
class OuterClass {
    interface NestedInterface {
        void print();
    }
}

class InnerClass implements OuterClass.NestedInterface {
    public void print() {
        System.out.println("Nested Interface Implemented");
    }
}
```

Use:

- Used for **logical grouping**.
- Improves **code organization**.

4. Importance of Interface in Java

Feature	Benefit
Full Abstraction	Interfaces hide implementation details and expose only the required behavior (i.e., method signatures).
Multiple Inheritance	A class can implement multiple interfaces, overcoming the limitation of single inheritance.
Loose Coupling	Classes can depend on **interfaces** instead of **concrete classes**, making the code

Feature	Benefit
	flexible and easier to maintain.

5. Example: Multiple Inheritance using Interface

```
interface Printable {
    void print();
}

interface Showable {
    void show();
}

class Document implements Printable, Showable {
    public void print() {
        System.out.println("Printing document");
    }

    public void show() {
        System.out.println("Showing document");
    }
}
```

What is a Package?

A **package** in Java is a mechanism to group related classes, interfaces, and sub-packages together.
It provides a way to **organize code**, avoid **name conflicts**, and offer **access protection**.
Think of it like a **folder** in your computer that contains files of the same type.

Example:

- java.util → contains utility classes like Scanner, ArrayList, etc.
- java.io → contains classes for input/output like File, BufferedReader.

Defining a Package

A package is defined using the **package keyword**, written as the **first statement** in a Java source file.

package mypackage;   // package declaration

public class MyClass {

```
    public void display() {
        System.out.println("Hello from MyClass");
    }
}
```

➡ Save the file as MyClass.java inside a folder named mypackage.

Rules for Creating a Package

1. Use package keyword as the first line of the program.
2. The file must be saved inside a folder with the same package name.
3. The folder structure should match the package hierarchy.

Example:

```
package college.student;   // multi-level package
```

➡ Save in folder path: college/student/MyClass.java

Concept of Classpath

- The **classpath** tells Java where to look for classes and packages.
- By default, Java looks in the current directory.
- You can set classpath in two ways:
    1. **Environment Variable:** set CLASSPATH=C:\Java\mypackage
    2. **Command Line Option:** javac -cp . MyClass.java

Access Protection in Packages

Access modifiers determine the **visibility** of classes, methods, and variables across packages.

Modifier	Within Same Class	Same Package	Subclass in Another Package	Other Packages
public	✅ Yes	✅ Yes	✅ Yes	✅ Yes
protected	✅ Yes	✅ Yes	✅ Yes	❌ No
default	✅ Yes	✅ Yes	❌ No	❌ No
private	✅ Yes	❌ No	❌ No	❌ No

Importing Packages

To use classes from another package, we use the **import keyword**.

Example 1: Import a specific class

```
import java.util.Scanner;


class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter name: ");
        String name = sc.nextLine();
        System.out.println("Hello, " + name);
    }
}
```

Example 2: Import all classes from a package

```
import java.util.*;


class Test {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        System.out.println(list);
    }
}
```

Advantages of Packages

1. **Code Reusability** – Classes can be reused across multiple projects.
2. **Encapsulation & Access Control** – Using access modifiers with packages.
3. **Avoids Name Conflicts** – Two classes with the same name can exist in different packages.
4. **Easier Maintenance** – Organized structure improves readability.
5. **Supports Modular Programming** – Encourages clean project structure.