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:-

Superclass (Parent/Base Class)

- वह class जिससे features (fields, methods, nested classes) inherit किए जाते हैं, उसे **superclass** कहते हैं।
- इसे **base class** या **parent class** भी कहते हैं।

Example of Superclass:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
```

Subclass (Child/Derived Class)

- वह class जो superclass की features को inherit करता है, उसे **subclass** कहते हैं।
- Subclass, superclass के members को access कर सकता है और साथ ही अपने नए members भी define कर सकता है।
- इसे **child class**, **derived class**, या **extended class** भी कहते हैं।

Example of Subclass:

```
class Dog extends Animal {   // Dog is subclass of Animal
    void bark() {
        System.out.println("Dog barks.");
    }
}
```

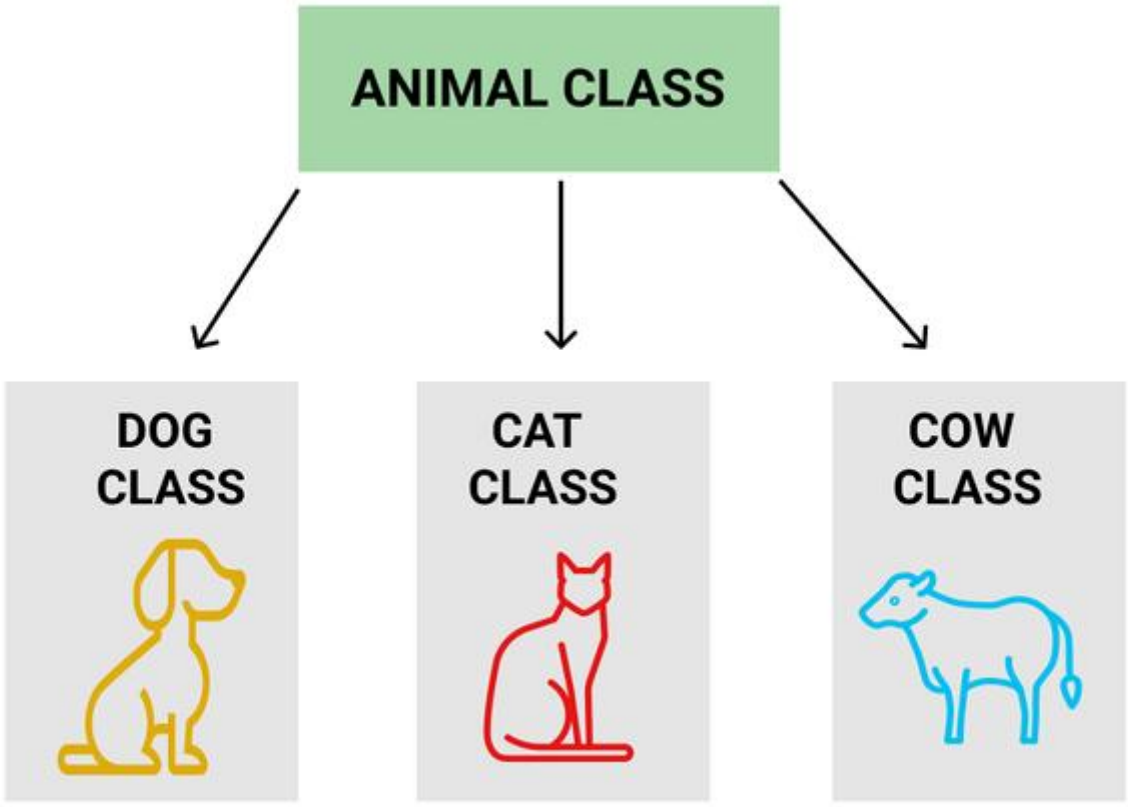
Example:-

```
// 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

- **Inheritance** Java में Object-Oriented Programming (OOP) का एक fundamental concept है।
- यह ऐसा mechanism है जिसमें **एक class (subclass/child)**, दूसरे **class (superclass/parent)** की **fields और methods** को inherit कर सकता है।
- Simple words में, Java Inheritance का मतलब है **नए classes बनाना existing classes के आधार पर**।

Example: निम्नलिखित उदाहरण में, **Animal** बेस क्लास है और **Dog, Cat और Cow** डेराइव्ड क्लासेज़ हैं, जो Animal क्लास को `extends` कीवर्ड से इनहेरिट करती हैं।.



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 Main {
    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** एक base class है।

- **Dog, Cat और Cow** derived classes हैं जो Animal class को extend करती हैं और sound() method की specific implementation देती हैं।

- **Main class** एक driver class है जो objects बनाती है और method overriding का use करके runtime polymorphism को demonstrate करती है।

• **Note:** Practically, Java में inheritance और polymorphism को साथ में use किया जाता है ताकि code fast और readable बने।

Syntax

class ChildClass extends ParentClass {

// Additional fields and methods

}

Note: Java में **Inheritance** को `extends` कीवर्ड से implement किया जाता है।

- जिस class से inherit किया जाता है उसे **Superclass (Parent Class)** कहते हैं।
- जो class inherit करती है उसे **Subclass (Child Class)** कहते हैं।

Java में Inheritance क्यों use किया जाता है?

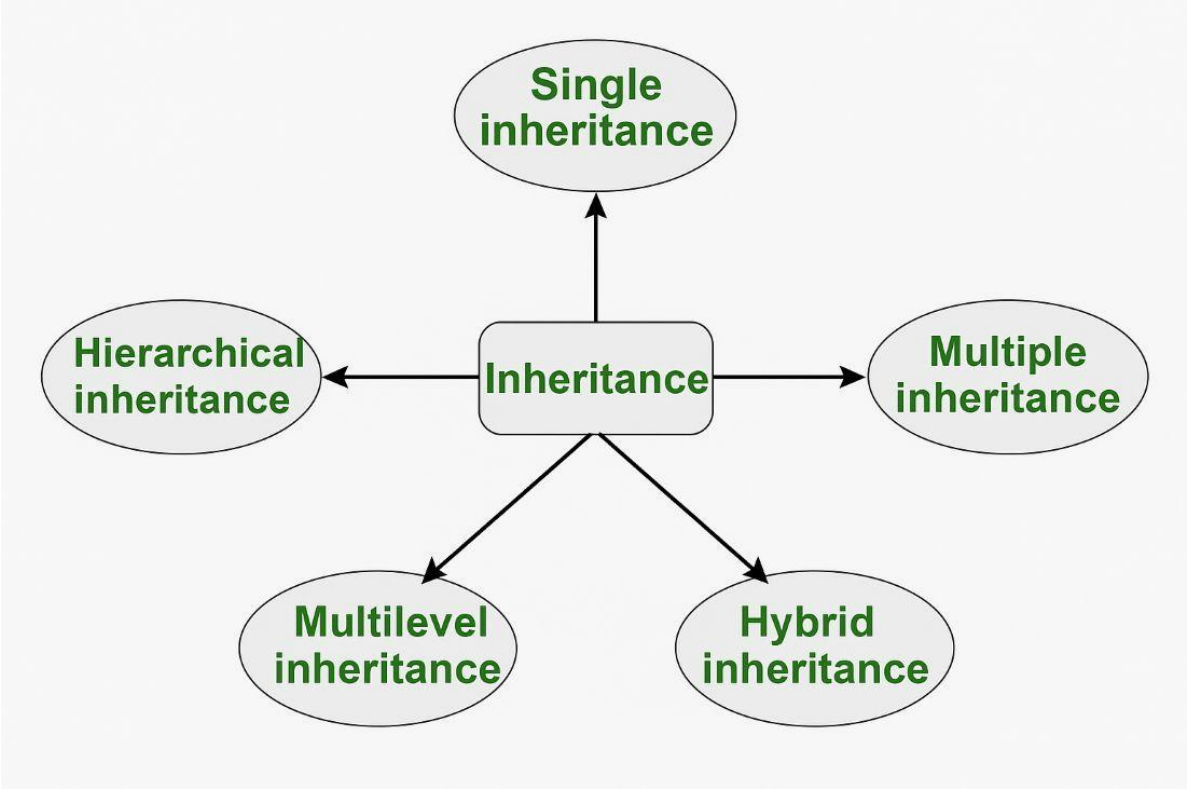
• **Code Reusability:** Superclass में लिखा गया code सभी subclasses के लिए common होता है। Child classes सीधे parent class का code use कर सकती हैं।

• **Method Overriding:** Method Overriding सिर्फ Inheritance के through possible है। यही तरीका है जिससे Java **Run Time Polymorphism** achieve करता है।

• **Abstraction:** Abstraction का concept, जहाँ हमें सारी details provide नहीं करनी होती, inheritance से possible होता है। Abstraction user को सिर्फ functionality दिखाता है, implementation details नहीं।

- .

Types of Inheritance in Java



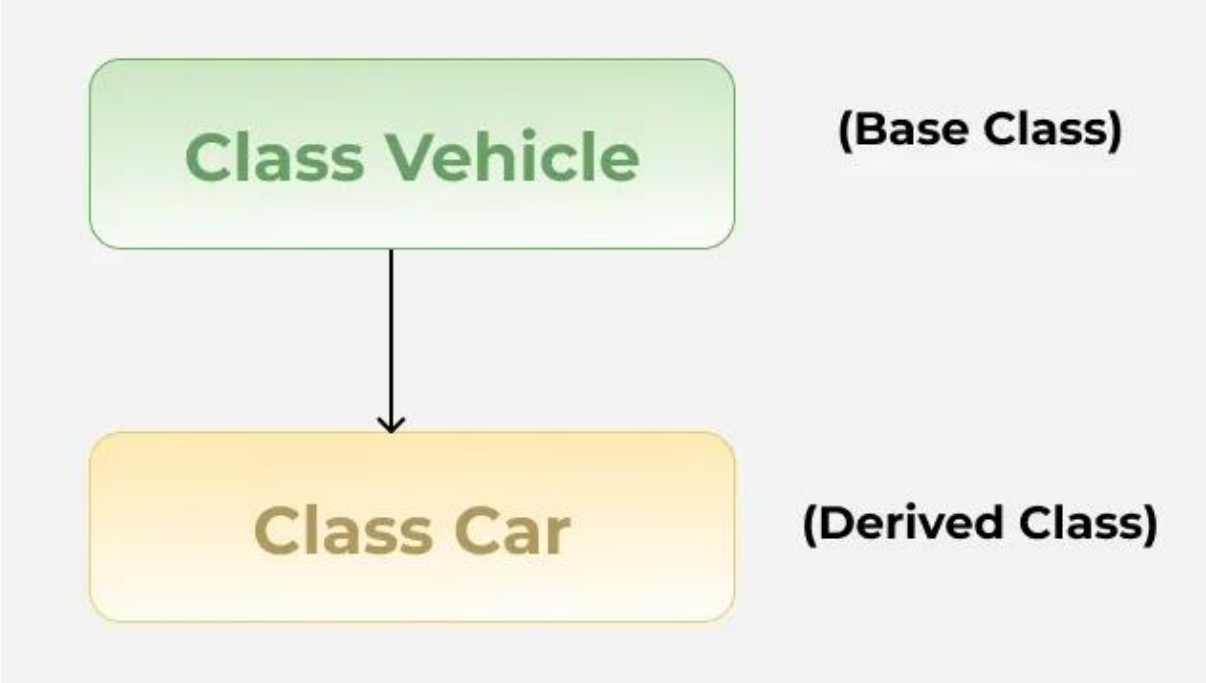
Types of Inheritance in Java

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**, एक **sub-class** सिर्फ एक **super class** से derive होता है।
- यह subclass अपने single-parent class की **properties और behavior** को inherit करता है।
- इसे कभी-कभी **simple inheritance** भी कहा जाता है।



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");
    }
}
```

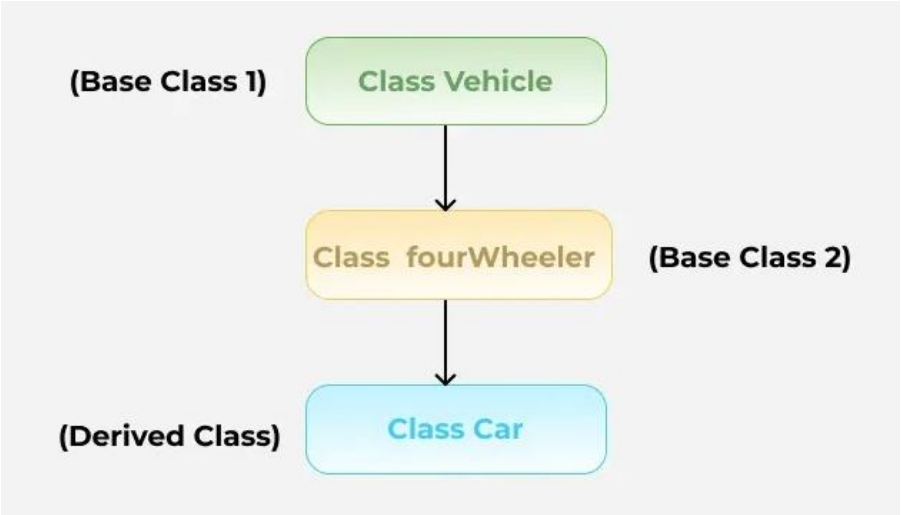
```
public class Test {
    public static void main(String[] args) {
        // Creating object of subclass invokes base class constructor
        Car obj = new Car();
    }
}
```

Output

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

2. Multilevel Inheritance

- In **Multilevel Inheritance**, एक **derived class** किसी **base class** से inherit करता है।
- साथ ही वही **derived class**, किसी दूसरे class के लिए **base class** की तरह काम करता है।
- यानी inheritance chain बनती है: `Base Class → Derived Class → Another Derived Class`



Example:

```
class Vehicle {
    Vehicle() {
```

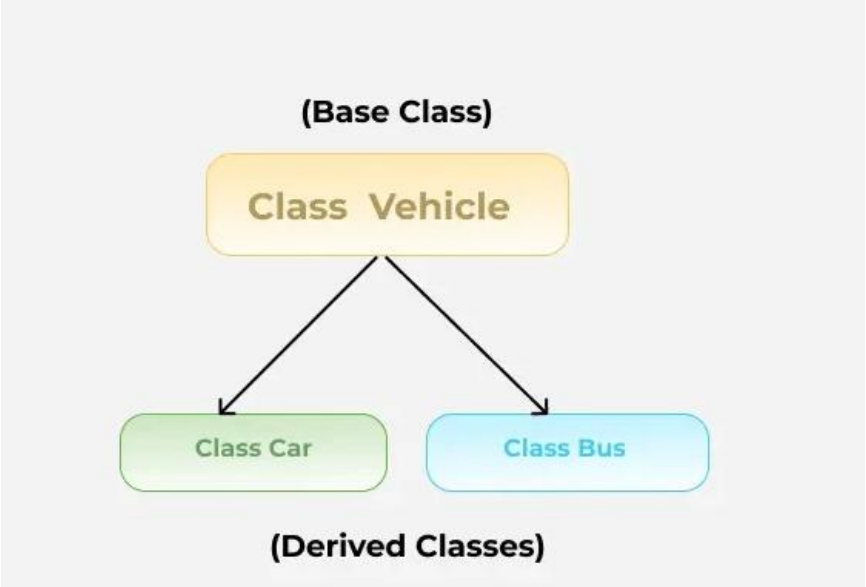
```
        System.out.println("This is a Vehicle");
    }
}
class FourWheeler extends Vehicle {
    FourWheeler() {
        System.out.println("4 Wheeler Vehicles");
    }
}
class Car extends FourWheeler {
    Car() {
        System.out.println("This 4 Wheeler Vehicle is a Car");
    }
}
public class 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**, एक single **base class** से एक से ज़्यादा **subclasses** inherit होते हैं।
- यानी एक base class से multiple derived classes बनाई जाती हैं।
- **Example:** `Vehicle` base class है और उससे **Car** और **Bus** दोनों subclasses inherit करते हैं।



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)

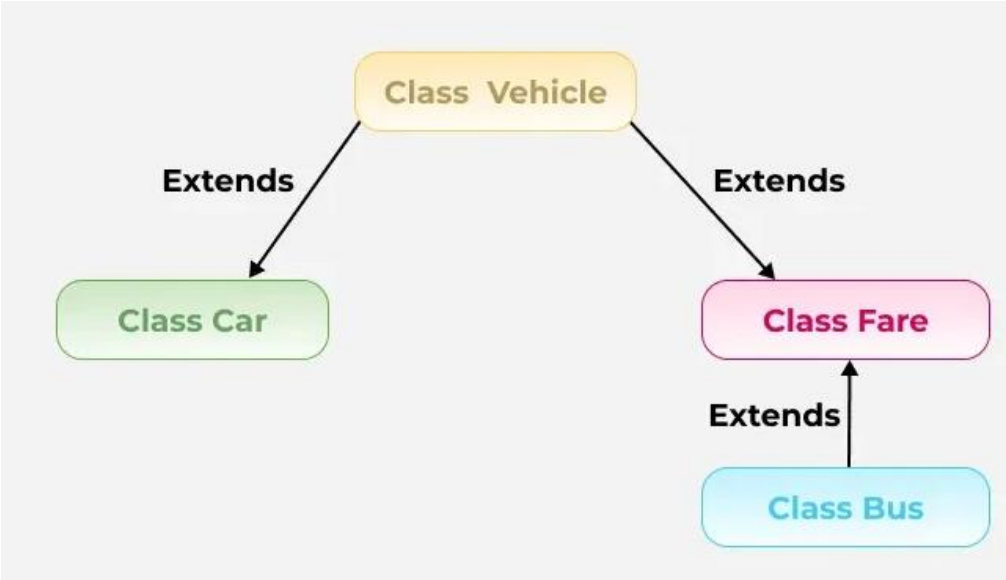
In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.

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

5. Hybrid Inheritance

- **Hybrid Inheritance** दो या उससे अधिक inheritance types का combination होता है।

- Java में direct hybrid inheritance possible नहीं है क्योंकि Java **multiple inheritance** को classes के साथ support नहीं करता।
- लेकिन अगर हमें **multiple inheritance** को use करके **hybrid inheritance** implement करना हो, तो यह केवल **Interfaces** के through possible है।



hybrid

Explanation:

- **class Car extends Vehicle → Single Inheritance**
- **class Bus extends Vehicle और class Bus extends Fare → Hybrid Inheritance** (क्योंकि Bus दो sources से inherit कर रहा है, जिससे Single + Multiple Inheritance का combination बनता है)।

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** Java में OOP (Object-Oriented Programming) का core concept है जो objects को उनकी specific class type के हिसाब से अलग-अलग तरह से behave करने देता है।
- "Polymorphism" का मतलब है **many forms** (Greek words से: *poly* = many, *morph* = forms)।
- यानी, **एक entity कई रूप ले सकती है**।
- Java में polymorphism allow करता है कि **एक ही method या object अलग-अलग तरीके से behave करे**, खासकर actual runtime class के context पर।

Key Features of Polymorphism

• **Multiple Behaviors:** एक ही method अलग-अलग objects के लिए अलग behavior show कर सकता है।
• **Method Overriding:** Child class, parent class की method को redefine कर सकता है।
• **Method Overloading:** एक ही नाम की कई methods हो सकती हैं लेकिन parameters अलग होंगे।
• **Runtime Decision:** Runtime पर Java decide करता है कि कौन-सी method call होगी object की actual class के हिसाब से।

Real-Life Example

एक **person** अपनी life में अलग-अलग roles play करता है:

- Father
- Husband
- Employee

हर role में उसका behavior अलग होता है। उसी तरह, Java में polymorphism allow करता है कि **same method different forms** ले depending on the object.

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**: ऊपर दिए गए example में:
  - **Person class** में एक method `role()` है जो general message print करता है।
  - **Father class** ने `role()` method को override करके specific message print किया।
  - Reference type **Person** का use करके **Father object** को point किया गया।
  - जब `role()` call किया गया तो **runtime polymorphism** हुआ और **Father की overridden method** invoke हुई।

Why Use Polymorphism in Java?

• **Code Reusability:** Polymorphism allow करता है कि same method या class different types के objects के साथ use हो सके, जिससे code ज्यादा reusable बनता है।

• **Flexibility:** Polymorphism enable करता है कि अलग-अलग classes के objects को एक common superclass के object की तरह treat किया जा सके, जिससे method execution और object interaction flexible होता है।

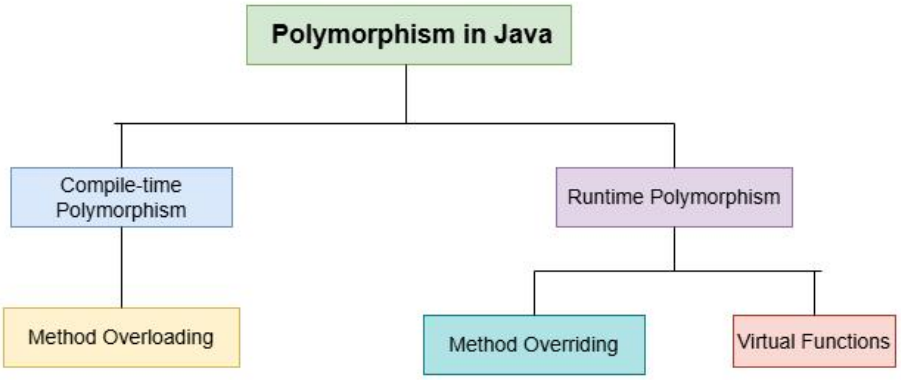
• **Abstraction:** Polymorphism abstract classes और interfaces के use को support करता है। इससे हम general types (superclass/interface) के साथ काम कर सकते हैं बजाय concrete subclasses के, जिससे object interaction simple हो जाता है।

• **Dynamic Behavior:** Polymorphism की वजह से Java runtime पर decide करता है कि कौन-सी method call होगी actual object type के हिसाब से, ना कि reference type के हिसाब से। इससे program में dynamic behavior आता है और flexibility बढ़ती है।

Types of Polymorphism in Java

In Java Polymorphism is mainly divided into two types:

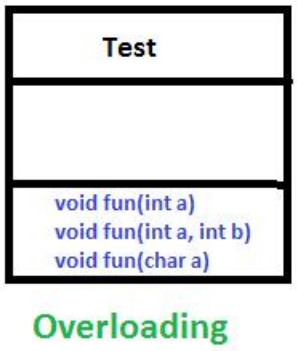
1. Compile-Time Polymorphism (Static)
2. Runtime Polymorphism (Dynamic)



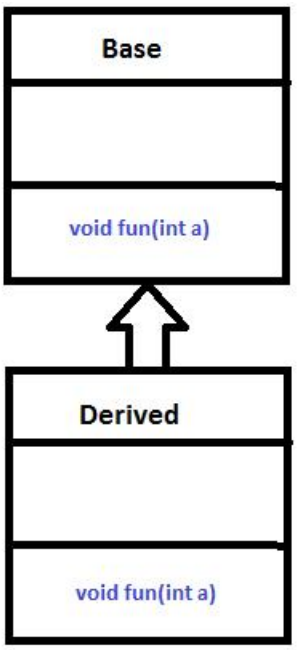
1. Compile-Time Polymorphism

- **Compile-Time Polymorphism** को **Static Polymorphism** भी कहा जाता है।
- इसे **Method Overloading** के नाम से भी जाना जाता है।
- यह तब होता है जब **एक ही class में कई methods का नाम same होता है लेकिन उनके parameters अलग होते हैं।**

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



Overloading



Overriding

Method Overloading

- जैसा कि ऊपर discuss किया गया, **Method Overloading** का मतलब है कि जब एक class में **multiple functions का नाम same हो लेकिन उनके parameters अलग हों**, तो उन functions को **overloaded methods** कहा जाता है।

- Methods को overload किया जा सकता है:

1. **Arguments की संख्या (number of arguments) बदलकर**
2. **Arguments के type बदलकर**

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** को **Dynamic Method Dispatch** भी कहा जाता है।
- यह एक process है जिसमें किसी **overridden method** की function call **runtime पर resolve होती है**।
- यह polymorphism **Method Overriding** के through achieve किया जाता है।

Method Overriding

- **Method Overriding** का मतलब है कि जब **subclass** अपने **superclass** में पहले से defined किसी method की specific implementation provide करती है।
- Overriding के लिए:
    1. Method का **name** same होना चाहिए।
    2. Method का **return type** same होना चाहिए।
    3. Method के **parameters** भी same होने चाहिए।
- **Purpose:** Method overriding allow करता है कि **subclass parent class के method के behavior को modify या extend कर सके**।

- इससे **Dynamic Method Dispatch** possible होता है, जहाँ runtime पर decide होता है कि कौन-सी method execute होगी — object के actual type के हिसाब से।

Example: यह program Java में **method overriding** को demonstrate करता है, जहाँ `Print()` method को subclasses (**Subclass1** और **Subclass2**) में redefine किया गया है ताकि वो अपनी **specific implementation** provide कर सकें।

जब हम superclass के reference से subclasses के objects को point करते हैं और `Print()` call करते हैं, तो **runtime पर** actual object की class के हिसाब से method execute होती है। यही **runtime polymorphism** है।

```
// 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

ऊपर दिए गए example में, जब **child class का object** create किया जाता है, तब **child class की method** call होती है।

- इसका कारण यह है कि **parent class की method को child class ने override कर दिया है**।
- इसलिए, child class में defined method को **parent class की method से ज़्यादा priority** मिलती है।
- Result: जब method call होती है, तो **child class वाली body execute होती है**, ना कि parent वाली।
- यही concept Java में **Method Overriding** और **Runtime Polymorphism** कहलाता है।

Advantages of Polymorphism

- **Code Reuse** – बार-बार code लिखने की जरूरत नहीं।
- **Easy Maintenance** – Code manage और modify करना आसान।

- **Dynamic Method Dispatch** – Runtime पर सही method call होता है।
- **Generic Programming** – ऐसा code लिख सकते हैं जो कई types पर काम करे।

Disadvantages of Polymorphism

- **Complexity** – Object का actual behavior समझना कभी-कभी मुश्किल हो जाता है।
- **Performance Issues** – Runtime पर method resolution होने से थोड़ा **extra computation** होता है।

Abstract Class

What is an Abstract Class?

- Java में **abstract class** वह class होती है जिसे `abstract` keyword के साथ declare किया जाता है।
- इसे **instantiate नहीं किया जा सकता** (यानी इसका object directly create नहीं कर सकते)।
- इसे **inherit करने के लिए design किया जाता है**, ताकि इसकी **abstract methods** की implementation subclasses provide कर सकें।
- इसे आप एक **blueprint (खाका)** मान सकते हैं, जो अन्य classes को design करने में use होता है।

Why Use Abstract Classes?

• सभी subclasses को **common functionality** (methods, variables) provide करने के लिए।
• Subclasses को force करने के लिए कि वे कुछ specific **methods implement करें**।
• **Partial abstraction** achieve करने के लिए (कुछ methods implemented हों, और कुछ सिर्फ declare हों)।

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?

Abstract class का use तब करना चाहिए जब:

• आप कई **closely related classes** के बीच code share करना चाहते हैं।
• आप चाहते हैं कि कुछ methods define हों और बाकी methods subclasses implement करें।
• आप subclasses के लिए एक **contract (structure)** enforce करना चाहते हैं, लेकिन साथ ही **common functionality** भी provide करना चाहते हैं।

Constructors in a multilevel inheritance

•**Definition:** Multilevel inheritance तब होता है जब एक class किसी दूसरी class से inherit करती है, और वह दूसरी class किसी और class से inherit करती है।

•Example: `Class C → Class B` से `inherit` करती है और `Class B → Class A` से `inherit` करती है।

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 `final` Keyword in Java

Java में `final` keyword के तीन main uses हैं:

1. **Constant values** बनाने के लिए (named constant)।

2. **Method overriding prevent** करने के लिए।
3. **Inheritance prevent** करने के लिए।

1. Using `final` to Prevent Overriding

- Normally, subclass parent class की method को override कर सकती है।
- लेकिन अगर किसी method को **`final` declare किया गया है**, तो उसे override नहीं किया जा सकता।

Example:

```
class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    // ERROR: Can't override final method
    void meth() {
        System.out.println("Illegal!");
    }
}
```

Output: Compile-time error (क्योंकि `meth()` को override करना allowed नहीं है)।

Performance Note:

- Final methods को compiler **inline** कर सकता है (early binding)।
- Normally methods **runtime पर resolve होती हैं (late binding)**, लेकिन final methods compile-time पर resolve हो जाती हैं।

2. Using `final` to Prevent Inheritance

- अगर किसी class को **`final` declare किया गया है**, तो उसे inherit नहीं किया जा सकता।
- Final class की सभी methods भी implicitly final होती हैं।

Example:

```
final class A {
    // class body
}

// ERROR: Can't inherit final class
class B extends A {
    // Not allowed
}
```

Result: Compile-time error, क्योंकि final class से subclass बनाना illegal है।

Note:

- **Abstract class** और **final class** को एक साथ declare करना illegal है।
- Reason: Abstract class को subclass की जरूरत होती है (implementation के लिए), जबकि final class subclass allow ही नहीं करती।

What is an Interface in Java?

- Java में **interface** एक **blueprint of a class** है।
- यह पूरी तरह से **abstract class** जैसा होता है, जिसमें शामिल हो सकते हैं:
  • **Abstract methods** (methods बिना body के)
  • **Constants** (`public static final` variables)
  • **Default और Static methods** (Java 8 से onward)
- एक interface एक **contract** define करता है जिसे implement करने वाली class को fulfill करना होता है।

1Defining 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:

- `interface` keyword से interface declare होता है।
- `implements` keyword से class, interface को implement करती है।
- Java 8 से पहले, interface की सभी methods **public और abstract** by default होती हैं।

2 Extending an Interface

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:

- एक interface, multiple interfaces को `extends` कर सकता है।
- यह तरीका है जिससे Java **multiple inheritance** support करता है (जो classes alone नहीं कर सकती)।

3 Nested Interface

Syntax:

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

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

Use:

- Logical grouping के लिए use होता है।
- Code organization improve करता है।

4 Importance of Interface in Java

Feature	Benefit
Full Abstraction	Implementation details hide करता है और सिर्फ method signatures expose करता है।
Multiple Inheritance	एक class multiple interfaces implement कर सकती है, जिससे single

Feature	Benefit
	inheritance की limitation दूर होती है।
Loose Coupling	Classes को concrete classes पर depend करने के बजाय interfaces पर depend करने देता है, जिससे code flexible और maintainable बनता है।

---

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");
    }
}
```

Output:

```
Printing document
Showing document
```

What is a Package in Java?

- A **package** Java में एक mechanism है जो **related classes, interfaces, और sub-packages** को group करता है।
- यह code को organize करने, name conflicts avoid करने और access protection provide करने में help करता है।
- आप इसे अपने computer के **folder** की तरह समझ सकते हैं जिसमें related files store होती हैं।

Examples:

- `java.util` → Scanner, ArrayList जैसी utility classes।
- `java.io` → File, BufferedReader जैसी I/O classes।

Defining a Package

```
package mypackage;   // package declaration

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

➡ File को `MyClass.java` के नाम से **mypackage folder** में save करना होगा।

Rules for Creating a Package

1. Program की पहली line में **package keyword** होना चाहिए।
2. File उसी नाम के folder में save होना चाहिए जिस नाम का package है।
3. Folder structure package hierarchy के बराबर होना चाहिए।

Example:

```
package college.student;
```

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

Concept of Classpath

- **Classpath** बताता है कि Java classes और packages कहाँ search करेगा।
- **Default:** Java current directory में search करता है।
- **Classpath set करने के तरीके:**
    1. **Environment Variable:**
    2. `set CLASSPATH=C:\Java\mypackage`
    3. **Command Line Option:**
    4. `javac -cp . MyClass.java`

Access Protection in 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

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 को multiple projects में reuse किया जा सकता है।
2. **Encapsulation & Access Control** – Access modifiers के साथ secure code।
3. **Avoids Name Conflicts** – Same नाम वाली classes अलग-अलग packages में exist कर सकती हैं।
4. **Easier Maintenance** – Organized structure से readability बढ़ती है।
5. **Supports Modular Programming** – Project structure clean और modular बनता है।