Inheritance

🔷 What is Inheritance?

Inheritance is a feature in Java that lets one class **acquire** (inherit) the **properties and methods** of another class.

- The **class that inherits** is called the **subclass (child class)**.
- The **class being inherited from** is the **superclass (parent class)**.

🔶 Syntax:

java

CopyEdit

```
class Parent {
   // fields and methods
}


class Child extends Parent {
   // additional fields and methods
}
```

🔷 Why Use Inheritance?

- **Code reuse** – Write common code once in the parent class.
- **Improves code organization**
- **Supports polymorphism** (used in method overriding)

🔷 Types of Inheritance in Java

Java supports:

1. **Single Inheritance**
2. **Multilevel Inheritance**

3. **Hierarchical Inheritance**

⚠️ Java does **not support multiple inheritance** with classes (to avoid ambiguity), but it supports it with **interfaces**.

🔷 **Member Access and Inheritance**

🔶 **Member = Variables + Methods**

When a class inherits another class, it **gets access** to its members, but access depends on **access modifiers**.

🔶 **Access Modifiers:**

Modifier	Same Class	Subclass (Same Package)	Subclass (Different Package)	Outside Class
private	✅	❌	❌	❌
(no modifier)	✅	✅	❌	❌
protected	✅	✅	✅	❌
public	✅	✅	✅	✅

🟡 **Key Points:**

- private members are **not inherited** (they are hidden).
- public and protected members are inherited.
- Default (no modifier) members are inherited **only within the same package**.

🔷 **Example:**

java

CopyEdit

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

```
    }

    protected void sleep() {
        System.out.println("This animal sleeps.");
    }

    private void secret() {
        System.out.println("Secret method.");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Dog barks.");
    }

    public void showActions() {
        eat();    // inherited - public
        sleep();  // inherited - protected
        // secret(); ❌ Not inherited - private
    }
}
```

Output:

nginx

CopyEdit

This animal eats food.

This animal sleeps.

Dog barks.

🔷 Constructor and Inheritance

- Constructors **are not inherited**, but the **child class constructor** calls the **parent class constructor** using super().

java

CopyEdit

```
class A {
  A() {
    System.out.println("A's constructor");
  }
}

class B extends A {
  B() {
    super(); // calls A's constructor
    System.out.println("B's constructor");
  }
}
```

🔷 Overriding and Inheritance

A **child class** can **override** a method of the parent class to give its own implementation.

java

CopyEdit

```
class Parent {
  void show() {
    System.out.println("Parent show");
  }
}
```

```
class Child extends Parent {
   @Override
   void show() {
      System.out.println("Child show");
   }
}
```

✅ **Summary:**

Concept	Explanation
Inheritance	One class acquires features of another class
Subclass/Superclass	Subclass inherits from Superclass using extends
Access to Members	Depends on access modifiers (public, protected, etc.)
Constructor Inheritance	Constructors are not inherited, but super() can call the parent constructor
Method Overriding	Subclass can redefine methods of superclass

🔷 **Types of Inheritance in Java**

Java supports the following types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance (Through Interfaces Only)**

❌ Java **does not support multiple inheritance** using **classes**, to avoid ambiguity (Diamond Problem).
✅ It supports multiple inheritance **using interfaces**.

✅ **1. Single Inheritance**

One subclass inherits from one superclass.

📘 **Example:**

java

CopyEdit

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // from Animal
        d.bark();  // from Dog
    }
}
```

✅ **2. Multilevel Inheritance**

A class is derived from a class that is also derived from another class (chain of inheritance).

📘 **Example:**

java

CopyEdit

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Puppy extends Dog {
    void weep() {
        System.out.println("Weeping...");
    }
}

public class Test {
    public static void main(String[] args) {
        Puppy p = new Puppy();
        p.eat();   // from Animal
        p.bark();  // from Dog
        p.weep();  // from Puppy
    }
}
```

✅ 3. Hierarchical Inheritance

Multiple classes inherit from a single parent class.

📘 Example:

java

CopyEdit

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("Meowing...");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
```

```
        d.bark();

        Cat c = new Cat();
        c.eat();
        c.meow();
    }
}
```

✅ 4. Multiple Inheritance (Using Interfaces)

Java does **not** allow a class to inherit from multiple classes, but it **can implement multiple interfaces**.

📘 Example:

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

interface Showable {
    void show();
}

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

    public void show() {
```

```
    System.out.println("Showing...");
  }
}

public class Test {
  public static void main(String[] args) {
    A obj = new A();
    obj.print();
    obj.show();
  }
}
```

🚫 Why No Multiple Inheritance with Classes?

⚠️ Ambiguity (Diamond Problem):

```
java
CopyEdit
class A {
  void show() { System.out.println("A"); }
}

class B extends A {
  void show() { System.out.println("B"); }
}

class C extends A {
  void show() { System.out.println("C"); }
}
```

// class D extends B, C ❌ Not allowed in Java

Java avoids this by **not allowing** multiple class inheritance.

✅ **Summary Table**

Type	Description	Supported in Java
Single Inheritance	One class inherits from one superclass	✅ Yes
Multilevel Inheritance	Inheritance chain of classes	✅ Yes
Hierarchical Inheritance	Multiple classes inherit from one superclass	✅ Yes
Multiple Inheritance	One class inherits from multiple classes	❌ No (via classes)
Multiple via Interface	One class implements multiple interfaces	✅ Yes

Super class Refrence

🔷 1. Superclass Reference in Java

A **superclass reference** can refer to an **object of its subclass**.

📌 Syntax:

java

CopyEdit

Superclass obj = new Subclass();

This is known as **upcasting**, and it's used for **runtime polymorphism**.

📘 Example: Superclass Reference

java

CopyEdit

```
class Animal {
   void sound() {
      System.out.println("Animal makes sound");
   }
}

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

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

```
    Animal obj = new Dog(); // superclass reference to subclass object
    obj.sound();  // Output: Dog barks (runtime polymorphism)
  }
}
```

✅ **Key Point:**

- Method call depends on **object type**, not reference type → called **dynamic method dispatch**.
- You can only call methods that are in the **superclass** using the reference.

🔷 **2. super Keyword in Java**

The super keyword refers to the **immediate parent class** of a subclass. It is used in **three main ways**:

✅ **a) To Call Parent Class Constructor**

Java automatically calls the parent constructor, but you can explicitly call it using super().

📘 **Example:**

java

CopyEdit

```
class Animal {
  Animal() {
    System.out.println("Animal constructor");
  }
}

class Dog extends Animal {
  Dog() {
    super(); // calls Animal constructor
    System.out.println("Dog constructor");
```

```
    }
}

public class Test {
    public static void main(String[] args) {
        new Dog();
    }
}
```

🟢 **Output:**

```
Animal constructor
Dog constructor
```

✅ **b) To Access Parent Class Method**

If the subclass overrides a method from the parent class, you can use super.methodName() to call the parent method.

📘 **Example:**

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void sound() {
```

```
        super.sound(); // calls parent method
        System.out.println("Dog barks");
    }
}

public class Test {
    public static void main(String[] args) {
        new Dog().sound();
    }
}
```

🟢 **Output:**

```
nginx
CopyEdit
Animal sound
Dog barks
```

✅ c) To Access Parent Class Variables

If subclass and superclass have variables with the same name, super.variableName is used to access the parent class variable.

📘 **Example:**

```
java
CopyEdit
class Animal {
    String type = "Animal";
}

class Dog extends Animal {
    String type = "Dog";
```

```
    void printType() {
        System.out.println(type);       // Dog
        System.out.println(super.type);  // Animal
    }
}

public class Test {
    public static void main(String[] args) {
        new Dog().printType();
    }
}
```

🟢 **Output:**

```
nginx
CopyEdit
Dog
Animal
```

✅ **Summary Table**

Usage of super	Purpose	Syntax Example
Call parent constructor	Initialize superclass	super();
Access parent method	When overridden	super.method();
Access parent variable	When same name exists in subclass	super.variable;

🔁 **Superclass Reference vs super Keyword**

Feature	Superclass Reference	super Keyword
Refers to	Object of subclass	Parent class
Used for	Polymorphism	Constructor, method, variable access
Syntax Example	Animal a = new Dog();	super.sound(); or super();
Method dispatch	Dynamic (runtime polymorphism)	Static call to parent method

Constructor call Sequence

🔷 What is a Constructor?

A **constructor** is a special method that is **called automatically when an object is created**. Its purpose is to **initialize** the object.

🔷 Constructor Call Sequence in Inheritance

When you create an object of a **subclass**, **constructors of parent classes are automatically called first**.

✅ Rule:

When a subclass object is created:

1. Java calls the **constructor of the topmost superclass** first.
2. Then it calls the constructors **down the inheritance chain** to the subclass.
3. This happens **even if you don't explicitly call super()** — Java inserts it automatically.

📘 Example: Constructor Call Sequence

java

CopyEdit

```
class A {
  A() {
    System.out.println("Constructor A");
  }
}

class B extends A {
  B() {
```

```
        System.out.println("Constructor B");
    }
}

class C extends B {
    C() {
        System.out.println("Constructor C");
    }
}

public class Test {
    public static void main(String[] args) {
        C obj = new C();  // Object of subclass C
    }
}
```

🟢 **Output:**

```
css
CopyEdit
Constructor A
Constructor B
Constructor C
```

🔎 **Explanation:**

- C is a subclass of B, which is a subclass of A.
- When new C() is called:
    - Java automatically calls A() → then B() → then C()
- Even though super() is not written explicitly, Java calls it behind the scenes.

🔶 Using super() Explicitly

You can use super() to **manually call** the superclass constructor.

📘 Example with super():

java

CopyEdit

```
class A {
   A() {
      System.out.println("Constructor A");
   }
}

class B extends A {
   B() {
      super();  // optional, Java adds it if missing
      System.out.println("Constructor B");
   }
}

class C extends B {
   C() {
      super();  // optional
      System.out.println("Constructor C");
   }
}
```

🔷 Constructor Overloading + Sequence

If a superclass has **multiple constructors**, you can choose which one to call using super(arguments).

📘 **Example with Parameters:**

java

CopyEdit

```
class A {
   A(int x) {
      System.out.println("Constructor A with value: " + x);
   }
}

class B extends A {
   B() {
      super(10);  // calls A(int x)
      System.out.println("Constructor B");
   }
}

public class Test {
   public static void main(String[] args) {
      B obj = new B();
   }
}
```

🟢 **Output:**

less

CopyEdit

```
Constructor A with value: 10
Constructor B
```

🧠 **Important Points to Remember**

Point	Explanation
Constructor calls start from superclass	Parent class constructors are always called before child class constructors
Default super() is inserted	If not explicitly written, Java adds super() as the first line
Constructor must call superclass constructor first	super() must be the **first statement** in constructor
Only one constructor is called per class	No constructor is called twice during object creation
Constructors are not inherited	But parent constructors are always invoked

---

✅ **Summary Table**

Concept	Rule/Behavior
Object of subclass created	Superclass constructor is called first
super()	Calls parent class constructor
Implicit call	Java adds super() if not written
Constructor order	Top → Bottom (from parent to child)
Parameterized constructor	Use super(args) to call parent constructor with arguments

Method Overriding

🔷 What is Method Overriding?

Method Overriding in Java means **redefining a method in the subclass** that is already defined in the superclass.

In other words:

When a subclass provides its **own version** of a method that is already present in its parent class.

✅ Conditions for Method Overriding:

1. The method in the **subclass must have the same name, return type, and parameters** as in the superclass.
2. The method in the **superclass must not be private, static, or final**.
3. The subclass method must have **equal or higher access level** than the superclass method.
4. It occurs in **inheritance**.

📘 Basic Example of Method Overriding

java

CopyEdit

```
class Animal {
  void sound() {
    System.out.println("Animal makes a sound");
  }
}

class Dog extends Animal {
  @Override
```

```
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Calls overridden method
    }
}
```

🟢 **Output:**

```
nginx
CopyEdit
Dog barks
```

🔍 **Why Use Method Overriding?**

- To achieve **runtime polymorphism**
- To give **specific behavior** to subclass methods
- To extend or customize the behavior of a parent class

📌 **Example with Superclass Reference (Polymorphism)**

```
java
CopyEdit
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
```

```
}

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

public class Test {
    public static void main(String[] args) {
        Animal a = new Cat();  // superclass reference
        a.sound();  // Calls Cat's version (runtime polymorphism)
    }
}
```

🟢 **Output:**

```
nginx
CopyEdit
Cat meows
```

🔶 **Using super to Call Superclass Method**

```
java
CopyEdit
class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}
```

```
class Car extends Vehicle {
    @Override
    void run() {
        super.run();  // call parent method
        System.out.println("Car is running");
    }
}

public class Test {
    public static void main(String[] args) {
        Car c = new Car();
        c.run();
    }
}
```

🟢 **Output:**

```
arduino
CopyEdit
Vehicle is running
Car is running
```

❌ **Method That Cannot Be Overridden**

Modifier	Can it be Overridden?
private	❌ No — not visible in subclass
static	❌ No — method hiding, not overriding
final	❌ No — cannot be changed

Modifier	Can it be Overridden?
constructor	❌ No — constructors are never inherited

✅ **Key Points Summary**

Feature	Explanation
Same method name	Must match exactly with the superclass
Same parameters	Must have same type, number, and order
Same return type	Or covariant return (subtype)
Inheritance needed	Must be a subclass
Access modifier	Should be same or more accessible
Runtime behavior	Supports runtime polymorphism
Annotation	@Override helps catch errors

✅ **Real-life Analogy:**

Think of a **Printer**:

- Superclass: Printer has a print() method that prints basic text.
- Subclass: PhotoPrinter overrides print() to print photos instead.

🧠 **Interview Tip:**

- Method Overloading = Compile time
- Method Overriding = Runtime (**polymorphism**)

Let me know if you'd like:

Dynamic method dispatch

🔷 What is Dynamic Method Dispatch in Java?

Dynamic Method Dispatch is the process in Java where a **superclass reference variable calls overridden methods** in the **subclass**, and the **method is chosen at runtime** — **not** during compilation.

👉 It is the **basis for runtime polymorphism** in Java.

✅ Definition:

Dynamic Method Dispatch allows Java to **decide which version of an overridden method to call based on the actual object**, not the reference type — at **runtime**.

📌 Key Concept:

java

CopyEdit

```
Superclass ref = new Subclass();  // upcasting
ref.overriddenMethod();         // resolved at runtime
```

Even though ref is of superclass type, if the method is overridden in the subclass, the **subclass version** is executed.

📘 Example of Dynamic Method Dispatch

java

CopyEdit

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}
```

```
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

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

public class Test {
    public static void main(String[] args) {
        Animal a;  // reference of superclass

        a = new Dog();
        a.sound();  // Output: Dog barks

        a = new Cat();
        a.sound();  // Output: Cat meows
    }
}
```

🟢 **Output:**

nginx

CopyEdit

Dog barks

Cat meows

Even though the reference type is Animal, the method is decided **at runtime** based on the object (Dog or Cat).

🔄 Why is it Called "Dynamic"?

Because the **method call is resolved during program execution**, not at compile time.

That's why it's also called:

- **Late binding**
- **Runtime polymorphism**

🧠 How it Works Internally

1. Compiler checks that sound() exists in Animal.
2. At **runtime**, Java checks the **actual object type** (e.g., Dog or Cat).
3. It calls the appropriate sound() method.

✅ Real-Life Analogy

Imagine you have a **remote control (reference)** that can control different devices:

java

CopyEdit

Remote ref = new TV(); // Calls TV's behavior

ref = new AC(); // Now calls AC's behavior

Even though you use the same remote (reference), the behavior depends on the **actual device (object)**.

❌ What Can't Be Dispatched Dynamically?

- **Static methods** (they use compile-time binding)
- **Private methods** (not inherited)
- **Constructors**
- **Final methods** (can't be overridden)

✅ **Summary Table**

Feature	Description
Type of polymorphism	Runtime polymorphism
When method is selected	At **runtime**, based on object type
Method must be	**Overridden** method
Reference type	Superclass
Object type	Subclass
Benefit	Flexibility and extensibility in OOP

📌 **Important Interview Tip**

- **Overriding + Superclass Reference + Subclass Object** → Dynamic Method Dispatch
- Enables polymorphic behavior in Java

Abstract

🔷 What is abstract in Java?

In Java, abstract is a **keyword** used with **classes and methods** that are **incomplete** and meant to be **extended** or **implemented by subclasses**.

It is part of **abstraction**, one of the four main OOP principles (Abstraction, Encapsulation, Inheritance, Polymorphism).

🔶 Why use abstract?

- To **define a common structure** without implementation.
- To **force subclasses** to provide their own implementation.

✅ Types of Abstract Usage:

1. Abstract Class

- Declared using the abstract keyword.
- Cannot be instantiated (you **cannot create objects** of an abstract class).
- Can have:
    - Abstract methods (without body)
    - Concrete methods (with body)
    - Constructors, variables

2. Abstract Method

- A method without a body.
- Ends with a semicolon ;
- Must be **overridden** in the subclass.

📘 Example of Abstract Class and Method

java

CopyEdit

```
abstract class Animal {
   abstract void sound();  // abstract method

   void sleep() {        // concrete method
      System.out.println("Sleeping...");
   }
}

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

public class Test {
   public static void main(String[] args) {
      // Animal a = new Animal(); ❌ Cannot create object of abstract class
      Dog d = new Dog();
      d.sound();   // Overridden method
      d.sleep();   // Inherited concrete method
   }
}
```

🟢 **Output:**

nginx

CopyEdit

Dog barks

Sleeping...

✅ Rules of Abstract in Java

Rule	Description
Cannot create objects of abstract class	new AbstractClass() ❌ Not allowed
Can have constructors	Yes, used when subclass is instantiated
Can have final/static methods	Yes, but they **cannot be abstract**
Must be extended	By a concrete class that overrides all abstract methods
Can be partially implemented	Abstract class can have some concrete methods

---

🔍 Real-life Analogy

Imagine a class called Shape:

- You **can't draw a generic shape**, but you can **draw a circle, square, etc.**
- So, Shape can be abstract, and Circle, Square can implement its methods.

📌 Another Example: Abstract + Polymorphism

java

CopyEdit

```
abstract class Shape {
   abstract void draw();
}

class Circle extends Shape {
   void draw() {
      System.out.println("Drawing Circle");
```

```
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Drawing Square");
    }
}

public class Test {
    public static void main(String[] args) {
        Shape s;

        s = new Circle();
        s.draw();  // Output: Drawing Circle

        s = new Square();
        s.draw();  // Output: Drawing Square
    }
}
```

❌ **Difference Between abstract and interface**

Feature	Abstract Class	Interface
Can have concrete methods	✅ Yes	✅ (from Java 8 onwards)
Constructor	✅ Yes	❌ No

Feature	Abstract Class	Interface
Multiple inheritance	❌ No (only one abstract class)	✅ Yes (can implement multiple)
Use	Partial abstraction	Full abstraction

✅ **Summary**

Feature	Explanation
abstract class	A class that **cannot be instantiated** and may contain abstract methods
abstract method	A method with **no body**, to be defined in subclass
Must be inherited	A subclass must provide implementation for abstract methods
Can have variables/methods	Yes, even concrete ones
Helps in	Achieving **abstraction and polymorphism**

Object class

🔷 What is the Object class in Java?

In Java, the Object class is the **root (superclass) of all classes**.
Every class in Java **directly or indirectly inherits** from the Object class.

✅ Key Point:

All Java classes are subclasses of the Object class, either explicitly or implicitly.

🔷 Syntax

java

CopyEdit

```
class MyClass {
   // Implicitly extends Object class
}
```

Even if you don't write extends Object, the class still inherits from it.

📦 Package

The Object class is defined in the **java.lang** package, which is automatically imported in every Java program.

🧠 Why is the Object class important?

- Provides **common methods** that every Java class can use.
- Enables **polymorphism** — you can refer to any object using Object reference.
- Acts as a **universal type** in collections, arrays, etc.

✅ Common Methods of Object Class

Method	Description
toString()	Returns string representation of the object
equals(Object obj)	Checks if two objects are "equal"
hashCode()	Returns hash code of the object
getClass()	Returns the runtime class of the object
clone()	Creates a copy (shallow) of the object
finalize()	Called by garbage collector before object is destroyed
wait(), notify(), notifyAll()	For thread communication

📘 Example: Using toString() and equals()

java

CopyEdit

```
class Student {
  int id;
  String name;

  Student(int id, String name) {
    this.id = id;
    this.name = name;
  }

  @Override
  public String toString() {
    return "Student[id=" + id + ", name=" + name + "]";
  }

  @Override
```

```
    public boolean equals(Object obj) {
        Student s = (Student) obj;
        return this.id == s.id && this.name.equals(s.name);
    }
}

public class Test {
    public static void main(String[] args) {
        Student s1 = new Student(101, "John");
        Student s2 = new Student(101, "John");

        System.out.println(s1);            // Uses toString()
        System.out.println(s1.equals(s2));    // Uses overridden equals()
    }
}
```

🟢 **Output:**

```
bash
CopyEdit
Student[id=101, name=John]
true
```

🔍 **Explanation**

- toString() is used when you print an object — we overrode it for readable output.
- equals() is overridden to **compare content**, not just reference (default behavior).

📌 **Default Behavior of Object Methods**

Method	Default behavior (if not overridden)
toString()	Prints class name + @ + hash code (hexadecimal)
equals()	Compares **references** (not actual content)
hashCode()	Returns a unique integer per object (used in HashMap etc.)

🧠 Real-life Analogy

Think of the Object class as a **basic template** — every other class inherits from it and customizes it.

✅ Summary

Feature	Description
Root class	All Java classes extend Object
Package	java.lang.Object
Cannot be avoided	Even custom classes inherit it
Common methods	toString(), equals(), hashCode(), etc.
Supports polymorphism	Yes — any object can be referred using Object type

---

📘 Bonus: Object Reference Example

java

CopyEdit

```
public class Test {
    public static void main(String[] args) {
        Object obj = "Hello";  // Can hold any object
        System.out.println(obj);  // Output: Hello (calls String.toString())
    }
}
```

Package

Sure! Here's a **detailed and easy-to-understand explanation** of **Packages in Java** covering:

- Defining a package
- Finding packages and CLASSPATH
- Access protection in packages
- Importing packages

with examples for each.

📦 Packages in Java

A **package** is a way to **group related classes and interfaces together**.
It helps in **organizing your code** and **avoiding name conflicts**.

1️⃣ Defining a Package

- You define a package using the package keyword at the **very top** of your Java source file.
- It must be the first statement (except comments).
- Syntax:

java

CopyEdit

package packagename;

Example: Defining a package

java

CopyEdit

package mypackage;

```
public class Hello {
  public void sayHello() {
    System.out.println("Hello from mypackage!");
  }
}
```

Here, mypackage is the package name.

2️⃣ Finding Packages and CLASSPATH

- **CLASSPATH** is an environment variable or parameter which tells Java **where to look for classes and packages** during compilation and execution.
- By default, the current directory (.) is included.
- If you create packages, your directory structure **must match the package structure**.

Example:

For package mypackage;, the .java file must be inside folder:

CopyEdit

```
project_root/
  mypackage/
    Hello.java
```

Compiling and running with packages:

- Compile:

bash

CopyEdit

```
javac mypackage/Hello.java
```

- Run (from project_root):

bash

CopyEdit

java mypackage.Hello

Setting CLASSPATH:

If your classes are in a different folder, you must set CLASSPATH to point there:

bash

CopyEdit

export CLASSPATH=/path/to/classes

3️⃣ Access Protection in Packages

Java provides access modifiers to control **visibility of classes, methods, and variables** inside packages:

Modifier	**Accessible In**
public	Everywhere
protected	Same package + subclasses (even outside package)
default (no modifier)	Same package only (package-private)
private	Within the same class only

Example of access protection:

java

CopyEdit

```
package mypackage;


public class A {
   public int a = 10;
   protected int b = 20;
   int c = 30;  // default access
```

```
  private int d = 40;
}
```

- a accessible anywhere.
- b accessible in same package and subclasses.
- c accessible only inside mypackage.
- d accessible only inside class A.

4️⃣ Importing Packages

To use classes from other packages, use the import keyword.

- Import a specific class:

```
java
CopyEdit
import mypackage.Hello;
```

- Import all classes in a package:

```
java
CopyEdit
import mypackage.*;
```

Example of Import:

File: mypackage/Hello.java

```
java
CopyEdit
package mypackage;

public class Hello {
  public void sayHello() {
    System.out.println("Hello from mypackage!");
  }
```

}

File: Test.java (default package or another package)

java

CopyEdit

```
import mypackage.Hello;


public class Test {
   public static void main(String[] args) {
      Hello h = new Hello();
      h.sayHello();
   }
}
```

Important:

- You **don't** need to import classes from java.lang package (like String, System).
- Use **fully qualified name** instead of import, if you want:

java

CopyEdit

```
mypackage.Hello h = new mypackage.Hello();
```

Summary Table

Concept	Explanation
Package	Group of related classes/interfaces
Defining package	package packagename; at top of source file
Directory structure	Must match package name (folder hierarchy)
CLASSPATH	Tells JVM where to find packages/classes
Access protection	Controls visibility (public, protected, default, private)

Concept	Explanation
Importing packages	Use import to use classes from other packages

Interface

Sure! Here's a **detailed, simple explanation** of **Interfaces in Java** covering:

- Defining interfaces
- Implementing interfaces
- Nesting interfaces
- Applying interfaces

with examples for each.

🔷 Interfaces in Java

An **interface** is a **reference type** in Java, similar to a class, but it can contain only **abstract methods** (before Java 8), **default methods**, **static methods**, and **constants**.

1️⃣ Defining an Interface

- Use the interface keyword.
- Methods inside are **abstract by default** (no body) unless default or static.
- Variables are public static final by default (constants).

Syntax:

java

CopyEdit

```
interface InterfaceName {
   // constant variables
   int MAX_VALUE = 100;

   // abstract method (no body)
   void method1();
```

```
    // from Java 8 onwards: default method (with body)
    default void method2() {
        System.out.println("Default implementation");
    }
}
```

Example:

java

CopyEdit

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

2️⃣ Implementing an Interface

- A class **implements** an interface using the implements keyword.
- It **must override all abstract methods** of the interface (unless the class is abstract).
- A class can implement **multiple interfaces** (multiple inheritance support).

Example:

java

CopyEdit

```
interface Animal {
    void sound();
}

class Dog implements Animal {
    public void sound() {
```

```
    System.out.println("Dog barks");
  }
}

public class Test {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.sound();
  }
}
```

Output:

nginx

CopyEdit

Dog barks

3️⃣ Nesting Interfaces

- An interface can be **nested inside a class or another interface**.
- Nested interfaces are by default static.
- Useful for grouping related interfaces or inner interface APIs.

Example: Nested Interface inside a class

java

CopyEdit

```
class OuterClass {
  interface InnerInterface {
    void display();
  }
}
```

```
class Implementer implements OuterClass.InnerInterface {
   public void display() {
      System.out.println("Inner Interface method implemented");
   }
}

public class Test {
   public static void main(String[] args) {
      Implementer obj = new Implementer();
      obj.display();
   }
}
```

4️⃣ Applying Interfaces

Why use interfaces?

- To achieve **full abstraction** (only method signatures).
- To support **multiple inheritance** (Java classes cannot inherit multiple classes but can implement multiple interfaces).
- To define **contracts** that implementing classes must follow.
- To enable **loose coupling** and improve flexibility.

Multiple interfaces implementation example:

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

```
interface Showable {
   void show();
}

class Printer implements Printable, Showable {
   public void print() {
      System.out.println("Printing...");
   }
   public void show() {
      System.out.println("Showing...");
   }
}

public class Test {
   public static void main(String[] args) {
      Printer p = new Printer();
      p.print();
      p.show();
   }
}
```

Output:

CopyEdit

Printing...

Showing...

Summary Table:

Concept	Description
Interface	A reference type with abstract methods
Define	Use interface keyword
Implement	Class uses implements and overrides methods
Methods	Abstract by default; from Java 8 can have default and static methods
Multiple Inheritance	Supported by implementing multiple interfaces
Nested Interfaces	Interfaces can be declared inside classes or interfaces
Purpose	Define contracts, support abstraction and polymorphism

🔷 Variables in Interfaces

1️⃣ What kinds of variables are allowed in interfaces?

- All variables declared in an interface are implicitly:
    - public
    - static
    - final

This means:

- They are **constants** (values cannot be changed).
- They belong to the interface itself (shared by all implementing classes).
- You **must initialize them** when declared.

Example:

java

CopyEdit

```
interface MyInterface {
```

```
    int MAX_VALUE = 100;  // implicitly public static final


    void display();
}
```

Here, MAX_VALUE is a constant.

Using the variable:

java

CopyEdit

```
class TestClass implements MyInterface {
    public void display() {
        System.out.println("Max value is: " + MAX_VALUE);
    }
}

public class Test {
    public static void main(String[] args) {
        TestClass obj = new TestClass();
        obj.display();  // Output: Max value is: 100
    }
}
```

Important points:

- You **cannot** declare instance variables (non-final, non-static) in interfaces.
- You **cannot** change the value of the variables after initialization.
- You can access interface variables using InterfaceName.VARIABLE or directly inside implementing classes.

🔷 Extending Interfaces

2️⃣ Can interfaces extend other interfaces?

- Yes, interfaces **can extend one or more interfaces**.
- This is similar to **inheritance** for classes but applies to interfaces.
- Supports **multiple inheritance** for interfaces.

Syntax:

java

CopyEdit

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

interface InterfaceB extends InterfaceA {
   void methodB();
}
```

Here, InterfaceB inherits methodA() from InterfaceA and declares its own method methodB().

Example with extending interfaces:

java

CopyEdit

```
interface Vehicle {
   void move();
}

interface FourWheeler extends Vehicle {
```

```
    void openTrunk();
}

class Car implements FourWheeler {
    public void move() {
        System.out.println("Car is moving");
    }
    public void openTrunk() {
        System.out.println("Opening car trunk");
    }
}

public class Test {
    public static void main(String[] args) {
        Car c = new Car();
        c.move();       // Output: Car is moving
        c.openTrunk();  // Output: Opening car trunk
    }
}
```

Extending multiple interfaces:

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

interface Showable {
```

```
    void show();
}

interface Readable extends Printable, Showable {
    void read();
}

class Document implements Readable {
    public void print() {
        System.out.println("Printing document");
    }
    public void show() {
        System.out.println("Showing document");
    }
    public void read() {
        System.out.println("Reading document");
    }
}

public class Test {
    public static void main(String[] args) {
        Document doc = new Document();
        doc.print();
        doc.show();
        doc.read();
    }
}
```

Summary Table

Feature	Description
Interface variables	Always public static final (constants)
Variables must be initialized	Yes
Extending interfaces	An interface can extend one or more interfaces
Implementing class	Must override all methods from all extended interfaces
Multiple inheritance	Supported via interface extension

Exception Handeling

Absolutely! Here's a **detailed explanation** of the **instanceof operator**, **exceptions in Java** including types, handling keywords, nested try, built-in and custom exceptions — all with clear examples for your exam.

1️⃣ instanceof Operator

What is instanceof?

- It is a **binary operator** used to test whether an object is an instance of a specific class or interface.
- Returns true or false.

Syntax:

java

CopyEdit

object instanceof ClassName

Example:

java

CopyEdit

class Animal {}

class Dog extends Animal {}

public class Test {

  public static void main(String[] args) {

    Animal a = new Dog();

```
        if (a instanceof Dog) {
            System.out.println("a is a Dog");
        } else {
            System.out.println("a is not a Dog");
        }
    }
}
```

Output:

```
css
CopyEdit
a is a Dog
```

2️⃣ Exception Handling in Java

What is an Exception?

- An **exception** is an event that disrupts normal program flow.
- Exceptions are objects that describe an error or unexpected event.

Types of Exceptions:

Type	Description	Example
Checked Exception	Must be handled or declared (compile-time)	IOException, SQLException
Unchecked Exception	Runtime exceptions, don't require explicit handling	NullPointerException, ArithmeticException
Error	Serious system errors, not meant to be caught	OutOfMemoryError, StackOverflowError

---

3️⃣ Uncaught Exceptions

- If an exception is **not caught**, the program terminates abnormally and prints the stack trace.
- Example: dividing by zero without handling.

4️⃣ Keywords for Exception Handling

---

try block

- Code that may throw an exception is placed here.

catch block

- Handles the exception thrown in try.

finally block

- Code inside it executes **always** whether exception occurs or not.

throw statement

- Used to **explicitly throw** an exception.

throws clause

- Used in method declaration to **declare possible exceptions** the method may throw.

5️⃣ Example demonstrating try, catch, throw, throws, finally

java

CopyEdit

```
import java.io.*;


class Test {
  // Method declaring IOException
  static void readFile() throws IOException {
    FileReader file = new FileReader("nonexistent.txt");
    BufferedReader fileInput = new BufferedReader(file);
```

```
        throw new IOException("Forced IOException");  // explicit throw
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Caught IOException: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

Output:

vbnet

CopyEdit

Caught IOException: Forced IOException

Finally block executed

6️⃣ Multiple catch Clauses

- You can have multiple catch blocks to handle different exceptions.

java

CopyEdit

```
try {
    int a = 10/0;
} catch (ArithmeticException e) {
    System.out.println("Arithmetic Exception caught");
```

```
} catch (Exception e) {
   System.out.println("General Exception caught");
}
```

7️⃣ Nested try Statements

- You can have try-catch blocks inside other try or catch blocks.

```
java
CopyEdit
try {
   try {
      int arr[] = new int[5];
      arr[6] = 10;  // ArrayIndexOutOfBoundsException
   } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Inner catch: " + e);
   }
   int a = 10/0;  // ArithmeticException
} catch (ArithmeticException e) {
   System.out.println("Outer catch: " + e);
}
```

8️⃣ Built-in Exceptions (Common ones)

Exception Name	Cause
ArithmeticException	Divide by zero
NullPointerException	Accessing null object
ArrayIndexOutOfBoundsException	Invalid array index
ClassCastException	Invalid casting
IOException	Input/Output failure

Exception Name	Cause
NumberFormatException	Invalid number conversion

9️⃣ Creating Custom Exceptions

Steps to create custom exceptions:

- Extend the Exception class (checked exception) or RuntimeException (unchecked).
- Provide constructors.

Example:

java

CopyEdit

```
class MyException extends Exception {
  public MyException(String message) {
    super(message);
  }
}

public class Test {
  static void checkNumber(int num) throws MyException {
    if (num < 1) {
      throw new MyException("Number is less than 1");
    }
    System.out.println("Number is valid");
  }

  public static void main(String[] args) {
```

```
        try {
            checkNumber(0);
        } catch (MyException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }
}
```

Output:

vbnet

CopyEdit

Caught custom exception: Number is less than 1

Summary Table:

Keyword	Purpose
try	Block of code to monitor for exceptions
catch	Handle exception thrown from try block
finally	Code that always runs after try/catch
throw	Used to explicitly throw an exception
throws	Declares exceptions a method might throw
instanceof	Tests whether an object is an instance of a class/interface

Stream

Sure! Here's a detailed explanation of **Managing I/O in Java** covering:

- Streams
- Byte Streams and Character Streams
- Predefined Streams
- Reading Console Input
- Writing Console Output
- PrintWriter class

with simple examples.

1️⃣ Streams in Java I/O

- A **stream** is a sequence of data.
- It represents **input or output** of data.
- Two main types:
    - **InputStream** (reading data)
    - **OutputStream** (writing data)

2️⃣ Byte Streams vs Character Streams

Aspect	**Byte Streams**	**Character Streams**
Data unit	8-bit bytes	16-bit Unicode characters
Classes start with	InputStream / OutputStream	Reader / Writer
Used for	Binary data (images, files)	Text data (characters, strings)
Examples	FileInputStream, FileOutputStream	FileReader, FileWriter

---

Example of Byte Stream:

java

CopyEdit

```
import java.io.*;

public class ByteStreamExample {
   public static void main(String[] args) throws IOException {
      FileInputStream fin = new FileInputStream("input.txt");
      int i;
      while ((i = fin.read()) != -1) {
         System.out.print((char) i);
      }
      fin.close();
   }
}
```

Example of Character Stream:

java

CopyEdit

```
import java.io.*;

public class CharStreamExample {
   public static void main(String[] args) throws IOException {
      FileReader fr = new FileReader("input.txt");
      int i;
      while ((i = fr.read()) != -1) {
         System.out.print((char) i);
      }
      fr.close();
   }
```

```
}
```

3️⃣ Predefined Streams

Java provides three standard predefined streams:

Stream	Description
System.in	Standard input stream (console input)
System.out	Standard output stream (console output)
System.err	Standard error output stream

4️⃣ Reading Console Input

Using System.in and InputStreamReader + BufferedReader:

java

CopyEdit

```
import java.io.*;

public class ConsoleInputExample {
   public static void main(String[] args) throws IOException {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);

      System.out.print("Enter your name: ");
      String name = br.readLine();

      System.out.println("Hello, " + name);
   }
}
```

Using Scanner class (more modern):

java

CopyEdit

```
import java.util.Scanner;

public class ScannerExample {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      System.out.print("Enter your age: ");
      int age = sc.nextInt();

      System.out.println("Your age is: " + age);
      sc.close();
   }
}
```

5️⃣ Writing Console Output

Using System.out.println()

java

CopyEdit

```
System.out.println("Hello World");
```

Using PrintWriter

- PrintWriter class can wrap any OutputStream (including System.out).
- Provides convenient methods to print formatted output.
- Can autoflush.

Example: Writing output with PrintWriter

java

CopyEdit

```
import java.io.*;

public class PrintWriterExample {
   public static void main(String[] args) throws IOException {
      PrintWriter pw = new PrintWriter(System.out, true); // autoflush enabled

      pw.println("This is a line printed using PrintWriter");
      pw.printf("Formatted number: %.2f\n", 12.3456);

      pw.close();
   }
}
```

Summary Table:

Concept	Description
Stream	Sequence of data for I/O
Byte Streams	InputStream/OutputStream, work with bytes
Character Streams	Reader/Writer, work with characters
Predefined streams	System.in (input), System.out (output), System.err (error)
Console Input	Use InputStreamReader+BufferedReader or Scanner
Console Output	Use System.out or PrintWriter