CHAPTER NO: 03
Inheritance, Interfaces and Packages {20 MARKS}

3.1 Inheritance Basics,

3.2 Member Access and Inheritance,

3.3 A Superclass Variable Can Reference a Subclass Object

3.4 Use of super keyword

3.5 Creating a Multilevel Hierarchy,

3.6 Method Overriding, Dynamic Method Dispatch, Abstract classes

3.7 Defining an Interface, Implementing Interfaces, applying interfaces, Variables in Interfaces,

3.8 Implementing Multiple Inheritance (Multiple Inheritance), Interfaces Can Be Extended

3.9 Packages, defining a Package, Finding Packages and CLASSPATH,

3.10 Access Protection, Importing Packages

3.1 Inheritance Basics:

What is Inheritance?

Inheritance is one of the core concepts of **Object-Oriented Programming (OOP)** that allows a new class to acquire (inherit) properties and behaviors (fields and methods) from an existing class.

- The existing class is called the **parent class** or **superclass**.
- The new class is called the **child class** or **subclass**.

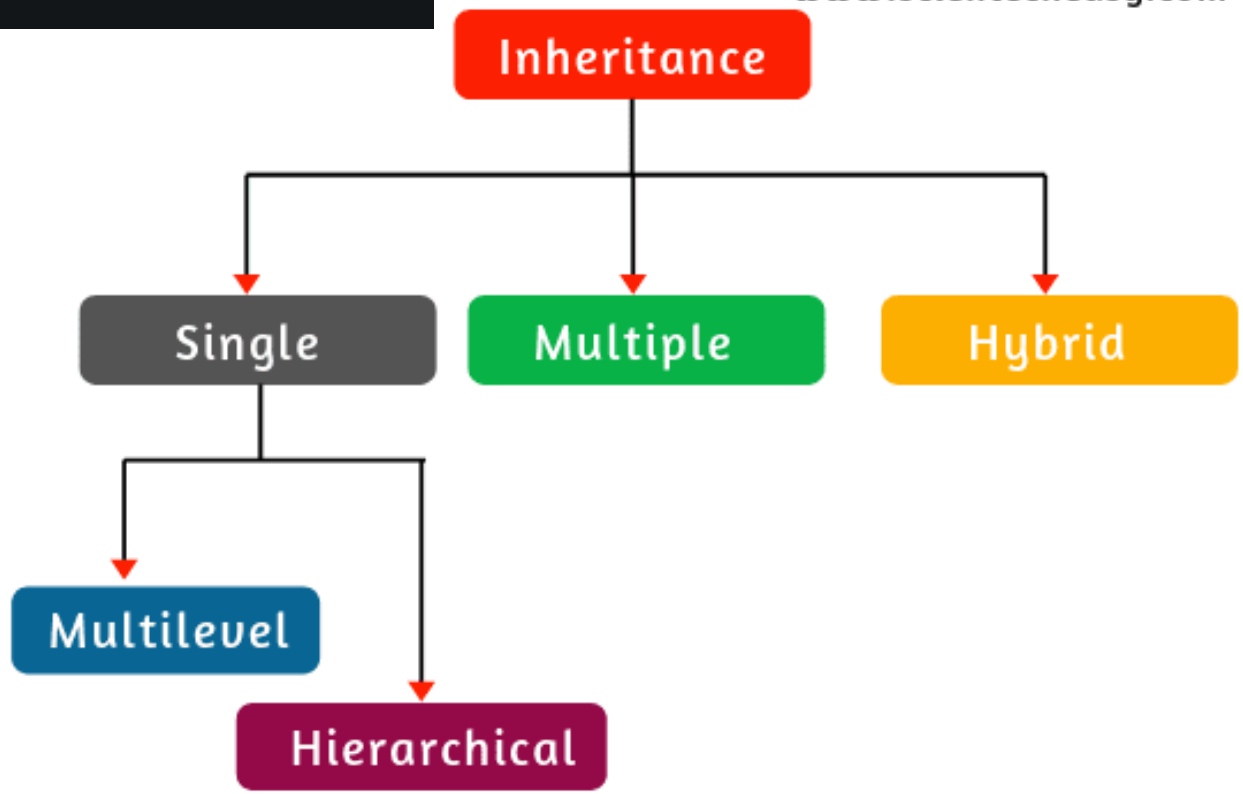
Why use Inheritance?

- **Code reuse**: Reuse existing code without rewriting it.
- **Extensibility**: Add new features or modify existing behavior.
- **Method overriding**: Modify or customize behavior of inherited methods.
- **Hierarchy**: Models real-world relationships.

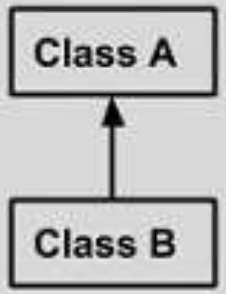
Types of Inheritance in Java

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

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

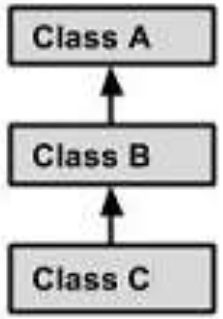


Single Inheritance



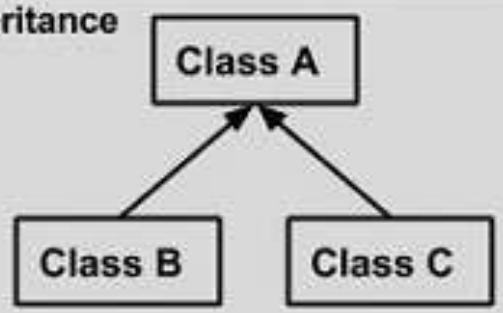

```
public class A {
   .......
}
public class B extends A {
   .........
}
```

Multi Level Inheritance



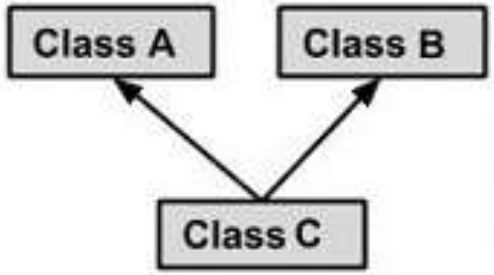

```
public class A { ..................}

public class B extends A {.....................}

public class C extends  B {..................... }
```

Hierarchical Inheritance



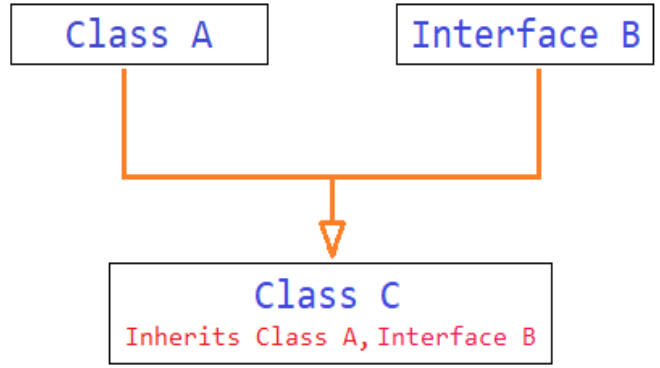

```
public class A { ..................}

public class B extends A {.....................}

public class C extends A {..................... }
```

Multiple Inheritance




```
public class A { ..................}

public class B {.....................}

public class C extends  A,B {
   .....................
} // Java does not support mutiple Inheritance
```

JAVA MULTIPLE INHERITANCE



Syntax

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

class ChildClass extends ParentClass {
    // additional fields and methods
}
```

`extends` keyword is used to inherit from a superclass.

Example

```java
// Parent class
class Student {
    String name;
    int rollNumber;

    void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Roll Number: " + rollNumber);
    }
}

// Child class
class ScienceStudent extends Student {
    String stream = "Science";

    void displayStream() {
        System.out.println("Stream: " + stream);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // Creating object for Akhilesh
        ScienceStudent akhilesh = new ScienceStudent();
        akhilesh.name = "Akhilesh";
        akhilesh.rollNumber = 101;

        // Creating object for Ankit
        ScienceStudent ankit = new ScienceStudent();
        ankit.name = "Ankit";
        ankit.rollNumber = 102;

        // Display Akhilesh's details
        System.out.println("Details of Student 1:");
        akhilesh.displayDetails();
        akhilesh.displayStream();

        System.out.println();

        // Display Ankit's details
        System.out.println("Details of Student 2:");
        ankit.displayDetails();
        ankit.displayStream();
    }
}
```

Output:

```
Details of Student 1:
Name: Akhilesh
Roll Number: 101
Stream: Science

Details of Student 2:
Name: Ankit
Roll Number: 102
Stream: Science
```

3.2 Member Access and Inheritance:

When a class inherits another, the **access modifiers** (`private`, `default` (package-private), `protected`, and `public`) control which members (variables and methods) of the superclass can be accessed by the subclass.

Access Modifiers and Their Effect on Inheritance

Modifier	Accessible in Same Class	Subclass (Same Package)	Subclass (Different Package)	Other Classes (Same Package)	Other Classes (Different Package)
`private`	Yes	No	No	No	No
default (no modifier)	Yes	Yes	No	Yes	No
`protected`	Yes	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes	Yes

- **private** members are **not inherited** in a way that the subclass can directly access them.
- **default** members are accessible only within the same package.
- **protected** members are accessible in subclasses even if they are in different packages.
- **public** members are accessible everywhere.

Example Demonstrating Member Access and Inheritance

```java
// Parent class in package pkg1
package pkg1;

public class Parent {
    private int privateVar = 1;
    int defaultVar = 2;       // default (package-private)
    protected int protectedVar = 3;
    public int publicVar = 4;

    public void show() {
        System.out.println("privateVar = " + privateVar);
        System.out.println("defaultVar = " + defaultVar);
        System.out.println("protectedVar = " + protectedVar);
        System.out.println("publicVar = " + publicVar);
    }
}
```

```java
// Child class in same package pkg1
package pkg1;

public class ChildSamePackage extends Parent {
    public void display() {
        // System.out.println(privateVar); // Error: privateVar not accessible
        System.out.println("defaultVar = " + defaultVar);       // accessible
        System.out.println("protectedVar = " + protectedVar);   // accessible
        System.out.println("publicVar = " + publicVar);         // accessible
    }
}
```

```java
// Child class in different package pkg2
package pkg2;
import pkg1.Parent;

public class ChildDifferentPackage extends Parent {
    public void display() {
        // System.out.println(defaultVar);   // Error: defaultVar not accessible
        System.out.println("protectedVar = " + protectedVar);  // accessible because protected
        System.out.println("publicVar = " + publicVar);        // accessible
        // System.out.println(privateVar);   // Error: privateVar not accessible
    }
}
```

`privateVar` is **not accessible** outside the `Parent` class.

`defaultVar` is accessible only within the same package.

`protectedVar` is accessible in subclasses even if they are in different packages.

`publicVar` is accessible everywhere.

3.3 A Superclass Variable Can Reference a Subclass Object:

In Java, **a variable of a superclass type can hold a reference to an object of any subclass type**. This is called **polymorphism** — one of the core OOP features.

- This allows writing flexible and reusable code.
- The superclass reference can only access methods and fields declared in the superclass (or overridden in the subclass).
- To access subclass-specific methods, **downcasting** is needed.

Syntax:

```
Superclass obj = new Subclass();
```

Example:

```java
// Superclass
class Student {
    void display() {
        System.out.println("I am a student.");
    }
}

// Subclass
class SportsStudent extends Student {
    void display() {
        System.out.println("I am a sports student.");
    }

    void playSport() {
        System.out.println("I play football.");
    }
}
```

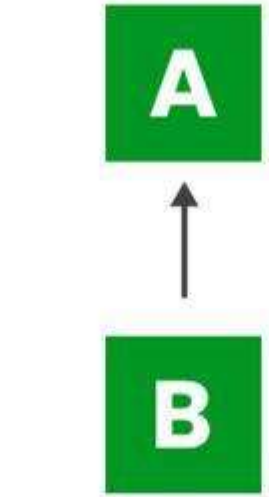
```java
public class Main {
    public static void main(String[] args) {
        // Superclass reference referring to subclass object
        Student s = new SportsStudent();

        s.display();  // Calls overridden method in SportsStudent

        // s.playSport();  // Error! playSport() is not a method of Student

        // Downcasting to call subclass-specific method
        ((SportsStudent) s).playSport();
    }
}
```

Output:

```
I am a sports student.
I play football.
```

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. In the below figure, 'A' is a parent class and 'B' is a child class. The class 'B' inherits all the properties of the class 'A'.



Single Inheritance

```java
// Base class
class Student {
    protected int age = 35;                 // updated age
    protected String name = "akhil";        // updated name

    void getStudent() {
        // Input nahi le rahe, method empty
    }

    void putStudent() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Derived class
class Test extends Student {
    private int m1 = 85;    // marks 1
    private int m2 = 90;    // marks 2

    void getMarks() {
        // Input nahi le rahe, method empty
    }
```

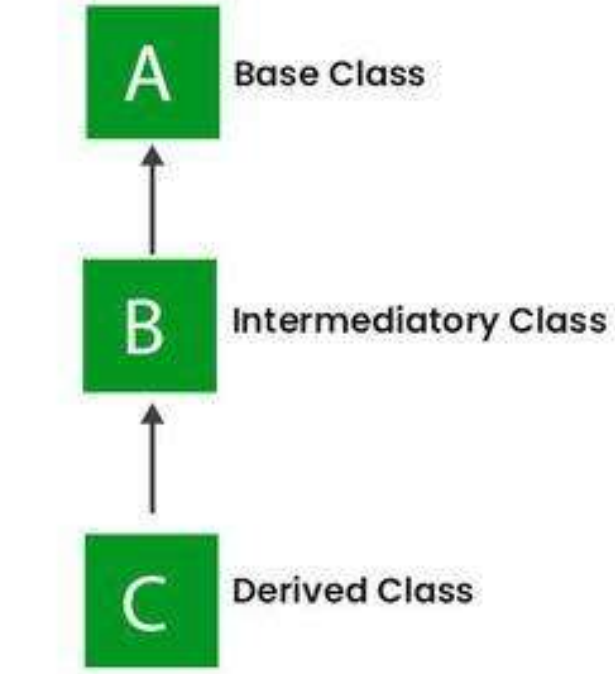
```
    void display() {
        double average = (m1 + m2) / 2.0;
        System.out.println("Average marks: " + average);
    }
}

public class Main {
    public static void main(String[] args) {
        Test t = new Test();

        t.putStudent();
        t.display();
    }
}
```

Output:

```
Name: akhil
Age: 35
Average marks: 87.5
```

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. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members if they are private.



Multilevel Inheritance

```java
// Base class
class Student {
    protected int age = 35;
    protected String name = "akhil";

    void getStudent() {
        // Empty, values khud se initialized hain
    }

    void putStudent() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Derived class from Student
class Test extends Student {
    protected int m1 = 85;
    protected int m2 = 90;

    void average() {
        double avg = (m1 + m2) / 2.0;
        System.out.println("Average marks: " + avg);
    }
}
```

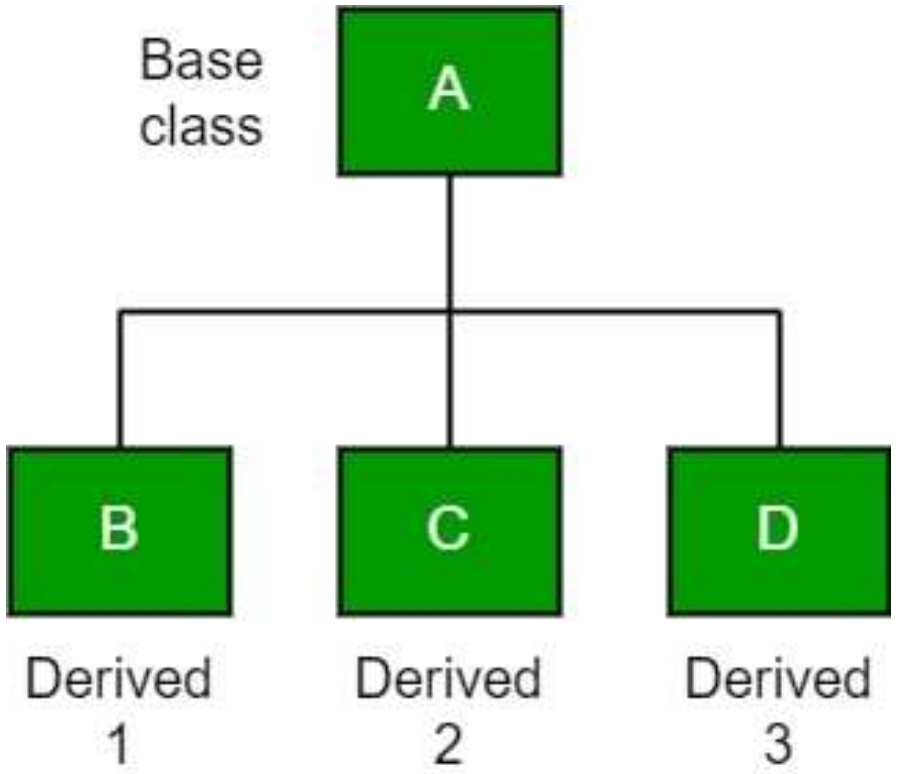
```java
// Derived class from Test (multilevel inheritance)
class Sports extends Test {
    private String sportsName = "Cricket";
    private int points = 50;

    void display() {
        average();  // Test class ka average method call
        System.out.println("Sports Name: " + sportsName);
        System.out.println("Points: " + points);
    }
}

public class Main {
    public static void main(String[] args) {
        Sports s = new Sports();

        s.putStudent();    // name aur age print karega
        s.display();       // marks average aur sports details print karega
    }
}
```

Output:

```
Name: akhil
Age: 35
Average marks: 87.5
Sports Name: Cricket
Points: 50
```

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



```java
// Base class
class A {
    void show() {
        System.out.println("This is the base class A");
    }
}

// Derived class 1
class B extends A {
    void display() {
        System.out.println("This is derived class B");
    }
}

// Derived class 2
class C extends A {
    void display() {
        System.out.println("This is derived class C");
    }
}

// Derived class 3
class D extends A {
    void display() {
        System.out.println("This is derived class D");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        B objB = new B();
        C objC = new C();
        D objD = new D();

        objB.show();       // Base class method
        objB.display();    // Derived class B method

        objC.show();       // Base class method
        objC.display();    // Derived class C method

        objD.show();       // Base class method
        objD.display();    // Derived class D method
    }
}
```

Output:

```
This is the base class A
This is derived class B
This is the base class A
This is derived class C
This is the base class A
This is derived class D
```

```java
// Base class
class Student {
    protected int age = 35;
    protected String name = "akhil";

    void putStudent() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Derived class 1
class Test extends Student {
    private int m1 = 85;
    private int m2 = 90;

    void average() {
        double avg = (m1 + m2) / 2.0;
        System.out.println("Average marks: " + avg);
    }
}
```

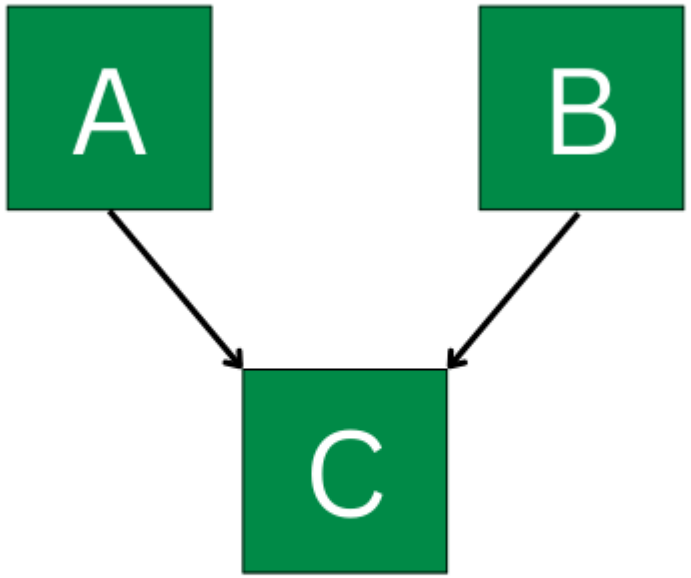
```java
// Derived class 2
class Sports extends Student {
    private String sportsName = "Cricket";
    private int points = 50;

    void display() {
        System.out.println("Sports Name: " + sportsName);
        System.out.println("Points: " + points);
    }
}

public class Main {
    public static void main(String[] args) {
        Test t = new Test();
        Sports s = new Sports();

        System.out.println("Test class details:");
        t.putStudent();
        t.average();

        System.out.println("\nSports class details:");
        s.putStudent();
        s.display();
    }
}
```

Output:

```
Test class details:
Name: akhil
Age: 35
Average marks: 87.5

Sports class details:
Name: akhil
Age: 35
Sports Name: Cricket
Points: 50
```

4. Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



Multiple Inheritance

```java
// Base class
class Student {
    protected int age = 35;
    protected String name = "akhil";

    void putStudent() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Derived class
class Test extends Student {
    protected int m1 = 85;
    protected int m2 = 90;

    void average() {
        double avg = (m1 + m2) / 2.0;
        System.out.println("Average marks: " + avg);
    }
}
```

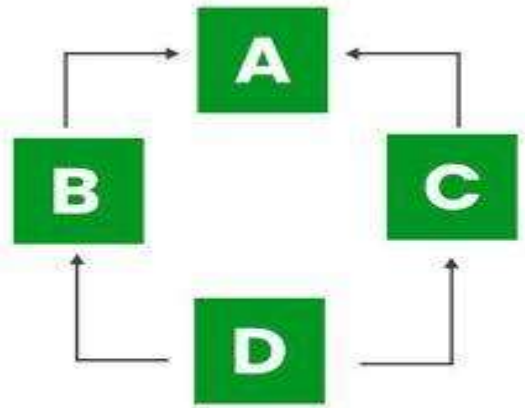
```java
// Sports interface
interface Sports {
    void display();
}

// Result class inherits Test and implements Sports interface
class Result extends Test implements Sports {
    private String sportsName = "Cricket";
    private int points = 50;


    public void display() {
        System.out.println("Sports Name: " + sportsName);
        System.out.println("Points: " + points);
    }
}

public class Main {
    public static void main(String[] args) {
        Result r = new Result();

        r.putStudent();  // from Student class
        r.average();     // from Test class
        r.display();     // from Sports interface implemented in Result
    }
}
```

Output:

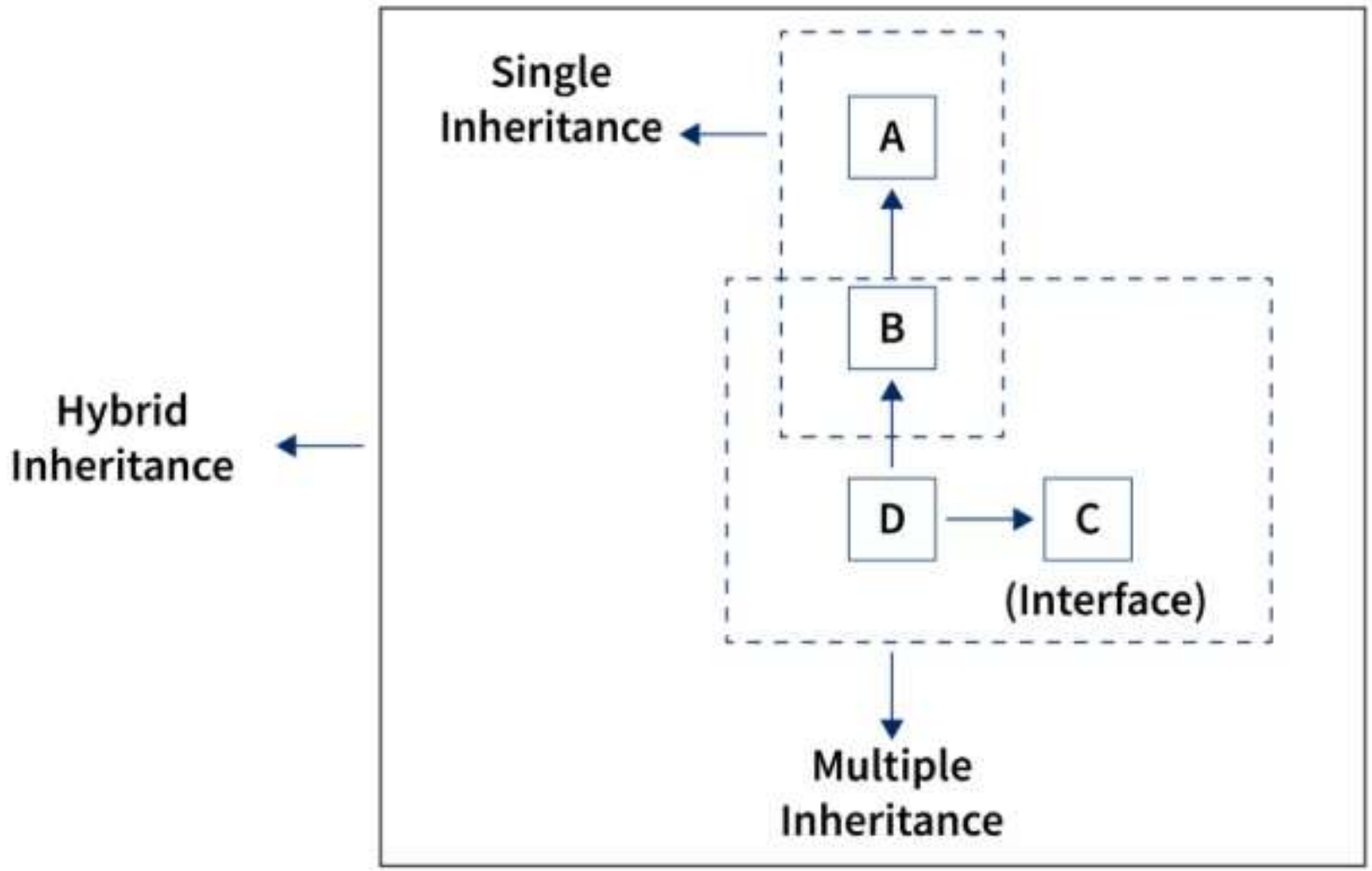
```
Name: akhil
Age: 35
Average marks: 87.5
Sports Name: Cricket
Points: 50
```

5. Hybrid Inheritance

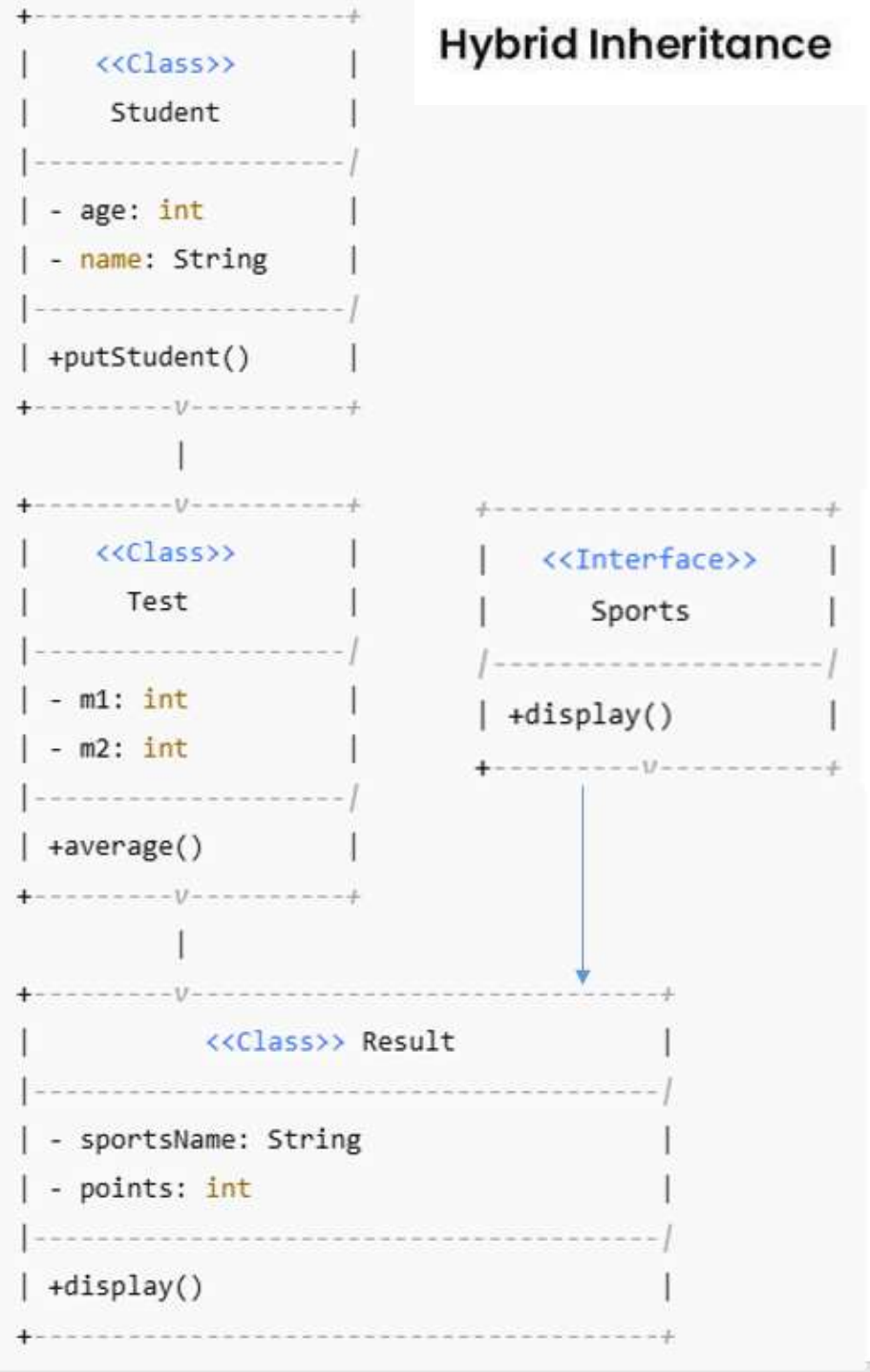
It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.
It is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



Hybrid Inheritance


Single
Inheritance
A
B
Hybrid
Inheritance
D
C
(Interface)
Multiple
Inheritance

Hybrid Inheritance



```java
// Base class
class Student {
    protected int age = 35;
    protected String name = "akhil";

    void putStudent() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Derived class from Student
class Test extends Student {
    protected int m1 = 85;
    protected int m2 = 90;

    void average() {
        double avg = (m1 + m2) / 2.0;
        System.out.println("Average marks: " + avg);
    }
}
```

```java
// Interface
interface Sports {
    void display();
}

// Derived class from Test and implementing Sports interface
class Result extends Test implements Sports {
    private String sportsName = "Cricket";
    private int points = 50;

    @Override
    public void display() {
        System.out.println("Sports Name: " + sportsName);
        System.out.println("Points: " + points);
    }
}
```

```java
// Main class to run the program
public class Main {
    public static void main(String[] args) {
        Result r = new Result();

        r.putStudent();   // from Student class
        r.average();      // from Test class
        r.display();      // from Sports interface implemented in Result
    }
}
```

✅ Output:

```
Name: akhil
Age: 35
Average marks: 87.5
Sports Name: Cricket
Points: 50
```

3.4 Use of super keyword:

The `super` keyword in Java is used to refer to the **immediate parent class** of the current class. It helps to:

1. Call superclass (parent) constructor from subclass constructor.
2. Access superclass variables (if hidden by subclass variables).
3. Call superclass methods (if overridden in subclass).

1. Calling superclass constructor

- Syntax:

```
super();
```

  - It must be the **first statement** in the subclass constructor.
  - Used to initialize the parent class.

2. Accessing superclass variable

If subclass has a variable with the same name as the superclass, use `super` to refer to the superclass variable.

Syntax:

```
super.variableName;
```

3. Calling superclass method

If subclass overrides a method, `super` can be used to call the parent class version of the method.

Syntax:

```
super.methodName();
```

Example Program Using `super`

```
class Student {
    String name;

    Student(String name) {
        this.name = name;
    }

    void display() {
        System.out.println("Student name: " + name);
    }
}
```

```java
class ScienceStudent extends Student {
    String name;  // hides superclass variable

    ScienceStudent(String studentName, String scienceName) {
        super(studentName);    // call superclass constructor
        this.name = scienceName;
    }

    void display() {
        super.display();  // call superclass method
        System.out.println("Science student name: " + name);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        ScienceStudent s = new ScienceStudent("Akhilesh", "Akhilesh S.");

        s.display();
    }
}
```

Output:

```
Student name: Akhilesh
Science student name: Akhilesh S.
```

`super(studentName);` calls the parent class constructor to initialize `name` in `Student`.

In `display()`, `super.display();` calls the `Student` class's `display` method.

`ScienceStudent` has its own `name` variable which hides the superclass variable.

Using `super` helps access the hidden superclass members.

3.5 Creating a Multilevel Hierarchy:

Multilevel Inheritance is a type of inheritance where a class is derived from another derived class. It forms a **"chain"** of inheritance.

- Class A → Base Class (Superclass)
- Class B → Derived from A
- Class C → Derived from B

This hierarchy allows class C to inherit features from both B and A (indirectly).

Syntax:

```
class A {
    // properties and methods of A
}

class B extends A {
    // properties and methods of B
}

class C extends B {
    // properties and methods of C
}
```

```java
// Base class Student
class Student {
    String name;
    int age;

    void setStudentDetails(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```java
// Derived class Sports
class Sports extends Student {
    int point1, point2;
    String sportName;

    void setSportsDetails(String sportName, int p1, int p2) {
        this.sportName = sportName;
        point1 = p1;
        point2 = p2;
    }
}
```

```
// Derived class Test from Sports
class Test extends Sports {
    int marks1, marks2;

    void setTestMarks(int m1, int m2) {
        marks1 = m1;
        marks2 = m2;
    }
```

```java
    // Single display function
    void displayAll() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);

        System.out.println("Sport: " + sportName);
        System.out.println("Sports Point 1: " + point1);
        System.out.println("Sports Point 2: " + point2);

        System.out.println("Test Marks 1: " + marks1);
        System.out.println("Test Marks 2: " + marks2);

        // Calculate average of test marks
        double average = (marks1 + marks2) / 2.0;
        System.out.println("Average Test Marks: " + average);
    }
}
```

```java
// Main class to test
public class MultilevelStudentExample {
    public static void main(String[] args) {
        Test student = new Test();

        // Setting values
        student.setStudentDetails("Alice", 20);
        student.setSportsDetails("Basketball", 85, 90);
        student.setTestMarks(75, 80);

        // Display all details with average test marks and sport name
        student.displayAll();
    }
}
```

Output:

```
Name: Alice
Age: 20
Sport: Basketball
Sports Point 1: 85
Sports Point 2: 90
Test Marks 1: 75
Test Marks 2: 80
Average Test Marks: 77.5
```

3.6 Method Overriding, Dynamic Method Dispatch, Abstract classes:

Method Overriding

- **Method Overriding** occurs when a subclass provides its own implementation of a method that is already defined in its superclass.
- It allows runtime polymorphism — the subclass's version of the method is called instead of the superclass's.
- The method signature (name, parameters) must be the same in both superclass and subclass.
- Access level cannot be more restrictive in the subclass.

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

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

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Superclass reference, subclass object
        myAnimal.sound();  // Calls Dog's sound method at runtime
    }
}
```

Output:

```
Dog barks
```

Abstract Class Example

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

    void sleep() {
        System.out.println("Animal sleeps");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
 public class TestAbstract {
     public static void main(String[] args) {
         Animal myDog = new Dog();
         myDog.sound();
         myDog.sleep();
     }
 }
```

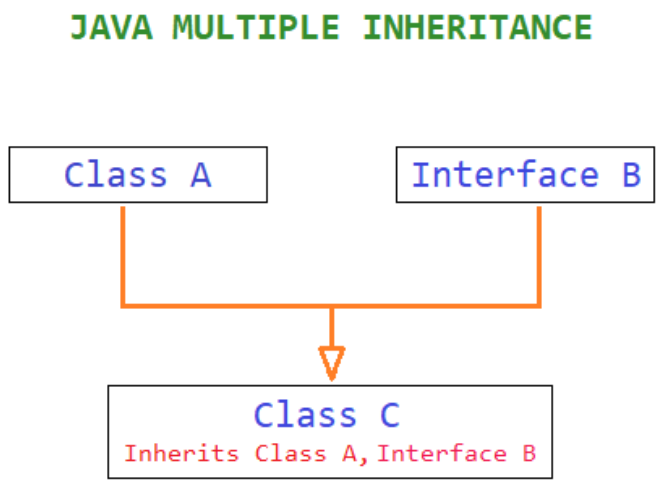
Output:

```
Dog barks
Animal sleeps
```

3.7 Defining an Interface, Implementing Interfaces, applying interfaces, Variables in Interfaces:

Defining an Interface

- An **interface** in Java is a reference type, similar to a class, that can contain **only abstract methods** (until Java 7) and **constants** (static final variables).
- From Java 8 onwards, interfaces can also have **default methods** (methods with implementation) and **static methods**.
- Interfaces define a **contract** that implementing classes must follow.
- They provide a way to achieve **multiple inheritance** in Java (since Java doesn't support multiple inheritance with classes).

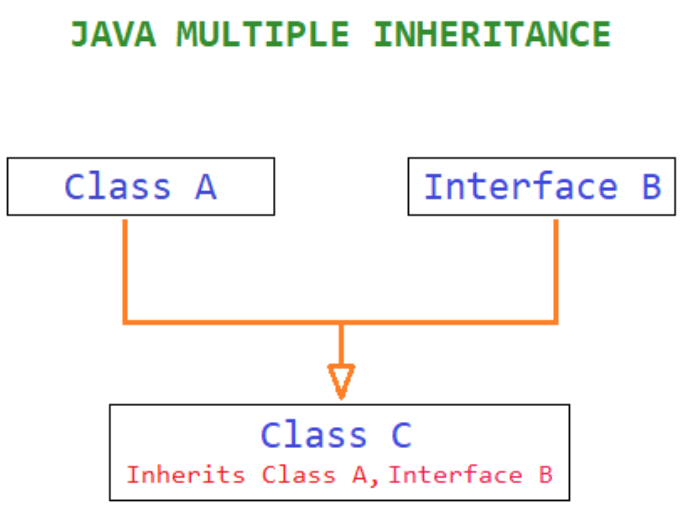


Implementing Interfaces

A class **implements** an interface using the `implements` keyword.

The class must provide concrete implementations of all abstract methods declared in the interface (unless the class is abstract).

A class can implement **multiple interfaces** separated by commas.



Applying Interfaces

- Interfaces are used to enforce certain behaviors on classes without restricting the inheritance hierarchy.
- They enable polymorphism: you can write code that works with interface types, regardless of the implementing class.

Variables in Interfaces

- Variables declared inside interfaces are implicitly:
  - `public`
  - `static`
  - `final` (constant)
- You **cannot** change these variables inside implementing classes.

```java
// Interface definition
interface Sports {
    int MAX_SCORE = 100;  // public static final by default

    void setScore(int score);
    int getScore();

    default void showMaxScore() {
        System.out.println("Max score in sports: " + MAX_SCORE);
    }
}
```

```java
// Class implementing the interface
class Football implements Sports {
    private int score;

    public void setScore(int score) {
        if (score <= MAX_SCORE)
            this.score = score;
        else
            this.score = MAX_SCORE;
    }

    public int getScore() {
        return score;
    }
}
```

```java
// Main class to test
public class InterfaceExample {
    public static void main(String[] args) {
        Football fb = new Football();

        fb.setScore(95);
        System.out.println("Football Score: " + fb.getScore());

        fb.showMaxScore();  // calling default method in interface
    }
}
```

Output:

```
Football Score: 95
Max score in sports: 100
```

Topic	Description
Interface	Blueprint with abstract methods and constants
Implementing Interface	Class uses `implements` keyword, provides method bodies
Applying Interfaces	Achieve polymorphism and multiple inheritance
Variables in Interfaces	Implicitly public static final constants

3.8 Implementing Multiple Inheritance (Multiple Inheritance), Interfaces Can Be Extended:

1. Multiple Inheritance in Java

- Java **does not support multiple inheritance with classes** to avoid complexity and ambiguity (e.g., the Diamond Problem).
- However, **Java supports multiple inheritance through interfaces**.
- A class can implement multiple interfaces, allowing it to inherit method declarations from multiple sources.

Why no multiple inheritance with classes?

- Ambiguity arises when two parent classes have methods with the same signature.
- Java resolves this by allowing multiple inheritance only via interfaces (which only declare methods but do not implement them).

Example:

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

interface Showable {
    void show();
}

class MultipleInheritanceExample implements Printable, Showable {
    public void print() {
        System.out.println("Print method");
    }

    public void show() {
        System.out.println("Show method");
    }

    public static void main(String[] args) {
        MultipleInheritanceExample obj = new MultipleInheritanceExample();
        obj.print();  // Output: Print method
        obj.show();   // Output: Show method
    }
}
```

Output:

```
Print method
Show method
```

2. Extending Interfaces in Java

- Interfaces can **extend one or more other interfaces**.
- This allows you to combine multiple interfaces into a single one.
- The extending interface inherits all abstract methods from the parent interfaces.
- Classes implementing the extended interface must provide implementations for all inherited methods.

Example:

```
interface Walkable {
    void walk();
}

interface Runnable {
    void run();
}

// Combined interface extending both Walkable and Runnable
interface Movable extends Walkable, Runnable {
    void stop();
}
```

```java
class Animal implements Movable {
    public void walk() {
        System.out.println("Animal is walking");
    }

    public void run() {
        System.out.println("Animal is running");
    }

    public void stop() {
        System.out.println("Animal stopped");
    }

    public static void main(String[] args) {
        Animal a = new Animal();
        a.walk();  // Animal is walking
        a.run();   // Animal is running
        a.stop();  // Animal stopped
    }
}
```

Output:

```
Animal is walking
Animal is running
Animal stopped
```

3.9 Packages, defining a Package, Finding Packages and CLASSPATH:

Package in Java?

- A **package** in Java is a namespace that organizes classes and interfaces.
- It helps to **avoid name conflicts** and **control access**.
- Packages also make searching/locating and usage of classes easier.
- Think of packages as **folders/directories** in your file system that contain related `.java` files.

Defining a Package

- You define a package at the **very beginning** of a Java source file using the `package` keyword.
- Syntax:

```
package package_name;
```

- Example:

```
package com.mycompany.project;

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

- The directory structure should mirror the package structure.

  For example:

```
com/mycompany/project/MyClass.java
```

Using Classes from Packages

- Use the `import` statement to access classes from other packages.
- Syntax:

```
import package_name.ClassName;
```

- Or import all classes from a package:

```
import package_name.*;
```

- Example:

```
import com.mycompany.project.MyClass;

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

Finding Packages (How Java Finds Packages and Classes)

- Java looks for classes based on the **package structure**.
- The directory structure of source files or compiled `.class` files must match the package name.
- The **classpath** tells the Java compiler (`javac`) and the Java runtime (`java`) where to look for packages and classes.

What is CLASSPATH?

- **CLASSPATH** is an environment variable or command-line option that specifies the location(s) of user-defined classes and packages.
- Java uses the CLASSPATH to locate classes and packages during compilation and runtime.
- By default, Java uses the current directory (`.`) as part of the classpath.

Setting CLASSPATH

- You can set the CLASSPATH in your environment variables or specify it when running Java commands.

Example setting CLASSPATH in terminal:

- Linux/macOS:

```
export CLASSPATH=/home/user/myclasses:/home/user/libs/mylib.jar
```

- Windows:

```
set CLASSPATH=C:\Users\user\myclasses;C:\Users\user\libs\mylib.jar
```

Using CLASSPATH with javac and java

- To compile a class that depends on other packages/classes:

```
javac -classpath /path/to/classes MyProgram.java
```

- To run a program specifying the classpath:

```
java -classpath /path/to/classes MainClass
```

Example

File: `com/example/Hello.java`

```java
package com.example;

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

File: `TestHello.java`

```java
import com.example.Hello;

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

Directory Structure:

```
project/
 ├─ com/
 │   └─ example/
 │       └─ Hello.java
 └─ TestHello.java
```

Compilation (from project directory):

```bash
javac com/example/Hello.java
javac -classpath . TestHello.java
```

Run:

```bash
java -classpath . TestHello
```

Output:

```csharp
Hello from com.example package!
```

3.10 Access Protection, Importing Packages:

1. Access Protection in Java

Access protection controls **visibility** and **accessibility** of classes, methods, and variables. Java provides **access modifiers** to enforce this:

Access Modifier	Class	Package	Subclass (same/different package)	World (anywhere)
`private`	Yes	No	No	No
default (no modifier)	Yes	Yes	No	No
`protected`	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes

Access Modifiers Details

- `private`

  Accessible **only within the class** itself.

- **Default (package-private)**

  If no modifier is used, accessible **within the same package** only.

- `protected`

  Accessible in the same package and also in subclasses (even if subclass is in a different package).

- `public`

  Accessible from **anywhere**.

Example showing access modifiers:

```java
package com.example;

public class AccessDemo {
    private int privateVar = 1;
    int defaultVar = 2;       // package-private
    protected int protectedVar = 3;
    public int publicVar = 4;

    public void show() {
        System.out.println("Private: " + privateVar);
        System.out.println("Default: " + defaultVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Public: " + publicVar);
    }
}
```

2. Importing Packages in Java

Why Import?

- To use classes/interfaces from other packages without writing their **fully qualified names** every time.

- Import a single class:

```
import packageName.ClassName;
```

- Import all classes in a package (wildcard):

```
import packageName.*;
```

- Example:

```
import java.util.ArrayList;
import java.util.*;
```

- `import` statements must be **after** the `package` declaration and **before** the class definition.
- Java does **not** import sub-packages automatically.
  For example, importing `java.util.*` does **not** import `java.util.concurrent.*`.
- You can use fully qualified names without imports, e.g.:

```
java.util.ArrayList<String> list = new java.util.ArrayList<>();
```

Example: Using Import

```java
import java.util.ArrayList;

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

Output:

```csharp
[Hello, World]
```

Feature	Description
private	Accessible only within the class
default	Accessible within the same package
protected	Accessible in same package & subclasses
public	Accessible from anywhere
import statement	Brings classes/interfaces into scope without fully qualified names
wildcard import	Imports all classes/interfaces in a package, but **not** sub-packages

✅ 3rd Point — `protected`:

> "Can be accessed in a subclass even if the subclass is in a different package."

Example:

java Copy Edit

```java
// File: com/example/Person.java
package com.example;

public class Person {
    protected String name = "Akhilesh";
}
```

java Copy Edit

```java
// File: com/other/Student.java
package com.other;
import com.example.Person;

public class Student extends Person {
    void show() {
        System.out.println(name);  // ✅ Allowed because it's a subclass, even though package is
    }
}
```

✔ `protected` = accessible in subclasses **even from another package**.

✅ Example for `public` (Access from subclass in another package)

📁 File 1: `com/example/Person.java`

java

```java
package com.example;

public class Person {
    public String name = "Akhilesh";
}
```

📁 File 2: `com/other/Student.java`

java

```java
package com.other;

import com.example.Person;

public class Student extends Person {
    void show() {
        System.out.println(name);  // ✅ Allowed – public is visible everywhere
    }

    public static void main(String[] args) {
        new Student().show();
    }
}
```

✅ Also Access from **non-subclass**

You don't even need to extend `Person`!

java

```java
package com.other;

import com.example.Person;

public class AnyClass {
    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(p.name);  // ✅ Still allowed – because it's public
    }
}
```

🧠 So in short:

Modifier	Subclass (other package)	Non-subclass (other package)
`public`	✅ Allowed	✅ Allowed
`protected`	✅ Allowed	❌ Not allowed

🔐 Java Access Modifiers Recap:

Modifier	Same Class	Same Package	Subclass (different package)	Outside Package
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
(default)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

Package: mypkg

```java
public class Parent {
    protected int data = 42;
}
```

Package: otherpkg

```java
import mypkg.Parent;

public class Other {
    public static void main(String[] args) {
        Parent p = new Parent();
        System.out.println(p.data); // ❌ Error: 'data' is protected
    }
}
```

But if you do:

```java
public class Child extends Parent {
    void show() {
        System.out.println(data);  // ✅ Allowed (subclass)
    }
}
```

Modifier	Same Class	Same Package	Subclass (Other Package)	Other Package
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
(default)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

🔓 1. `public` — Sab jagah se access ho sakta hai

- **Koi bhi class**, chahe kisi bhi package mein ho, us method/variable ko access kar sakti hai.

```java
public int x = 10;

public void show() {
    System.out.println("This is public");
}
```

✅ Accessible **inside same class, same package, different package, and subclasses**

🔒 2. `private` — Sirf usi class ke andar

- **Kisi aur class se access nahi ho sakta**, even subclass bhi nahi.

```java
private int y = 20;

private void secret() {
    System.out.println("This is private");
}
```

❌ Not accessible **outside the class**, not even in subclass.