

Module-3

Chapter-1

Inheritance

Inheritance is an object-oriented programming (OOP) feature that allows one class to acquire the properties and behaviors of another class. It promotes **code reusability**, **hierarchical classification**, and **easy maintenance**. The class whose members are inherited is called the **superclass (parent class)**. The class that inherits these members is called the **subclass (child class)**. A subclass inherits all **accessible fields and methods** of the superclass and can also add its own variables and methods. In Java, inheritance is implemented using the **extends** keyword.

I. INHERITANCE BASICS

Extending a Class in Java

A **subclass** is created by inheriting a **superclass** using the **extends** keyword. The subclass automatically acquires all **accessible variables and methods** of the superclass and can add its own members.

Java supports **single inheritance**, meaning a subclass can inherit from **only one superclass**. However, a subclass can itself become a superclass for another class, forming an **inheritance hierarchy**.

Syntax

```
class SuperClass {  
    // members  
}  
class SubClass extends SuperClass {  
    // additional members  
}
```

Example 1: Basic Inheritance

```
class A {  
    int i, j;  
    void showIJ() {  
        System.out.println("i = " + i + ", j = " + j);  
    }  
}  
class B extends A {  
    int k;  
    void sum() {  
        System.out.println("Sum = " + (i + j + k));  
    }  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
public class Demo {  
    public static void main(String args[]) {  
        B obj = new B();  
        obj.i = 10;  
        obj.j = 20;  
        obj.k = 30;  
        obj.showIJ(); // Inherited method  
        obj.sum();    // Own method  
    }  
}
```

Output

i = 10, j = 20
Sum = 60

Member Access in Inheritance

A subclass **cannot access private members** of its superclass. Only **public, protected, and default (same package)** members are inherited and accessible.

Example

```
class A {  
    private int x = 10;  
}  
class B extends A {  
    void display() {  
        // System.out.println(x); // Error: x is private  
    }  
}
```

Note: Private members remain accessible **only within the class in which they are declared**.

Example 2: Practical Inheritance (Box and BoxWeight)

```
class Box {  
    double width, height, depth;  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}  
class BoxWeight extends Box {  
    double weight;  
  
    BoxWeight(double w, double h, double d, double wt) {
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
        super(w, h, d);
        weight = wt;
    }

    void showWeight() {
        System.out.println("Weight = " + weight);
    }
}

public class Demo {
    public static void main(String args[]) {
        BoxWeight box = new BoxWeight(10, 20, 15, 25);
        System.out.println("Volume = " + box.volume());
        box.showWeight();
    }
}
```

Output

Volume = 3000.0

Weight = 25.0

II. USING SUPER

The **super** keyword is used in Java to refer to the **immediate superclass** of a subclass. It has two main uses: **calling the superclass constructor** and **accessing superclass members (variables or methods)** that are hidden by the subclass. Using super avoids code duplication and supports proper inheritance.

When **super()** is used, it must be the **first statement** inside a subclass constructor. It invokes the appropriate constructor of the parent class. The **super.member** syntax is used to access a superclass variable or method when the subclass has a member with the same name.

Example 1: Using super() to Call a Superclass Constructor

```
class Box {
    int width, height, depth;
    Box(int w, int h, int d) {
        width = w;
        height = h;
        depth = d;
    }
}

class BoxWeight extends Box {
    int weight;
    BoxWeight(int w, int h, int d, int wt) {
        super(w, h, d); // Calls Box constructor
        weight = wt;
    }
    void display() {
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
        System.out.println("Width = " + width);
        System.out.println("Weight = " + weight);
    }
}

public class Demo {
    public static void main(String[] args) {
        BoxWeight obj = new BoxWeight(10, 20, 15, 25);
        obj.display();
    }
}
```

Output:

Width = 10
Weight = 25

Example 2: Using super to Access Hidden Variables

```
class A {
    int x = 100;
}
class B extends A {
    int x = 200;
    void show() {
        System.out.println("Subclass x = " + x);
        System.out.println("Superclass x = " + super.x);
    }
}

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

Output:

Subclass x = 200
Superclass x = 100

III. CREATING A MULTILEVEL HIERARCHY

Multilevel inheritance is a feature in Java where a class inherits from another subclass, forming a chain of inheritance. Each subclass inherits all the properties and methods of its parent classes while adding its own unique features.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

// Superclass

```
class Box {  
    double width, height, depth;  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}
```

// Subclass

```
class BoxWeight extends Box {  
    double weight;  
    BoxWeight(double w, double h, double d, double wt) {  
        super(w, h, d);  
        weight = wt;  
    }  
}
```

// Subclass of BoxWeight

```
class Shipment extends BoxWeight {  
    double cost;  
    Shipment(double w, double h, double d, double wt, double c) {  
        super(w, h, d, wt);  
        cost = c;  
    }  
    void display() {  
        System.out.println("Volume: " + volume());  
        System.out.println("Weight: " + weight);  
        System.out.println("Shipping Cost: " + cost);  
    }  
}  
  
public class MultiLevelDemo {  
    public static void main(String args[]) {  
        Shipment s = new Shipment(10, 20, 15, 12.5, 250);  
        s.display();  
    }  
}
```

Output

Volume: 3000.0

Weight: 12.5

Shipping Cost: 250.0

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

IV. WHEN CONSTRUCTORS ARE EXECUTED

In Java inheritance, constructors are executed in the **order of inheritance**, starting from the **superclass** and ending with the **subclass**. This ensures that the parent class is initialized before the child class.

Example

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

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

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

Output

Constructor of A
Constructor of B
Constructor of C

When new C() is executed, Java first calls the constructor of the superclass (A). After A finishes, the constructor of B is executed. Finally, the constructor of C is executed. If super() is not written explicitly, Java automatically inserts a call to the default constructor of the superclass. Constructors execute from **superclass** → **subclass**. super() must be the **first statement** in a subclass constructor. If super() is omitted, Java automatically calls the superclass's default constructor. This order ensures proper initialization of inherited members before subclass-specific initialization.

V. METHOD OVERRIDING

Method overriding occurs when a subclass defines a method with the **same name and same parameter list (type signature)** as its superclass. The subclass method replaces the superclass method, enabling **runtime polymorphism**.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example 1: Basic Method Overriding

```
class A {  
    void show() {  
        System.out.println("Inside Class A");  
    }  
}  
class B extends A {  
    void show() {  
        System.out.println("Inside Class B");  
    }  
}  
public class OverrideDemo {  
    public static void main(String args[]) {  
        B obj = new B();  
        obj.show();  
    }  
}
```

Output

Inside Class B

Explanation:

The show() method in class **B** overrides the show() method in class **A**. Therefore, when obj.show() is called, the subclass version is executed.

Using super to Call the Superclass Method

If you want to execute the superclass version of an overridden method, use the **super** keyword.

Example

```
class A {  
    void show() {  
        System.out.println("Inside Class A");  
    }  
}  
class B extends A {  
    void show() {  
        super.show();  
        System.out.println("Inside Class B");  
    }  
}  
public class SuperDemo {  
    public static void main(String args[]) {  
        B obj = new B();  
        obj.show();  
    }  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Output

Inside Class A

Inside Class B

- `super.show()` calls the superclass method. Then the subclass method continues its execution.

VI. DYNAMIC METHOD DISPATCH

Dynamic Method Dispatch is the process by which Java determines **at runtime** which overridden method to execute. It enables **runtime polymorphism**, where the method called depends on the actual object, not the reference type.

Example

```
class A {  
    void callMe() {  
        System.out.println("Inside A");  
    }  
}  
class B extends A {  
    void callMe() {  
        System.out.println("Inside B");  
    }  
}  
class C extends A {  
    void callMe() {  
        System.out.println("Inside C");  
    }  
}  
public class DynamicDispatchDemo {  
    public static void main(String args[]) {  
        A r;      // Superclass reference  
        A a = new A();  
        B b = new B();  
        C c = new C();  
        r = a;  
        r.callMe();  
        r = b;  
        r.callMe();  
        r = c;  
        r.callMe();  
    }  
}
```

Output

Inside A

Inside B

Inside C

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

r is a reference variable of type A. It first refers to an object of class A, so A's callMe() is executed. Next, r refers to an object of class B, so B's overridden callMe() is executed. Finally, r refers to an object of class C, so C's overridden callMe() is executed. The method that runs depends on the **actual object** (A, B, or C), not on the reference variable (r).

Dynamic Method Dispatch is also called **Runtime Polymorphism**. A **superclass reference** can refer to a **subclass object**. The overridden method is selected **at runtime** based on the object type. It is possible only when **method overriding** is used. It improves flexibility and supports object-oriented programming.

VII. USING ABSTRACT CLASSES

An **abstract class** is a class that cannot be instantiated and is used as a base class for other classes. An **abstract method** has no body and must be implemented (overridden) by all subclasses.

Example 1: Abstract Class and Abstract Method

```
abstract class A {  
    abstract void callMe(); // Abstract method  
    void callMeToo() {      // Concrete method  
        System.out.println("This is a concrete method.");  
    }  
}  
class B extends A {  
    void callMe() {  
        System.out.println("B's implementation of callMe().");  
    }  
}
```

```
public class AbstractDemo {  
    public static void main(String args[]) {  
        B obj = new B();  
        obj.callMe();  
        obj.callMeToo();  
    }  
}
```

Output

B's implementation of callMe().
This is a concrete method.

A is an **abstract class**, so objects of A cannot be created. callMe() is an **abstract method** with no implementation. Class B overrides callMe() and provides its implementation. callMeToo() is a normal (concrete) method inherited from class A.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example 2: Abstract Class Using Figure

```
abstract class Figure {
    double dim1, dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }
    double area() {
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }
    double area() {
        return (dim1 * dim2) / 2;
    }
}

public class AbstractFigureDemo {
    public static void main(String args[]) {
        Figure fig;
        fig = new Rectangle(10, 5);
        System.out.println("Area of Rectangle = " + fig.area());
        fig = new Triangle(10, 5);
        System.out.println("Area of Triangle = " + fig.area());
    }
}
```

Output

Area of Rectangle = 50.0
Area of Triangle = 25.0

An **abstract class** cannot be instantiated using new. An **abstract method** has no body and ends with a semicolon (;). Any subclass must implement all abstract methods or be declared abstract. Abstract classes can contain both **abstract** and **concrete** methods. Abstract classes are mainly used to achieve **abstraction** and support **runtime polymorphism** through superclass references.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

VIII. USING 'FINAL' WITH INHERITANCE

The **final** keyword is used to prevent modification in Java. It can be used with **variables** (constant values), **methods** (prevent overriding), and **classes** (prevent inheritance).

1. Using final to Prevent Method Overriding

A method declared as **final** cannot be overridden by any subclass.

Example

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

class B extends A {
    // Error: Cannot override the final method
    /*
    void display() {
        System.out.println("Inside B");
    }
    */
}

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

Output

This is a final method.

The display() method in class A is declared as final. Class B inherits the method but **cannot override it**. Attempting to override it results in a **compile-time error**.

2. Using final to Prevent Inheritance

A class declared as **final** cannot be extended by another class.

Example

```
final class A {
    void show() {
        System.out.println("Final class");
    }
}

/*
// Error: Cannot inherit from final class A
class B extends A {
}
*/

public class FinalClassDemo {
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

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

Output

Final class

Class A is declared as final. No other class can inherit from A. Attempting to extend A causes a **compile-time error**. final **variable** → Value cannot be changed (constant). final **method** → Cannot be overridden by subclasses. final **class** → Cannot be inherited. A class **cannot be both abstract and final** because an abstract class requires inheritance, while a final class prevents it.

IX. LOCAL VARIABLE TYPE INFERENCE AND INHERITANCE

Local Variable Type Inference, introduced in **JDK 10** using the `var` keyword, automatically determines the type of a local variable from its initializer. In an inheritance hierarchy, the inferred type is always based on the **declared type** of the initializer, not on the actual object returned at runtime.

Therefore, if a method returns a superclass type, a variable declared with `var` will also be inferred as the superclass type, even if the returned object belongs to a subclass.

For example, if a method `getObj()` has a return type `MyClass` but returns objects of `FirstDerivedClass` or `SecondDerivedClass`, then `var obj = getObj();` will infer `obj` as **MyClass**. This supports polymorphism but restricts access to only the members defined in the superclass unless explicit casting is used.

X. THE OBJECT CLASS

Object is the root (super) class of Java, and every class in Java automatically inherits from it, either directly or indirectly.

It allows an Object reference to store any class object or array, providing common methods like `toString()`, `equals()`, and `hashCode()`.

Object defines the following methods, which means that they are available in every object.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Method	Purpose
Object clone()	Creates a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is recycled. (Deprecated for removal by JDK 18.)
Class<?> getClass()	Obtains the class of an object at run time.
int hashCode()	Returns the hash code associated with the invoking object.
void notify()	Resumes execution of a thread waiting on the invoking object.
void notifyAll()	Resumes execution of all threads waiting on the invoking object.
String toString()	Returns a string that describes the object.
void wait() void wait(long milliseconds) void wait(long milliseconds, int nanoseconds)	Waits on another thread of execution.

Object Class Methods: equals() and toString()

The **equals()** method compares two objects and returns **true** if they are considered equal; otherwise, it returns **false**. The **toString()** method returns a string representation of an object and is automatically called when the object is printed using `System.out.println()`.

Example of equals()

```
class Test {
    int x;
    Test(int x) {
        this.x = x;
    }
    public boolean equals(Object obj) {
        Test t = (Test) obj;
        return this.x == t.x;
    }
}

public class EqualsDemo {
    public static void main(String[] args) {
        Test t1 = new Test(10);
        Test t2 = new Test(10);

        System.out.println(t1.equals(t2));
    }
}
```

Output:

true

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example of toString()

```
class Student {  
    String name = "Anu";  
  
    public String toString() {  
        return "Student Name: " + name;  
    }  
}  
public class ToStringDemo {  
    public static void main(String[] args) {  
        Student s = new Student();  
        System.out.println(s);  
    }  
}
```

Output:

Student Name: Anu

equals() compares the contents of two objects (when overridden). **toString()** returns a readable description of an object. **println(object)** automatically invokes the object's **toString()** method. Both methods are inherited from the **Object** class and are commonly overridden in user-defined classes.

CHAPTER-2

PACKAGES

An interface in Java provides complete abstraction by defining a set of methods that classes must implement without specifying their implementation. It acts as a contract between a class and its users, ensuring a common method structure while allowing different implementations. Unlike abstract classes, a class can implement multiple interfaces, enabling multiple inheritance of type, whereas a class can inherit only one superclass.

I. PACKAGES

A **package** in Java is a mechanism used to organize related classes and interfaces into separate namespaces. It helps avoid **name collisions**, making it possible for different programmers to create classes with the same name without conflict. Packages also provide **access control**, allowing certain classes and members to be accessible only within the same package. This improves code organization, security, and maintainability in large applications.

Defining a Package

A package is created by placing the **package** statement as the **first line** in a Java source file. All classes declared in that file become part of the specified package. If the package statement is omitted, the class belongs to the **default package**, which has no name.

Syntax:

```
package pkg;
```

Example:

```
package mypackage;
```

This statement creates a package named **mypackage**, and every class in that source file belongs to this package.

Package Directory Structure

Java stores packages as directories in the file system. The directory name must exactly match the package name, including uppercase and lowercase letters.

Example:

```
package mypackage;
```

The compiled .class file must be stored in the directory:

```
mypackage
```

Multiple Files in the Same Package

Many Java source files can belong to the same package by including the same package statement.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

File 1

```
package mypackage;  
class A {  
    // Class A  
}
```

File 2

```
package mypackage;  
class B {  
    // Class B  
}
```

Both classes **A** and **B** belong to the package **mypackage**.

Hierarchical Packages

Java supports packages with multiple levels by separating package names using dots (.). Each level corresponds to a directory in the file system.

Syntax:

```
package pkg1.pkg2.pkg3;
```

Example:

```
package a.b.c;
```

In Windows, the directory structure will be:

```
a  
├── b  
│   └── c
```

or

a\b\c

Finding Packages (CLASSPATH)

Java locates packages using one of the following methods:

1. **Current Working Directory** – Java first searches in the current directory.
2. **CLASSPATH Environment Variable** – Specifies additional directories where Java searches for classes.
3. **-classpath Option** – Used with java and javac commands to specify the class path.

Example of CLASSPATH

Suppose the package declaration is:

```
package mypack;
```

If the package is stored in:

C:\MyPrograms\Java\mypack

Then the class path should be:

C:\MyPrograms\Java

Not

C:\MyPrograms\Java\mypack

because Java automatically looks inside the **mypack** directory.

Prof. Suma H C, Dept of CSE-AI&ML, ATMECE

Page 16 of 33

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Short Package Example

Create a file named **AccountBalance.java** inside a folder called **mypack**.

```
package mypack;
class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }
    void show() {
        if (bal < 0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}

public class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];
        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);
        for (int i = 0; i < 3; i++)
            current[i].show();
    }
}
```

Save this file in the directory:

mypack

Compile the program:

```
javac mypack/AccountBalance.java
```

Run the program from the directory above **mypack**:

```
java mypack.AccountBalance
```

Output

K. J. Fielding: \$123.23

Will Tell: \$157.02

--> Tom Jackson: \$-12.33

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Since AccountBalance belongs to the package **mypack**, it **cannot** be executed as:

```
java AccountBalance
```

This is **incorrect** because Java requires the package name.

The correct command is:

```
java mypack.AccountBalance
```

Here, mypack identifies the package and AccountBalance is the class containing the main() method.

II. PACKAGES AND MEMBER ACCES

Access Protection

Java provides an **access control mechanism** to protect variables, methods, and classes from unauthorized access. Earlier, you learned that **private** members can be accessed only within the same class. Packages extend this access control by defining how members can be accessed within the same package, subclasses, and different packages.

This enables developers to achieve **encapsulation**, improve security, and control the visibility of program components.

Packages act as **containers for classes**, while classes act as **containers for data and methods**.

Java classifies member accessibility into four categories:

- **Subclasses in the same package**
- **Non-subclasses in the same package**
- **Subclasses in different packages**
- **Classes that are neither subclasses nor in the same package**

Java provides four access levels to control visibility:

1. Private Access Modifier (private)

A member declared as **private** is accessible **only within the same class**. It cannot be accessed by subclasses or classes in the same or different packages.

Example

```
class Test {  
    private int x = 10;  
    void display() {  
        System.out.println(x);  
    }  
}
```

Here, x can only be accessed inside the Test class.

2. Default Access (No Modifier)

If no access modifier is specified, Java uses **default access** (also called package-private).

Members with default access are accessible:

- Within the same class
- By other classes in the same package

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

They **cannot** be accessed from another package.

Example

```
class Test {  
    int x = 20;    // Default access  
}
```

Any class in the same package can access x.

3. Protected Access Modifier (protected)

A **protected** member is accessible:

- Within the same class
- By all classes in the same package
- By subclasses in different packages

Example

```
class Test {  
    protected int x = 30;  
}
```

A subclass outside the package can access x.

4. Public Access Modifier (public)

A **public** member is accessible from **any class** in **any package**.

Example

```
class Test {  
    public int x = 40;  
}
```

The variable x can be accessed everywhere.

Access Protection Example

The book uses **two packages**:

p1
and
p2

Package p1

Protection.java

This class declares four variables with different access levels.

```
package p1;  
public class Protection {  
    int n = 1;           // Default  
    private int n_pri = 2; // Private  
    protected int n_pro = 3; // Protected  
    public int n_pub = 4;  // Public  
    public Protection() {  

```

Prof. Suma H C, Dept of CSE-AI&ML, ATMECE

Page 19 of 33

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
        System.out.println("Base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

Derived.java

Derived extends Protection and belongs to the same package.

```
package p1;
class Derived extends Protection {
    Derived() {
        System.out.println("Derived constructor");
        System.out.println("n = " + n);
        // System.out.println(n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

SamePackage.java

This class is **not a subclass**, but belongs to the same package.

```
package p1;
class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("n = " + p.n);
        // System.out.println(p.n_pri);
        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Package p2

Protection2.java

This class extends Protection but belongs to another package.

```
package p2;
import p1.Protection;
class Protection2 extends Protection {
    Protection2() {
        System.out.println("Protection2 constructor");
    }
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
// System.out.println(n);
// System.out.println(n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

OtherPackage.java

This class neither belongs to package p1 nor extends Protection.

```
package p2;
import p1.Protection;
class OtherPackage {
    OtherPackage() {
        Protection p = new Protection();
        // System.out.println(p.n);
        // System.out.println(p.n_pri);
        // System.out.println(p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Class Accessibility

Java classes themselves have only **two access levels public and default**.

III. IMPORTING PACKAGES

Importing Packages

Java stores all built-in classes inside **packages** instead of the default package. Since package names can be long (for example, java.util.Date), writing the complete package name every time becomes tedious. To simplify programming, Java provides the **import** statement, which allows classes or entire packages to be used directly without writing their full package names repeatedly. The import statement is optional because a class can always be accessed using its **fully qualified name**.

Syntax of Import Statement

The general syntax is:

```
import pkg1[.pkg2].ClassName;
```

or

```
import pkg1[.pkg2].*;
```

- **ClassName** → Imports a specific class.
- ***** → Imports all public classes in the package.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example 1: Importing a Single Class

```
import java.util.Date;

class Demo {
    public static void main(String args[]) {
        Date d = new Date();
        System.out.println(d);
    }
}
```

Only the **Date** class is imported from the java.util package. The class can be used directly as Date.

Example 2: Importing an Entire Package

```
import java.util.*;

class Demo {
    public static void main(String args[]) {
        Date d = new Date();
        Scanner sc = new Scanner(System.in);
    }
}
```

java.util.* imports **all public classes** in the java.util package. Classes like Date, Scanner, ArrayList, etc., can be used directly.

Location of Import Statement

The import statement must appear: After the package statement (if present), Before any class definition

Example:

```
package mypack;
import java.util.Date;
public class Test {
}
```

The java.lang Package

The package **java.lang** contains essential Java classes such as:

- String
- System
- Math
- Object
- Integer

This package is **automatically imported** by the Java compiler. is equivalent to: import java.lang.*; Therefore, programmers do not need to write this import statement explicitly.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
class Test {  
    public static void main(String args[]) {  
        String s = "Java";  
        System.out.println(s.length());  
    }  
}
```

No import statement is needed because String and System belong to java.lang.

Name Conflict While Importing

Sometimes two packages contain classes with the **same name**.

Example:

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

```
import java.sql.*;
```

Both packages contain a class named **Date**.

If you write:

```
Date d = new Date();
```

Java produces a **compile-time error** because it cannot determine which Date class you mean.

The solution is to use the **fully qualified name**.

Example:

```
java.util.Date d = new java.util.Date();
```

or

```
java.sql.Date d;
```

Using Fully Qualified Names

Instead of importing a class, you can always specify its complete package name.

With Import

```
import java.util.Date;
```

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

Without Import

```
class Test {  
    public static void main(String args[]) {  
        java.util.Date today = new java.util.Date(); }  
}
```

Both programs work exactly the same. The second program does not use the import statement because the class name is fully qualified.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Public Classes and Import

Only **public classes** can be accessed from another package.

Suppose package **mypack** contains the class:

```
package mypack;
public class Balance {
    String name;
    double bal;
    public Balance(String n, double b) {
        name = n;
        bal = b;
    }
    public void show() {
        if (bal < 0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    } }
```

The class **Balance** is declared **public**. Its constructor is **public**. The method **show()** is **public**. Therefore, any program outside the package can use this class.

Using the Imported Class

```
import mypack.Balance;
class TestBalance {
    public static void main(String args[]) {
        Balance current = new Balance("John", 1250.75);
        current.show();
    }
}
```

Output

John: \$1250.75

What Happens if public is Removed?

Suppose the class is declared as:

```
package mypack;
class Balance {
}
```

Now compile:

```
import mypack.Balance;
class TestBalance {
    public static void main(String args[]) {
        Balance b = new Balance();
    }
}
```

The program **will not compile** because **Balance** has **default access** and cannot be accessed from another package.

CHAPTER-3

INTERFACES

I. INTERFACES

An **interface** in Java is a blueprint that specifies **what a class should do, but not how it should do it**. It contains method declarations without implementation (in the traditional form). Any number of classes can implement the same interface, allowing Java to achieve "**one interface, multiple methods**" and **run-time polymorphism**.

Example

```
interface Callback {  
    void callback(int param);  
}
```

An interface is declared using the **interface** keyword. All methods are **public and abstract** by default, and all variables are **public, static, and final**. An interface does not contain instance variables. Since JDK 8, interfaces can also contain **default** and **static** methods, and from JDK 9, **private methods** are allowed.

Example

```
interface Callback {  
    void callback(int param);  
}
```

Implementing an Interface

A class implements an interface using the **implements** keyword. The implementing class must provide the implementation of every method declared in the interface. These methods must be declared **public**. A class may also define additional methods besides those in the interface.

Example

```
class Client implements Callback {  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
}
```

Accessing Implementations Through Interface References

An interface reference can refer to any object of a class that implements the interface. The appropriate method implementation is selected dynamically during program execution, which is known as **dynamic method resolution** or **run-time polymorphism**.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
Callback c = new Client();  
c.callback(42);
```

Output

callback called with 42

Run-Time Polymorphism

Different classes can implement the same interface differently. When an interface reference points to different objects, Java automatically invokes the correct method according to the object it references at run time. This demonstrates the concept of **one interface, multiple implementations**.

Example

```
Callback c;  
c = new Client();  
c.callback(42);  
c = new AnotherClient();  
c.callback(42);
```

Partial Implementations

If a class implements an interface but does not implement all its methods, the class must be declared **abstract**. Any subclass inheriting it must either implement the remaining methods or also be declared abstract.

Example

```
abstract class Incomplete implements Callback {  
    // callback() not implemented  
}
```

Nested Interfaces

An interface can be declared inside another class or interface. Such an interface is called a **nested interface**. Outside the enclosing class, it must be referred to using its fully qualified name.

Example

```
class A {  
    public interface NestedIF {  
        boolean isNotNegative(int x);  
    }  
}  
class B implements A.NestedIF {  
    public boolean isNotNegative(int x) {  
        return x >= 0;  
    }  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Applying Interfaces (Stack Example)

Interfaces separate the **interface from the implementation**. The book demonstrates this using an **IntStack** interface implemented by **FixedStack** and **DynStack**. A single interface reference can work with both implementations, allowing run-time polymorphism.

Interface

```
interface IntStack {  
    void push(int item);  
    int pop();  
}
```

Implementations

```
class FixedStack implements IntStack  
class DynStack implements IntStack
```

Using Interface Reference

```
IntStack mystack;  
mystack = new FixedStack();  
mystack = new DynStack();
```

Variables in Interfaces

Variables declared in an interface are automatically **public, static, and final**. These variables act as constants and can be shared among multiple classes implementing the interface. The book demonstrates this using the **SharedConstants** interface.

Example

```
interface SharedConstants {  
    int YES = 0;  
    int NO = 1;  
    int MAYBE = 2;  
}
```

Extending Interfaces

One interface can inherit another interface using the **extends** keyword. A class implementing the derived interface must implement all methods inherited from the parent interface as well as the methods declared in the child interface.

Example

```
interface A {  
    void meth1();  
}  
interface B extends A {  
    void meth2();  
}  
class MyClass implements B {  
    public void meth1() {  
        // implementation  
    }  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
public void meth2() {  
    // implementation  
}  
}
```

II. DEFAULT INTERFACE METHODS

Before **JDK 8**, an interface could not contain any method implementation. All interface methods were **abstract** and had no body. JDK 8 introduced **default methods**, allowing an interface to provide a method implementation using the **default** keyword. Because of this feature, interfaces can now define optional behavior while still serving as a contract between classes. Default methods are also called **extension methods**.

Default methods were introduced mainly to **extend existing interfaces without breaking old programs**. Earlier, if a new method was added to an interface, every implementing class had to implement it, causing compilation errors in existing code. A default method provides a predefined implementation that is automatically used if the implementing class does not override it. It also allows optional methods to be included in an interface without forcing every class to provide an unnecessary implementation.

Features of Default Methods

A default method contains a method body and is declared using the **default** keyword. It does not affect the fundamental nature of interfaces. An interface still cannot have instance variables or maintain object state, and an interface cannot be instantiated directly. Classes must implement the interface to create objects.

Syntax of a Default Method

A default method is declared by placing the **default** keyword before the method declaration.

Example

```
interface MyIF {  
    int getNumber();  
    default String getString() {  
        return "Default String";  
    }  
}
```

Here:

- `getNumber()` is an abstract method.
- `getString()` is a default method with an implementation.

An implementing class **does not have to override** a default method. If it provides no implementation, Java automatically uses the default implementation defined in the interface.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
class MyIFImp implements MyIF {  
    public int getNumber() {  
        return 100;  
    }  
}
```

Using the Interface

```
MyIF obj = new MyIFImp();  
System.out.println(obj.getNumber());  
System.out.println(obj.getString());
```

Output

100

Default String

The method `getString()` is executed from the interface because the class does not override it.

Overriding a Default Method

A class implementing an interface can provide its own implementation of a default method. In that case, the class implementation overrides the interface version.

Example

```
class MyIFImp2 implements MyIF {  
    public int getNumber() {  
        return 200;  
    }  
    public String getString() {  
        return "This is a different string.";  
    }  
}
```

Now the class version of `getString()` is executed instead of the default version.

Practical Example – IntStack Interface

Suppose the **IntStack** interface is already used in many programs. A new method **clear()** needs to be added without affecting the existing classes.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
interface IntStack {  
    void push(int item);  
    int pop();  
    default void clear() {  
        System.out.println("clear() not implemented.");  
    }  
}
```

Existing classes such as **FixedStack** and **DynStack** continue to work without modification because they automatically inherit the default implementation of `clear()`. Existing programs continue to work. New functionality can be added safely. Classes implement `clear()` only if required.

Multiple Inheritance Issues

A class can implement multiple interfaces that contain default methods. Java follows specific rules to resolve conflicts.

Rule 1: Class Implementation Has Highest Priority

If the class overrides the default method, the class version is always executed.

```
class MyClass implements Alpha, Beta {  
    public void reset() {  
        System.out.println("Class version");  
    }  
}
```

The class implementation overrides both interface methods.

Rule 2: Two Interfaces with Same Default Method

If two interfaces contain the same default method and the class does not override it, Java reports a **compile-time error**.

```
interface Alpha {  
    default void reset() { }  
}  
interface Beta {  
    default void reset() { }  
}  
class MyClass implements Alpha, Beta {  
    // Error if reset() is not overridden  
}
```

The class must override `reset()` to remove the ambiguity.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Rule 3: Derived Interface Gets Priority

If one interface extends another and both define the same default method, the child interface's version is used.

```
interface Alpha {  
    default void reset() {  
        System.out.println("Alpha");  
    }  
}  
  
interface Beta extends Alpha {  
    default void reset() {  
        System.out.println("Beta");  
    }  
}
```

Here, **Beta's** reset() method is executed.

Calling a Parent Interface's Default Method

An implementing class can explicitly invoke a default method from a parent interface using:

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

Example

```
Alpha.super.reset();
```

This calls the reset() method defined in the Alpha interface.

III. USE STATIC METHODS IN AN INTERFACE

JDK 8 introduced another important feature called static methods in interfaces. Like static methods in a class, these methods belong to the interface itself and can be called without creating an object or implementing the interface. They are accessed directly using the interface name, followed by the method name. Static interface methods are useful for defining utility methods related to the interface. They are not inherited by implementing classes or sub interfaces.

Syntax of a Static Interface Method

A static method is declared using the static keyword inside an interface.

Example

```
interface MyIF {  
    int getNumber();  
    default String getString() {  
        return "Default String";  
    }  
    static int getDefaultNumber() {  
        return 0;  
    }  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Here,

- `getNumber()` is an abstract method.
- `getString()` is a default method.
- `getDefaultNumber()` is a static method that returns 0.

Calling a Static Interface Method

A static interface method is called using the interface name. No object or implementation class is required.

Example

```
int defNum = MyIF.getDefaultNumber();  
System.out.println(defNum);
```

Output

0

The method `getDefaultNumber()` is executed directly from the interface.

Characteristics of Static Interface Methods

- Introduced in JDK 8.
- Declared using the static keyword.
- Can be called without creating an object.
- Accessed using `InterfaceName.methodName()`.
- Cannot be overridden by implementing classes.
- Are not inherited by implementing classes or subinterfaces.
- Mainly used for utility or helper methods related to the interface.

General Syntax

```
InterfaceName.staticMethodName();
```

Example

```
MyIF.getDefaultNumber();
```

IV. PRIVATE INTERFACE METHODS

Beginning with **JDK 9**, Java introduced **private methods in interfaces**. A private interface method can only be called by **default methods** or **other private methods** within the same interface. Since it is declared with the **private** access modifier, it cannot be accessed by implementing classes, external code, or even subinterfaces because private methods are **not inherited**.

The main advantage of a private interface method is **code reusability**. When multiple default methods require the same functionality, the common code can be placed inside a private method. This avoids **code duplication**, improves readability, and makes the interface easier to maintain.

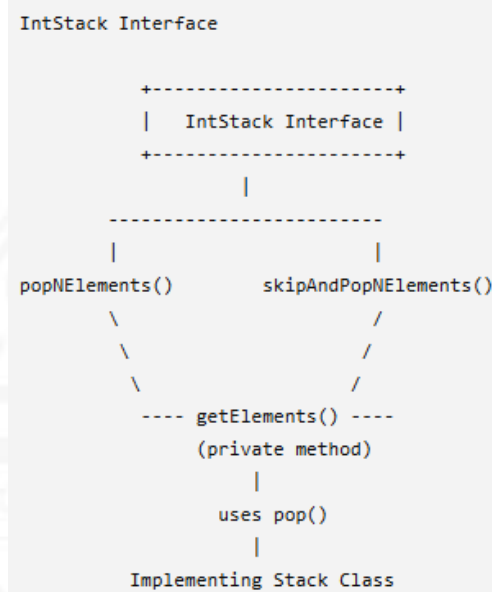
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example: IntStack Interface

Suppose the **IntStack** interface contains two default methods:

- **popNElements()** – Returns an array containing the top N elements from the stack.
- **skipAndPopNElements()** – Skips a specified number of elements and then returns the next N elements as an array.

Both of these default methods need similar code to retrieve elements from the stack. Instead of repeating the same logic, they call a **private method** named **getElements()**.



It is Introduced in **JDK 9**. Declared using the **private** keyword. Can only be called by **default methods** or **private methods** within the same interface. Cannot be accessed outside the interface or inherited by subinterfaces. Helps eliminate duplicate code by placing common functionality in one method. In the **IntStack** example, both **popNElements()** and **skipAndPopNElements()** use the private **getElements()** method.

The **getElements()** method internally calls **pop()**, which executes the implementation provided by the class implementing **IntStack**. This ensures the private method works correctly for any class that implements the interface.