ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: Object Oriented Programming with JAVA

COURSE CODE: 1BCS302

SEMESTER: III

COURSE COORDINATOR: Mrs.DIVYA N

(ACADEMIC YEAR 2026-27)

INSTITUTIONAL VISION AND MISSION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

• To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
• To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
• To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT VISION AND MISSION

Vision

To be a global leader in Computer Science and Design Engineering striving for design excellence, cultural awareness, a profound commitment to environmental stewardship, and societal responsibility.

Mission

M1: To establish a technology-enabled experiential learning environment, prioritizing and cultivating problem-solving and design thinking skills among students.

M2: To foster collaboration with industries, research and development organizations, jointly addressing socially relevant challenges in Computer Science with a core emphasis on design.

Program Educational Objective (PEO's)

- Demonstrate the proficiency and adaptability needed to seamlessly integrate with the latest advancements and changes in Computer Science and Design.
- Our graduates will possess the adaptability to explore various career paths, such as Research & Development, Manufacturing, and Entrepreneurship with independent design firms, Design Services, and Consultancies.
- Our students will be well-prepared to excel in collaborative, multi-disciplinary environments, showcasing their versatility and skills.

Program Specific Outcomes (PSO's)

- To develop stand-alone, embedded, and web-based solutions with easy-to-operate interfaces using software engineering practices and contemporary programming languages.
- Design and develop computer-based systems in various areas of Multimedia, Graphics data visualization and computer vision.

Inheritance: Inheritance Basics, using super, creating a Multilevel Hierarchy, When Constructors Are Executed, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, Local Variable Type Inference and Inheritance, The Object Class.

Packages: Packages, Packages and Member Access, Importing Packages.

Interfaces: Interfaces, Default Interface Methods, Use static Methods in an Interface, Private Interface Methods.

Inheritance Basics

Inheritance is a fundamental concept in object-oriented programming, and it plays a crucial role in Java. It allows programmers to create new classes based on existing classes, enabling code reusability and organisation. Inheritance establishes an ***IS-A*** relationship between classes, where a subclass inherits the properties and behaviours of its superclass. At its core, inheritance in Java promotes code reuse and modularity. Extending existing classes allows you to avoid duplicating code and build upon a well-defined foundation.

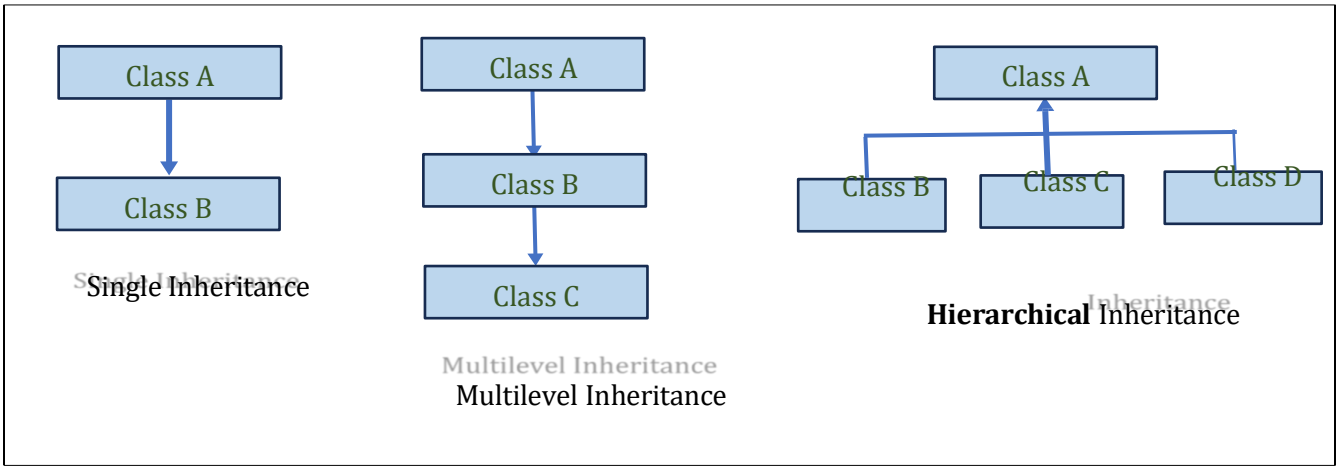
To understand inheritance in Java better, let us familiarise ourselves with some common terms used in inheritance programs in Java.

- **Class:** A blueprint or template that defines the properties and behaviours of objects.
- **Subclass/Child class:** A class that inherits from another class. It extends the functionality of the superclass by adding new methods or overriding existing ones.
- **Superclass/Parent class:** The class from which another class inherits. It provides a base for subclasses to build upon.
- **Reusability:** The ability to reuse code from an existing class in a new class, reducing redundancy and promoting efficiency.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: **single**, **multilevel** and **hierarchical**.

Note: Multiple inheritance is not supported in Java through class.



Single Inheritance

Single inheritance in Java refers to the scenario where a subclass extends only one superclass. It creates a simple hierarchy where one subclass inherits the properties and behaviours of a single superclass. Consider the following inheritance example in Java:

Example:

```
class CSDept {
    void showDetails() {
        System.out.println("ATME College CS Department");
    }
}

class CSD extends CSDept {
    void displayBranch() {
        System.out.println("COMPUTER SCIENCE & DESIGN Engineering Department.");
    }
}

public class ATME {
    public static void main(String[] args) {
        CSD obj = new CSD();  // Object of the child class
        obj.showDetails();    // Inherited method from the parent class
        obj.displayBranch();  // Method of the child class
    }
}
```

- **Parent Class (CSDept):** Defines a generic representation of the Computer Science department.
- **Child Class (CSD):** Inherits the showDetails() method from CSDept and adds its own functionality with the displayBranch() method.
- **Single Inheritance:** A single parent (CSDept) is inherited by one child class (CSD).

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
ATME College CS Department
COMPUTER SCIENCE & DESIGN Engineering Department.
```

Multilevel Inheritance:

Multi-level inheritance in Java occurs when a subclass becomes a superclass for another class, creating a chain of inheritance. In this type of inheritance, each class in the hierarchy inherits properties and behaviours from its immediate superclass. Consider the following inheritance example in Java:

Example:

- **Base Class (CSDept):** Represents the general Computer Science department.
- **Intermediate Class (CSD):** Inherits from CSDept and includes details about the Computer Science & Design department.
- **Derived Class (CSDProductDesign):** Extends CSD to focus on a specialization in Design and Development, relevant to design engineering.

```java
// Base class representing the CS Department
class CSDept {
    void showDetails() {
        System.out.println("ATME College CS Department");
    }
}

// Intermediate class representing a specific branch
class CSD extends CSDept {
    void displayBranch() {
        System.out.println("COMPUTER SCIENCE & DESIGN Engineering Department.");
    }
}

// Derived class representing a specific specialization in Design Engineering
class CSDProductDesign extends CSD {
    void displaySpecialization() {
        System.out.println("Specialization: Design and Development.");
    }
}

public class ATME {
    public static void main(String[] args) {
        // Object of the most derived class
        CSDProductDesign obj = new CSDProductDesign();

        obj.showDetails();            // Inherited from the base class
        obj.displayBranch();          // Inherited from the intermediate class
        obj.displaySpecialization();  // Defined in the derived class
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
ATME College CS Department
COMPUTER SCIENCE & DESIGN Engineering Department.
Specialization: Design and Development.
```

Hierarchical Inheritance

Hierarchical inheritance occurs when multiple subclasses inherit from a single superclass. It allows multiple classes to share common properties and behaviours defined in the superclass.

Example:

- **Base Class (CSDept)**:
  - Represents the general Computer Science department.
  - Provides a common method, showDetails, accessible to all derived classes.
- **Derived Classes**:
  - **CSD**: Represents the Computer Science and Design branch.
  - **AIML**: Represents the Artificial Intelligence and Machine Learning branch.
  - **CyberSecurity**: Represents the Cybersecurity branch.

Each derived class has its own method (displayBranch) to display specific branch details, but all inherit the general showDetails method from the base class. This structure demonstrates **hierarchical inheritance**, where multiple derived classes inherit from a single base class.

Example:

```java
// Base class representing the CS Department
class CSDept {
    void showDetails() {
        System.out.println("ATME College Computer Science Department");
    }
}

// Derived class for Computer Science and Design branch
class CSD extends CSDept {
    void displayBranch() {
        System.out.println("Branch: Computer Science and Design (CSD)");
    }
}

// Derived class for Artificial Intelligence and Machine Learning branch
class AIML extends CSDept {
    void displayBranch() {
        System.out.println("Branch: Artificial Intelligence and Machine Learning (AIML)");
    }
}

// Derived class for Cybersecurity branch
class CyberSecurity extends CSDept {
    void displayBranch() {
        System.out.println("Branch: Cybersecurity");
    }
}

public class ATME {
    public static void main(String[] args) {
        // Creating objects for each branch
        CSD csd = new CSD();
        AIML aiml = new AIML();
        CyberSecurity cy = new CyberSecurity();

        // Accessing methods of the base and derived classes
        csd.showDetails();
        csd.displayBranch();

        aiml.showDetails();
        aiml.displayBranch();

        cy.showDetails();
        cy.displayBranch();

    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
ATME College Computer Science Department
Branch: Computer Science and Design (CSD)
ATME College Computer Science Department
Branch: Artificial Intelligence and Machine Learning (AIML)
ATME College Computer Science Department
Branch: Cybersecurity
```

Multiple Inheritance:

Java doesn't support multiple inheritance to avoid the 'diamond problem', where ambiguity arises when a class inherits from two classes with the same method names. Instead, Java uses interfaces to achieve similar functionality while maintaining clarity and simplicity.

Super Keyword:

The super keyword in Java is used to access members (methods, variables, or constructors) of a parent class from a child class. It is primarily used in the context of **inheritance** to resolve ambiguity when both the parent and child class have the same members or to explicitly call a parent class's constructor.

Usage of Super Keyword:

- **Access Parent Class Methods**:
  - Used to call a method from the parent class that is overridden in the child class.
- **Access Parent Class Variables**:
  - Access variables of the parent class when the child class has a variable with the same name.
- **Invoke Parent Class Constructor**:
  - Used to call a specific constructor of the parent class from the child class.

Example:

```java
// Base class representing the CS Department
class CSDept {
    String departmentName = "ATME College Computer Science Department";

    // Constructor of the base class
    CSDept() {
        System.out.println("CSDept Constructor Called");
    }

    // Method to show the department details
    void showDetails() {
        System.out.println(departmentName);
    }
}

// Subclass representing the Computer Science and Design (CSD) branch
class CSD extends CSDept {
    String departmentName = "Computer Science and Design Branch"; // Overriding the variable

    // Constructor of the CSD class
    CSD() {
        super();  // Calling the parent class constructor (CSDept)
        System.out.println("CSD Constructor Called");
    }

    // Method to display the branch details
    void displayBranch() {
        super.showDetails();  // Calls the showDetails() method from CSDept
        System.out.println("Branch: " + departmentName);  // Uses the CSD class's departmentName
    }
}

public class ATME {
    public static void main(String[] args) {
        // Creating an object of CSD (Computer Science and Design) branch
        CSD csdBranch = new CSD();
        csdBranch.displayBranch();  // Display details of the CSD branch
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
CSDept Constructor Called
CSD Constructor Called
ATME College Computer Science Department
Branch: Computer Science and Design Branch
```

Method Overriding

Method Overriding in Java is a feature that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is a fundamental concept in Object-Oriented Programming (OOP) that promotes dynamic polymorphism, allowing objects to behave differently based on their actual type rather than their reference type.

In Java, **method overriding** is used when:

- The **method signature** (name, return type, and parameters) in the subclass matches the method signature in the superclass.
- The method in the subclass **replaces** or **modifies** the behavior of the inherited method

Some important points of Method Overriding

- **Method Signature**: The method signature in the subclass must match the one in the superclass.
- **Access Modifiers**: The access modifier of the overriding method in the subclass must be the same or more permissive than the one in the superclass (e.g., a method with protected in the superclass can be overridden with protected or public).
- **Return Type**: The return type in the subclass can either be the same as the superclass's return type or a subtype (covariant return type).
- **@Override Annotation**: Although optional, the @Override annotation is recommended as it ensures that the method is correctly overriding the superclass method and avoids mistakes like typos in method signatures.
- **Inheritance**: Method overriding is only possible in the context of inheritance. A subclass can override a method of its superclass, but cannot override a method from another class that is not in the same hierarchy.

Why is Method Overriding Important?

1. **Polymorphism**: Method overriding allows for **dynamic method dispatch**, meaning that the appropriate method is called based on the actual object type, not the reference type. This is a form of **runtime polymorphism**.

2. **Customization**: It allows subclasses to provide their own implementation of a method without changing the interface or the method signature of the parent class. This promotes flexibility in object-oriented systems.

3. **Code Reusability**: By overriding methods, subclasses can reuse code from the parent class and still customize it to their needs. This avoids redundant code and encourages code reuse.

Example:

```java
// Base class representing the CS Department
class CSDept {
    String departmentName = "ATME College Computer Science Department";

    // Method to show general department details (to be overridden)
    void showDetails() {
        System.out.println(departmentName);
    }
}

// Subclass representing the Computer Science and Design (CSD) branch
class CSD extends CSDept {
    String departmentName = "Computer Science and Design Branch"; // Overriding the variable

    // Overriding the showDetails method to display branch-specific information
    @Override
    void showDetails() {
        super.showDetails();  // Calling the parent class's showDetails method
        System.out.println("Branch: " + departmentName);
        System.out.println("Focus: Design, Development, and Architecture of Software");
    }
}

public class ATME {
    public static void main(String[] args) {
        // Creating an object for CSD (Computer Science and Design) branch
        CSDept csdDept = new CSD();

        // Calling the showDetails method for the CSD branch
        System.out.println("Displaying details of CSD Branch:");
        csdDept.showDetails();  // Calls the overridden method in CSD
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
Displaying details of CSD Branch:
ATME College Computer Science Department
Branch: Computer Science and Design Branch
Focus: Design, Development, and Architecture of Software
```

Dynamic Method Dispatch in Java

Dynamic Method Dispatch is a mechanism in Java that allows the Java Virtual Machine (JVM) to determine at runtime which method to invoke, based on the object being referred to by the reference variable, rather than the type of the reference variable itself. This is closely related to **runtime polymorphism** and is a key feature of object-oriented programming in Java.

Key Points:

- **Runtime Polymorphism**: The method that is executed is determined at runtime, not compile-time.
- **Method Overriding**: It works when a method in the subclass overrides a method in the superclass.
- **Upcasting**: The reference variable is of the superclass type, but the object it points to is of the subclass type.

How Dynamic Method Dispatch Works:

- When you call a method on a reference variable, Java decides which version of the method to call (superclass or subclass method) based on the actual object the reference variable points to.
- Even if the reference variable is of the superclass type, Java will call the overridden method from the subclass if the object being referred to is an instance of that subclass.

Example:

```java
// Base class representing the CS Department
class CSDept {
    // Method to show general department details (to be overridden)
    void showDetails() {
        System.out.println("ATME College Computer Science Department");
    }
}

// Subclass representing the Computer Science and Design (CSD) branch
class CSD extends CSDept {
    // Overriding the showDetails method to display branch-specific information
    @Override
    void showDetails() {
        System.out.println("ATME College Computer Science and Design Branch");
        System.out.println("Focus: Design, Development, and Architecture of Software");
    }
}

public class ATME {
    public static void main(String[] args) {
        // Creating a reference of CSDept but pointing to an object of CSD
        CSDept dept = new CSD();

        // Calling the showDetails method - this will invoke the method of CSD at runtime
        dept.showDetails();  // Dynamic Method Dispatch

        // Creating another reference of CSDept but pointing to an object of CSDept
        CSDept generalDept = new CSDept();

        // Calling the showDetails method - this will invoke the method of CSDept
        generalDept.showDetails();
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
ATME College Computer Science and Design Branch
Focus: Design, Development, and Architecture of Software
ATME College Computer Science Department
```

Why is Dynamic Method Dispatch Important?

- **Polymorphism**: It enables polymorphism, where the same method name can behave differently based on the object being referred to, even though the reference type is the same.
- **Flexibility and Extensibility**: Dynamic method dispatch makes it possible to write more flexible and reusable code. New subclasses can be added without changing the existing code, as long as they adhere to the method signature in the superclass.

Using Abstract Classes

Abstract Classes in Java

An abstract class in Java is a class that cannot be instantiated on its own and is meant to be inherited by other classes. It can have both abstract methods (methods without implementation) and concrete methods (methods with implementation). Abstract classes allow you to define a template for other classes, forcing them to implement certain methods while providing default functionality where needed.

Key Features of Abstract Classes:

1. **Abstract Methods:** These methods have no implementation in the abstract class. Subclasses are required to provide their implementation for these methods.
2. **Concrete Methods:** These methods have a body (implementation) in the abstract class, and the subclasses can either use them as is or override them.
3. **Cannot be Instantiated:** You cannot create an instance of an abstract class directly. Instead, you instantiate subclasses that provide concrete implementations for the abstract methods.
4. **Can Have Constructors:** Abstract classes can have constructors that can be called by the subclass constructors.

Example:

```java
// Abstract class representing the CS Department
abstract class CSDept {
    // Abstract method (does not have a body)
    abstract void showDetails();
}

// Subclass representing the Computer Science and Design (CSD) branch
class CSD extends CSDept {
    // Implementing the abstract method showDetails()
    @Override
    void showDetails() {
        System.out.println("ATME College Computer Science and Design Branch");
        System.out.println("Focus: Design, Development, and Architecture of Software");
    }
}

public class ATME {
    public static void main(String[] args) {
        // Creating an object for CSD (Computer Science and Design) branch
        CSDept dept = new CSD(); // Creating an instance of the subclass

        // Calling the showDetails method which is implemented in CSD
        dept.showDetails();  // Calls the overridden method in CSD
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
ATME College Computer Science and Design Branch
Focus: Design, Development, and Architecture of Software
```

Using final with Inheritance

In Java, the final keyword can be used in various ways to restrict inheritance or modification of classes, methods, and variables. Here's how the final keyword works in the context of inheritance:

- **Using final with Classes:** When a class is declared as final, it cannot be subclassed. This means no other class can extend a final class. This is useful when you want to prevent further modification of a class through inheritance.
- **Using final with Methods:** If a method in a class is declared as final, it cannot be overridden by any subclass. This ensures that the behavior defined in the final method is preserved across all subclasses.

Example:

```java
// Final class representing the CS Department
final class CSDept {
    // Final method that cannot be overridden
    final void showDetails() {
        System.out.println("ATME College Computer Science Department");
    }
}

// Attempting to subclass the final class would result in a compile-time error
// class CSD extends CSDept {   // This will give a compile-time error
//     void showDetails() {    // Cannot override showDetails() because it is final
//         System.out.println("Computer Science and Design Branch");
//     }
// }

public class ATME {
    public static void main(String[] args) {
        // Creating an object of CSDept
        CSDept dept = new CSDept();

        // Calling the showDetails method
        dept.showDetails();  // Calls the final method from CSDept
    }
}
```

- **Using final with Variables:** If a variable is declared as final, its value cannot be changed once it is initialized. This is useful when you want to define constants or prevent reassignment of variables.

```java
class CSDept {
    final String departmentName = "ATME College Computer Science Department";

    void showDetails() {
        System.out.println(departmentName);
    }
}

class CSD extends CSDept {
    void showDetails() {
        // departmentName = "Computer Science and Design Branch";
        // Compile-time error: cannot assign a value to final variable
        System.out.println("Computer Science and Design Branch");
    }
}

public class ATME {
    public static void main(String[] args) {
        CSD csdDept = new CSD();
        csdDept.showDetails();
    }
}
```

Local Variable Type Inference and Inheritance in Java:

Java 10 introduced **Local Variable Type Inference** with the var keyword, which allows the compiler to infer the type of local variables from the context. This feature helps reduce verbosity by allowing developers to write cleaner and more readable code. It is particularly useful when the type of the variable is obvious from the initialization expression.

Inheritance and var Keyword:

The var keyword does not affect inheritance in Java. Inheritance operates as usual, and var will only help you infer the type of local variables within methods. For inheritance, you still need to work with the actual class types, whether they are inferred using var or explicitly declared.

Example:

```java
// Base class representing the CS Department
class CSDept {
    void showDetails() {
        System.out.println("ATME College Computer Science Department");
    }
}

// Subclass representing Computer Science and Design Branch
class CSD extends CSDept {
    @Override
    void showDetails() {
        System.out.println("ATME College Computer Science and Design Branch");
    }
}

public class ATME {
    public static void main(String[] args) {
        // Using var to refer to the parent class type
        var dept = new CSD();  // var infers the type CSD

        dept.showDetails();  // Calls the overridden method in CSD
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
ATME College Computer Science and Design Branch
```

The Object Class in Java

The Object class is the root class of the Java class hierarchy. Every class in Java, directly or indirectly, inherits from the Object class. It provides several important methods that are common to all objects in Java, such as toString(), equals(), hashCode(), and getClass().

Key Methods of the Object Class:

1. **toString()**: Returns a string representation of the object (often overridden).
2. **equals(Object obj)**: Compares the object with another object to check for equality.
3. **hashCode()**: Returns a hash code value for the object.
4. **getClass()**: Returns the runtime class of the object.
5. **clone()**: Creates a clone of the object (requires the class to implement Cloneable).
6. **wait() / notify() / notifyAll()**: Methods for thread synchronization (from java.lang.Object).

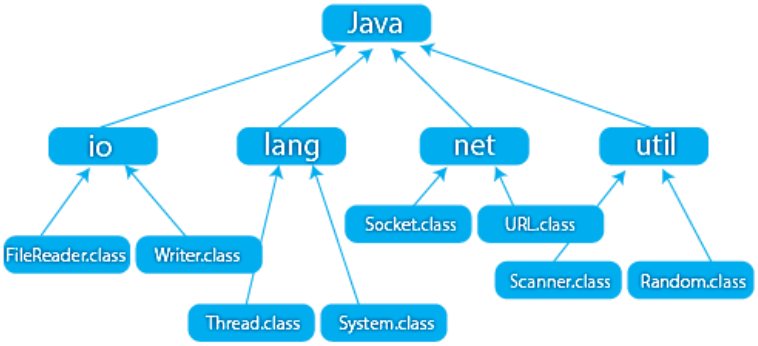
Example:

```java
// Base class representing the CS Department
class CSDept {
    // Overriding toString() method from the Object class
    @Override
    public String toString() {
        return "ATME College Computer Science Department";
    }
}

// Subclass representing Computer Science and Design Branch
class CSD extends CSDept {
    @Override
    public String toString() {
        return "ATME College Computer Science and Design Branch";
    }
}

public class ATME {
    public static void main(String[] args) {
        CSDept dept = new CSDept();
        CSD csdDept = new CSD();

        // Using the toString method from the Object class (overridden in both classes)
        System.out.println(dept.toString());  // Calls CSDept's toString()
        System.out.println(csdDept.toString());  // Calls CSD's toString()

        // Using getClass() method from the Object class to get runtime class info
        System.out.println(dept.getClass());  // class CSDept
        System.out.println(csdDept.getClass());  // class CSD
    }
}
```

Output:

```
C:\TestJava>java ATME
ATME College Computer Science Department
ATME College Computer Science and Design Branch
class CSDept
class CSD
```

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.



Examples of Built-in Packages:

- java.util: Contains utility classes like ArrayList, HashMap, etc.
- java.io: Includes classes for input and output operations.
- java.lang: Automatically imported; contains essential classes like String, Math, etc.

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Creating a Package

To create a package, follow the steps given below:

1) Define a package in your class file

- Use the package keyword at the top of your Java file.

```
package mypack;

public class MyClass {
    public void displayMessage() {
        System.out.println("Message from MyClass in mypack!");
    }
}
```

2) Save the file

- Save the above code in a directory structure matching the package name. File path: mypackage/MyClass.java

Package Naming Convention

We follow the naming convention rules to name a package. Java has some predefined packages and allows us to create our own package. So, it is possible that a programmer can create a class with the same name as a package that already contains that type in a predefined package.

- Package names should be written in **lowercase** to avoid conflicts with class names and to improve readability.
- Use meaningful names to describe the purpose of the package. This enhances code readability and maintainability.
- Do not use Java reserved keywords (e.g., class, public, static) in package names.
- Use a hierarchical structure for subpackages to logically group related classes.
- Avoid Abbreviations and Special Characters

Importing a package

1) Write a new Java program in another file

- Use the import statement to include the package.
- Example:

```
import mypack.MyClass;

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

2) Save this file in the same directory as mypack or elsewhere, depending on your setup.

3) Compile the program

Compile the program with the javac command, ensuring the mypack folder is in the classpath: ***javac Main.java***

4) Run the program

Run the program using the java command: ***java Main***

Packages and Member Access in Java

Java provides **access modifiers** to control the visibility of members (fields, methods, and constructors) across different classes and packages. Access levels are especially important when working with packages, as they determine how members are shared between classes in the same or different packages.

Access Levels and Modifiers

Modifier	Same Class	Same Package	Subclass (Different Package)	Other Packages
Public	✅	✅	✅	✅
Protected	✅	✅	✅	❌
Default	✅	✅	❌	❌
Private	✅	❌	❌	❌

Public

- Members declared as public are accessible **from any class**, regardless of whether the class is in the same or a different package.
- Use this modifier for members you want to make globally accessible.

```java
package package1;

public class ClassA {
    public void display() {
        System.out.println("Public method in ClassA");
    }
}

// In another package
package package2;

import package1.ClassA;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display();  // Accessible
    }
}
```

Protected

- protected members are accessible:
    - Within the same package.
    - In subclasses in other packages through inheritance.

In subclasses in other packages through inheritance

```java
package package1;

public class ClassA {
    protected void display() {
        System.out.println("Protected method in ClassA");
    }
}

// In the same package
package package1;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display();  // Accessible
    }
}

// In a different package (through inheritance)
package package2;

import package1.ClassA;

public class ClassC extends ClassA {
    public static void main(String[] args) {
        ClassC obj = new ClassC();
        obj.display();  // Accessible through inheritance
    }
}
```

Default (No Modifier)

- Members without an explicit modifier are accessible only **within the same package**.
- This is also called **package-private** access

```java
class ClassA {
    void display() {
        System.out.println("Default method in ClassA");
    }
}

// In the same package
package package1;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display();  // Accessible
    }
}

// In a different package
package package2;

import package1.ClassA;

public class ClassC {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display();  // NOT accessible
    }
}
```

Private

- Members declared as private are accessible **only within the class** where they are defined.
- Not accessible from other classes, even in the same package.

```
package package1;

public class ClassA {
    private void display() {
        System.out.println("Private method in ClassA");
    }

    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display();  // Accessible within the same class
    }
}

// In the same package
package package1;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display();  // NOT accessible
    }
}
```

Interfaces:

In Java, **interfaces** are used to represent a contract that classes must adhere to. An interface defines a set of abstract methods (methods without implementation) that the implementing classes must provide. Interfaces cannot have method implementations (except in the case of default or static methods). They can, however, include constants and abstract methods.

Key Points about Interfaces

- **Abstract Methods**: In Java 7 and earlier, interfaces could only have abstract methods, meaning methods without implementations. All methods in an interface were implicitly public and abstract.
- **Default Methods**: From Java 8 onward, interfaces can have default methods that provide a body. This allows interfaces to evolve over time by adding new methods without breaking existing implementations.
- **Static Methods**: Static methods can also be defined in interfaces from Java 8 onwards. They cannot be overridden by implementing classes.
- **Private Methods**: Introduced in Java 9, private methods in interfaces allow you to create helper methods that are not accessible from outside the interface but can be used in default and static methods.

S.No.	Abstract Class	Interface
1.	An abstract class can contain both abstract and non-abstract methods.	Interface contains only abstract methods.
2.	An abstract class can have all four; static, non-static and final, non-final variables.	Only final and static variables are used.
3.	To declare abstract class abstract keywords are used.	The interface can be declared with the interface keyword.
4.	It supports multiple inheritance.	It does not support multiple inheritance.
5.	The keyword 'extend' is used to extend an abstract class	The keyword implement is used to implement the interface.
6.	It has class members like private and protected, etc.	It has class members public by default.

Basic Interface

An interface can be defined with the interface keyword, and it contains abstract methods (by default, methods in interfaces are abstract unless they are static or default methods).

```
interface CSDept {
    // Abstract method (no implementation)
    void showDetails();
}
```

Implementing an Interface

A class that implements an interface must provide concrete implementations for all the abstract methods declared in the interface. The class uses the implements keyword to implement the interface.

```
interface CSDept {
    // Abstract method (no implementation)
    void showDetails();
}

class CSD implements CSDept {
    @Override
    public void showDetails() {
        System.out.println("Computer Science and Design Branch at ATME College");
    }
}
```

Default Interface Methods

In **Java 8 and later**, interfaces can have **default methods**. A default method has a body (implementation) in the interface itself. This allows new methods to be added to interfaces without breaking existing implementations. Default methods are defined using the default keyword.

Example:

```
interface CSDept {
    // Abstract method
    void showDetails();

    // Default method
    default void printDepartmentName() {
        System.out.println("ATME College Computer Science Department");
    }
}

class CSD implements CSDept {
    @Override
    public void showDetails() {
        System.out.println("Computer Science and Design Branch at ATME College");
    }
}

public class ATME {
    public static void main(String[] args) {
        CSD csd = new CSD();
        csd.showDetails();          // Calls the overridden method
        csd.printDepartmentName(); // Calls the default method from the interface
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
Computer Science and Design Branch at ATME College
ATME College Computer Science Department
```

- printDepartmentName() is a default method in the CSDept interface. It is automatically inherited by the CSD class, so there is no need to implement it in the CSD class unless you want to override it.
- showDetails() is an abstract method that must be implemented by the CSD class.

Static Methods in an Interface

Java 8 introduced **static methods** in interfaces. Static methods in interfaces work similarly to static methods in classes. They can be called on the interface itself, not on an instance of the interface. Static methods can have a body (implementation), but they cannot be overridden in implementing classes.

Example:

```
interface CSDept {
    // Static method in an interface
    static void displayInfo() {
        System.out.println("Welcome to ATME College");
    }
}

public class ATME {
    public static void main(String[] args) {
        // Calling the static method on the interface
        CSDept.displayInfo();
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
Welcome to ATME College
```

Private Methods in an Interface

Java 9 introduced **private methods** in interfaces. Private methods allow you to write helper methods within the interface that cannot be accessed from outside the interface. These private methods can be used to reduce code duplication in default methods or static methods.

Example:

```java
interface CSDept {
    // Abstract method
    void showDetails();

    // Default method using a private method
    default void departmentInfo() {
        printDepartmentDetails();
        System.out.println("Welcome to ATME College");
    }

    // Private helper method
    private void printDepartmentDetails() {
        System.out.println("Department: Computer Science");
    }
}

class CSD implements CSDept {
    @Override
    public void showDetails() {
        System.out.println("Computer Science and Design Branch");
    }
}

public class ATME {
    public static void main(String[] args) {
        CSD csd = new CSD();
        csd.showDetails();
        csd.departmentInfo(); // Calls the default method, which uses the private method
    }
}
```

Output:

```
C:\TestJava>javac ATME.java

C:\TestJava>java ATME
Computer Science and Design Branch
Department: Computer Science
Welcome to ATME College
```

- The printDepartmentDetails() method is private and can only be called within the interface. It helps avoid code duplication.
- The departmentInfo() default method uses printDepartmentDetails() to print details about the department.
- The CSD class does not need to implement the private method, but it can use the public/default methods that internally use the private methods.

Summary of Interface Features:

- **Abstract methods**: Methods without implementation; must be implemented by classes that implement the interface.
- **Default methods**: Methods with a default implementation; can be used by implementing classes without needing to be overridden.
- **Static methods**: Methods with a body that belong to the interface and are called on the interface, not instances.
- **Private methods**: Helper methods used within the interface, primarily for use in default and static methods, and cannot be accessed outside the interface.

When to Use These Features:

- **Abstract Methods**: When you want to define a contract that must be implemented by the classes.
- **Default Methods**: When you want to provide a default implementation that can be optionally overridden by implementing classes.
- **Static Methods**: When you need utility methods that belong to the interface itself, not instances.
- **Private Methods**: When you need to avoid code duplication inside the interface and encapsulate common logic used in default or static methods.