

**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.

**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.

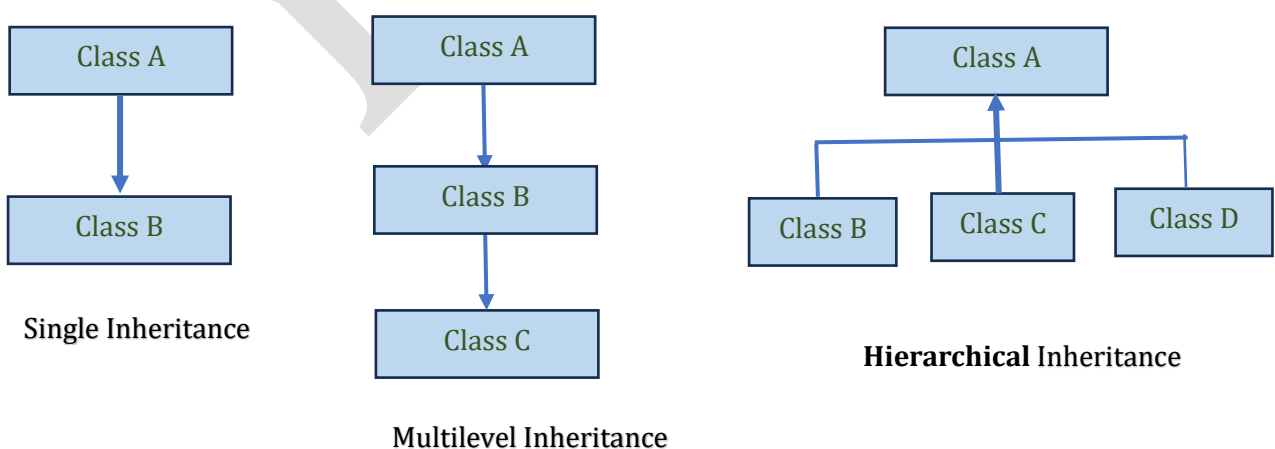
To understand inheritance in Java better, let us familiarise ourselves with some common terms used in inheritance programs in Java.

- **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:

```
// Parent class
class ComputerApplication {
    void showDetails() {
        System.out.println("ATME College - Department of Computer Applications");
    }
}

// Child class representing a program under the department
class BCA extends ComputerApplication {
    void displayProgram() {
        System.out.println("Bachelor of Computer Applications (BCA) Program");
    }
}

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

- Parent Class (ComputerApplication): Defines a generic representation of the Department of Computer Applications at ATME College.
- Child Class (BCA): Inherits the showDetails() method from ComputerApplication and adds its own functionality with the displayProgram() method to represent the BCA Program.
- Single Inheritance: A single parent class (ComputerApplication) is inherited by one child class (BCA).

### 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 representing the Computer Applications Department
class ComputerApplication {
    void showDepartment() {
        System.out.println("ATME College - Department of Computer Applications");
    }
}

// Derived class representing the BCA Program under the department
class BCA extends ComputerApplication {
    void showProgram() {
        System.out.println("Program: Bachelor of Computer Applications (BCA)");
    }
}

// Derived class representing a specialized BCA program
class BCA_DataScience extends BCA {
    void showSpecialization() {
        System.out.println("Specialization: Data Science");
    }
}

// Main class
public class Multilevel {
    public static void main(String[] args) {
        // Creating object of the most derived class
        BCA_DataScience student = new BCA_DataScience();

        // Accessing methods from all levels of inheritance
        student.showDepartment(); // Method from base class
        student.showProgram();    // Method from intermediate class
        student.showSpecialization(); // Method from derived class
    }
}
```

### Output:

```
C:\Users\YESHASHWINI>javac Multilevel.java

C:\Users\YESHASHWINI>java Multilevel
ATME College - Department of Computer Applications
Program: Bachelor of Computer Applications (BCA)
Specialization: Data Science
```

### 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 (ComputerApplication):**
  - Represents the general *Department of Computer Applications*.
  - Provides a common method `showDetails()` accessible to all derived classes.
- **Derived Classes:**
  - BCA: Represents the *Bachelor of Computer Applications* general program.
  - BCA\_DS: Represents the *BCA – Data Science* specialization.
  - BCA\_AI: Represents the *BCA – Artificial Intelligence* specialization.

Each derived class defines its own method (displayProgram()) to show specific program details, while 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:**

```
// Base class representing the Computer Applications Department
class ComputerApplication {
    void showDetails() {
        System.out.println("ATME College - Department of Computer Applications");
    }
}

// Derived class for BCA General Program
class BCA extends ComputerApplication {
    void displayProgram() {
        System.out.println("Program: Bachelor of Computer Applications (BCA)");
    }
}

// Derived class for BCA with Data Science specialization
class BCA_DS extends ComputerApplication {
    void displayProgram() {
        System.out.println("Program: BCA - Data Science Specialization");
    }
}

// Derived class for BCA with Artificial Intelligence specialization
class BCA_AI extends ComputerApplication {
    void displayProgram() {
        System.out.println("Program: BCA - Artificial Intelligence Specialization");
    }
}

// Main class
public class ATME {
    public static void main(String[] args) {
        // Creating objects for each program
        BCA general = new BCA();
        BCA_DS ds = new BCA_DS();
        BCA_AI ai = new BCA_AI();

        // Accessing methods of the base and derived classes
        general.showDetails();
        general.displayProgram();

        ds.showDetails();
        ds.displayProgram();

        ai.showDetails();
        ai.displayProgram();
    }
}
```

**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:

```
// Base class representing the Computer Applications Department
class ComputerApplication {
    String departmentName = "ATME College - Department of Computer Applications";

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

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

// Subclass representing the BCA program
class BCA extends ComputerApplication {
    String departmentName = "Bachelor of Computer Applications (BCA) Program"; // Overriding variable

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

    // Method to display program details
    void displayProgram() {
        super.showDetails(); // Calls showDetails() from ComputerApplication
        System.out.println("Program: " + departmentName); // Uses BCA's departmentName
    }
}

// Main class
public class SuperKeyword {
    public static void main(String[] args) {
        // Creating an object of the BCA program
        BCA bp = new BCA();
        bp.displayProgram(); // Display details of the BCA program
    }
}
```

**Output:**

```
C:\Users\YESHASHWINI>javac SuperKeyword.java  
C:\Users\YESHASHWINI>java SuperKeyword  
ComputerApplication Constructor Called  
BCA Constructor Called  
ATME College – Department of Computer Applications  
Program: Bachelor of Computer Applications (BCA) Program
```

**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:**

```
// Base class representing the Computer Applications Department
class ComputerApplication {
    String departmentName = "ATME College - Department of Computer Applications";

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

// Subclass representing the BCA Program
class BCA extends ComputerApplication {
    String departmentName = "Bachelor of Computer Applications (BCA) Program"; // Overriding the variable

    // Overriding the showDetails method to display program-specific information
    @Override
    void showDetails() {
        super.showDetails(); // Calling the parent class's showDetails() method
        System.out.println("Program: " + departmentName);
        System.out.println("Focus: Software Development, Web Technologies, and Data Analytics");
    }
}

// Main class
public class MethodOverriding {
    public static void main(String[] args) {
        // Creating an object for BCA Program
        ComputerApplication dept = new BCA();

        // Calling the showDetails method for the BCA Program
        System.out.println("Displaying details of BCA Program:");
        dept.showDetails(); // Calls the overridden method in BCA
    }
}
```

**Output:**

```
C:\Users\YESHASHWINI>javac MethodOverriding.java

C:\Users\YESHASHWINI>java MethodOverriding
Displaying details of BCA Program:
ATME College - Department of Computer Applications
Program: Bachelor of Computer Applications (BCA) Program
Focus: Software Development, Web Technologies, and Data Analytics
```

**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:

```
// Base class
class ComputerApplication {
    void showDetails() {
        System.out.println("ATME College - Department of Computer Applications");
    }
}

// Derived class 1
class BCA extends ComputerApplication {
    @Override
    void showDetails() {
        System.out.println("Program: Bachelor of Computer Applications (BCA)");
    }
}

// Derived class 2
class MCA extends ComputerApplication {
    @Override
    void showDetails() {
        System.out.println("Program: Master of Computer Applications (MCA)");
    }
}

// Main class
public class DMDATME {
    public static void main(String[] args) {
        // Reference variable of parent class
        ComputerApplication ref;

        // Refers to BCA object
        ref = new BCA();
        ref.showDetails(); // Calls BCA's showDetails() method

        // Refers to MCA object
        ref = new MCA();
        ref.showDetails(); // Calls MCA's showDetails() method
    }
}
```

### Output:

```
C:\Users\YESHASHWINI>javac DMDATME.java

C:\Users\YESHASHWINI>java DMDATME
Program: Bachelor of Computer Applications (BCA)
Program: Master of Computer Applications (MCA)
```

### 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:

```
// Abstract class representing the Computer Application Department
abstract class CompAppDept {
    // Abstract method (does not have a body)
    abstract void showProgramDetails();
}

// Subclass representing the Bachelor of Computer Applications (BCA) program
class BCA extends CompAppDept {
    // Implementing the abstract method showProgramDetails()
    @Override
    void showProgramDetails() {
        System.out.println("ATME College Department of Computer Applications");
        System.out.println("Program: Bachelor of Computer Applications (BCA)");
        System.out.println("Focus: Application Development, Software Programming, and Business IT");
    }
}

public class ATMECollege {
    public static void main(String[] args) {
        // Creating an object for BCA program (Computer Applications)
        // Demonstrates Polymorphism: Referencing a subclass object with an abstract superclass type
        CompAppDept program = new BCA(); // Creating an instance of the subclass

        // Calling the showProgramDetails method which is implemented in BCA
        program.showProgramDetails(); // Calls the overridden method in BCA
    }
}
```

### Output:

```
C:\Users\YESHASHWINI>javac ATMECollege.java
C:\Users\YESHASHWINI>java ATMECollege
ATME College Department of Computer Applications
Program: Bachelor of Computer Applications (BCA)
Focus: Application Development, Software Programming, and Business IT
```

### 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:

```
// Final class representing the Computer Application Department
// A final class cannot be inherited (subclassed)
final class CompAppDept {

    // Final method that cannot be overridden by any subclass (though a subclass isn't possible here anyway)
    final void showDetails() {
        System.out.println("ATME College Department of Computer Applications");
    }
}

/*
// Attempting to subclass the final class would result in a compile-time error
// class BCA extends CompAppDept {
//     // This will give a compile-time error: cannot inherit from final class CompAppDept
// }
*/

public class FinalExample {
    public static void main(String[] args) {
        // Creating an object of the final class CompAppDept
        CompAppDept dept = new CompAppDept();

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

- **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.

```
// Class representing the Computer Application Department
class CompAppDept {
    // Final variable (a constant) - its value cannot be changed after initialization
    final String collegeName = "ATME College Department of Computer Applications";

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

// Subclass representing the BCA Program
class BCA extends CompAppDept {
    @Override
    void showDetails() {
        // Attempting to change the final variable 'collegeName' would cause a compile-time error
        // collegeName = "New College Name";
        // Compile-time error: cannot assign a value to final variable 'collegeName'

        System.out.println("Program: Bachelor of Computer Applications (BCA)");
        // We can still access the final variable from the superclass
        System.out.println("College: " + collegeName);
    }
}

public class FinalVariable {
    public static void main(String[] args) {
        BCA bp = new BCA();
        bp.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:

```
// Base class representing the Computer Application Department
class CompAppDept {
    void showDetails() {
        System.out.println("ATME College Department of Computer Applications");
    }
}

// Subclass representing the Bachelor of Computer Applications (BCA) program
class BCA extends CompAppDept {
    // Overriding the showDetails() method from the parent class
    @Override
    void showDetails() {
        System.out.println("ATME College Bachelor of Computer Applications (BCA) Program");
    }
}

public class UpdatedJava {
    public static void main(String[] args) {
        // Creating an object of the subclass BCA
        // The type 'var' is inferred to be BCA here.
        var program = new BCA();

        // Calling the showDetails method
        // This executes the overridden method defined in the BCA subclass
        program.showDetails();
    }
}
```

### Output:

```
C:\Users\YESHASHWINI>javac UpdatedJava.java
C:\Users\YESHASHWINI>java UpdatedJava
ATME College Bachelor of Computer Applications (BCA) Program
```

### 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:**

```
// Base class representing the Computer Application Department
class CompAppDept {
    // Overriding toString() method from the Object class
    @Override
    public String toString() {
        return "ATME College Department of Computer Applications";
    }
}

// Subclass representing the Bachelor of Computer Applications (BCA) program
class BCA extends CompAppDept {
    // Overriding toString() method to provide specific BCA details
    @Override
    public String toString() {
        return "ATME College Bachelor of Computer Applications (BCA) Program";
    }
}

public class ObjectClass {
    public static void main(String[] args) {
        // Creating objects of the superclass and subclass
        CompAppDept compAppDept = new CompAppDept();
        BCA bp = new BCA();

        // --- Using the toString() method (Overridden in both classes) ---
        System.out.println("--- toString() Output ---");
        // Calls CompAppDept's toString()
        System.out.println(compAppDept.toString());
        // Calls BCA's toString()
        System.out.println(bp.toString());

        // --- Using getClass() method from the Object class ---
        System.out.println("\n--- getClass() Output ---");
        // Returns the runtime class info: class CompAppDept
        System.out.println(compAppDept.getClass());
        // Returns the runtime class info: class BCA
        System.out.println(bp.getClass());
    }
}
```

**Output:**

```
C:\Users\YESHASHWINI>javac ObjectClass.java

C:\Users\YESHASHWINI>java ObjectClass
--- toString() Output ---
ATME College Department of Computer Applications
ATME College Bachelor of Computer Applications (BCA) Program

--- getClass() Output ---
class CompAppDept
class BCA
```

### 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 CADept {
    // Abstract method
    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 CADept {
    // Abstract method
    void showDetails();
}

class BCA implements CADept {
    @Override
    public void showDetails() {
        System.out.println("Bachelor of Computer Applications");
    }
}
```

## 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 CADept {
    // 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 Applications");
    }
}

class BCA implements CADept {
    @Override
    public void showDetails() {
        System.out.println("Bachelor of Computer Applications");
    }
}

public class ATME1 {
    public static void main(String[] args) {
        BCA ca = new BCA();
        ca.showDetails();
        ca.departmentInfo(); // Calls the default method, which uses the private method
    }
}
```

**Output:**

```
C:\Users\YESHASHWINI>javac ATME1.java

C:\Users\YESHASHWINI>java ATME1
Bachelor of Computer Applications
Department: Computer Applications
Welcome to ATME College
```

- `printDepartmentDetails()` is a default method in the **CADept** interface. It is automatically inherited by the **BCA** class, so there is no need to implement it in the **BCA** class unless you want to override it.
- `showDetails()` is an abstract method that must be implemented by the **BCA** 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:**

```
interface CADept {
    // 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 Applications");
    }
}

class BCA implements CADept {
    @Override
    public void showDetails() {
        System.out.println("Bachelor of Computer Applications");
    }
}

public class ATME1 {
    public static void main(String[] args) {
        BCA ca = new BCA();
        ca.showDetails();
        ca.departmentInfo(); // Calls the default method, which uses the private method
    }
}
```

### Output:

```
C:\Users\YESHASHWINI>javac ATME1.java
C:\Users\YESHASHWINI>java ATME1
Bachelor of Computer Applications
Department: Computer Applications
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.