# Module 2: Object-Oriented Design: Classes, Inheritance, and Polymorphism


Premanand S

Assistant Professor
School of Electronics Engineering
Vellore Institute of Technology
Chennai Campus

*premanand.s@vit.ac.in*


February 10, 2026

**In Module 1, we learned:**

# From Module 1 to Module 2

**In Module 1, we learned:**

- How to write Java programs
- Variables, data types, operators
- Control structures and loops
- Arrays and strings
- Using Java libraries

# From Module 1 to Module 2

**In Module 1, we learned:**

- How to write Java programs
- Variables, data types, operators
- Control structures and loops
- Arrays and strings
- Using Java libraries

**Focus of Module 1:**

- **Procedural thinking**

# Why Object-Oriented Design?

So far, our programs are:

# Why Object-Oriented Design?

So far, our programs are:

- Logic-based
- Function-focused
- Suitable for small problems

# Why Object-Oriented Design?

So far, our programs are:

- Logic-based
- Function-focused
- Suitable for small problems

**But real-world scenario is:**

- Large
- Complex
- Built using objects

# Why Object-Oriented Design?

So far, our programs are:

- Logic-based
- Function-focused
- Suitable for small problems

**But real-world scenario is:**

- Large
- Complex
- Built using objects

**This is where OOP comes in.**

# What Will Module 2 Teach You?

In this module, you will learn how to:

# What Will Module 2 Teach You?

In this module, you will learn how to:

- Model real-world entities using **classes and objects**

# What Will Module 2 Teach You?

In this module, you will learn how to:

- Model real-world entities using **classes and objects**
- Control access using **access specifiers**

# What Will Module 2 Teach You?

In this module, you will learn how to:

- Model real-world entities using **classes and objects**
- Control access using **access specifiers**
- Reuse code using **inheritance**

# What Will Module 2 Teach You?

In this module, you will learn how to:

- Model real-world entities using **classes and objects**
- Control access using **access specifiers**
- Reuse code using **inheritance**
- Achieve flexibility using **polymorphism**

# What Will Module 2 Teach You?

In this module, you will learn how to:

- Model real-world entities using **classes and objects**
- Control access using **access specifiers**
- Reuse code using **inheritance**
- Achieve flexibility using **polymorphism**
- Design robust systems using **interfaces and abstract classes**

# Module 2 — Big Picture

## Module 2 Topics

## Module 2 Topics

- Class Fundamentals
- Access and Non-Access Specifiers
- Declaring Objects and Assigning Object Reference Variables
- Array of Objects
- Constructors and Destructors
- Usage of `this` and `static` Keywords
- Enum Types and Their Iterations
- Inheritance and Its Types
- Use of `super` Keyword
- `final` Keyword
- Polymorphism
- Method Overloading and Method Overriding
- Abstract Classes
- Interfaces

A **class** is a blueprint or template.

# Class Fundamentals — What is a Class?

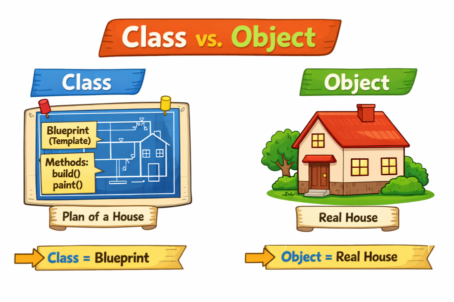
A **class** is a blueprint or template.

**It defines:**

- Data (variables)
- Behavior (methods)

# What is a Class?

- Class = Blueprint
- Object = Real entity



# Why Do We Need Classes?

Classes help us to:

# Why Do We Need Classes?

Classes help us to:

- Group related data and methods

# Why Do We Need Classes?

Classes help us to:

- Group related data and methods
- Represent real-world entities

# Why Do We Need Classes?

Classes help us to:

- Group related data and methods
- Represent real-world entities
- Build reusable and maintainable code

# Why Do We Need Classes?

Classes help us to:

- Group related data and methods
- Represent real-world entities
- Build reusable and maintainable code

**Example entities:**

- Student
- Employee
- BankAccount

# Class — Basic Syntax

```
class ClassName {
 // data members
 // methods
}
```

# Class Example 1

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

 void display() {
  System.out.println(id + " " + name);
 }
}

public class Main {
 public static void main(String[] args) {
  Student s1 = new Student();
  s1.id = 52818;
  s1.name = "Prem";
  s1.display();
 }
}
```

# Class Example 2

```
class BankAccount {
 int accountNumber;
 double balance;

 void deposit(double amount) {
  balance = balance + amount;
 }
 void showBalance() {
  System.out.println("Balance: " + balance);
 }
}
public class Main {
 public static void main(String[] args) {
  BankAccount acc = new BankAccount();
  acc.deposit(5000);
  acc.deposit(2000);
  acc.showBalance();
 }
}
```

**Consider the statement:**

# Class vs Object vs Reference

**Consider the statement:**

```
Student s1 = new Student();
```

# Class vs Object vs Reference

**Consider the statement:**

```
Student s1 = new Student();
```

- **Student** $\rightarrow$ Class (blueprint)

# Class vs Object vs Reference

**Consider the statement:**

```
Student s1 = new Student();
```

- **Student** $\rightarrow$ Class (blueprint)
- **new Student()** $\rightarrow$ Object (created in heap)

**Consider the statement:**

```
Student s1 = new Student();
```

- **Student** → Class (blueprint)
- **new Student()** → Object (created in heap)
- **s1** → Reference variable (stores address)

# Class vs Object vs Reference

**Consider the statement:**

```
Student s1 = new Student();
```

- **Student** → Class (blueprint)
- **new Student()** → Object (created in heap)
- **s1** → Reference variable (stores address)

**Key idea:**

- Reference is NOT the object

# Default Values of Instance Variables

**Example:**

# Default Values of Instance Variables

**Example:**

```
class Test {
 int a;
 double b;
 String s;
}
```

# Default Values of Instance Variables

**Example:**

```
class Test {
 int a;
 double b;
 String s;
}
```

**Default values:**

- `a → 0`

# Default Values of Instance Variables

**Example:**

```
class Test {
 int a;
 double b;
 String s;
}
```

**Default values:**

- a → 0
- b → 0.0

# Default Values of Instance Variables

**Example:**

```
class Test {
 int a;
 double b;
 String s;
}
```

**Default values:**

- `a → 0`
- `b → 0.0`
- `s → null`

# Default Values of Instance Variables

**Example:**

```
class Test {
 int a;
 double b;
 String s;
}
```

**Default values:**

- `a → 0`
- `b → 0.0`
- `s → null`

**Note:**

- Only instance variables get defaults

# Instance Variables vs Local Variables

```
class Demo {
 int x;   // instance variable

 void show() {
  int y;   // local variable
 }
}
```

# Instance Variables vs Local Variables

```
class Demo {
 int x;   // instance variable

 void show() {
  int y;   // local variable
 }
}
```

- Instance variable → belongs to object

# Instance Variables vs Local Variables

```
class Demo {
 int x;   // instance variable

 void show() {
  int y;   // local variable
 }
}
```

- Instance variable → belongs to object
- Local variable → belongs to method

# Instance Variables vs Local Variables

```
class Demo {
 int x;   // instance variable

 void show() {
  int y;   // local variable
 }
}
```

- Instance variable → belongs to object
- Local variable → belongs to method
- Local variables have NO default values

# Multiple Objects from Same Class

```
Student s1 = new Student();
Student s2 = new Student();

s1.id = 10;
s2.id = 20;
```

# Multiple Objects from Same Class

```
Student s1 = new Student();
Student s2 = new Student();

s1.id = 10;
s2.id = 20;
```

**Observation:**

- s1 and s2 are independent objects

# Multiple Objects from Same Class

```
Student s1 = new Student();
Student s2 = new Student();

s1.id = 10;
s2.id = 20;
```

**Observation:**

- s1 and s2 are independent objects
- Same class $\rightarrow$ different data

# Multiple Objects from Same Class

```
Student s1 = new Student();
Student s2 = new Student();

s1.id = 10;
s2.id = 20;
```

**Observation:**

- s1 and s2 are independent objects
- Same class → different data

**Analogy:**

- One blueprint, many houses

# Object Lifecycle — What Does It Mean?

- **Class is loaded** JVM reads class structure (blueprint)

# Object Lifecycle — What Does It Mean?

- **Class is loaded** JVM reads class structure (blueprint)
- **Object is created in heap** Memory allocated for real object

# Object Lifecycle — What Does It Mean?

- **Class is loaded** JVM reads class structure (blueprint)
- **Object is created in heap** Memory allocated for real object
- **Object is used via reference** Reference variable accesses object

# Object Lifecycle — What Does It Mean?

- **Class is loaded** JVM reads class structure (blueprint)
- **Object is created in heap** Memory allocated for real object
- **Object is used via reference** Reference variable accesses object
- **Reference is lost** No variable points to the object

# Object Lifecycle — What Does It Mean?

- **Class is loaded** JVM reads class structure (blueprint)
- **Object is created in heap** Memory allocated for real object
- **Object is used via reference** Reference variable accesses object
- **Reference is lost** No variable points to the object
- **Garbage Collector removes object** JVM automatically frees memory

# Complete Example

**An object is destroyed not when you want, but when no one points to it.**

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

   Student s1 = new Student();  // object created
   s1.id = 10;                  // object used

   s1 = null;                   // reference lost
   // object eligible for garbage collection
 }
}
```

# Methods Work on Object Data

```
class Student {
 int marks;

 void addMarks(int m) {
  marks = marks + m;
 }
}
```

# Methods Work on Object Data

```
class Student {
 int marks;

 void addMarks(int m) {
  marks = marks + m;
 }
}
```

**Key idea:**

- Method changes data of the calling object

# Class Fundamentals Practice

1. Create a class `Student` with variables `id` and `name`. Create an object and print the details.

# Class Fundamentals Practice

1. Create a class `Student` with variables `id` and `name`. Create an object and print the details.
2. Create a class `Rectangle` with `length` and `breadth`. Write a method to calculate and print area.

# Class Fundamentals Practice

1. Create a class `Student` with variables `id` and `name`. Create an object and print the details.
2. Create a class `Rectangle` with `length` and `breadth`. Write a method to calculate and print area.
3. Create a class `Car` with variables `brand` and `speed`. Create two objects and assign different values.

# Class Fundamentals Practice

1. Create a class `BankAccount` with `balance`. Add a method `deposit()` and update balance.

# Class Fundamentals Practice

1. Create a class `BankAccount` with `balance`. Add a method `deposit()` and update balance.
2. Create a class `Employee` with `salary`. Write a method to add bonus only if bonus is positive.

# Class Fundamentals Practice

1. Create a class `BankAccount` with `balance`. Add a method `deposit()` and update balance.
2. Create a class `Employee` with `salary`. Write a method to add bonus only if bonus is positive.
3. Create two objects from the same class and show that changing one object does not affect the other.

# Class Fundamentals Practice

1. Create a class `Counter` with a variable `count`. Increment count using a method and observe object behavior.

# Class Fundamentals Practice

1. Create a class `Counter` with a variable `count`. Increment count using a method and observe object behavior.
2. Create a class `Product` with `price`. Apply discount only if price is greater than 1000.

# Class Fundamentals Practice

1. Create a class `Counter` with a variable `count`. Increment count using a method and observe object behavior.
2. Create a class `Product` with `price`. Apply discount only if price is greater than 1000.
3. Write a program where an object becomes unreachable (reference lost) and explain what happens.

# Class Fundamentals Practice

1. What happens if an object is created but no reference is stored?

# Class Fundamentals Practice

1. What happens if an object is created but no reference is stored?
2. Predict the output:

```
Student s1 = new Student();
Student s2 = s1;
s2.id = 50;
System.out.println(s1.id);
```

# Class Fundamentals Practice

1. What happens if an object is created but no reference is stored?
2. Predict the output:

```
Student s1 = new Student();
Student s2 = s1;
s2.id = 50;
System.out.println(s1.id);
```

3. Can a class exist without an object? Can an object exist without a class? Explain.

# Access vs Non-Access Specifiers

- Specifiers control how class members are used.

# Access vs Non-Access Specifiers

- Specifiers control how class members are used.
- Java provides:
  - **Access Specifiers** – control visibility
  - **Non-Access Specifiers** – control behavior

# Access Specifiers – Concept

- Access specifiers control **visibility** of:
  - Variables
  - Methods
  - Constructors
  - Classes

# Access Specifiers – Concept

- Access specifiers control **visibility** of:
  - Variables
  - Methods
  - Constructors
  - Classes
- They help achieve:
  - Encapsulation
  - Security
  - Controlled access

# Types of Access Specifiers

1. **private**

# Types of Access Specifiers

1. **private**
2. **default** (no keyword)

# Types of Access Specifiers

1. **private**
2. **default** (no keyword)
3. **protected**

# Types of Access Specifiers

1. **private**
2. **default** (no keyword)
3. **protected**
4. **public**

# Access Specifier Scope

| Specifier | Class | Package | Subclass | World |
|---|---|---|---|---|
| private | Yes | No | No | No |
| default | Yes | Yes | No | No |
| protected | Yes | Yes | Yes | No |
| public | Yes | Yes | Yes | Yes |

# private Access Specifier – Concept

- Most restrictive access level

# private Access Specifier – Concept

- Most restrictive access level
- Accessible:
  - Only inside the same class

# private Access Specifier – Concept

- Most restrictive access level
- Accessible:
    - Only inside the same class
- Used to:
    - Hide data
    - Protect sensitive information

# Private Access Specifier – Allowed Access

```
class Student {
 private int id;
 void setId(int i) {
  id = i;
 }
 void showId() {
  System.out.println("Student ID is: " + id);
 }
}
public class Main {
 public static void main(String[] args) {
  Student s1 = new Student();
  s1.setId(101);
  s1.showId();
 }
}
```

# Private Access Specifier – Direct Access Not Allowed

```
class Student {
 private int id;
}
public class Main {
 public static void main(String[] args) {
  Student s1 = new Student();
  s1.id = 101;
  System.out.println(s1.id);
 }
}
```

# default Access Specifier – Concept

- No keyword is used

# default Access Specifier – Concept

- No keyword is used
- Accessible:
  - Within the same class
  - Within the same package

# default Access Specifier – Concept

- No keyword is used
- Accessible:
  - Within the same class
  - Within the same package
- Not accessible outside the package

# Default Access Specifier

```
class Employee {
 int salary;   // default access

 void showSalary() {
  System.out.println(salary);
 }
}
```

# Default Access Specifier

```
class Employee {
 int salary;   // default access

 void showSalary() {
  System.out.println(salary);
 }
}
```

- Accessible within same package
- Not accessible from other packages

# Default Access Specifier – Incorrect Usage

**Concept:** Default access variables belong to an **object**, not to the class itself.

**Java Code:**

```
class Student {
 int id;   // default access

 void showId() {
  System.out.println("ID: " + id);
 }
}

public class Main {
 public static void main(String[] args) {
  showId();   //  invalid call
 }
}
```

# protected Access Specifier – Concept

- Less restrictive than default

# protected Access Specifier – Concept

- Less restrictive than default
- Accessible:
  - Same class
  - Same package
  - Subclasses (even in different package)

## protected Access Specifier – Concept

- Less restrictive than default
- Accessible:
  - Same class
  - Same package
  - Subclasses (even in different package)
- Commonly used in inheritance

# Protected Access Specifier – Full Code

```
class Parent {
 protected int value = 100;
}
class Child extends Parent {
 void show() {
  System.out.println("Value is: " + value);
 } }
  public class Main {
 public static void main(String[] args) {
  Child c = new Child();
  c.show();
 }  }
```

# Protected Access Specifier – Full Code

```
class Parent {
 protected int value = 100;
}
class Child extends Parent {
 void show() {
  System.out.println("Value is: " + value);
 } }
  public class Main {
 public static void main(String[] args) {
  Child c = new Child();
  c.show();
 }  }
```

- Protected members are accessible in subclasses
- Used mainly with inheritance
- Accessed using child class object

# public Access Specifier – Concept

- Least restrictive access level

# public Access Specifier – Concept

- Least restrictive access level
- Accessible:
    - Anywhere in the program
    - Any package

# public Access Specifier – Concept

- Least restrictive access level
- Accessible:
  - Anywhere in the program
  - Any package
- Used for:
  - APIs
  - Entry points (main method)

# public Access Specifier – Code

```
public class Test {
 public int number = 10;

 public void display() {
  System.out.println("Number is: " + number);
 }

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

# public Access Specifier – Code

```
public class Test {
 public int number = 10;

 public void display() {
  System.out.println("Number is: " + number);
 }

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

- Accessible from anywhere

# Non-Access Specifiers

- Do **NOT** control visibility.

# Non-Access Specifiers

- Do **NOT** control visibility.
- They define:
  - Behavior
  - Nature of data or methods

# Non-Access Specifiers

- **static**

# Non-Access Specifiers

- **static**
- **final**

# Non-Access Specifiers

- **static**
- **final**
- **abstract**

# Non-Access Specifiers

- **static**
- **final**
- **abstract**
- **synchronized**

# Non-Access Specifiers

- **static**
- **final**
- **abstract**
- **synchronized**
- **volatile**

# static Keyword

**Concept:** Static members belong to the class.

```
class Test {
 static int x = 10;

 static void show() {
  System.out.println(x);
 }

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

# final Keyword

**Concept:** Final means constant.

```
class Test {
 final int x = 10;

 void show() {
  System.out.println(x);
 }

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

**Concept:** Abstract methods must be implemented by child class.

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

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

public class Main {
 public static void main(String[] args) {
  Shape s = new Circle();
  s.draw();
 } }
```

# synchronized Keyword

**Concept:** Controls thread access.

```
class Table {
 synchronized void print(int n) {
  for(int i=1;i<=3;i++) {
   System.out.println(n*i);
  }
 }
}

public class Main {
 public static void main(String[] args) {
  Table t = new Table();
  t.print(5);
 }
}
```

# volatile Keyword

**Concept:** Guarantees visibility, not atomicity.

```
class Test {
 volatile boolean flag = true;

 void stop() {
  flag = false;
 }
}
```

# volatile Keyword

**Concept:** Guarantees visibility, not atomicity.

```
class Test {
 volatile boolean flag = true;

 void stop() {
  flag = false;
 }
}
```

**Why Needed:**

- Prevents cached value usage
- Threads see updated value immediately

# Practice

1. Create a class with a `private` variable and show how it can be accessed safely.

# Practice

1. Create a class with a `private` variable and show how it can be accessed safely.
2. Write a program where direct access to a `private` variable causes a compile-time error.

# Practice

1. Create a class with a `private` variable and show how it can be accessed safely.
2. Write a program where direct access to a `private` variable causes a compile-time error.
3. Design a class using `default` access members and explain when they can be accessed.

# Practice

1. Create a class with a `private` variable and show how it can be accessed safely.
2. Write a program where direct access to a `private` variable causes a compile-time error.
3. Design a class using `default` access members and explain when they can be accessed.
4. Create a program demonstrating `protected` access using inheritance.

# Practice

1. Create a class with a `private` variable and show how it can be accessed safely.
2. Write a program where direct access to a `private` variable causes a compile-time error.
3. Design a class using `default` access members and explain when they can be accessed.
4. Create a program demonstrating `protected` access using inheritance.
5. Write a Java program to show that `public` members are accessible from anywhere.

# Practice

1. Identify the error when a `private` variable is accessed directly outside the class.

# Practice

1. Identify the error when a `private` variable is accessed directly outside the class.
2. What happens if a `protected` member is accessed without inheritance?

# Practice

1. Identify the error when a `private` variable is accessed directly outside the class.
2. What happens if a `protected` member is accessed without inheritance?
3. Why does a program fail when a default access class is accessed incorrectly?

# Practice

1. Identify the error when a `private` variable is accessed directly outside the class.
2. What happens if a `protected` member is accessed without inheritance?
3. Why does a program fail when a default access class is accessed incorrectly?
4. Analyze why the compiler restricts access to certain class members.

# Practice

1. Write a program to demonstrate the use of the `static` keyword.

# Practice

1. Write a program to demonstrate the use of the `static` keyword.
2. Create a program where modifying a `final` variable causes a compilation error.

# Practice

1. Write a program to demonstrate the use of the `static` keyword.
2. Create a program where modifying a `final` variable causes a compilation error.
3. Design an abstract class and implement it using inheritance.

## Practice

1. Write a program to demonstrate the use of the `static` keyword.
2. Create a program where modifying a `final` variable causes a compilation error.
3. Design an abstract class and implement it using inheritance.
4. Write a multithreaded scenario explaining the need for `synchronized`.

# Practice

1. Write a program to demonstrate the use of the `static` keyword.
2. Create a program where modifying a `final` variable causes a compilation error.
3. Design an abstract class and implement it using inheritance.
4. Write a multithreaded scenario explaining the need for `synchronized`.
5. Explain a situation where `volatile` is required.

# Practice

1. Why can we not create an object of an abstract class?

# Practice

1. Why can we not create an object of an abstract class?
2. What happens if a final method is overridden?

# Practice

1. Why can we not create an object of an abstract class?
2. What happens if a final method is overridden?
3. Explain the issue caused when `synchronized` is not used in shared resources.

# Practice

1. Why can we not create an object of an abstract class?
2. What happens if a final method is overridden?
3. Explain the issue caused when `synchronized` is not used in shared resources.
4. Why does `volatile` not guarantee thread safety?

# Practice

1. Why can we not create an object of an abstract class?
2. What happens if a final method is overridden?
3. Explain the issue caused when `synchronized` is not used in shared resources.
4. Why does `volatile` not guarantee thread safety?
5. Identify the mistake when static members are accessed incorrectly.

# Practice

1. You are designing a banking application. Which access specifier should be used for account balance and why?

# Practice

1. You are designing a banking application. Which access specifier should be used for account balance and why?
2. In a multithreaded ticket booking system, which keyword ensures data consistency?

# Practice

1. You are designing a banking application. Which access specifier should be used for account balance and why?
2. In a multithreaded ticket booking system, which keyword ensures data consistency?
3. When should you prefer `volatile` over `synchronized`?

# Practice

1. You are designing a banking application. Which access specifier should be used for account balance and why?
2. In a multithreaded ticket booking system, which keyword ensures data consistency?
3. When should you prefer `volatile` over `synchronized`?
4. Why are utility methods usually declared static?

# Practice

1. You are designing a banking application. Which access specifier should be used for account balance and why?
2. In a multithreaded ticket booking system, which keyword ensures data consistency?
3. When should you prefer `volatile` over `synchronized`?
4. Why are utility methods usually declared static?
5. Explain why constants are declared using the final keyword.

# Thank You!

## Stay Connected


## Premanand S

| | |
|---|---|
| **Email:** | premanand.s@vit.ac.in |
| **Phone:** | +91-7358679961 |
| **LinkedIn:** | linkedin.com/in/premsanand |
| **Instagram:** | instagram.com/premsanand |
| **WhatsApp Channel:** | anandsDataX |
| **Google Scholar:** | Google Scholar Profile |
| **GitHub:** | github.com/anandprems |