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

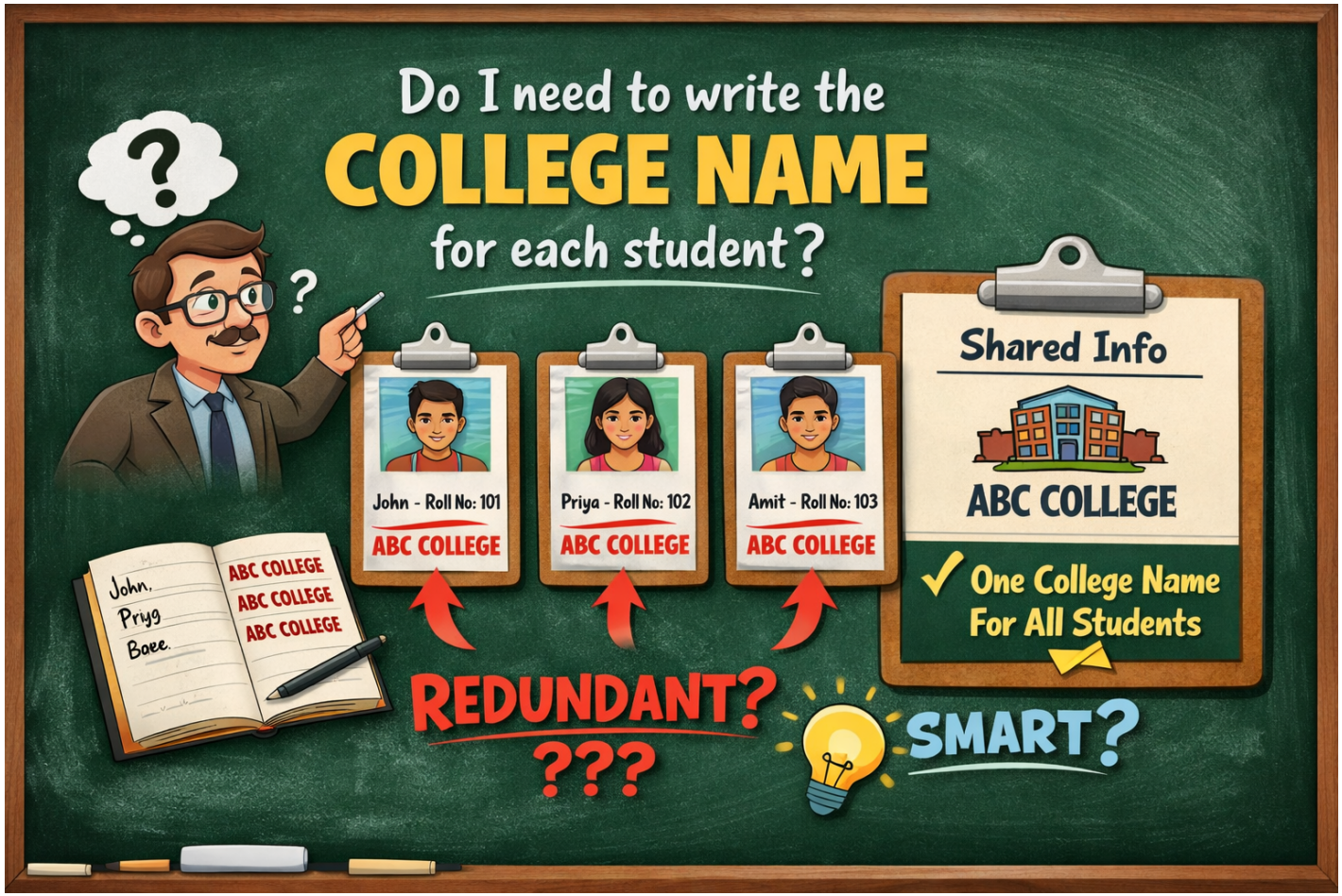
Premanand S

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

premanand.s@vit.ac.in

February 17, 2026

Understanding `static` – A Real-Life View



Key Idea

If something is common to all objects, store it once.

Understanding `static` Using a Real-Life Analogy

Java Concept	**Real Life Analogy**
Class	College
Object	Student
Instance variable	Student's name, roll number
Static variable	College name
Instance method	Student studying
Static method	College holiday notice

What Does `static` Mean?

Definition

`static` members belong to the class rather than to objects.

- Only ONE copy exists
- Shared by all objects
- Created when class is loaded

Why Do We Need `static`?

- To store common data
- To save memory
- To represent class-level behavior

Key Rule

If data is common to all objects, make it `static`.

Where Can We Use `static`?

- Static variables
- Static methods
- Static blocks
- Static nested classes (advanced)

Static Variable – Syntax

```
class Student {
 static String college = "VIT";
 int id;
}
```

- Single shared variable
- Memory allocated once

Static Variable – Example

```
class Student {
 static String college = "VIT";
 int id;

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

- All objects share same college name

Static Variable – Memory Insight

- Instance variables $\rightarrow$ created per object
- Static variables $\rightarrow$ created once per class

Interview Line

Static variables are class-level data.

Static Method – Syntax

```
class MathUtil {
 static int add(int a, int b) {
  return a + b;
 }
}
```

- Can be called without object
- Accessed using class name

Calling Static Method

```
public class Main {
 public static void main(String[] args) {
  int sum = MathUtil.add(10, 20);
  System.out.println(sum);
 }
}
```

Key Point

No object creation required.

Rules of Static Methods

- Cannot access non-static members directly
- Cannot use `this`
- Can access only static data

Static Block

```
class Demo {
 static {
  System.out.println("Static block executed");
 }
}
```

- Executes when class is loaded
- Used for initialization

Execution Order

1. Static block
2. Static variables
3. Constructor
4. Instance methods

Static vs Instance Members

Static	Instance
Class-level	Object-level
Single copy	Multiple copies
Access via class name	Access via object
No `this`	Uses `this`

Brushing Up

Predict Output:

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

public class Main {
 public static void main(String[] args) {
  Test t1 = new Test();
  Test t2 = new Test();
  t1.x = 20;
  System.out.println(t2.x);
 }
}
```

Brushing Up

```
class Counter {
 static int count = 0;

 Counter() {
  count++;
 }
}
```

- Create 3 objects
- Observe value of count

Brushing Up

```
class Demo {
 static {
  System.out.println("Block 1");
 }
 static {
  System.out.println("Block 2");
 }
}
```

Question: When are these blocks executed?

Practice Problems

- Create a class with static variable
- Show that it is shared among objects

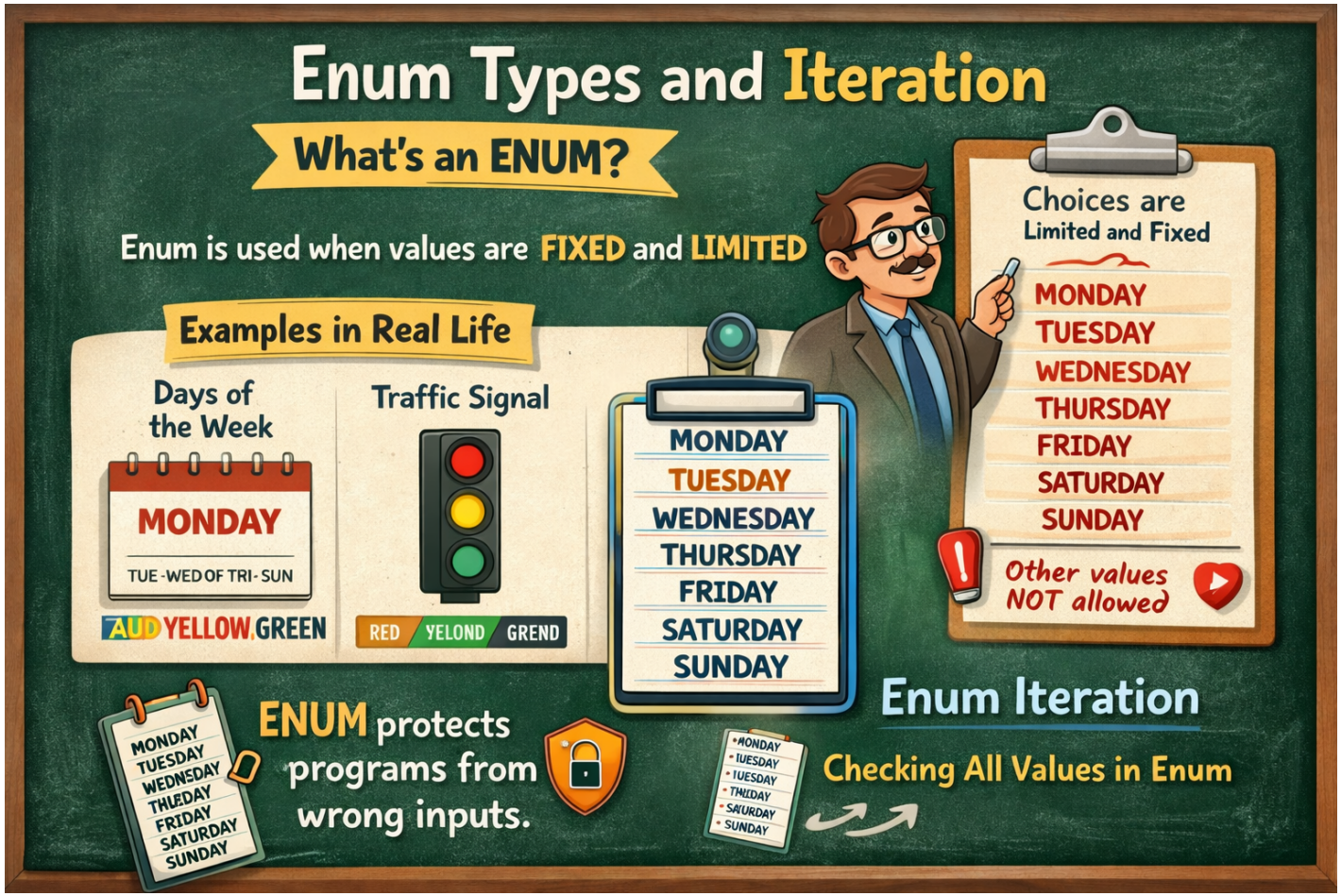
Practice Problems

- Write a program using static counter
- Count number of objects created

Practice Problems

- Why is `main()` static?
- Can static methods be overridden?
- Explain static block execution order

Understanding enum – A Real-Life View



Key Idea

Enum = Fixed choices

What is an Enum?

Definition

An `enum` is a special data type that represents a fixed set of constants.

- Values are predefined
- No new values allowed at runtime
- Improves safety and readability

Why Do We Need Enum?

- To restrict values to a fixed set
- To avoid invalid inputs
- To replace error-prone `int` or `String` constants

Key Idea

Use enum when choices are fixed and limited.

Enum Syntax

```
enum Day {
 MONDAY, TUESDAY, WEDNESDAY,
 THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

- Enum constants are public, static, and final by default
- Enum name starts with capital letter (convention)

Using Enum in a Program

```
class Test {
 Day d;

 Test(Day d) {
  this.d = d;
 }

 void display() {
  System.out.println(d);
 }
}
```

Enum Types (Classification)

- Simple enum (constants only)
- Enum with fields and constructors
- Enum with methods
- Enum with switch statement

Simple Enum Example

```
enum Signal {
 RED, YELLOW, GREEN
}
```

- Most commonly used
- Ideal for status and state values

Enum with Fields and Constructor

```
enum Level {
 LOW(1), MEDIUM(2), HIGH(3);

 int value;

 Level(int value) {
  this.value = value;
 }
}
```

- Enum can have variables
- Constructor is always private

Enum with Method

```
enum Operation {
 ADD, SUBTRACT;

 int apply(int a, int b) {
  if(this == ADD) return a + b;
  else return a - b;
 }
}
```

- Behavior can be associated with enum constants

Enum Iteration – values()

Concept

values() returns an array of all enum constants.

```
Day[] days = Day.values();
```

Enum Iteration Using For-Each Loop

```
for(Day d : Day.values()) {
 System.out.println(d);
}
```

- Most common way to iterate enums
- Clean and readable

Enum Methods – `ordinal()` and `name()`

```
Day d = Day.MONDAY;
System.out.println(d.ordinal());
System.out.println(d.name());
```

- `ordinal()` $\rightarrow$ position (starts from 0)
- `name()` $\rightarrow$ constant name

Enum with Switch Statement

```
switch(d) {
 case MONDAY:
 System.out.println("Start of week");
 break;
 case FRIDAY:
 System.out.println("End of week");
 break;
 default:
 System.out.println("Mid week");
}
```

Brushing Up

Task: Predict Output

```
enum Color { RED, BLUE }

public class Main {
 public static void main(String[] args) {
  Color c = Color.RED;
  System.out.println(c);
 }
}
```

Brushing Up

```
enum Status { SUCCESS, FAILURE }

for(Status s : Status.values()) {
 System.out.println(s.ordinal());
}
```

- Observe index values

Brushing Up

```
enum Grade {
 A, B, C, D, F
}

Grade g = Grade.A;
if(g == Grade.A) {
 System.out.println("Excellent");
}
```

Brushing Up

- Create an enum for days of the week
- Print all values using iteration

Brushing Up

- Create enum with numeric priority values
- Access and print associated data

Brushing Up

- Why is enum safer than constants?
- Can enum have constructors? Why private?
- Difference between enum and final static constants

Understanding Destructors – A Real-Life View

Key Idea

Destructor = Not in Java

What is a Destructor?

General Meaning

A destructor is a special method that is executed when an object is destroyed.

Important

Java does NOT support destructors like C++.

Does Java Have a Destructor?

- Java uses **automatic garbage collection**
- Programmer does NOT control object destruction
- JVM decides when memory is freed

Key Idea

In Java, object destruction is automatic.

Object Life Cycle in Java

1. Class loaded
2. Object created using `new`
3. Object used via reference
4. Reference lost
5. Object becomes eligible for garbage collection

Garbage Collection (GC)

Definition

Garbage Collection is the process by which JVM automatically removes unused objects from memory.

- Runs automatically
- Improves memory safety
- Cannot be forced by programmer

The `finalize()` Method

What is `finalize()`?

A method that may be called by the garbage collector before object removal.

```
protected void finalize() {
 // cleanup code
}
```

- Defined in `Object` class
- NOT a true destructor

Example Using `finalize()`

```
class Demo {
 protected void finalize() {
  System.out.println("Object destroyed");
 }

 public static void main(String[] args) {
  Demo d = new Demo();
  d = null;
  System.gc();
 }
}
```

Important

Output is NOT guaranteed.

Rules of `finalize()`

- May or may not be executed
- Called at most once per object
- Execution time is unpredictable
- Deprecated in modern Java

Why `finalize()` is Deprecated

- Unreliable execution
- Performance overhead
- Can cause memory leaks

Industry Rule

Never rely on `finalize()` for cleanup.

Destructor in C++ vs Java

Feature	C++	Java
Destructor	Yes	No
Manual deletion	Yes	No
Garbage collection	No	Yes
Predictable cleanup	Yes	No

Memory Cleanup vs Resource Cleanup

- Memory cleanup $\rightarrow$ Garbage Collector
- Resource cleanup (files, DB) $\rightarrow$ Programmer

Key Distinction

GC cleans memory, not external resources.

Brushing Up

Question: Is object destroyed here?

```
Demo d = new Demo();
d = null;
```

- Object becomes eligible for GC
- Not destroyed immediately

Brushing Up

```
Demo d1 = new Demo();
Demo d2 = new Demo();
d1 = d2;
```

Question: Which object is eligible for garbage collection?

Brushing UP

```
Demo d = new Demo();
System.gc();
```

Question: Does this guarantee object destruction?

Brushing Up

- Explain why Java does not support destructors
- Describe object life cycle in Java

Brushing Up

- Write a program showing GC eligibility
- Explain why finalize is unreliable

Brushing Up

- Difference between GC and destructor
- Why Java removed finalize?
- How does Java avoid memory leaks?

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