Object Oriented Programming

Content

Programmin Paradigm

A programming paradigm is a style or approach of programming.

Examples:

- ❑ Procedural Programming
- ❑ Object-Oriented Programming

Procedural Programming

❑ Procedural Programming is a programming paradigm that is based on the concept of procedure calls, where a program is divided into procedures or routines (also called functions or subroutines). It focuses on step-by-step instructions to perform tasks.

❑ Think: How would you design a program to manage 100 students?

Object-Oriented Programming

- Object-Oriented Programming(OOP) is an approach of looking at a problem using models which are organized around the real-world concepts.
- The fundamental construct of Object-Oriented Programming (OOP) is object which combines both data structure and behavior as a single entity.

OOPs Concepts:

- Class
- Objects
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Pillar of OOP

Think: How would you design a program to manage 100 students?

OOP vs Procedural Programming

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Top-down (step-by-step instructions)	Bottom-up (objects and classes first)
Focus	Focuses on functions and procedures	Focuses on objects and data
Code Structure	Organized using functions	Organized using classes and objects
Data Handling	Data is exposed and shared	Data is encapsulated (hidden) inside objects
Code Reuse	Limited reuse (use functions)	Easy reuse through inheritance and polymorphism
Security	Less secure – data is accessible to all functions	More secure – data hidden inside objects
Examples	C, Pascal, Basic	Python (with OOP), Java, C++, C#

Why OOP is Needed?

Procedural programming problems:

- ❑ Difficult to manage large code
- ❑ Poor reusability
- ❑ Data is not secure

OOP provides:

- ❑ Better structure
- ❑ Code reusability
- ❑ Security
- ❑ Easy maintenance

Basic OOP Terminology

Class

A blueprint or template for creating objects. Defines the structure and behaviour that objects will have.

Object

An instance of a class. A concrete entity created from the class blueprint with actual values.

Method

A function defined inside a class that describes the behaviour or actions an object can perform.

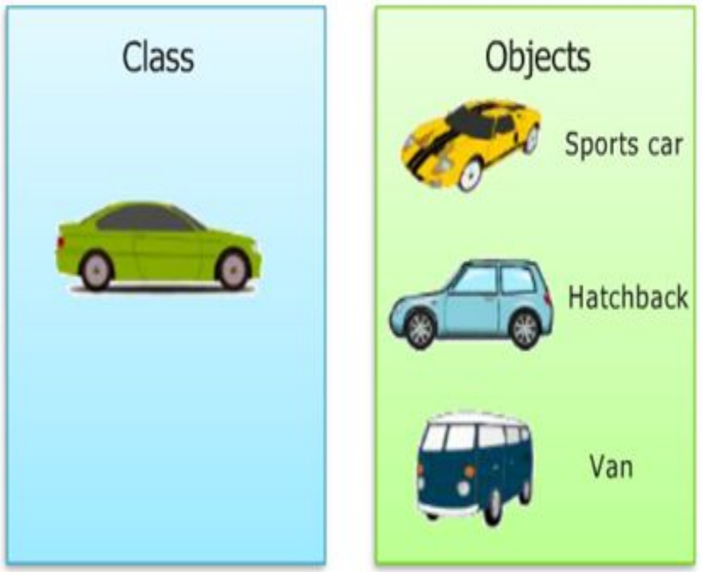
Attribute

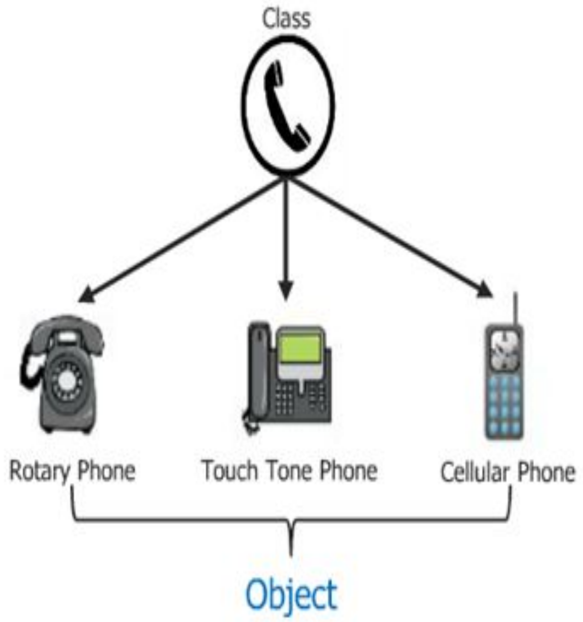
A variable defined inside a class that stores data about an object's characteristics or state.

Classes and Objects

→ Class is a blueprint used to create objects having same property or attribute as its class

→ An Object is an instance of class which contains variables and methods

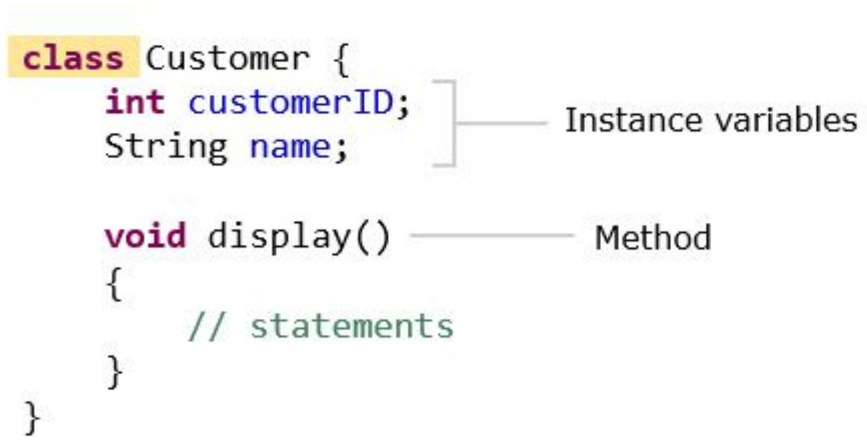




Classes and Objects

- A class is a blueprint for creating objects. It defines the attributes (data) and methods (functions) that describe the characteristics and behavior of real-world entities.
- An object is an instance of a class, created during runtime. The process is called instantiation.
- Attributes (also called data members, properties, or instance variables) are variables defined inside a class.
- Methods (also known as member functions or behaviors) define what actions the object can perform.
- By default, attributes are default and accessed using the dot (.) operator, e.g., object.attribute.

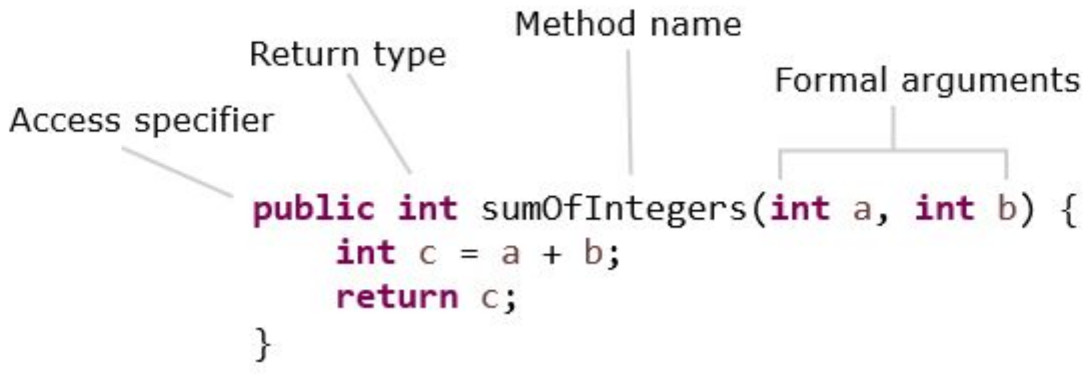
What is a Class?



```
class Customer {
    int customerID;
    String name;

    void display()
    {
        // statements
    }
}
```

Method ?



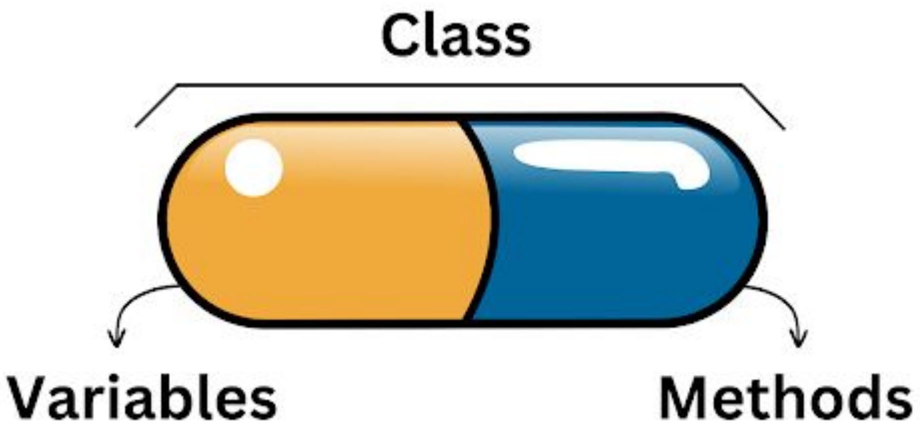
```java
public int sumOfIntegers(int a, int b) {
    int c = a + b;
    return c;
}
```

Example – Class and Object

```
class Student {
    int roll;
    String name;
    void display() {
        System.out.println(roll + " " + name);
    }
}
Student s1 = new Student();
Student s2 = new Student();
```

- ❑ Multiple objects can be created from one class
- ❑ Each object has its own data

Encapsulation



Encapsulation

- ❑ Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP).
- ❑ It refers to the bundling of data (attributes) and methods (functions) into a single unit — a class — and restricting access to some of the object's components.

How Encapsulation Works :

- ❑ **Data Hiding:** private or protected, not accessible directly from outside , acessed or modified through the methods.
- ❑ **Access through Methods:** Methods act as the interface through which external code interacts with the data stored in the variables. For instance, getters and setters are
- ❑ **Control and Security:** By encapsulating the variables and only allowing their manipulation via methods, the class can enforce rules on how the variables are accessed or modified, thus maintaining control and security over the data.

Encapsulation

- ❑ Java achieves encapsulation through:
  - ❑ public attributes.
  - ❑ protected attributes.
  - ❑ private attributes.
  - ❑ Default attributes.

Modifier	Class	Variable	Method	Syntax Example
public	✅	✅	✅	`public class A { }public int x;public void show() { }`
protected	❌	✅	✅	`protected int marks;protected void display() { }`
default *(no keyword)*	✅	✅	✅	`class A { }int count;void print() { }`
private	❌	✅	✅	`private int salary;private void calculate() { }`

- ❑ Getter and setter methods.

Benefits of Encapsulation

Access Modifier	Accessible Within Class	Accessible in Subclass	Accessible Outside Class
public	✅ Yes	✅ Yes	✅ Yes
protected	✅ Yes	✅ Yes	❌ No
default *(no keyword)*	✅ Yes	❌ No	❌ No
private	✅ Yes	❌ No	❌ No

- ❑ Data security
- ❑ Controlled access
- ❑ Easy maintenance
- ❑ Flexible code

Polymorphism

Polymorphism

Polymorphism means "many forms".
In Object-Oriented Programming, it allows the same method name to behave differently based on the object (class) calling it.

Why Use Polymorphism?

- To write flexible and reusable code
- To allow different classes to implement the same interface (i.e., same method names) with different behavior
- Supports dynamic method binding (method resolution happens at runtime)

Type of Polymorphism

Two Types of Polymorphism:

- **Compile-time/Static Polymorphism**
  - Method Overloading (same name, different arguments)

- **Run-time/Dynamic Polymorphism**
  - Method Overriding (child class modifies parent method)

Advantages of Polymorphism

- ❑ Code readability
- ❑ Flexibility
- ❑ Reusability

Abstraction

Abstraction means hiding internal implementation details and showing only the essential features of an object.
It allows you to focus on what an object does, instead of how it does it.

Why Use Abstraction?

1. To reduce complexity
2. To increase code modularity and reusability
3. To define rules or blueprints for subclasses using abstract classes

Real-Life Example:

- ❏ ATM machine:
- ❏ User knows: withdraw, deposit
- ❏ User doesn’t know: internal processing

What is a Constructor?

- ❑ A constructor is a special method used to initialize an object
- ❑ Automatically called when an object is created using new
- ❑ Constructor name must be same as class name
- ❑ Constructors do not have return type

```
Student s = new Student();
```

About Constructors?

- ❑ To initialize instance variables
- ❑ To allocate resources
- ❑ To ensure object is created in a valid state
- ❑ To avoid uninitialized objects
- ❑ Same name as the class
- ❑ No return type (not even void)
- ❑ Called automatically
- ❑ Executes only once per object
- ❑ Can be overloaded

Types of Constructors

- ❑ Default Constructor
- ❑ No-Argument Constructor
- ❑ Parameterized Constructor

1. Default Constructor:

- ❑ Provided by the Java compiler
- ❑ Created only if no constructor is defined
- ❑ Initializes variables with default values

```
class Test {
   int x;
}
Test t = new Test();
```

Types of Constructors

2. No-Argument Constructor:

❑ Defined explicitly by programmer
❑ Takes no parameters
❑ Used for custom initialization

```
class Demo {
    Demo() {
        System.out.println("No-arg constructor called");
    }
}
```

3. Parameterized Constructor:

❑ Accepts parameters
❑ Allows dynamic initialization
❑ Most commonly used constructor

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

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
Student s1 = new Student(101, "Rahul");
Student s2 = new Student(102, "Anita");
```

Constructor Overloading

- ❑ Multiple constructors in the same class
- ❑ Different parameter lists
- ❑ Example of compile-time polymorphism

```
class Demo {
   Demo() { }
   Demo(int a) { }
   Demo(int a, int b) { }
}

Demo d = new Demo();
Demo d = new Demo(10);
Demo d = new Demo(10,20,30);
```

this Keyword

It refers to the current object
Used inside class methods
Used when instance variable and local variable have same name
Helps avoid naming confusion
Improves code clarity

```
class Student {
    int id;

    Student(int id) {
        this.id = id;
    }

    public static void main(String[] args) {
        Student s = new Student(101);
        System.out.println(s.id);
    }
}
```

```
class Test {
    Test() {
        this(10);   // calls parameterized
constructor
        System.out.println("Default
constructor");
    }
    Test(int x) {
        System.out.println("Value: " + x);
    }
    public static void main(String[]
args) {
        new Test();
    }
}
```

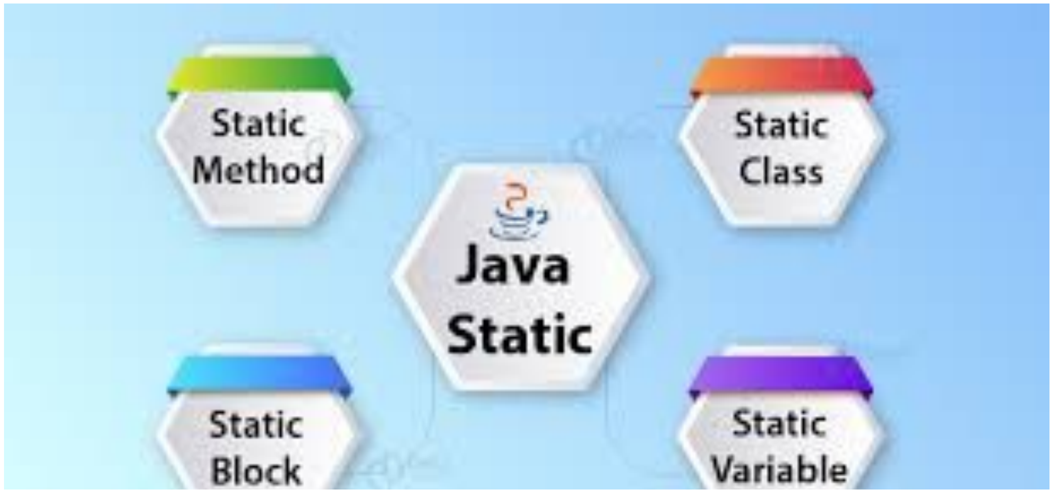
Constructor vs Method

Constructor	Method
Same name as class	Any name
No return type	Has return type
Called automatically	Called explicitly
Initializes object	Performs logic

Important Rules:

- ❑ Constructors cannot be inherited
- ❑ Constructors can be overloaded
- ❑ Constructors cannot be static
- ❑ Constructor cannot return a value

static Keyword



- static is used to create class-level members
- Static members belong to the class, not to objects
- Only one copy exists for entire class
- Static members are shared by all objects
- Can be called using class name.

static Variable

```
class Student {
    int id;
    static String college = "C-DAC";

    Student(int id) {
        this.id = id;
    }

    void display() {
        System.out.println(id + " " + college);
    }
}
Student s1 = new Student(101);
Student s2 = new Student(102);
```

Static Method

- ❑ Can be called using class name
- ❑ Does not require object creation
- ❑ Can access only static members
- ❑ Cannot access non-static variables directly

```
class Demo {
    static void show() {
        System.out.println("Static method");
    }
}
public class Test {
    public static void main(String[] args) {
        Demo.show();
    }
}
```

Static Block

- ❑ Executes once, when class is loaded
- ❑ Used for static initialization

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

Order of Excecution:

1. Static block
2. main() method
3. Constructor
4. Instance methods

Why main() is static?
JVM calls main() without creating object
Static allows direct access

Types of variable ?

Feature	Instance Variable	Static Variable (Class Variable)	Local Variable
Declared inside class	✅ Yes	✅ Yes	❌ No
Declared inside method	❌ No	❌ No	✅ Yes
Uses `static` keyword	❌ No	✅ Yes	❌ No
Belongs to	Object	Class	Method / Block
Number of copies	One per object	Single copy	New copy per method call
Memory location	Heap	Method Area	Stack
Lifetime	As long as object exists	Till class is loaded	Till method execution
Default value	✅ Yes	✅ Yes	❌ No
Used for	Object state	Shared/common data	Temporary data

Relationships in OOP(Association)

- ❑ One object contains another object
- ❑ Also called association

```
class Address { }
class Student {
    Address addr;
}
```

Parameter Passing

- ❑ Parameter passing means sending data (values/objects) to a method when it is called.
- ❑ Java supports ONLY Call by Value
  - ❑ There is no call by reference in Java
  - ❑ Even object references are passed by value

```
class Test {

    static void change(int x) {
        x = 50;
    }

    public static void main(String[] args) {
        int a = 10;
        change(a);
        System.out.println(a);
    }
}
```

- ❑ Copy of a is passed to method
- ❑ Original value does not change

Parameter Passing

```
class Student {
    int marks = 60;
}

class Test {

    static void change(Student s) {
        s.marks = 85;
    }

    public static void main(String[] args) {
        Student obj = new Student();
        change(obj);
        System.out.println(obj.marks);
    }
}
```

Output: 85

- ❑ Reference is passed by value
- ❑ Object data can change

Call by Value vs Call by Reference

Feature	Call by Value	Call by Reference
Java supports	✅ Yes	❌ No
Copy passed	Value copy	Reference
Primitive affected	❌ No	✅ Yes
Object data change	✅ Yes	✅ Yes

Common Confusion:

If object changes, why not call by reference?

Because reference itself is copied, not the object.

Assignments

1. Create a Book class with:
   variables: bookId, title, price
   one no-argument constructor
   one parameterized constructor

WAP for constructor overloading by creating objects using both constructors.

2. Create a class Student with:
   instance variable id
   static variable collegeName
   Create multiple objects and print id and collegeName.
3. Create a class with:
   static block
   static method
   constructor
   and put print statement in every block and Print execution order.

Assignments

4. Create a Calculator class and overload method add():

   two integers
   three integers
   two doubles

THANK YOU!!