Introduction of OOPs

24 October 2025 18:44

Why OOPs:-

- Before oops (Procedural Approach)

→ Programs were written as a sequence of instruction (Step by Step).

→ Data and function were separate function operated on data, but there is no binding b/w them.

Problems with procedural approach.

1. Hard to manage large project:- Code become lengthy and confusing.
2. Poor reusability:- same logic often hard to be written.
3. Less secure:- Any function could modify global data.
4. Tough to scale and maintain:- changing one part often broken other parts.

Procedural Programming focus on functions and procedures, while oops focus on object and their interaction, making modern application more organized reusable and scalable.

Before OOPs, most program followed a procedural approach, where functions and data were written separately. This caused problems like poor code reusability, weak security, and difficult in maintain large projects.

```
public class StudentProcedural {
    // Global data (no encapsulation)
    static String name;
    static int age;
    static int marks;

 // Functions operate on global data
    static void inputData(String n, int a, int m) {
        name = n;
        age = a;
        marks = m;
    }

    static void displayData() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Marks: " + marks);
    }

    public static void main(String[] args) {
        inputData("Vishal", 20, 85);
        displayData();
    }
}
```

Explanation:

- **Data and functions are separate** — there's no real binding between them.
- Anyone can modify the global data (e.g., StudentProcedural.name = "Hacker";).
- Not reusable — if you want another student, you'd need new global variables.
- Hard to scale — as project grows, global variables and functions become unmanageable.

⇒ **To overcome these limitation, object - oriented programming was introduced.**

- Java is built around oops principles, which organize code into classes(blueprint) and object(real-world entities). This make programs:-

→ Programs are structure into class and object.

→ Keeps related data and method together.

→ Make code modular, reusable, and scalable.

→ Prevent unauthorized access of data.

→ Follow the dry (Don't repeat yourself) principle.

OOPs stand for object oriented programming system.

It is a programming approach that organizes code into objects and classes and makes it more structured and easy to manage. A class is a blueprint that defines properties and behaviors, while an object is an instance of a class representing real-world entities.

The main advantage of oops are reusability modularity, flexibility, and easy maintenances.

Procedural programming:- Hard to manage, less reusable.

OOPs:- Classes + objects = Organized and secure.

Java Support OOPs principles such as:-

1. **Encapsulation:-** Keep data and method together and give security from direct access.
2. **Inheritance:-** Allowing one class to inherit properties and behavior form another class.
3. **Polymorphism:-** Ability of an object to take many form.
4. **Abstraction:-** Handling complex implementation details and showing only the necessary fractures.

Classes and Objects in Java

In Java, **classes and objects** are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities.

Class:-

A **class in Java** is a template with the help of which we create real-world entities known as objects that share common characteristics and properties.
In other words, we can say that a class is a blueprint for objects.

Properties of Java Classes

- Class is not a real-world entity. It is just a template or blueprint, or a prototype from which objects are created.
- A class itself does not occupy memory for its attributes and methods until an object is instantiated.
- A Class in Java can contain Data members, methods, a Constructor, Nested classes, and interfaces.

-> Syntax :

```
access-modifiers class ClassName extends
ParentClassName implements InterfaceName
{
    //variables
    //blocks
    //constructors
    //methods
    //nested class, interfaces
}
```

→ **Modifiers**:- A class can be public or has default access.

→ **Class keyword**:- Class keyword is used to create a class.

→ **Class Name:-** The name should begin with an initial letter(capitalized by conventions).

→ **Body:-** The class body is surrounded by braces {}.

→ **Fields:-** Fields are variables defines within a class that hold the data or state of an object.

→ **Constructor:-** It is special method in class, that is automatically called when an object is created.

```
class Student {
    // Fields (data members)
    String name;
    int age;

    // Constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }

    // Method
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

Objects:-

The World is nothing it is surrounded with object and every object belong to some class. Object is real world entity that have has part and dose part.

Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods.

→ e:g Car has wheel, Colour, mirror, name, model…

→ Car does:- move(), stop(), accelerated()…

Objects are real world of class that are created to use the attributes and method of a class.

Attributes/states:- It is represented as data member of class.

Behaviour:- It is reparented as method of class.

Objects correspond to things found in real world.

Note: Objects (non-primitive types) are always allocated on the heap, while their reference variables are stored on the stack.

Object Instantiation (Declaring Objects)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behaviour of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Object instantiation is the process of creating an actual instance of a class so that we can use its data and methods.

ClassName objectRef; // Declaration

Car myCar; // myCar is a reference variable of type car.

Instance:- Highlights that the object is a specific occurrence of a particular class.

```
Class Car
{
 String brand;
String colour;
}
Car myCar = new Car(); // myCar is an instance of car. We can think of it as representing an actual car in the real
```

}
Car myCar = new Car(); // myCar is an instance of car. We can think of it as representing an actual car in the real world.

If we declare a **reference variable** its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

Instantiating Object:-

→ To creating the actual object in memory,we use the new keyword.

→ A new operator instantiates a class by allocating memory for new object and returning a references to that memory.

ObjectRef = new className();

e:g myCar = new Car(); // Instantiation.

Declaring and Instantiating in one step:-

Car myCar = new Car();

- All instance share the same method but hold unique data.
- We can create multiple object from a single class definition.

Declaration :- Car myCar; Creates a references but not an objects.

Instantiation :- new Car(); creates the actual objects.

- Every instances is an object.
- Every object is an instances of some class.

How Objects Are Created in Java

When you create an object using the new keyword, a lot of things happen inside the **Java Virtual Machine (JVM)**.

```
class Animal {
    String name = "Dog";
}

public class AnimalMain {
    public static void main(String[] args) {
        Animal ob = new Animal();
    }
}
```

Step-by-Step Object Creation Process

1. **Compilation Phase**
   - When you compile the program using:

javac AnimalMain.java

- The **Java Compiler** checks for syntax errors.
- If the syntax is correct, it generates .class files (bytecode) for each class.
  → Example: Animal.class and AnimalMain.class

2. **Execution Phase**

When you run:

java AnimalMain

the JVM starts executing the program step-by-step:

2.1 Loading Phase

- The AnimalMain.class file is **loaded** into the **Method Area** of JVM.

- An object of java.lang.Class is created in the **Heap Area**, storing metadata (info) about the AnimalMain class.

2.2 Main Thread Creation

- The **main method** is located and executed.
- JVM creates a new **thread** called the **main thread**.
- A **stack** is created in the **Stack Area** for this main thread.

2.3 Object Creation Statement

When the line

Animal ob = new Animal();

executes:

(a) Class Loading

- The Animal.class file is loaded into the **Method Area** (if not already loaded).
- A corresponding java.lang.Class object is created in the **Heap Area** for the Animal class metadata.

(b) Memory Allocation

- The new keyword tells the JVM to **create a new object** of the Animal class.
- JVM instructs the **Heap Manager** to allocate memory.
- Heap Manager asks for object size → JVM calculates it based on **instance variables** (like String name).
- Heap Manager then allocates that much memory in the **Heap Area**.

(c) Hashcode and Reference

- Each object is assigned a **unique integer value** called **hashcode**.
- JVM converts this hashcode into a **hexadecimal reference value** (memory address-like).
- This **reference value** is stored in the **reference variable** (ob), which is on the **stack**.

(d) Initialization

- The object is **initialized**:
  - All instance variables are given **default values** (e.g., 0, null, false).
  - If constructors assign any values, they overwrite defaults.

Final Result

- An **object** of Animal is successfully created in the heap.
- Its **reference variable (ob)** holds the memory address in hexadecimal form, pointing to that object.

```
Source Code (AnimalMain.java)
        |
   └─> javac → AnimalMain.class + Animal.class
        |
   └─> java AnimalMain
        |
        ├── Load classes into Method Area
        ├── Create java.lang.Class objects in Heap
        ├── Start Main Thread → Main Stack
        ├── new Animal()
        |       ├── Heap allocation (object)
        |       ├── Assign hashcode + reference
```

```
Source Code (AnimalMain.java)
        |
   └─> javac → AnimalMain.class + Animal.class
        |
   └─> java AnimalMain
        |
        ├── Load classes into Method Area
        ├── Create java.lang.Class objects in Heap
        ├── Start Main Thread → Main Stack
        ├── new Animal()
        |       ├── Heap allocation (object)
        |       ├── Assign hashcode + reference
        |       ├── Initialize variables
        |       └── Object ready for use
```

Ways to Create an Object of a Class in Java

In Java, there are **5 main ways** to create an object of a class.

Each method is used in different situations depending on performance, flexibility, or reflection requirements.

1. Using new Keyword (Most Common Way)

This is the **standard and most widely used** method.

How it works:

- JVM allocates memory for the object in the **Heap Area**.
- The **constructor** is called automatically to initialize the object.

2. Using Class.forName() Method (Reflection)

Used when you **don't know the class name at compile time** (dynamic class loading).

How it works:

- Class.forName("ClassName") loads the class into memory.
- newInstance() creates an object of that class using the **default constructor**.

3. Using clone() Method

Creates a **copy (duplicate)** of an existing object.

To use this, the class must **implement Cloneable interface**.

How it works:

- Creates a **new object with the same values** as the original.
- **Constructor is not called** during cloning.

4. Using Object Deserialization

When an object is **stored (serialized)** into a file and later **retrieved (deserialized)**, a **new object** is created in memory.

How it works:

- JVM reads the object data from a file or stream and **creates a new object**.
- **Constructor is not called** during deserialization.

5. Using Constructor.newInstance() (Reflection API)

This method is part of the **Reflection API** and provides more control than Class.newInstance().

How it works:

- You get the Constructor object of the class.

- Then you call newInstance() on it to create an object.
- Supports **parameterized constructors** too.

Anonymous Objects in Java

Anonymous objects are objects that are **instantiated** without storing their reference in a variable. They are used for one-time operations (e.g., method calls) and are discarded immediately after use.

- No reference variable: Cannot reuse the object.
- Created & used instantly: Saves memory for short-lived tasks.
- Common in event handling (e.g., button clicks).

```
class Demo {
    void show() {
        System.out.println("Hello from anonymous object!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Anonymous object: no reference variable
        new Demo().show();   // Object created and method called instantly
    }
}
```

✅ **Output:**

```
csharp

Hello from anonymous object!
```

Explanation

- new Demo() → creates an object of class Demo.
- .show() → immediately calls the show() method.
- After execution, the object is **eligible for garbage collection**, since **no reference variable** points to it.

🧩 Characteristics of Anonymous Objects

Feature	Description
No reference variable	The object is not assigned to any variable.
Used once	Commonly used for a single method call or short operation.
Memory efficient	Avoids creating extra references for temporary use.
Not reusable	You cannot call another method on the same object later.
Garbage collected quickly	Automatically destroyed after use.