

Day_3 OOPJ_Sanket_Shalukar

Thursday, August 28, 2025 10:35 AM

Topics -

- Class and objects
- Variables
- ▢ • Reference
- ProgramsP

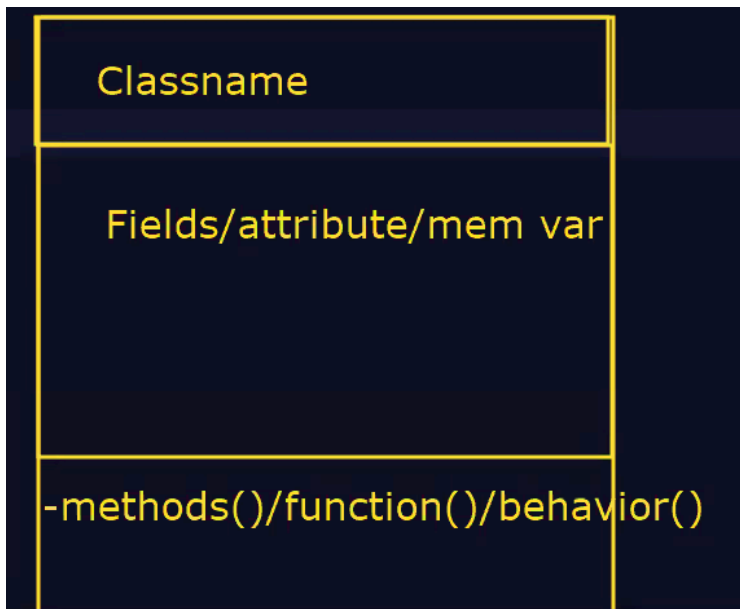
Object oriented Programing!

OOP is a programing paradigm based on the concept of object which contain data (attributes) and methods(behaviour)

Java follows the OOP paradigm, making it easy to organize and manage code effective.

Class:

A Class is a blueprint or template that defines the properties (attributes/fields) and behavior (define by methods)



Object:

An object is an instance of a class.

```
Ex:
class Abc{
    //Fields
    Datatype variable;

    //Methods
    void m1(){

    }

}

Syntax:
Classname objectname = new Classname();
EX:
Abc a1 = new Abc();
```

- Class : Blueprint/template
- Object : Instance of class
- Fields : Variables inside the class
- Methods : Functions inside the class
- Constructors : Special method used to initialize object

Features of OOP

- Object and Classes : Objects are real world entities and classes are their blueprint.
- Encapsulation : Hiding implementation details and expressing only necessary.
- Abstraction : Simplifying complex system by focusing on essential details.
- Inheritance : Enabling code reuse by creating new classes from existing one.
- Polymorphism : Allowing one interface to be used for different implementation.

Object-Oriented Programming Concepts with Real-Life Examples

1. Encapsulation (Data Hiding)

- Definition: Wrapping data and methods together in a single unit (class). Internal data is hidden and accessed only through methods.
- **Real Life Example:** Bank Account. The account balance is private; it can only be modified through deposit() or withdraw() methods, not directly.

2. Abstraction

- Definition: Hiding implementation details and showing only the essential features.
- **Real Life Example:** Car. A driver uses the steering, accelerator, and brakes without knowing the complex internal working of the engine.

3. Inheritance

- Definition: Mechanism where one class acquires the properties and behavior of another class.
- **Real Life Example:** SportsCar inheriting from Car. The SportsCar gets all features of Car and can also have additional features like turbo.

4. Polymorphism

- Definition: The ability of one entity to take many forms. It allows methods to behave differently based on context.
- **Real Life Example:** A person acts as an employee in the office, as a parent at home, and as a friend in society. Same person, different behaviors depending on the situation.

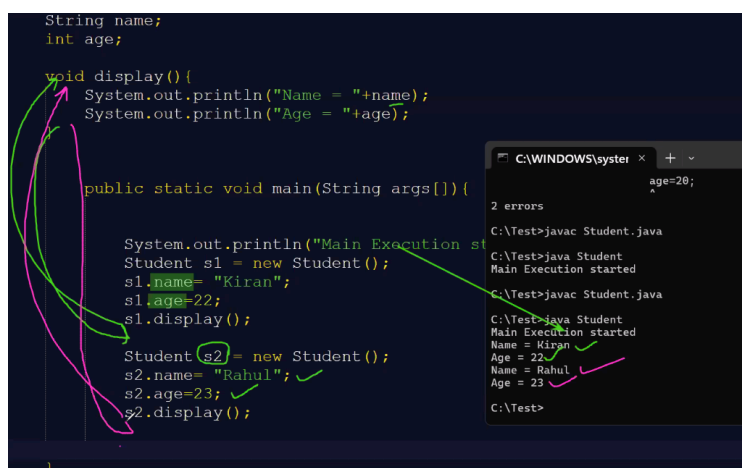
```
class Student {
    String name;
    int age;

    void display() {
        System.out.println("Name = " + name);
        System.out.println("Age = " + age);
    }

    public static void main(String args[]) {
        System.out.println("Main Execution started");

        Student s1 = new Student();
        s1.name = "Sanket";
        s1.age = 20;

        s1.display(); // Call method to print details
    }
}
```



```
String name;
int age;

void display() {
    System.out.println("Name = "+name);
    System.out.println("Age = "+age);
}

public static void main(String args[]){
    System.out.println("Main Execution started");
    Student s1 = new Student();
    s1.name= "Kiran";
    s1.age=22;
    s1.display();
    Student s2= new Student();
    s2.name= "Rahul";
    s2.age=23;
    s2.display();
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
C:\Test>javac Student.java
C:\Test>java Student
Main Execution started
Name = Kiran
Age = 22
Name = Rahul
Age = 23
C:\Test>
```

Overwrite -

```

void display(){
    System.out.println("Name = "+name);
    System.out.println("Age = "+age);
}

public static void main(String args[]){

    System.out.println("Main Execution started");
    Student s1 = new Student();
    s1.name= "Kiran";
    s1.age=22;
    s1.display();

    Student s2 = new Student();
    s1.name= "Rahul";
    s1.age=23;

    s2.display();
    s1.display();
}

```

```

C:\WINDOWS\system x + v
Age = 22
Name = null
Age = 0
C:\Test>javac Student1.java
C:\Test>java Student1
Main Execution started
Name = Kiran
Age = 22
Name = null
Age = 0
Name = Rahul
Age = 23
C:\Test>

```

1. Class and instance fields

- String name; int age; are instance variables.
- Instance variables get default values if not set: name (reference) → null, age (int) → 0.
- Each new Student() object gets its own copy of these fields.

2. display() method

- Prints the current object's state.
- Using name and age inside the method is equivalent to this.name and this.age.
- If name is still null, it will literally print Name = null. If age is unset, it prints Age = 0.

3. main method begins

- System.out.println("Main Execution started"); prints the banner line.

4. Create first object (s1) and set its fields

- Student s1 = new Student(); allocates object #1 on the heap and puts a reference into s1 (on the stack).
- s1.name = "Kiran"; s1.age = 22; updates object #1's state.
- s1.display(); outputs:
Name = Kiran
Age = 22

5. Create second object (s2)

- Student s2 = new Student(); allocates object #2 with default state name=null, age=0.

6. Important detail (the bug in the screenshot)

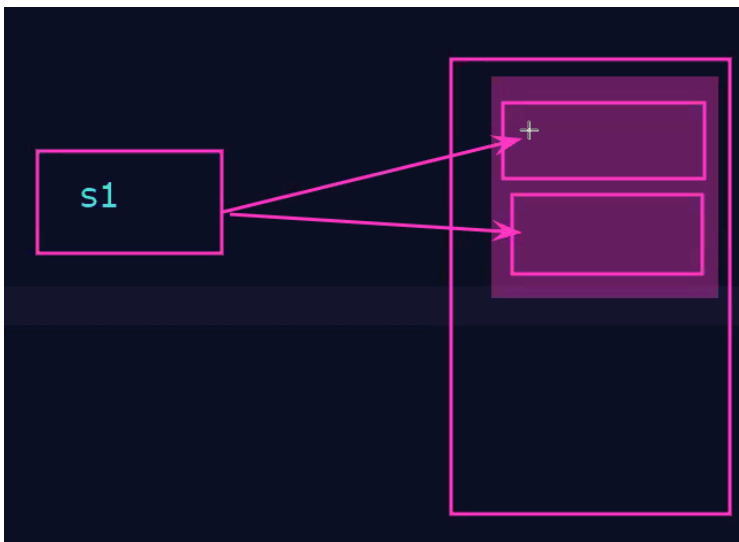
- You then wrote s1.name = "Rahul"; s1.age = 23; right after creating s2.
- That line modifies object #1 again, not object #2.
- So now:
- Object #1 (via s1) → name="Rahul", age=23.
- Object #2 (via s2) → still name=null, age=0.

7. Printing s2 and s1

- s2.display(); prints the defaults for object #2:
Name = null
Age = 0
- s1.display(); prints the updated values for object #1:
Name = Rahul
Age = 23

8. Why the output appears in that order

- First s1.display() shows Kiran/22.
- Then s2.display() shows null/0 because s2 was never assigned values.
- Finally s1.display() shows Rahul/23 after you reassigned s1's fields.



- Variables data will be stored in heap memory
- And references will be stored into stack

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

    System.out.println("Main Execution started");
    Student s1 = new Student();
    s1.name= "Kiran";
    s1.age=22;
    s1.display();

    s1.name= "Rahul";
    s1.age=23;
    s1.display();

    Student s2 = new Student();
    s2.display();

    Student s3 = new Student();
    s3.name= "Rahul";
    s3.age=23;
    s3.display();
}
```

1. Primitive Variable

- Holds the **actual value** of the data.
- Stored directly in **stack memory**.
- Examples: int, float, double, char, boolean, byte, short, long.

2. Reference Variable

- Stores the **address (reference)** of an object, not the actual object itself.
- The object is stored in **heap memory**, the reference variable is stored in stack.
- Examples: String, arrays, user-defined classes, etc.

Key Differences

Feature	Primitive Variable	Reference Variable
Stores	Actual value	Address of object in heap
Memory	Value stored in stack	Reference in stack, object in heap
Examples	int, char, float, boolean, etc.	String, Array, Object, custom classes
Copy Behavior	Copy creates independent value	Copy creates another reference pointing same object
Default Value	0, false, \u0000 (depending on type)	null

```
class MathOperation{
```

```

}


class MathOperationsDemo{
    int x = 10;
    int y = 20;

    public static void main(String args[]){

    }

}

```



```

class MathOperation{
}

class MathOperationsDemo{
    static int x = 10;
    static int y = 20;
    int z = 30;

    public static void main(String args[]){
        System.out.println(x);
        System.out.println(y);

        MathOperationsDemo m1 = new MathOperationsDemo();
        System.out.println(m1.z);

    }

}

```

Output:

```

10
20
30

```

Key Points:

- **Static variables** (x, y) → shared by all objects, can be accessed without creating object.
- **Instance variable** (z) → belongs to object, so you need new keyword.

Static Variable (Class Variable):

A **static variable** in Java is a variable that is declared with the keyword `static`.

Key Points:

- **Belongs to the class, not to objects** → Only one copy exists for the entire class.
- **Shared by all objects** → If one object changes it, all other objects see the updated value.
- **Accessed directly using class name** (no need to create an object).

Example: `ClassName.variableName;`

```

class MathOperation{
    static int a = 30;
    static int b = 40;
}

class MathOperationsDemo{
    static int x = 10;
    static int y = 20;
    int z = 30;

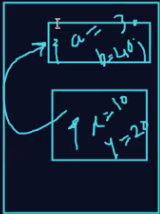
    public static void main(String args[]){
        System.out.println(x);
        System.out.println(y);

        //MathOperationsDemo m1 = new MathOperationsDemo();
        //System.out.println(m1.z);
        System.out.println(MathOperation.a);
        System.out.println(MathOperation.b);

    }

}

```



Static Method :

A **static method** is a method that belongs to the **class** rather than to any specific object.

Key Points:

- Declared using the keyword **static**.
- Can be called **without creating an object** of the class.
→ `ClassName.methodName()`;
- Can **access only static variables** and **other static methods** directly.
- Cannot use `this` or `super` (because they belong to objects, not the class).

Types of variables:

Primitive variables : Stores the basic values

Reference variables (Object)

Local variable

Static variable

```
static void m1(){
    int aal=100;
    System.out.println("Executing : m1()");
}

static void m3(){
    int aa=100; //Local variable
    System.out.println("Executing : m3()");
}
```

```
class TestDemo{
    int a;
    private String name;

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

```
return_type methodname(parameters){
    return 0;
}

Ex: void display(){
}

Ex: int display(){
    ....
    ....
    return sum;
}

Ex: boolean display(){
    ....
    ....
    return true;
}
```

