

MODULE-2

CHAPTER-1

INTRODUCING CLASSES

A class is the fundamental building block of Java that defines the properties (data) and behaviors (methods) of objects. It serves as a blueprint for creating objects and forms the foundation of object-oriented programming (OOP). Every concept in a Java program is implemented using classes.

I. CLASS FUNDAMENTALS

A **class** in Java is a user-defined data type that acts as a blueprint for creating objects. It contains **instance variables (data)** and **methods (functions)**, collectively known as **class members**.

General Syntax:

```
class ClassName {  
    // instance variables  
    // methods  
}
```

Instance Variables

Instance variables are variables declared inside a class. Each object of the class gets its **own separate copy** of these variables.

Methods

Methods are functions defined inside a class that perform operations on the object's data. They determine how the instance variables are accessed and modified.

Simple Class

A simple class contains only data members or methods and serves as a template for creating objects. It does not create an object until it is instantiated using the **new** keyword.

Example:

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

Creating an Object

An **object** is an instance of a class and is created using the **new** keyword.

Example:

```
Box myBox = new Box();
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Accessing Members

The **dot (.) operator** is used to access an object's variables and methods.

Example:

```
myBox.width = 10;
```

```
myBox.height = 5;
```

Program Example

```
class Box {  
    double width, height, depth;  
}  
  
public class BoxDemo {  
    public static void main(String[] args) {  
        Box myBox = new Box();  
        myBox.width = 10;  
        myBox.height = 20;  
        myBox.depth = 15;  
        double volume = myBox.width * myBox.height * myBox.depth;  
        System.out.println("Volume = " + volume);  
    }  
}
```

Output:

Volume = 3000.0

Multiple Objects

Each object has its **own copy** of the instance variables. Changes made to one object do **not** affect the data of another object.

Example:

```
Box box1 = new Box();  
Box box2 = new Box();  
box1.width = 10;  
box2.width = 20;  
System.out.println(box1.width); // 10  
System.out.println(box2.width); // 20
```

II. DECLARING OBJECTS

Creating an object in Java is a two-step process: first, declare a reference variable of the class type, and second, create the object using the new keyword. The new operator allocates memory at runtime and returns a reference to the object.

Example:

```
Box myBox;           // Declaration  
myBox = new Box();    // Object creation
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

The new Operator

The new operator dynamically allocates memory for an object during program execution and returns a reference to it. If no constructor is defined, Java automatically calls the default constructor.

Syntax:

```
ClassName object = new ClassName();
```

Example:

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

Class vs Object

A class is a blueprint or template that defines the properties and behavior of objects. An object is a real instance of a class that occupies memory and stores its own data.

Class	Object
Blueprint or template	Instance of a class
Logical entity	Physical entity
Does not occupy memory	Occupies memory
Used to create objects	Created using new keyword

Example:

```
class Car {  
    String color;  
}
```

```
Car c1 = new Car(); // Object  
c1.color = "Red";
```

III. ASSIGNING OBJECT REFERENCE VARIABLES

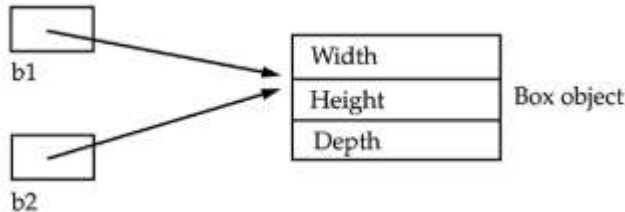
In Java, object reference variables store the reference (memory address) of an object, not the object itself. Assigning one reference variable to another makes both variables refer to the same object, rather than creating a copy.

```
Box b1 = new Box();  
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
```

```
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

IV. INTRODUCING METHODS

Methods in Java

A **method** is a block of code inside a class that performs a specific task. Methods can accept **parameters**, process data, and **return a value** or void.

General Syntax:

```
returnType methodName(parameterList) {
    // method body
    return value;
}
```

Adding a Method to a Class

Methods provide access to and operations on a class's instance variables. They define the behavior of an object and can be called using the **dot (.) operator**.

Example:

```
class Box {
    double width, height, depth;

    void volume() {
        System.out.println("Volume = " + width * height * depth);
    }
}
```

Method Call:

```
Box b = new Box();
b.width = 10;
b.height = 20;
b.depth = 15;

b.volume();
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Output:

Volume = 3000.0

Returning a Value from a Method

A method can return a value to the caller using the **return** statement. The returned value must match the method's return type.

Example:

```
class Box {  
    double width, height, depth;  
    double volume() {  
        return width * height * depth;  
    }  
}  
  
Box b = new Box();  
b.width = 10;  
b.height = 20;  
b.depth = 15;  
double vol = b.volume();  
System.out.println("Volume = " + vol);
```

Output:

Volume = 3000.0

Methods with Parameters

A **parameterized method** accepts input values (parameters), making it reusable for different data.

Example:

```
class Square {  
    int square(int n) {  
        return n * n;  
    }  
}  
  
public class Demo {  
    public static void main(String[] args) {  
        Square s = new Square();  
        System.out.println(s.square(5));  
    }  
}
```

Output:

25

Parameter vs Argument

Parameter	Argument
Variable declared in a method.	Actual value passed to the method.
Receives the value.	Supplies the value.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
int square(int n) // n is a parameter
square(10);      // 10 is an argument
```

Example: Method to Set Box Dimensions

```
class Box {
    double width, height, depth;

    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    double volume() {
        return width * height * depth;
    }
}

public class BoxDemo {
    public static void main(String[] args) {
        Box box = new Box();
        box.setDim(10, 20, 15);
        System.out.println("Volume = " + box.volume());
    }
}
```

Output:

Volume = 3000.0

V. CONSTRUCTORS

A **constructor** is a special member of a class that is automatically called when an object is created. It has the **same name as the class** and **does not have any return type**, not even void.

Syntax:

```
class ClassName {
    ClassName() {
        // Initialization code
    }
}
```

Default Constructor

A **default constructor** is automatically provided by Java if no constructor is defined by the programmer. It initializes instance variables to their default values such as **0**, **null**, or **false**.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
class Box {  
    int width, height, depth;  
}  
public class Demo {  
    public static void main(String[] args) {  
        Box b = new Box();  
        System.out.println(b.width);  
    }  
}
```

Output:

0

User-Defined Constructor

A **user-defined constructor** is explicitly created by the programmer to initialize objects with desired values.

Example:

```
class Box {  
    int width, height, depth;  
    Box() {  
        width = 10;  
        height = 10;  
        depth = 10;  
    }  
    int volume() {  
        return width * height * depth;  
    }  
}  
public class Demo {  
    public static void main(String[] args) {  
        Box b = new Box();  
        System.out.println("Volume = " + b.volume());  
    }  
}
```

Output:

Volume = 1000

Parameterized Constructor

A **parameterized constructor** accepts arguments during object creation, allowing each object to be initialized with different values.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
class Box {
    int width, height, depth;
    Box(int w, int h, int d) {
        width = w;
        height = h;
        depth = d;
    }
    int volume() {
        return width * height * depth;
    }
}

public class Demo {
    public static void main(String[] args) {
        Box b1 = new Box(10, 20, 15);
        Box b2 = new Box(3, 6, 9);
        System.out.println("Volume of Box1 = " + b1.volume());
        System.out.println("Volume of Box2 = " + b2.volume());
    }
}
```

Output:

Volume of Box1 = 3000
Volume of Box2 = 162

VI. THE 'this' KEYWORD

Definition

The **this** keyword is a reference variable in Java that refers to the **current object** (the object that invoked the method or constructor). It can be used inside any method or constructor to access the current object's members.

Key Points:

- Refers to the current object.
- Used inside methods and constructors.
- Helps distinguish instance variables from local variables or parameters.

Syntax

```
this.variableName;
this.methodName();
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example 1: Using this

```
class Box {
    double width, height, depth;

    Box() {
        this.width = 10;
        this.height = 20;
        this.depth = 15;
    }
    double volume() {
        return width * height * depth;
    }
}

class Demo {
    public static void main(String args[]) {
        Box b = new Box();
        System.out.println("Volume = " + b.volume());
    }
}
```

Output

Volume = 3000.0

Explanation:

- this.width refers to the current object's width.
- Here, using this is **correct but optional**, because there is no naming conflict.

Instance Variable Hiding

Definition

Instance Variable Hiding occurs when a **local variable or method parameter** has the **same name as an instance variable**. In such cases, the local variable hides the instance variable.

Example Without this

```
class Box {
    int width;
    Box(int width) {
        width = width;
    }
}
```

Explanation

Here,

- The parameter width hides the instance variable width.
- Statement width = width; assigns the parameter to itself.
- The instance variable remains unchanged.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example Using this

```
class Box {  
    int width;  
    Box(int width) {  
        this.width = width;  
    }  
    void display() {  
        System.out.println("Width = " + width);  
    }  
}
```

```
class Demo {  
    public static void main(String args[]) {  
        Box b = new Box(25);  
        b.display();  
    }  
}
```

Output

Width = 25

Explanation

In the constructor:

this.width = width;

- this.width → Instance variable
- width → Constructor parameter

So,

Instance variable = Parameter value

VII. GARBAGE COLLECTION

Garbage Collection (GC) is an automatic memory management process in Java that **reclaims memory occupied by objects that are no longer in use.**

Unlike languages such as C++, Java does **not require programmers to manually destroy objects.** The Java Virtual Machine (JVM) automatically performs garbage collection.

Why Garbage Collection is Needed

Objects are created dynamically using the **new** operator.

Example:

```
Box b = new Box();
```

When the object is no longer referenced by any variable, it becomes **eligible for garbage collection**, and its memory can be reused.

How Garbage Collection Works

1. Objects are created using the **new** operator.
2. As long as at least one reference points to an object, it remains in memory.
3. When **no references** to an object exist, the object becomes **eligible for garbage collection.**
4. The JVM automatically reclaims the memory occupied by that object.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
class Box {  
}  
class Demo {  
    public static void main(String args[]) {  
        Box b = new Box();  
        b = null; // Object has no reference  
        System.out.println("Object is eligible for garbage collection.");  
    }  
}
```

Initially, b refers to a Box object., After b = null;, no reference points to that object. The object becomes **eligible for garbage collection**.

CHAPTER-2 METHODS AND CLASSES

This chapter explains advanced concepts of **methods and classes**, including **method overloading, parameter passing, and recursion**. It also discusses **access control, the static keyword, and the String class**, which are essential features of Java programming.

I. OVERLOADING METHODS

Method overloading is the feature in Java that allows **two or more methods in the same class to have the same name but different parameter lists** (different number or type of parameters). It is one of the ways Java supports **polymorphism**, allowing the same method name to perform different tasks based on the arguments passed.

Key Points

- Methods must have the **same name**.
- Methods must differ in the **number or type of parameters**.
- **Return type alone cannot distinguish** overloaded methods.
- Java selects the appropriate method based on the **arguments** supplied during the method call.
- If no exact match is found, Java may perform **automatic type conversion** to resolve the call.

Example

```
class Demo {  
    void test() {  
        System.out.println("No parameters");  
    }  
    void test(int a) {  
        System.out.println("One integer: " + a);  
    }  
    void test(int a, int b) {  
        System.out.println("Two integers: " + a + ", " + b);  
    }  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
void test(double a) {  
    System.out.println("One double: " + a);  
}  
public static void main(String args[]) {  
    Demo d = new Demo();  
    d.test();  
    d.test(10);  
    d.test(10, 20);  
    d.test(12.5);  
}  
}
```

Advantages

- Supports **compile-time polymorphism**.
- Improves **code readability**.
- Reduces the need for multiple method names.
- Allows related operations to be accessed using a common method name.

Overloading Constructors

Constructor overloading is the process of defining **multiple constructors with different parameter lists** in the same class. It allows objects to be initialized in different ways depending on the arguments passed during object creation.

- Constructors have the **same name as the class**.
- Constructors must have **different parameter lists**.
- Java automatically calls the appropriate constructor based on the arguments provided.
- Constructor overloading provides flexibility in object initialization.

Example

```
class Box {  
    double width, height, depth;  
  
    Box() {  
        width = height = depth = 0;  
    }  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    Box(double len) {  
        width = height = depth = len;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
    public static void main(String args[]) {  
        Box b1 = new Box();  
    }
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
Box b2 = new Box(10, 20, 15);
Box b3 = new Box(5);
System.out.println("Volume of b1 = " + b1.volume());
System.out.println("Volume of b2 = " + b2.volume());
System.out.println("Volume of b3 = " + b3.volume());
}
}
```

II. USING OBJECTS AS PARAMETERS

In Java, **objects can be passed as arguments to methods**, just like primitive data types. The method receives a **reference to the object**, allowing it to access the object's instance variables and methods.

Objects can be passed as method parameters. The parameter type is the **class name**. Methods can compare, modify, or use the data of the passed object. Object parameters are commonly used in **constructors**.

Example: Passing an Object to a Method

```
class Test {
    int a;

    Test(int i) {
        a = i;
    }
    boolean equalTo(Test o) {
        return a == o.a;
    }
}
class Demo {
    public static void main(String args[]) {
        Test ob1 = new Test(100);
        Test ob2 = new Test(100);
        System.out.println(ob1.equalTo(ob2));
    }
}
```

Output

true

equalTo() receives a Test object as a parameter. It compares the instance variable a of both objects. If the values are equal, it returns true; otherwise, it returns false.

Constructors with Object Parameters

A **constructor can accept an object of the same class as a parameter**. This is called an **object parameter constructor** or **copy constructor (user-defined)**. It is used to initialize one object using the values of another object.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
class Box {
    double width, height, depth;
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // Constructor with object parameter
    Box(Box ob) {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    double volume() {
        return width * height * depth;
    }
}
```

```
public static void main(String args[]) {
    Box b1 = new Box(10, 20, 15);
    Box b2 = new Box(b1);
    System.out.println("Volume of b1 = " + b1.volume());
    System.out.println("Volume of b2 = " + b2.volume());
}
```

b1 is created using the parameterized constructor. b2 is created using the constructor that accepts another Box object. The values of width, height, and depth are copied from b1 to b2. Both objects have the same dimensions and volume.

III. ARGUMENT PASSING

Argument passing is the process of sending values or objects to a method. Java always uses **call-by-value**, but its behavior differs for primitive data types and object references.

For **primitive data types**, a copy of the actual value is passed to the method. Any changes made to the parameter inside the method do not affect the original variable.

For **objects**, a copy of the reference is passed to the method instead of the object itself. Since both references point to the same object, changes made to the object's data inside the method are reflected outside the method.

Although object references behave like **call-by-reference**, Java still uses **call-by-value** because only the reference value is copied, not the actual object.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

For example, consider the following program:

// Objects are passed through their references.

```
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // pass an object
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRef {
    public static void main(String[] args) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " +
            ob.a + " " + ob.b);

        ob.meth(ob);

        System.out.println("ob.a and ob.b after call: " +
            ob.a + " " + ob.b);
    }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside **meth()** have affected the object used as an argument.

IV. RETURNING OBJECTS

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen()** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
// Returning an object.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}

class RetOb {
    public static void main(String[] args) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
                           + ob2.a);
    }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

As you can see, each time **incrByTen()** is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out of scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

V. RECURSION

Recursion is a programming technique in which a method calls itself to solve a problem. A method that calls itself is known as a **recursive method**.

A classic example of recursion is calculating the **factorial** of a number. The method repeatedly calls itself with a smaller value until it reaches a **base case**, where it stops.

Each recursive call creates a new copy of its local variables and parameters on the **stack**. As each call finishes, control returns to the previous call until the final result is obtained.

Recursive methods are often simpler and easier to understand than iterative solutions for problems such as **factorial**, **QuickSort**, **tree traversal**, and **AI algorithms**. However, excessive recursion may cause a **stack overflow** due to repeated method calls.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Every recursive method must include a **base condition** (such as an if statement) to stop further recursive calls. Without a base case, the method will continue calling itself indefinitely, leading to a runtime error.

Example of recursion.

The recursive method **printArray()** prints the first **i** elements in the array **values**.

// Another example that uses recursion.

```
class RecTest {
    int[] values;
    RecTest(int i) {
        values = new int[i];
    }

    // display array -- recursively
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println "[" + (i-1) + " ] " + values[i-1];
    }
}

class Recursion2 {
    public static void main(String[] args) {
        RecTest ob = new RecTest(10);
        int i;

        for(i=0; i<10; i++) ob.values[i] = i;

        ob.printArray(10);
    }
}
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

VI. INTRODUCING ACCESS CONTROL

Access control is a feature of encapsulation that restricts access to the data and methods of a class. It protects data from unauthorized access and ensures controlled access through methods. Java provides four access modifiers: **public**, **private**, **protected**, and **default**. They determine where a class member can be accessed.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

public: Can be accessed from anywhere in the program.

```
class Test {  
    public int a = 10;  
}
```

```
Test obj = new Test();  
System.out.println(obj.a); // Accessible
```

private: Can be accessed only within the same class.

```
class Test {  
    private int a = 10;
```

```
    public int getA() {  
        return a;  
    }  
}
```

```
Test obj = new Test();  
// System.out.println(obj.a); // Error  
System.out.println(obj.getA()); // Correct
```

default (no modifier): Can be accessed only within the same package.

```
class Test {  
    int a = 20; // Default access  
}
```

protected: Can be accessed within the same package and by subclasses.

```
class Parent {  
    protected int x = 100;  
}
```

```
class Child extends Parent {  
    void display() {  
        System.out.println(x); // Accessible  
    }  
}
```

Using **private** variables with **public getter and setter methods** is the best practice for implementing encapsulation. It prevents direct modification of data and improves program security.

Example: Encapsulation using Access Control

```
class Student {  
    private String name;  
  
    public void setName(String n) {  
        name = n;  
    }  
    public String getName() {  
        return name;  
    }  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
public class Demo {  
    public static void main(String[] args) {  
        Student s = new Student();  
        s.setName("Ravi");  
        System.out.println(s.getName());  
    }  
}
```

Output:

Ravi

VII. UNDERSTANDING STATIC

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.

You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods of their class.
- They can only directly access **static** variables of their class.
- They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args) {
        meth(42);
    }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a*4** or **12**. Then **main()** is called, which calls **meth()**, passing **42** to **x**. The three **println()** statements refer to the two **static** variables **a** and **b**, as well as to the parameter **x**.

Here is the output of the program:

Static block initialized.

x = 42

a = 3

b = 12

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method()

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main()**, the **static** method **callme()** and the **static** variable **b** are accessed through their class name **StaticDemo**.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
class StaticDemo {
    static int a = 42;
    static int b = 99;

    static void callme() {
        System.out.println("a = " + a);
    }
}

class StaticByName {
    public static void main(String[] args) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Here is the output of this program:

```
a = 42
b = 99
```

VIII. INTRODUCING FINAL

The **final** keyword is used to declare constants in Java. Once a **final variable** is assigned a value, it **cannot be changed** during program execution.

A **final field** can be initialized either **at the time of declaration** or **inside a constructor**. By convention, constant names are written in **UPPERCASE**.

Example 1: Final Variable

```
class Demo {
    final int MAX = 100;

    void display() {
        System.out.println(MAX);
        // MAX = 200; // Error: Cannot assign a value to final variable
    }
}
```

Output:

```
100
```

A **final method parameter** cannot be modified inside the method, and a **final local variable** can be assigned only once.

Example 2: Final Parameter

```
class Demo {
    void show(final int x) {
        System.out.println(x);
        // x = 20; // Error
    }
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

2. Arrays Revisited

In Java, **arrays are objects**, and every array has a built-in **length** attribute. The **length** variable stores the total number of elements the array can hold.

The **length** attribute is useful for traversing arrays and avoiding **ArrayIndexOutOfBoundsException**.

Example: Using length

```
class ArrayDemo {  
    public static void main(String args[]) {  
        int nums[] = {10, 20, 30, 40, 50};  
  
        System.out.println("Array size = " + nums.length);  
  
        for(int i = 0; i < nums.length; i++) {  
            System.out.println(nums[i]);  
        }  
    }  
}
```

Output:

```
Array size = 5  
10  
20  
30  
40  
50
```

IX. INTRODUCING NESTED AND INNER CLASSES.

A **nested class** is a class defined inside another class. It exists only within the scope of its enclosing (outer) class and can access the members, including **private members**, of the outer class.

Java supports **two types of nested classes**:

1. **Static Nested Class**
2. **Inner Class (Non-static Nested Class)**

1. Static Nested Class

A **static nested class** is declared using the static keyword. It can access only the **static members** of the outer class directly and must use an object reference to access non-static members.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
class Outer {
    static int x = 100;
    static class Inner {
        void display() {
            System.out.println("x = " + x);
        }
    }
}

public class Demo {
    public static void main(String args[]) {
        Outer.Inner obj = new Outer.Inner();
        obj.display();
    }
}
```

Output

x = 100

2. Inner Class (Non-static Nested Class)

An **inner class** is a non-static nested class. It can directly access **all members** (including private members) of the outer class.

Example

```
class Outer {
    int outer_x = 100;
    class Inner {
        void display() {
            System.out.println("outer_x = " + outer_x);
        }
    }
    void test() {
        Inner in = new Inner();
        in.display();
    }
}

public class Demo {
    public static void main(String args[]) {
        Outer obj = new Outer();
        obj.test();
    }
}
```

Output

outer_x = 100

Outer Class Cannot Access Inner Class Members

The **inner class** can access outer class members, but the **outer class cannot directly access** the members of the inner class.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example

```
class Outer {  
    class Inner {  
        int y = 20;  
    }  
    void show() {  
        // System.out.println(y); // Error  
    }  
}
```

Local Inner Class

A nested class can also be declared inside a **method or block**. Such a class is called a **local inner class**.

Example

```
class Test {  
    void display() {  
        class Local {  
            void show() {  
                System.out.println("Inside Local Inner Class");  
            }  
        }  
        Local obj = new Local();  
        obj.show();  
    }  
    public static void main(String args[]) {  
        Test t = new Test();  
        t.display();  
    }  
}
```

Output

Inside Local Inner Class