Module 2: Objects and Classes - Java OOP Fundamentals

This document provides a comprehensive overview of Objects and Classes in Java, based on the NDU Curriculum DCSA250059. It covers fundamental concepts, methodologies, and best practices for object-oriented programming in Java.

---

1. Class Fundamentals

What is a Class?

A **class** serves as a blueprint or template that defines the structure and behavior of objects. It encapsulates data (fields) and methods that operate on that data.

What is an Object?

An **object** is a concrete instance of a class. It occupies memory and holds actual values, representing a real-world entity defined by its class blueprint.

Real-world Analogy:

```
Class: Car (Blueprint)
```

```
├── Attributes: make, model, color, speed
└── Behaviors: start(), accelerate(), brake(), stop()

Objects (Actual Cars):
├── Car 1: Toyota, Camry, Red, 0 km/h
├── Car 2: Honda, Civic, Blue, 0 km/h
```

```
└── Car 3: BMW, X5, Black, 0 km/h
```

Basic Class Structure:

```
// Class declaration
```

```
public class Student {
    // Instance variables (Fields)
    int rollNo;
```

```
    String name;
    double marks;

    // Methods
    void display() {
        System.out.println("Roll No: " + rollNo);
        System.out.println("Name: " + name);
        System.out.println("Marks: " + marks);
    }

    void study() {
        System.out.println(name + " is studying");
    }
}

// Using the class
public class Main {
    public static void main(String[] args) {
        // Creating objects
        Student s1 = new Student();
        Student s2 = new Student();

        // Accessing fields
        s1.rollNo = 101;
        s1.name = "Alice";
        s1.marks = 85.5;

        s2.rollNo = 102;
        s2.name = "Bob";
        s2.marks = 78.0;

        // Calling methods
        s1.display();
        s1.study();

        s2.display();
        s2.study();
    }
```

```
}
```

Output:

```
Roll No: 101
```

```
Name: Alice
Marks: 85.5
Alice is studying
Roll No: 102
Name: Bob
Marks: 78.0
```

```
Bob is studying
```

Declaring Objects

Memory Allocation:

```
// Step 1: Declaration (creates reference variable)
```

```
Student s;       // s is null, no object created

// Step 2: Instantiation (creates object in heap)
s = new Student(); // new allocates memory, calls constructor

// Combined declaration and instantiation
```

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

Memory Diagram:

```
Stack Memory                    Heap Memory
```

```
┌──────────────┐                ┌──────────────────────┐
│     s        │──────────────→│ Student Object       │
│  (reference)│                │ rollNo: 0            │
└──────────────┘                │ name: null           │
                                │ marks: 0.0           │
```

```
                                └──────────────────────┘
```

Multiple Objects Example:

```
class Rectangle {
```

```
    int length;
    int breadth;

    void area() {
        System.out.println("Area: " + (length * breadth));
    }
}

public class MultipleObjectsDemo {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        Rectangle r2 = new Rectangle();

        r1.length = 10;
        r1.breadth = 5;

        r2.length = 7;
        r2.breadth = 3;

        System.out.print("Rectangle 1 - ");
        r1.area();

        System.out.print("Rectangle 2 - ");
        r2.area();

        // Reference assignment - both point to same object
        Rectangle r3 = r1;
        r3.length = 20;  // Changes r1's length too!

        System.out.print("Rectangle 1 after change - ");
        r1.area();  // Area becomes 20 * 5 = 100
    }
```

```
}
```

---

2. Methods in Java

Method Declaration:

```
[access_modifier] [static] return_type method_name(parameters) {
    // method body
    [return value;]
}
```

Types of Methods:

```
class Calculator {
    // 1. Method with no parameters, no return value
    void displayWelcome() {
        System.out.println("Welcome to Calculator!");
    }

    // 2. Method with parameters, no return value
    void printSum(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }

    // 3. Method with parameters and return value
    int add(int a, int b) {
        return a + b;
    }

    // 4. Method with return value, no parameters
    int getRandomNumber() {
        return (int)(Math.random() * 100);
    }

    // 5. Method returning object
    Calculator getInstance() {
        return this;
    }
}

public class MethodDemo {
```

```
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        calc.displayWelcome();
        calc.printSum(10, 20);

        int result = calc.add(5, 7);
        System.out.println("Addition result: " + result);

        int random = calc.getRandomNumber();
        System.out.println("Random number: " + random);
    }
```

```
}
```

---

3. Constructors

A **constructor** is a special method used to initialize an object when it is created.

Characteristics:

- Same name as the class
- No return type (not even void)
- Automatically called when an object is created using new
- Can be overloaded

Types of Constructors:

```
class Employee {
```

```
    int empId;
    String name;
    double salary;

    // 1. Default Constructor (no parameters)
    Employee() {
        empId = 0;
        name = "Unknown";
```

```
        salary = 0.0;
        System.out.println("Default constructor called");
    }

    // 2. Parameterized Constructor
    Employee(int id, String n) {
        empId = id;
        name = n;
        salary = 25000;
        System.out.println("Parameterized constructor (2 params) called");
    }

    // 3. Parameterized Constructor with all fields
    Employee(int id, String n, double sal) {
        empId = id;
        name = n;
        salary = sal;
        System.out.println("Parameterized constructor (3 params) called");
    }

    void display() {
        System.out.println("ID: " + empId + ", Name: " + name + ", Salary: " + salary);
    }
}

public class ConstructorDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee();                // Default constructor
        Employee e2 = new Employee(101, "Alice");       // 2-param constructor
        Employee e3 = new Employee(102, "Bob", 50000);   // 3-param constructor

        e1.display();
        e2.display();
        e3.display();
    }
```

```
}
```

Output:

```
Default constructor called
```

```
Parameterized constructor (2 params) called
```

```
Parameterized constructor (3 params) called
ID: 0, Name: Unknown, Salary: 0.0
ID: 101, Name: Alice, Salary: 25000.0
```

```
ID: 102, Name: Bob, Salary: 50000.0
```

Constructor vs Method:

Constructor	Method
Same name as class	Any name (except class name)
No return type	Has return type (void or other)
Called automatically with new	Called explicitly
Cannot be called directly	Can be called any number of times
Used to initialize object	Used to perform operations

Constructor Overloading

Constructor overloading involves defining multiple constructors within the same class, each with a different parameter list.

```
class Book {
```

```
    String title;
    String author;
    double price;
    int pages;

    // Constructor 1: No parameters
    Book() {
        title = "Unknown";
        author = "Unknown";
        price = 0.0;
        pages = 0;
        System.out.println("Default book created");
    }

    // Constructor 2: Two parameters
```

```
    Book(String t, String a) {
        this.title = t;
        this.author = a;
        this.price = 299.0;
        this.pages = 200;
        System.out.println("Book with title and author created");
    }

    // Constructor 3: Three parameters
    Book(String t, String a, double p) {
        this.title = t;
        this.author = a;
        this.price = p;
        this.pages = 250;
        System.out.println("Book with title, author, price created");
    }

    // Constructor 4: All parameters
    Book(String t, String a, double p, int pg) {
        this.title = t;
        this.author = a;
        this.price = p;
        this.pages = pg;
        System.out.println("Complete book created");
    }

    void display() {
        System.out.println("Title: " + title + ", Author: " + author +
                           ", Price: Rs." + price + ", Pages: " + pages);
    }
}

public class ConstructorOverloadingDemo {
    public static void main(String[] args) {
        Book b1 = new Book();
        Book b2 = new Book("Java Programming", "Herbert Schildt");
        Book b3 = new Book("Python Basics", "John Doe", 450.0);
        Book b4 = new Book("Data Structures", "Alice Brown", 599.0, 500);

        System.out.println("\n--- Book Details ---");
        b1.display();
        b2.display();
        b3.display();
        b4.display();
```

```
    }
}
```

---

4. The this Keyword

The this keyword in Java refers to the current object instance.

Uses of this:

1. To refer to instance variables (when shadowed by parameters)

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

    // Without 'this' - parameter names shadow instance variables
    // Student(int rollNo, String name) {
    //     rollNo = rollNo;  // WRONG - assigns parameter to itself
    //     name = name;      // WRONG - assigns parameter to itself
    // }

    // With 'this' - correct way
    Student(int rollNo, String name) {
        this.rollNo = rollNo;  // this.rollNo refers to instance variable
        this.name = name;      // name refers to parameter
    }
}
```

2. To call another constructor (constructor chaining)

```
class Time {
    int hour;
    int minute;
```

```
    int second;

    // Constructor with all fields
    Time(int hour, int minute, int second) {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // Constructor with hour and minute (second = 0)
    Time(int hour, int minute) {
        this(hour, minute, 0);  // Calls the 3-parameter constructor
    }

    // Constructor with only hour (minute = 0, second = 0)
    Time(int hour) {
        this(hour, 0, 0);  // Calls the 3-parameter constructor
    }

    void display() {
        System.out.printf("%02d:%02d:%02d\n", hour, minute, second);
    }
}

public class ThisConstructorDemo {
    public static void main(String[] args) {
        Time t1 = new Time(10, 30, 45);
        Time t2 = new Time(14, 20);
        Time t3 = new Time(9);

        t1.display();  // 10:30:45
        t2.display();  // 14:20:00
        t3.display();  // 09:00:00
    }
```

```
}
```

3. To return current object

```
class Counter {
```

```
    int count = 0;
```

```
    Counter increment() {
        count++;
        return this;  // Return current object for chaining
    }

    Counter decrement() {
        count--;
        return this;
    }

    void display() {
        System.out.println("Count: " + count);
    }
}

public class ThisReturnDemo {
    public static void main(String[] args) {
        Counter c = new Counter();

        // Method chaining
        c.increment().increment().decrement().increment();
        c.display();  // Count: 2
    }
```

```
}
```

4. To pass current object as parameter

```
class Processor {
```

```
    void process(Data d) {
        System.out.println("Processing data: " + d.value);
    }
}

class Data {
    int value;
    Processor p;

    Data(int value) {
        this.value = value;
        p = new Processor();
```

```
    }

    void submit() {
        p.process(this);  // Passing current object
    }
}

public class ThisParameterDemo {
    public static void main(String[] args) {
        Data d = new Data(100);
        d.submit();  // Processing data: 100
    }
}
```

---

5. Garbage Collection

Java automatically manages memory through **garbage collection** (GC). The JVM automatically removes objects that no longer have any references pointing to them. The garbage collector runs in the background, and while it cannot be forced, it can be requested using System.gc().

Ways to make an object eligible for GC:

1. Assign null to its reference variable.
2. Reassign the reference variable to a new object.
3. An object created inside a method becomes eligible when the method scope ends.
4. **Island of Isolation**: A group of objects that reference each other but are not referenced by any active object outside the group.

```
class MyObject {

    int id;

    MyObject(int id) {
        this.id = id;
        System.out.println("Object " + id + " created");
    }
```

```
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Object " + id + " is being garbage collected");
    }
}

public class GCDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            MyObject obj = new MyObject(i);
            obj = null;  // Object becomes eligible for GC
        }

        System.gc(); // Request garbage collection (not guaranteed)

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main method ends");
    }
}
```

---

6. The finalize() Method

The finalize() method is called by the garbage collector just before an object is destroyed. It can be overridden to perform cleanup operations for resources held by the object.

```
class ResourceHandler {
    String resourceName;
    boolean isOpen;

    ResourceHandler(String name) {
        this.resourceName = name;
        this.isOpen = true;
```

```
        System.out.println("Resource '" + name + "' opened");
    }

    void close() {
        if (isOpen) {
            isOpen = false;
            System.out.println("Resource '" + resourceName + "' closed");
        }
    }

    @Override
    protected void finalize() throws Throwable {
        try {
            if (isOpen) {
                System.out.println("WARNING: Resource '" + resourceName +
                                   "' was not closed! Closing in finalize()");
                close();
            }
        } finally {
            super.finalize();
        }
    }
}

public class FinalizeDemo {
    public static void main(String[] args) {
        ResourceHandler r1 = new ResourceHandler("database_conn");
        ResourceHandler r2 = new ResourceHandler("file_handle");

        r1.close();  // Properly closed

        r2 = null;   // Forgot to close r2

        System.gc();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Program ends");
    }
```

```
}
```

Output:

```
Resource 'database_conn' opened
```

```
Resource 'file_handle' opened
Resource 'database_conn' closed
WARNING: Resource 'file_handle' was not closed! Closing in finalize()
Resource 'file_handle' closed
```

```
Program ends
```

---

7. Method Overloading

Method overloading allows a class to have multiple methods with the **same name** but **different parameters** (number, type, or order of parameters).

```
class MathOperations {
```

```
    // Overloaded methods for addition
    int add(int a, int b) {
        System.out.println("int + int");
        return a + b;
    }

    int add(int a, int b, int c) {
        System.out.println("int + int + int");
        return a + b + c;
    }

    double add(double a, double b) {
        System.out.println("double + double");
        return a + b;
    }

    double add(int a, double b) {
        System.out.println("int + double");
```

```
        return a + b;
    }

    double add(double a, int b) {
        System.out.println("double + int");
        return a + b;
    }
}

public class OverloadingDemo {
    public static void main(String[] args) {
        MathOperations m = new MathOperations();

        System.out.println("Result: " + m.add(5, 10));
        System.out.println("Result: " + m.add(5, 10, 15));
        System.out.println("Result: " + m.add(5.5, 10.5));
        System.out.println("Result: " + m.add(5, 10.5));
        System.out.println("Result: " + m.add(5.5, 10));
    }
```

```
}
```

Method Overloading vs Overriding:

Method Overloading	Method Overriding
Same class or different classes	Different classes (inheritance)
Same method name	Same method name
Different parameters	Same parameters
Compile-time polymorphism	Runtime polymorphism
Return type can be different	Return type must be same (or covariant)
No @Override annotation	@Override annotation recommended

Rules for Overloading:

- Must change parameter list.
- Can change return type.
- Can change access modifier.

- Can throw different exceptions.

---

8. Objects as Parameters

Objects can be passed as arguments to methods, allowing methods to operate on and potentially modify the state of those objects.

```
class Point {
    int x, y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void display() {
        System.out.println("(" + x + ", " + y + ")");
    }
}

class Geometry {
    // Method taking object as parameter
    double distance(Point p1, Point p2) {
        int dx = p1.x - p2.x;
        int dy = p1.y - p2.y;
        return Math.sqrt(dx*dx + dy*dy);
    }

    // Method that modifies object
    void translate(Point p, int dx, int dy) {
        p.x += dx;
        p.y += dy;
    }

    // Method that creates and returns new object
    Point createPoint(int x, int y) {
        return new Point(x, y);
    }
}
```

```
public class ObjectParameterDemo {
    public static void main(String[] args) {
        Point p1 = new Point(0, 0);
        Point p2 = new Point(3, 4);

        Geometry geo = new Geometry();

        double dist = geo.distance(p1, p2);
        System.out.println("Distance: " + dist);

        geo.translate(p1, 5, 5);
        System.out.print("After translation: ");
        p1.display();

        Point p3 = geo.createPoint(10, 20);
        System.out.print("New point: ");
        p3.display();
    }
}
```

---

9. Argument Passing: Call by Value

Important: Java is strictly **call by value**. This means that when arguments are passed to a method, a copy of the value is passed, not a reference to the original variable.

- For **primitive types**, a copy of the primitive value is passed. Changes to the parameter inside the method do not affect the original variable.
- For **object types**, a copy of the *reference value* (the memory address) is passed. This means both the original reference and the parameter inside the method point to the same object in memory. Therefore, modifications to the object's state through the parameter *will* affect the original object. However, reassigning the parameter to a *new* object inside the method will *not* affect the original reference.

```
public class ArgumentPassingDemo {

    // For primitive types - changes don't affect original
    static void changePrimitive(int x) {
```

```
    System.out.println("Inside method before change: " + x);
    x = x + 10;
    System.out.println("Inside method after change: " + x);
}

// For objects - reference value is passed (copy of reference)
static void changeObject(Point p) {
    System.out.println("Inside method before change: (" + p.x + ", " + p.y + ")");
    p.x = 100;
    p.y = 200;
    System.out.println("Inside method after change: (" + p.x + ", " + p.y + ")");
}

// Trying to reassign reference - doesn't affect original
static void reassignReference(Point p) {
    System.out.println("Inside method before reassign: (" + p.x + ", " + p.y + ")");
    p = new Point(999, 999);  // Creates new object locally
    System.out.println("Inside method after reassign: (" + p.x + ", " + p.y + ")");
}

public static void main(String[] args) {
    // Primitive type - call by value
    int num = 5;
    System.out.println("Before method call: " + num);
    changePrimitive(num);
    System.out.println("After method call: " + num);

    System.out.println("\n---\n");

    // Object type - still call by value (reference value is copied)
    Point p = new Point(10, 20);
    System.out.println("Before method call: (" + p.x + ", " + p.y + ")");
    changeObject(p);
    System.out.println("After method call: (" + p.x + ", " + p.y + ")");

    System.out.println("\n---\n");

    // Reassigning reference inside method
    Point p2 = new Point(1, 2);
    System.out.println("Before method call: (" + p2.x + ", " + p2.y + ")");
    reassignReference(p2);
    System.out.println("After method call: (" + p2.x + ", " + p2.y + ")");
    System.out.println("After method call: (" + p2.x + ", " + p2.y + ")");
}
```

```
}
```

Output:

```
Before method call: 5
```

```
Inside method before change: 5
Inside method after change: 15
After method call: 5

---

Before method call: (10, 20)
Inside method before change: (10, 20)
Inside method after change: (100, 200)
After method call: (100, 200)

---

Before method call: (1, 2)
Inside method before reassign: (1, 2)
Inside method after reassign: (999, 999)
```

```
After method call: (1, 2)
```

Key Understanding:

- Java passes a **copy of the value** for primitive types.
- Java passes a **copy of the reference value** for object types.
- Object contents can be modified through the copied reference, but reassignment of the reference itself within the method does not affect the caller's original reference.

---

10. Returning Objects

Methods in Java can return objects, allowing for complex data structures and method chaining.

```
class Matrix {
```

```
    int[][] data;
```

```
int rows, cols;

Matrix(int rows, int cols) {
    this.rows = rows;
    this.cols = cols;
    data = new int[rows][cols];
}

void fill(int value) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            data[i][j] = value;
        }
    }
}

// Method returning a Matrix object
Matrix add(Matrix other) {
    if (this.rows != other.rows || this.cols != other.cols) {
        System.out.println("Cannot add - dimensions mismatch");
        return null;
    }

    Matrix result = new Matrix(rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result.data[i][j] = this.data[i][j] + other.data[i][j];
        }
    }
    return result;
}

// Method returning itself for chaining
Matrix multiplyByScalar(int scalar) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            data[i][j] *= scalar;
        }
    }
    return this;  // Returning current object
}

void display() {
    for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
                System.out.print(data[i][j] + "\t");
            }
            System.out.println();
        }
        System.out.println();
    }
}

public class ReturningObjectsDemo {
    public static void main(String[] args) {
        Matrix m1 = new Matrix(2, 2);
        Matrix m2 = new Matrix(2, 2);

        m1.fill(5);
        m2.fill(3);

        System.out.println("Matrix 1:");
        m1.display();

        System.out.println("Matrix 2:");
        m2.display();

        Matrix m3 = m1.add(m2);
        System.out.println("Sum Matrix:");
        m3.display();

        // Method chaining using returned 'this'
        m1.multiplyByScalar(2).multiplyByScalar(3);
        System.out.println("Matrix after chain operations:");
        m1.display();
    }
```

```
}
```

---

11. Recursion

Recursion is a programming technique where a method calls itself to solve a problem. A recursive method must have a **base case** (a condition to stop recursion) and a **recursive case** (the step where the method calls itself).

```
public class RecursionDemo {


    // Factorial: n! = n * (n-1) * (n-2) * ... * 1
    int factorial(int n) {
        if (n == 0 || n == 1) {
            return 1;  // Base case
        }
        return n * factorial(n - 1);  // Recursive case
    }

    // Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...
    int fibonacci(int n) {
        if (n <= 1) {
            return n;  // Base case
        }
        return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive case
    }

    // Sum of digits: 1234 -> 1+2+3+4 = 10
    int sumOfDigits(int n) {
        if (n == 0) {
            return 0;
        }
        return (n % 10) + sumOfDigits(n / 10);
    }

    // Power: a^b
    int power(int base, int exponent) {
        if (exponent == 0) {
            return 1;
        }
        return base * power(base, exponent - 1);
    }

    // Palindrome check
    boolean isPalindrome(String str, int start, int end) {
        if (start >= end) {
            return true;
        }
        if (str.charAt(start) != str.charAt(end)) {
            return false;
        }
```

```
        return isPalindrome(str, start + 1, end - 1);
    }

    // Tower of Hanoi
    void towerOfHanoi(int n, char from, char to, char aux) {
        if (n == 1) {
            System.out.println("Move disk 1 from " + from + " to " + to);
            return;
        }
        towerOfHanoi(n - 1, from, aux, to);
        System.out.println("Move disk " + n + " from " + from + " to " + to);
        towerOfHanoi(n - 1, aux, to, from);
    }

    public static void main(String[] args) {
        RecursionDemo rd = new RecursionDemo();

        System.out.println("Factorial of 5: " + rd.factorial(5));
        System.out.println("Fibonacci of 10: " + rd.fibonacci(10));
        System.out.println("Sum of digits of 1234: " + rd.sumOfDigits(1234));
        System.out.println("2^8: " + rd.power(2, 8));
        System.out.println("Is 'radar' palindrome? " + rd.isPalindrome("radar", 0, 4));
        System.out.println("\nTower of Hanoi with 3 disks:");
        rd.towerOfHanoi(3, 'A', 'C', 'B');
    }
```

```
}
```

Output:

```
Factorial of 5: 120
```

```
Fibonacci of 10: 55
Sum of digits of 1234: 10
2^8: 256
Is 'radar' palindrome? true
Is 'hello' palindrome? false

Tower of Hanoi with 3 disks:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
```

```
Move disk 1 from A to C
```

Recursion vs Iteration:

Recursion	Iteration
Cleaner, more readable for some problems	Can be more complex for some problems
Uses more memory (stack frames)	Uses less memory
Risk of stack overflow for deep recursion	No stack overflow risk
Generally slower (function call overhead)	Generally faster
Natural for tree/graph problems	Natural for simple loops

---

12. Access Control (Access Modifiers)

Java provides four access levels to control the visibility of classes, fields, and methods:

5 **private**: Accessible only within the declaring class.
6 **default (no modifier)**: Accessible only within the same package.
7 **protected**: Accessible within the same package and by subclasses (even in different packages).
8 **public**: Accessible from anywhere.

```
package com.example.access;
```

```
public class AccessDemo {
    private int privateVar = 1;     // Only within this class
    int defaultVar = 2;             // Within package (no modifier)
    protected int protectedVar = 3;  // Package + subclasses
    public int publicVar = 4;       // Everywhere

    private void privateMethod() {
        System.out.println("Private method");
    }
```

```
    void defaultMethod() {
        System.out.println("Default method");
    }

    protected void protectedMethod() {
        System.out.println("Protected method");
    }

    public void publicMethod() {
        System.out.println("Public method");
    }

    // Within same class - all accessible
    void testWithinClass() {
        System.out.println(privateVar);    // ✓
        System.out.println(defaultVar);    // ✓
        System.out.println(protectedVar);  // ✓
        System.out.println(publicVar);     // ✓
        privateMethod();                   // ✓
        defaultMethod();                   // ✓
        protectedMethod();                 // ✓
        publicMethod();                    // ✓
    }
}

// Same package class
class SamePackageClass {
    void test() {
        AccessDemo obj = new AccessDemo();
        // System.out.println(obj.privateVar);  // ✗ - private
        System.out.println(obj.defaultVar);    // ✓ - default
        System.out.println(obj.protectedVar);  // ✓ - protected
        System.out.println(obj.publicVar);     // ✓ - public

        // obj.privateMethod();                // ✗
        obj.defaultMethod();                   // ✓
        obj.protectedMethod();                 // ✓
        obj.publicMethod();                    // ✓
    }
```

```
}
```

Subclass in different package:

```
package com.example.subclass;
```

```
import com.example.access.AccessDemo;

class SubclassDemo extends AccessDemo {
    void test() {
        SubclassDemo obj = new SubclassDemo();
        // System.out.println(obj.privateVar);   // ✗ - private
        // System.out.println(obj.defaultVar);   // ✗ - default (different package)
        System.out.println(obj.protectedVar);   // ✓ - protected (via inheritance)
        System.out.println(obj.publicVar);      // ✓ - public

        // obj.privateMethod();                 // ✗
        // obj.defaultMethod();                 // ✗
        obj.protectedMethod();                  // ✓
        obj.publicMethod();                     // ✓
    }
```

```
}
```

Access Modifier Summary:

Modifier	Class	Package	Subclass	World
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

Best Practices:

- Use private for implementation details to enforce encapsulation.
- Use public for API methods that should be exposed.
- Use protected for methods intended for extension by subclasses.
- Keep fields private and provide public getter/setter methods.

---

13. The static Keyword

The static keyword is used to create class-level members (variables and methods) that belong to the class itself, rather than to any specific object instance.

Static Variables (Class Variables):

- One copy of the variable is shared by all objects of the class.
- Accessed using the class name (e.g., ClassName.variableName).

```
class Employee {
```

```
    // Instance variables (each object has its own copy)
    int empId;
    String name;
    double salary;

    // Static variable (shared by all objects - one copy)
    static String companyName = "ABC Corporation";
    static int totalEmployees = 0;

    Employee(int empId, String name, double salary) {
        this.empId = empId;
        this.name = name;
        this.salary = salary;
        totalEmployees++;  // Increment shared counter
    }

    void display() {
        System.out.println("ID: " + empId + ", Name: " + name +
                           ", Salary: " + salary + ", Company: " + companyName);
    }

    static void showTotalEmployees() {
        System.out.println("Total employees: " + totalEmployees);
        // Cannot access non-static members from a static method
        // System.out.println(name);  // ERROR - non-static
    }
}

public class StaticVariableDemo {
    public static void main(String[] args) {
        System.out.println("Company: " + Employee.companyName);  // Access via class
```

```
        Employee e1 = new Employee(101, "Alice", 50000);
        Employee e2 = new Employee(102, "Bob", 55000);
        Employee e3 = new Employee(103, "Charlie", 60000);

        e1.display();
        e2.display();
        e3.display();

        Employee.showTotalEmployees();  // Total employees: 3

        // Changing static variable affects all objects
        Employee.companyName = "XYZ Technologies";

        System.out.println("\nAfter company name change:");
        e1.display();  // Shows new company name
    }
```

```
}
```

Static Methods:

- Belong to the class, not to any specific object.
- Can be called directly using the class name (e.g., ClassName.methodName()) without creating an object.
- Cannot access non-static (instance) variables or methods directly.

```
class MathHelper {
```

```
    // Static methods - can be called without object
    static int max(int a, int b) {
        return (a > b) ? a : b;
    }

    static int min(int a, int b) {
        return (a < b) ? a : b;
    }

    static double average(int[] arr) {
        int sum = 0;
        for (int num : arr) {
            sum += num;
        }
```

```
        return (double) sum / arr.length;
    }

    // Non-static method - requires object
    void display() {
        System.out.println("MathHelper instance");
    }
}

public class StaticMethodDemo {
    public static void main(String[] args) {
        // Call static methods without creating object
        System.out.println("Max of 10 and 20: " + MathHelper.max(10, 20));
        System.out.println("Min of 10 and 20: " + MathHelper.min(10, 20));

        int[] numbers = {10, 20, 30, 40, 50};
        System.out.println("Average: " + MathHelper.average(numbers));

        // Cannot call non-static method without object
        // MathHelper.display();  // ERROR

        // Need object for non-static
        MathHelper helper = new MathHelper();
        helper.display();
    }
}
```

Static Block:

- A block of code that is executed only once when the class is loaded into memory.
- Used for static initializations that cannot be done in a single line.

```
class DatabaseConfig {

    static String url;
    static String username;
    static String password;

    // Static block - executes when class is loaded (once)
    static {
        System.out.println("Static block: Loading database configuration");
```

```
        url = "jdbc:mysql://localhost:3306/mydb";
        username = "root";
        password = "secret";

        // Can perform complex initialization
        System.out.println("Configuration loaded");
    }

    // Instance block - executes before constructor for each object
    {
        System.out.println("Instance block: Creating new database connection");
    }

    DatabaseConfig() {
        System.out.println("Constructor called");
    }

    static void displayConfig() {
        System.out.println("URL: " + url);
        System.out.println("Username: " + username);
    }
}

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

        // Class loads here, static block executes
        DatabaseConfig.displayConfig();

        System.out.println("\nCreating first object:");
        DatabaseConfig db1 = new DatabaseConfig();

        System.out.println("\nCreating second object:");
        DatabaseConfig db2 = new DatabaseConfig();
    }
}
```

Output:

```
Main method started
```

```
Static block: Loading database configuration
```

```
Configuration loaded
URL: jdbc:mysql://localhost:3306/mydb
Username: root

Creating first object:
Instance block: Creating new database connection
Constructor called

Creating second object:
Instance block: Creating new database connection
```

```
Constructor called
```

Static vs Non-static Summary:

Static Member	Non-static Member
Belongs to class	Belongs to object
One copy shared by all objects	Each object has its own copy
Accessed using class name	Accessed using object reference
Can only access static members	Can access both static and non-static
Memory allocated once	Memory allocated per object
Loaded when class loads	Loaded when object created

---

14. The final Keyword

The final keyword in Java is used to restrict the user. It can be applied to variables, methods, and classes.

Final Variable (Constant):

- A final variable's value cannot be changed after it has been initialized.
- Must be initialized at the time of declaration or within a constructor/initializer block.

```
class Constants {
```

```
    // Final variable - cannot be changed after initialization
    final double PI = 3.14159;
    final int MAX_SIZE = 100;

    // Blank final variable - must be initialized in constructor
    final int MIN_SIZE;

    Constants() {
        MIN_SIZE = 1;  // Initialize blank final
    }

    void attemptChange() {
        // PI = 3.14;  // ERROR - cannot assign to final variable
    }
}

public class FinalVariableDemo {
    public static void main(String[] args) {
        Constants c = new Constants();
        System.out.println("PI: " + c.PI);
        System.out.println("MAX_SIZE: " + c.MAX_SIZE);
        System.out.println("MIN_SIZE: " + c.MIN_SIZE);

        // final local variable
        final int LOCAL_CONSTANT = 50;
        // LOCAL_CONSTANT = 60;  // ERROR
    }
```

```
}
```

Final Method (Cannot be overridden):

- A final method cannot be overridden by any subclass.
- Ensures that the implementation of the method remains consistent across the inheritance hierarchy.

```
class Parent {
```

```
    // Final method - cannot be overridden by subclasses
    final void display() {
        System.out.println("Parent's display method");
    }
```

```
    void normalMethod() {
        System.out.println("Normal method - can be overridden");
    }
}

class Child extends Parent {
    // @Override
    // void display() {  // ERROR - cannot override final method
    //     System.out.println("Child's display");
    // }

    @Override
    void normalMethod() {  // OK - not final
        System.out.println("Child's normal method");
    }
}

public class FinalMethodDemo {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();       // Calls Parent's version
        c.normalMethod();  // Calls Child's version
    }
}
```

Final Class (Cannot be inherited):

- A final class cannot be extended by any other class.
- Prevents inheritance and ensures the class's behavior cannot be modified.

```
final class FinalClass {
    void method() {
        System.out.println("Final class method");
    }
}

// class SubClass extends FinalClass {  // ERROR - cannot inherit final class
// }
```

```
// But can be used normally
public class FinalClassDemo {
    public static void main(String[] args) {
        FinalClass obj = new FinalClass();
        obj.method();
    }
}
```

---

15. Arrays Revisited

Arrays of Objects:

- Arrays can hold references to objects. Each element in an object array is a reference, which must be initialized to point to an actual object.

```
class Student {
    int rollNo;
    String name;
    double marks;

    Student(int rollNo, String name, double marks) {
        this.rollNo = rollNo;
        this.name = name;
        this.marks = marks;
    }

    void display() {
        System.out.printf("Roll: %d | Name: %-10s | Marks: %.2f\n",
                          rollNo, name, marks);
    }
}

public class ObjectArrayDemo {
    public static void main(String[] args) {
        // Creating array of objects
        Student[] students = new Student[5];
```

```
        // Initializing each element
        students[0] = new Student(101, "Alice", 85.5);
        students[1] = new Student(102, "Bob", 78.0);
        students[2] = new Student(103, "Charlie", 92.5);
        students[3] = new Student(104, "Diana", 88.0);
        students[4] = new Student(105, "Eve", 75.5);

        // Array initialization shorthand
        Student[] moreStudents = {
            new Student(201, "Frank", 82.0),
            new Student(202, "Grace", 91.0),
            new Student(203, "Henry", 79.5)
        };

        // Displaying all students
        System.out.println("=== Student List ===");
        for (Student s : students) {
            if (s != null) {
                s.display();
            }
        }

        System.out.println("\n=== More Students ===");
        for (Student s : moreStudents) {
            s.display();
        }

        // Finding top scorer
        Student top = students[0];
        for (Student s : students) {
            if (s != null && s.marks > top.marks) {
                top = s;
            }
        }
        System.out.println("\nTop scorer: " + top.name + " with " + top.marks + " marks");
    }
}
```

Multidimensional Arrays:

- Arrays of arrays, used to represent tables or grids.
- Can be rectangular (all rows same length) or jagged (rows with different lengths).

```java
public class MultiDimArrayDemo {

    public static void main(String[] args) {
        // 2D array declaration and initialization
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Jagged array (rows with different lengths)
        int[][] jagged = new int[3][];
        jagged[0] = new int[]{1};
        jagged[1] = new int[]{2, 3};
        jagged[2] = new int[]{4, 5, 6};

        // 3D array
        int[][][] cube = new int[3][3][3];

        // Fill cube with values
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                for (int k = 0; k < 3; k++) {
                    cube[i][j][k] = i + j + k;
                }
            }
        }

        // Display matrix
        System.out.println("Matrix:");
        for (int[] row : matrix) {
            for (int val : row) {
                System.out.print(val + " ");
            }
            System.out.println();
        }

        // Display jagged array
        System.out.println("\nJagged Array:");
        for (int i = 0; i < jagged.length; i++) {
            for (int val : jagged[i]) {
                System.out.print(val + " ");
            }
```

```
            System.out.println();
        }

        // Calculate sum of matrix
        int sum = 0;
        for (int[] row : matrix) {
            for (int val : row) {
                sum += val;
            }
        }
        System.out.println("\nSum of matrix: " + sum);
    }
}
```

Array Utility Methods (java.util.Arrays):

```
import java.util.Arrays;

public class ArrayUtilityDemo {
    public static void main(String[] args) {
        int[] numbers = {5, 2, 8, 1, 9, 3, 7, 4, 6};

        // Sorting
        Arrays.sort(numbers);
        System.out.println("Sorted: " + Arrays.toString(numbers));

        // Binary search (array must be sorted)
        int index = Arrays.binarySearch(numbers, 7);
        System.out.println("Index of 7: " + index);

        // Fill array
        int[] filled = new int[5];
        Arrays.fill(filled, 10);
        System.out.println("Filled array: " + Arrays.toString(filled));

        // Copy array
        int[] copy = Arrays.copyOf(numbers, numbers.length);
        int[] range = Arrays.copyOfRange(numbers, 2, 5);

        // Compare arrays
```

```
        boolean isEqual = Arrays.equals(numbers, copy);
        System.out.println("Arrays equal: " + isEqual);

        // Convert to string
        String str = Arrays.toString(numbers);
        System.out.println("String representation: " + str);

        // Parallel sort (for large arrays)
        int[] large = {9, 8, 7, 6, 5, 4, 3, 2, 1};
        Arrays.parallelSort(large);
        System.out.println("Parallel sorted: " + Arrays.toString(large));

        // 2D array comparison
        int[][] m1 = {{1, 2}, {3, 4}};
        int[][] m2 = {{1, 2}, {3, 4}};
        System.out.println("2D arrays equal: " + Arrays.deepEquals(m1, m2));
        System.out.println("2D array deep toString: " + Arrays.deepToString(m1));
    }
```

```
}
```

---

16. Complete Example: Student Management System

This section presents a complete Java program demonstrating the application of various OOP concepts discussed in this module, including classes, objects, constructors, methods, access modifiers, static members, and arrays of objects. The system allows for adding students, displaying their details, searching, and viewing statistics.

```
import java.util.Scanner;
```

```

class Student {
    // Private fields - encapsulation
    private int rollNo;
    private String name;
    private double[] marks;
    private static int totalStudents = 0;
    private static final int MAX_SUBJECTS = 5;
```

```
// Constructors
Student() {
    this(0, "Unknown");
}

Student(int rollNo, String name) {
    this.rollNo = rollNo;
    this.name = name;
    this.marks = new double[MAX_SUBJECTS];
    totalStudents++;
}

// Getter methods
public int getRollNo() { return rollNo; }
public String getName() { return name; }

// Setter methods
public void setName(String name) { this.name = name; }

public void setMarks(double[] marks) {
    if (marks.length <= MAX_SUBJECTS) {
        System.arraycopy(marks, 0, this.marks, 0, marks.length);
    }
}

public void setMark(int subject, double mark) {
    if (subject >= 0 && subject < MAX_SUBJECTS) {
        marks[subject] = mark;
    }
}

// Methods
double calculateAverage() {
    double sum = 0;
    for (double m : marks) {
        sum += m;
    }
    return sum / MAX_SUBJECTS;
}

char calculateGrade() {
    double avg = calculateAverage();
    if (avg >= 90) return 'A';
    else if (avg >= 80) return 'B';
```

```
        else if (avg >= 70) return 'C';
        else if (avg >= 60) return 'D';
        else return 'F';
    }

    void display() {
        System.out.println("\n--- Student Details ---");
        System.out.println("Roll No: " + rollNo);
        System.out.println("Name: " + name);
        System.out.print("Marks: ");
        for (int i = 0; i < MAX_SUBJECTS; i++) {
            System.out.print(marks[i] + " ");
        }
        System.out.println();
        System.out.printf("Average: %.2f\n", calculateAverage());
        System.out.println("Grade: " + calculateGrade());
    }

    static void showTotalStudents() {
        System.out.println("Total students enrolled: " + totalStudents);
    }
}

class StudentDatabase {
    private Student[] students;
    private int count;

    StudentDatabase(int capacity) {
        students = new Student[capacity];
        count = 0;
    }

    void addStudent(Student s) {
        if (count < students.length) {
            students[count++] = s;
            System.out.println("Student added: " + s.getName());
        } else {
            System.out.println("Database full!");
        }
    }

    Student searchByRollNo(int rollNo) {
        for (int i = 0; i < count; i++) {
            if (students[i].getRollNo() == rollNo) {
```

```
            return students[i];
        }
    }
    return null;
}

Student searchByName(String name) {
    for (int i = 0; i < count; i++) {
        if (students[i].getName().equalsIgnoreCase(name)) {
            return students[i];
        }
    }
    return null;
}

void displayAll() {
    if (count == 0) {
        System.out.println("No students in database");
        return;
    }
    System.out.println("\n=== All Students ===");
    for (int i = 0; i < count; i++) {
        students[i].display();
    }
}

void showStatistics() {
    if (count == 0) return;

    double highest = Double.MIN_VALUE;
    double lowest = Double.MAX_VALUE;
    double total = 0;
    Student topStudent = null, bottomStudent = null;

    for (int i = 0; i < count; i++) {
        double avg = students[i].calculateAverage();
        total += avg;

        if (avg > highest) {
            highest = avg;
            topStudent = students[i];
        }
        if (avg < lowest) {
            lowest = avg;
```

```
            bottomStudent = students[i];
        }
    }

    System.out.println("\n=== Statistics ===");
    System.out.printf("Class Average: %.2f\n", total / count);
    System.out.println("Highest Average: " + topStudent.getName() +
            " with " + highest);
    System.out.println("Lowest Average: " + bottomStudent.getName() +
            " with " + lowest);

    // Grade distribution
    int[] gradeCount = new int[5]; // A, B, C, D, F
    for (int i = 0; i < count; i++) {
        char grade = students[i].calculateGrade();
        switch (grade) {
            case 'A': gradeCount[0]++; break;
            case 'B': gradeCount[1]++; break;
            case 'C': gradeCount[2]++; break;
            case 'D': gradeCount[3]++; break;
            case 'F': gradeCount[4]++; break;
        }
    }

    System.out.println("Grade Distribution:");
    System.out.println("A: " + gradeCount[0]);
    System.out.println("B: " + gradeCount[1]);
    System.out.println("C: " + gradeCount[2]);
    System.out.println("D: " + gradeCount[3]);
    System.out.println("F: " + gradeCount[4]);
  }
}

public class StudentManagementSystem {
  static Scanner sc = new Scanner(System.in);

  static Student inputStudent() {
    System.out.print("Enter Roll Number: ");
    int roll = sc.nextInt();
    sc.nextLine();  // consume newline
    System.out.print("Enter Name: ");
    String name = sc.nextLine();

    Student s = new Student(roll, name);
```

```
        System.out.println("Enter marks for 5 subjects:");
        double[] marks = new double[5];
        for (int i = 0; i < 5; i++) {
            System.out.print("Subject " + (i+1) + ": ");
            marks[i] = sc.nextDouble();
        }
        s.setMarks(marks);

        return s;
    }

    public static void main(String[] args) {
        System.out.print("Enter database capacity: ");
        int capacity = sc.nextInt();
        StudentDatabase db = new StudentDatabase(capacity);

        int choice;
        do {
            System.out.println("\n=== Student Management System ===");
            System.out.println("1. Add Student");
            System.out.println("2. Display All Students");
            System.out.println("3. Search by Roll No");
            System.out.println("4. Search by Name");
            System.out.println("5. Show Statistics");
            System.out.println("6. Show Total Students");
            System.out.println("7. Exit");
            System.out.print("Enter choice: ");

            choice = sc.nextInt();

            switch (choice) {
                case 1:
                    if (capacity > 0) {
                        Student s = inputStudent();
                        db.addStudent(s);
                    }
                    break;
                case 2:
                    db.displayAll();
                    break;
                case 3:
                    System.out.print("Enter Roll No: ");
                    int roll = sc.nextInt();
```

```
                Student s = db.searchByRollNo(roll);
                if (s != null) s.display();
                else System.out.println("Student not found");
                break;
            case 4:
                sc.nextLine();
                System.out.print("Enter Name: ");
                String name = sc.nextLine();
                s = db.searchByName(name);
                if (s != null) s.display();
                else System.out.println("Student not found");
                break;
            case 5:
                db.showStatistics();
                break;
            case 6:
                Student.showTotalStudents();
                break;
            case 7:
                System.out.println("Exiting...");
                break;
            default:
                System.out.println("Invalid choice");
        }
    } while (choice != 7);

    sc.close();
  }
```

```
}
```

Quick Reference - Module 2

Class Syntax:

```
[access] class ClassName {
```

```
    // Fields
```

```
    [access] type fieldName;

    // Constructors
    [access] ClassName([parameters]) { }

    // Methods
    [access] returnType methodName([parameters]) { }
}
```

Object Creation:

```
ClassName obj = new ClassName([arguments]);
```

Access Modifiers (from least to most restrictive):

```
public > protected > default > private
```

Key Keywords:

Keyword	Purpose
this	Reference to current object
static	Class-level member
final	Constant or cannot override/inherit
new	Creates new object

End of Module 2: Objects and Classes