Advance Educational Activities Pvt. Ltd.

Unit 2: Object-Oriented Programming – I

2.1.1 Classes and Objects

What is a Class?

A **class** is a user-defined data type that serves as a **blueprint** for creating objects. It contains:

- **Fields** (also called attributes or properties) → store data
- **Methods** → define behavior (functions inside a class)

Think of a class as a **template**, and an object as a **real-world instance** of that template.

Real-Life Analogy:

- **Class** → Car (definition of what a car is)
- **Object** → car1, car2 (specific cars like Honda City, BMW)

Syntax of a Class in Java:

```
class ClassName {
    // Fields (data)
    dataType variableName;

    // Methods (behavior)
    returnType methodName(parameters) {
        // code
    }
}
```

Example: Defining a Class

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

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

This class:

- Has two fields: name and age
- Has one method: displayInfo()

2.1.1 What is an Object?

An **object** is a runtime entity created from a class using the new keyword.

Each object has:

- Its own copy of the class fields
- The ability to use methods defined in the class

Mini Activity 1:

Think of 3 real-world entities you can model using classes and objects (e.g., Book, Employee, MobilePhone). What fields and methods would you include?

2.2.1 Defining and Instantiating Classes

Instantiating (Creating) an Object:

ClassName obj = new ClassName();

Full Example with Object:

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

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();  // Object creation
        s1.name = "Alice";
        s1.age = 21;
        s1.displayInfo();  // Method call
    }
}
```

Output:

```
Name: Alice
Age: 21
```

2.2.2 Object vs Class (Summary):

CONCEPT	CLASS	OBJECT
TYPE	Blueprint / definition	Instance of a class
CREATED	At compile time	At runtime using new
MEMORY	No memory unless instantiated	Takes memory (fields & refs)

2.2.3 Practice Activities:

1. Create a class Book with fields title, author, price. Write a method displayBook().
2. Write a program to create 2 objects of class Car and assign different values.
3. Create a class Employee and display their name and salary.
4. Try creating a Person object without setting values. What is the default value of String and int?

2.3.1 Constructors in Java

Analogy:

Imagine you're assembling a **new phone** in a factory:

- When the phone (object) is created, the factory sets its initial configuration (model, battery, screen).
- This “setup process” is like a **constructor**—it **initializes the object** with meaningful default or passed values.

What is a Constructor?

A **constructor** is a special method in Java that:

- **Has the same name** as the class.
- **Has no return type** (not even void).
- **Is automatically called** when an object is created using new.

Why Use Constructors?

- To **initialize** objects at the time of creation.
- To **avoid calling a separate method** after object creation just to assign values.

Syntax of a Constructor

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

2.3.2 Default Constructor

```
class Student {
    Student() {
        System.out.println("Constructor called");
    }
}
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(); // Constructor called automatically
    }
}
```

Output:

Constructor called

This is called a **default (no-argument) constructor**.

2.3.3 Parameterized Constructor

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

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

    void display() {
        System.out.println(name + " - " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("John", 20);
        s1.display();
    }
}
```

Output:

```
John - 20
```

2.3.4 Constructor Overloading

You can define **multiple constructors** in the same class with **different parameters**.

Example:

```
class Rectangle {
    int length, width;

    Rectangle() {
        length = 0;
        width = 0;
    }

    Rectangle(int l, int w) {
        length = l;
        width = w;
    }

    void display() {
        System.out.println("Area: " + (length * width));
    }
}
```

Usage:

```
Rectangle r1 = new Rectangle();        // Area: 0
Rectangle r2 = new Rectangle(5, 3);    // Area: 15
```

Real-world Analogy for Constructor Overloading

Think of **pizza ordering**:

- You can order a **default pizza** (no toppings, small size).
- Or, you can **customize** it (extra cheese, medium, mushrooms).

In code, you have:

- Pizza() → default constructor
- Pizza(String size, String toppings) → parameterized constructor

2.3.5 Summary Table

FEATURE	DESCRIPTION
CONSTRUCTOR NAME	Same as class name
RETURN TYPE	None (not even void)
WHEN CALLED	Automatically at object creation
CAN BE OVERLOADED	Yes
CAN TAKE PARAMETERS	Yes (Parameterized Constructor)
CAN INITIALIZE VARIABLES	Yes

2.3.6 Practice Activities

1. Create a Car class with fields brand, price. Use a constructor to initialize and a method to display.
2. Overload the constructor in a Box class: one with no dimensions, one with length, breadth, height.
3. Write a class Account with name and balance, and use constructor overloading for different account types.
4. Create a class Laptop and use the constructor to auto-set brand and RAM size.

2.4.1 Method Overloading in Java

What is Method Overloading?

Method Overloading is a feature in Java that allows a class to have **more than one method with the same name** but **different parameters** (type, number, or order).

It's a way of performing **polymorphism** (compile-time / static polymorphism).

Analogy: Multiple Contact Numbers for the Same Person

Imagine you save a contact as **"Mom"** in your phone:

- "Mom (Mobile)"
- "Mom (Work)"
- "Mom (Home)"

All entries have the **same name**, but different **numbers** (parameters).
When you call "Mom", your phone chooses the correct number based on the **context**.

Similarly, in Java:

- Same method name
- Different parameter list
- Java decides **which version** to run based on the arguments passed.

Example of Method Overloading

```
public class Calculator {

    // Method 1: Add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method 2: Add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}
```

Usage:

```
Calculator calc = new Calculator();

System.out.println(calc.add(10, 20));          // Output: 30
System.out.println(calc.add(10, 20, 30));      // Output: 60
System.out.println(calc.add(5.5, 3.3));        // Output: 8.8
```

➡️ All methods are named add, but **Java differentiates them** based on **number and type of arguments**.

How Java Resolves Overloaded Methods

Java compiler looks at:

1. **Number of parameters**
2. **Type of parameters**
3. **Order of parameters**

Rules of Method Overloading

RULE	DESCRIPTION
✔	Must have **same method name**
✔	Must have **different parameter list**
❌	Changing only **return type** does NOT count as overloading
❌	Changing only **access modifiers or static/non-static** is NOT valid

Invalid Overload Example:

```
public int show() { return 1; }
public double show() { return 1.0; } // ❌ Compile-time error
```

Use Cases

- Creating flexible APIs
- Performing similar operations with different input types
- Simplifying code readability and structure

Quick Summary Table

FEATURE	DESCRIPTION
WHAT	Same method name, different signatures
TYPE	Compile-time polymorphism
PARAMETER VARIATIONS	Number, type, or order
RETURN TYPE ALONE	Cannot be used to overload
USE CASE	Code readability and method flexibility

2.5.1 static Keyword in Java

The static keyword in Java is used for **memory management**. It is applied to:

- Variables (static variables)
- Methods (static methods)
- Blocks (static blocks)
- Nested classes (static nested classes)

The static keyword tells Java: **"This belongs to the class, not to a specific object."**

Analogy: Common Notice Board in a School

In a school:

- Every **student (object)** has their **own notebook (instance variable)**.
- But the **common notice board (static variable)** is **shared by all students**.

Anything marked static in Java is like that **notice board** — **shared and common** to all.

2.5.2. Static Variable (Class Variable)

Definition:

A variable declared with static inside a class is **shared among all instances** of that class.

```
class Student {
    int rollNo;
    static String college = "ABC College"; // shared by all students
}
```

Usage:

```
Student s1 = new Student();
Student s2 = new Student();

System.out.println(s1.college); // ABC College
System.out.println(s2.college); // ABC College
Change it in one place → reflects for all:
Student.college = "XYZ College";
```

2.5.3 Static Method

Definition:

A method marked static can be **called without creating an object** of the class.

```
class MathUtil {
    static int square(int x) {
        return x * x;
    }
}
```

Usage:

```
int result = MathUtil.square(5); // No object needed
System.out.println(result);      // 25
```

Rules of Static Methods:

RULE	EXPLANATION
✅	Can access **only static data** directly
✅	Can call **only static methods** directly
❌	Cannot use this or super
❌	Cannot access instance variables/methods directly

2.5.4. Static Block

Definition:

A static block is used to **initialize static variables**. It executes **once** when the class is loaded.

```
class Config {
    static int version;
    static {
        version = 1;
        System.out.println("Static block executed");
    }
}
```

Usage:

```
public class Test {
    public static void main(String[] args) {
        System.out.println(Config.version); // Triggers static block once
    }
}
```

2.5.5 Static Nested Class

You can declare a **class inside another class** using static.

```
class Outer {
    static class Inner {
        void display() {
            System.out.println("Inside static nested class");
        }
    }
}
```

Usage:

```
Outer.Inner obj = new Outer.Inner();
obj.display();
```

Unlike non-static inner classes, static nested classes **don't require an object** of the outer class.

2.5.6 Why Use static?

- Memory-efficient: one copy for all instances.
- Easy access: no object needed.
- Useful for **constants**, **utility methods**, **configuration data**, etc.

2.5.7 Quick Recap Table

ELEMENT	STATIC USE & MEANING
STATIC VARIABLE	One copy shared by all instances
STATIC METHOD	Called without object; can't access instance members
STATIC BLOCK	Initializes static members; runs once when class is loaded
STATIC CLASS	Inner class that doesn't depend on outer class instance

Example Combining All

```
public class Example {
    static int count;

    static {
        count = 10;
        System.out.println("Static block run");
    }

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

    static class Helper {
        void help() {
            System.out.println("Helping...");
        }
    }

    public static void main(String[] args) {
        Example.showCount();
        Example.Helper h = new Example.Helper();
        h.help();
    }
}
```

2.6.1 this Keyword in Java

What is this in Java?

this is a **reference variable** in Java that refers to the **current object** — the object on which a method or constructor is being called.

Think of this as **"myself"** for an object.

Analogy: Self-Introduction

In a classroom:

- A student says: "**Hi, I am Rahul**."
- Here, "**I**" refers to **Rahul himself**.

In Java:

- An object can refer to itself using this.

2.6.2 Why Use this Keyword?

1. To **refer to current class instance variables**
2. To **invoke current class methods or constructors**
3. To **pass the current object as a parameter**
4. To **return the current class object**

Differentiate Between Instance and Local Variables

Problem:

When constructor parameters **have the same name** as instance variables.

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

    Student(int id, String name) {
        id = id;        // ❌ does NOT assign to instance variable
        name = name;    // ❌ same here
    }
}
```

Solution:

Use this to refer to instance variables.

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

    Student(int id, String name) {
        this.id = id;          // now it's clear
        this.name = name;
    }
}
```

this.id refers to the **instance variable**, id refers to the **parameter**.

To Invoke Current Class Method

```
class Demo {
    void display() {
        System.out.println("Display method called");
    }
    void show() {
        this.display(); // Calls display() of this object
    }
}
```

To Invoke Constructor from Another Constructor

This is called **constructor chaining** using this().

```
class Car {
    String brand;
    int year;

    Car() {
        this("Unknown", 0);  // calling parameterized constructor
    }

    Car(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }
}
```

Must be the **first statement** in the constructor.

To Pass Current Object as a Parameter

```
class Printer {
    void print(Student s) {
        System.out.println("Printing student: " + s.name);
    }
}

class Student {
    String name = "Amit";

    void show() {
        Printer p = new Printer();
        p.print(this); // passing current object
    }
}
```

To Return Current Object

Useful for **method chaining**.

```
class Person {
    Person getObject() {
        return this;
    }
}
```

2.6.3 Key Points About this

USE CASE	DESCRIPTION
REFERRING INSTANCE VARIABLE	When local and instance variable names clash
INVOKING INSTANCE METHOD	Call another method of the same class
CALLING ANOTHER CONSTRUCTOR	To chain constructors
PASS CURRENT OBJECT	Pass this to another class or method
RETURN CURRENT OBJECT	Enable method chaining

2.6.4 Summary Table

CONTEXT	EXAMPLE	MEANING
THIS.VARIABLE	this.name = name;	Refers to current object's variable
THIS.METHOD()	this.display();	Calls method on current object
THIS()	this("car", 2020);	Calls another constructor in same class
AS PARAMETER	p.print(this);	Passes current object
AS RETURN	return this;	Returns the current object

2.6.4 Real-Life Example: Method Chaining

```
class Builder {
    Builder start() {
        System.out.println("Started");
        return this;
    }

    Builder build() {
        System.out.println("Building...");
        return this;
    }

    Builder end() {
        System.out.println("Finished");
        return this;
    }
}

public class Main {
    public static void main(String[] args) {
        new Builder().start().build().end();  // Chaining using this
    }
}
```

2.7.1 Access Modifiers in Java

What Are Access Modifiers?

Access Modifiers in Java **define the visibility/scope** of:

- Classes
- Variables
- Methods
- Constructors

They control **who can access what** in your code.

Analogy: House Rooms and Keys

Imagine a house:

- **Private Room** – only you can enter.
- **Default Room** – only people inside the house (package) can enter.
- **Protected Room** – family (subclass) and housemates (package) can enter.
- **Public Room** – anyone can enter.

In Java, these levels are:

private Access Modifier

- Accessible **only within the same class**
- Not accessible outside the class, not even in subclasses

```
class Account {
    private double balance = 5000;

    private void showBalance() {
        System.out.println("Balance: " + balance);
    }
}
```

❌ Cannot access balance from another class directly.

(default) — No Modifier

- Also called **package-private**
- Accessible **within the same package only**

```
class Employee {
    int empId = 101; // default
    void show() {
        System.out.println("Employee ID: " + empId);
    }
}
```

❌ Cannot access from a class in a **different package**.

protected Access Modifier

- Accessible:
    - Within the **same package**
    - In **subclasses**, even if they are in a **different package**

```
class Person {
    protected String name = "John";
}

class Student extends Person {
    void display() {
        System.out.println("Name: " + name); // Accessible in subclass
    }
}
```

public Access Modifier

- Accessible **from anywhere** — any class, any package

```
public class Calculator {
    public void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }
}
```

Can be accessed from other classes or packages.

Comparison Table

MODIFIER	SAME CLASS	SAME PACKAGE	SUBCLASS (DIFFERENT PACKAGE)	OTHER PACKAGES
PRIVATE	✅	❌	❌	❌
(DEFAULT)	✅	✅	❌	❌
PROTECTED	✅	✅	✅	❌
PUBLIC	✅	✅	✅	✅

Access Modifiers with Classes

- **Top-level classes** can only be:
    - public
    - *(default)* (no modifier)

You **cannot declare a top-level class as private or protected.**

2.7.2 Best Practices

USE CASE	SUGGESTED MODIFIER
INTERNAL HELPER METHODS	private
FIELDS (ENCAPSULATION)	private + getters/setters
PUBLIC API METHODS	public
INHERITANCE SUPPORT METHODS	protected
PACKAGE-ONLY UTILITY CLASSES	*(default)*

2.7.3 Example Combining All

```
public class Example {

    private int secret = 123;          // Only this class
    int packageValue = 100;            // Same package
    protected String name = "Java";    // Same package + subclasses
    public void show() {               // Accessible everywhere
        System.out.println("Public method");
    }

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

2.7.4 Summary Table

ACCESS LEVEL	USE FOR	KEYWORD
PRIVATE	Sensitive data, internal logic	private
DEFAULT	Package-level classes	*(no keyword)*
PROTECTED	Inheritance support	protected
PUBLIC	Public APIs or core features	public

2.8.1 Encapsulation in Java

Definition:

Encapsulation is the **hinding of data (variables)** and the **code (methods)** that operate on the data **into a single unit**, while **restricting direct access** to some components.

It's one of the **four pillars of OOP** (along with inheritance, polymorphism, and abstraction).

Real-world Analogy: Medicine Bottle

Imagine a **medicine bottle**:

- The **medicine (data)** is inside the bottle.
- The **bottle (class)** protects the medicine.
- The **cap (access control)** prevents direct access — you can only get the medicine **through a prescription or controlled dose**.
- You can't reach in and change the ingredients directly.

Like a medicine bottle, **encapsulation controls how data is accessed and modified**, keeping it safe from misuse.

This is **encapsulation** — hiding the internal complexity and providing a clean interface.

How to Achieve Encapsulation in Java

1. **Make variables private**
2. **Provide public getter and setter methods**

Example:

```
class BankAccount {
    private double balance;  // private data

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0)
            balance += amount;
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance)
            balance -= amount;
    }
}
```

Usage:

```
BankAccount account = new BankAccount();
account.deposit(1000);
account.withdraw(500);
System.out.println(account.getBalance()); // 500
```

Direct access to balance is not allowed.
Only controlled access via getBalance(), deposit(), withdraw().

Why Use Encapsulation?

REASON	BENEFIT
HIDE IMPLEMENTATION DETAILS	Reduces complexity
SECURE DATA	Prevents unauthorized access
EASY TO MAINTAIN	Internal code changes don't affect external classes
ADDS CONTROL	You can add logic in setters/getters

2.8.2 Best Practices

- Always make fields private
- Provide public getter/setter methods **only if needed**
- Add validation logic inside setters
- Avoid public setters if the field should be **read-only**

2.8.3 Quick Recap Table

TERM	DESCRIPTION
ENCAPSULATION	Binding data and code, hiding data
PRIVATE	Used to restrict direct access
GETTER	Method to read private variable
SETTER	Method to write/update private variable

2.9.1 Introduction to Packages in Java

What is a Package in Java?

A **package** is a **namespace** that groups **related classes and interfaces** together.

Think of a package as a **folder** or **directory** that helps organize your Java files in a logical way.

Real-world Analogy: File Cabinet

Imagine a **file cabinet** in an office:

- Each **drawer** is like a **package**.
- Inside each drawer, you keep **files of a specific type** — e.g., invoices, resumes, reports.

Packages in Java work the same way — they organize your classes so you don't lose track and avoid name conflicts.

Why Use Packages?

BENEFIT	🔍 DESCRIPTION
BETTER ORGANIZATION	Group similar classes (e.g., all GUI classes in one package)
AVOID CLASS NAME CONFLICTS	Two classes with the same name can exist in different packages
CONTROLLED ACCESS	Use access modifiers (public, protected, etc.)
REUSABILITY	Code can be shared and reused across multiple projects

2.9.2 Creating a Package

Step 1: Declare package at the top of the Java file

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass");
    }
}
```

package must be the **first statement** in the file.

Step 2: Compile with directory structure

```
javac -d . MyClass.java
```

- -d . tells the compiler to create the folder structure based on the package name.
- This creates a folder mypackage/ containing MyClass.class.

Step 3: Import and Use the Package

```
import mypackage.MyClass;

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

2.9.3 Types of Packages

TYPE	DESCRIPTION	EXAMPLE
BUILT-IN	Predefined packages in Java API	java.util, java.io
USER-DEFINED	Packages created by the programmer	mypackage, school.student

2.9.4 Common Built-in Packages

PACKAGE	USE
JAVA.LANG	Basic classes (String, Math, etc.)
JAVA.UTIL	Collections, Date, Scanner, etc.
JAVA.IO	File handling
JAVA.SQL	Database connectivity
JAVAX.SWING	GUI programming

2.9.5 Package Structure (Hierarchy)

You can create **sub-packages** using dot (.) notation:

```
package com.company.department;

public class Employee {
    // code here
}
```

This would create a folder path: com/company/department/Employee.class

2.9.6 Access Modifiers in Packages

MODIFIER	ACCESSIBLE WITHIN SAME PACKAGE?	ACCESSIBLE FROM OUTSIDE PACKAGE?
PRIVATE	❌	❌
(DEFAULT)	✅	❌
PROTECTED	✅	(only through subclass)
PUBLIC	✅	✅

Best Practices

- Use company/domain-style package names (e.g., com.techacademy.utils)
- Keep related classes together (e.g., DAO classes in com.app.dao)
- Avoid using default package (no package declaration)
- Use meaningful names (e.g., student.records, billing.invoice)

Example Summary

```
// File: com/example/Hello.java
package com.example;

public class Hello {
    public void greet() {
        System.out.println("Hello from package!");
    }
}

// File: Main.java
import com.example.Hello;

public class Main {
    public static void main(String[] args) {
        Hello h = new Hello();
        h.greet();
    }
}
```

2.10.1 Array in Java

An **array** is a **collection of elements** of the **same data type** stored in **contiguous memory locations**. It is used to store multiple values under a single variable name.

Think of an array as a row of lockers (indexed), where each locker holds a value of the same type.

Syntax

1. Declaration:

```
int[] numbers;        // Recommended
// OR
int numbers[];        // Also valid
```

2. Instantiation:

```
numbers = new int[5];  // Array of 5 integers (default values: 0)
```

3. Initialization:

```
numbers[0] = 10;
numbers[1] = 20;
```

4. Combined:

```
int[] numbers = new int[] {10, 20, 30, 40, 50};
OR
int[] numbers = {10, 20, 30, 40, 50};
```

Real-world Analogy

Imagine you run a delivery service and assign lockers for packages. Each locker is numbered (indexed). You can access a specific locker (array index) to retrieve the package (value).

Example:

```
public class ArrayDemo {
    public static void main(String[] args) {
        String[] products = {"Laptop", "Tablet", "Mobile"};

        for (int i = 0; i < products.length; i++) {
            System.out.println(products[i]);
        }
    }
}
```

Output:

```
Laptop
Tablet
Mobile
```

Types of Arrays in Java

TYPE	DESCRIPTION	EXAMPLE
SINGLE-DIMENSIONAL	Linear list of elements	int[] marks = new int[5];
MULTI-DIMENSIONAL	Array of arrays (matrix-style)	int[][] matrix = new int[3][3];
JAGGED ARRAY	Array of arrays with different lengths	int[][] arr = new int[3][];

Array Properties

- Fixed size (declared at the time of creation)
- Index starts from 0
- Can store **primitive** or **reference types**
- Default values:
    - 0 for int
    - false for boolean
    - null for objects

Common Use Cases

- Storing student marks
- Holding a list of products
- Representing 2D data (like a matrix or a chessboard)

Advantages

- Easy to use
- Memory-efficient for fixed-size data
- Fast data access using index

Limitations

- Fixed size – can't grow dynamically (use ArrayList instead for dynamic needs)
- All elements must be of the same type
- Insertion/deletion in the middle is expensive (shifting needed)

Best Practices

- Always check array.length before iterating to avoid ArrayIndexOutOfBoundsException
- For unknown sizes, use **collections** like ArrayList
- Use **enhanced for loop** for readability:

```
for (String item : products) {
    System.out.println(item);
}
```

Summary

FEATURE	DESCRIPTION
DEFINITION	Fixed-size container for same-type elements
ACCESS	Via index (starting at 0)
TYPE	Single or multi-dimensional
BETTER ALTERNATIVE (DYNAMIC)	Use ArrayList or other collections

2.10.2 Array Input/Output in Java

- **Array Input**: Taking values from the user and storing them in an array.
- **Array Output**: Displaying the values stored in the array.

Both input and output can be performed using loops like for or for-each.

Real-World Analogy

Think of a row of exam paper slots for students:

- Input: Each student drops their paper into a numbered slot (array index).
- Output: The examiner reads papers from the slots one by one.

Example: Taking Input and Printing Output of an Integer Array

```
import java.util.Scanner;

public class ArrayIOExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter size of the array: ");
        int size = scanner.nextInt();

        int[] numbers = new int[size];

        // Input
        System.out.println("Enter " + size + " numbers:");
        for (int i = 0; i < size; i++) {
            numbers[i] = scanner.nextInt();
        }

        // Output
        System.out.println("You entered:");
        for (int i = 0; i < size; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }

        scanner.close();
    }
}
```

Sample Output

```
Enter size of the array: 4
Enter 4 numbers:
10
20
30
40
You entered:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
```

Using Enhanced For Loop for Output

```
for (int num : numbers) {
    System.out.println(num);
}
For String Array Input/Output Example
String[] names = new String[3];
Scanner sc = new Scanner(System.in);

System.out.println("Enter 3 names:");
for (int i = 0; i < names.length; i++) {
    names[i] = sc.nextLine();
}

System.out.println("Names entered:");
for (String name : names) {
    System.out.println(name);
}
```

Common Mistakes to Avoid

MISTAKE	FIX
arrayindexoutofboundsexception	Always use array.length for loops
forgetting to close scanner	Use scanner.close() at the end
mixing nextint() and nextline()	Use scanner.nextLine() after nextInt() to consume leftover \n

Summary Table

OPERATION	CODE SNIPPET
DECLARE ARRAY	int[] arr = new int[5];
INPUT VALUES	arr[i] = sc.nextInt(); in loop
OUTPUT VALUES	System.out.println(arr[i]); or for-each
DYNAMIC SIZE	int size = sc.nextInt(); then create array

2.10.3 2D Array in Java

A **2D array** is an array of arrays. It stores data in **rows and columns**, like a **matrix or table**.

Definition: A 2D array in Java is declared as: dataType[][] arrayName;

Real-World Analogy

Think of a **spreadsheet** or **chessboard**:

- Rows and columns hold values.
- Each cell is accessed by its row and column number (like [i][j]).

Syntax of 2D Arrays

Declaration:

```
int[][] matrix;          // Recommended
int matrix[][];          // Also valid
```

Instantiation:

```
matrix = new int[3][4];  // 3 rows and 4 columns
```

Initialization:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Accessing Elements

```
matrix[0][1];  // Access element at 1st row, 2nd column (value: 2)
```

Taking Input and Printing a 2D Array

```
import java.util.Scanner;

public class TwoDArrayIO {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int[][] matrix = new int[2][3];  // 2 rows, 3 columns
```

```
// Input
        System.out.println("Enter elements (2 rows, 3 columns):");
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }

        // Output
        System.out.println("Matrix:");
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }

        sc.close();
    }
}
```

Sample Output:

```
Enter elements (2 rows, 3 columns):
1 2 3
4 5 6
Matrix:
1 2 3
4 5 6
```

Enhanced For Loop for Output

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

Common Use Cases of 2D Arrays

USE CASE	EXAMPLE
MATRIX OPERATIONS	Addition, multiplication
TABLES OR GRIDS	Marks of students (rows: students, columns: subjects)
BOARD GAMES	Chess, Sudoku, Tic Tac Toe
IMAGE REPRESENTATION	Pixel values in grayscale

Important Points

- Indexing starts at 0 → [0][0] is first row, first column.
- Default values:
  - 0 for int/float
  - false for boolean
  - null for objects
- matrix.length gives **number of rows**
  matrix[0].length gives **number of columns**

Summary

FEATURE	DESCRIPTION
DECLARATION	int[][] arr = new int[3][4];
INPUT	Nested for loop
OUTPUT	Nested for or for-each loop
USE CASES	Tables, matrices, grids, boards

2.11.1 What is a String in Java

A **String** in Java is a **sequence of characters**, treated as an object of the String class in the java.lang package.

Java Strings are **immutable**, meaning once created, their values **cannot be changed**.

Real-World Analogy

Think of a **string** as a **word written in ink** on paper — once written (created), you can read, compare, or copy it, but **can't change the ink directly** (immutable). To change it, you create a new paper (new String).

String Declaration and Initialization

```
// Using string literal (stored in string pool)
String s1 = "Hello";

// Using new keyword (stored in heap)
String s2 = new String("World");
```

Common String Methods

METHOD	DESCRIPTION	EXAMPLE
LENGTH()	Returns the number of characters	s.length()
CHARAT(INT INDEX)	Returns character at a specific index	s.charAt(1)
TOUPPERCASE()	Converts to uppercase	s.toUpperCase()
TOLOWERCASE()	Converts to lowercase	s.toLowerCase()
EQUALS()	Compares content (case-sensitive)	s1.equals(s2)
EQUALSIGNORECASE()	Compares ignoring case	s1.equalsIgnoreCase(s2)
CONTAINS()	Checks if string contains substring	s.contains("text")
SUBSTRING(START, END)	Extracts substring	s.substring(1, 4)

METHOD	DESCRIPTION	EXAMPLE
REPLACE(A, B)	Replaces characters	s.replace("a", "b")
SPLIT(" ")	Splits string into array	s.split(" ")
TRIM()	Removes leading/trailing spaces	s.trim()

String Immutability Explained

```
String s = "Hello";
s.concat(" World");  // does NOT change original string
System.out.println(s);  // Output: Hello
To reflect the change:
s = s.concat(" World");
System.out.println(s);  // Output: Hello World
String Comparison
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");

System.out.println(s1 == s2);      // true (same object in pool)
System.out.println(s1 == s3);      // false (different object)
System.out.println(s1.equals(s3)); // true (same content)
Example Program
public class StringExample {
    public static void main(String[] args) {
        String name = "Java Programming";

        System.out.println("Length: " + name.length());
        System.out.println("Upper: " + name.toUpperCase());
        System.out.println("First char: " + name.charAt(0));
        System.out.println("Contains 'Java': " + name.contains("Java"));
    }
}
```

Best Practices

- Prefer string **literals** for memory efficiency.
- Use equals() for comparison, **not ==**.
- Use StringBuilder for heavy string modifications (e.g., in loops).
- Avoid unnecessary string concatenations — it's memory-expensive.

Summary

TOPIC	KEY POINT
IMMUTABLE	Once created, cannot be changed
STORAGE	String Pool (literal), Heap (new)
METHODS	Powerful built-in methods for processing
COMPARISON	equals() for content, == for reference check

2.11.2 String Literal Vs String Object

What is a String Literal?

A **String literal** is any sequence of characters enclosed in double quotes, e.g.:

String s1 = "Java";

- Stored **in the String Constant Pool (SCP)** inside the **Method Area** of JVM memory.
- If "Java" already exists in the SCP, it **does not create a new object** — it just returns a reference to the existing one.

What is a String Object?

You can also create a String using the new keyword:

String s2 = new String("Java");

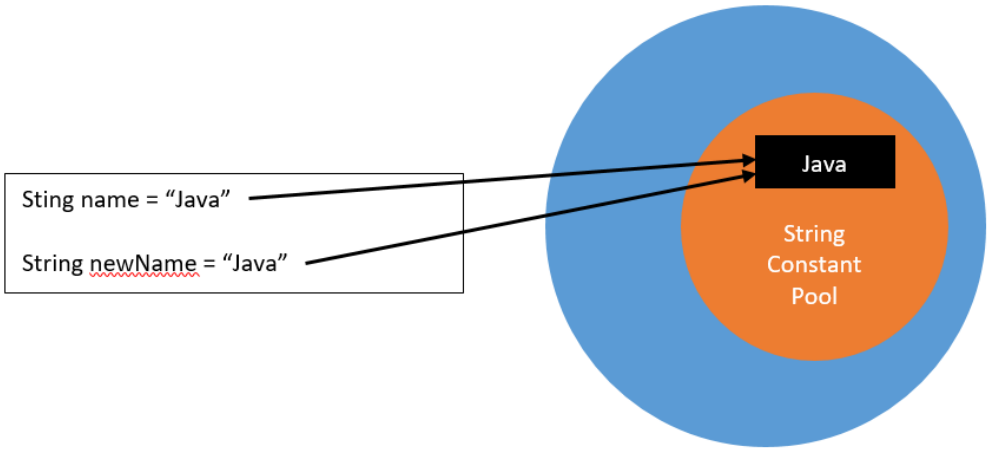
- Creates a **new object in the Heap** memory.
- Also refers to "Java" in the **SCP** (for internal character storage).
- So this creates **two objects**:
  → One in Heap (via new)
  → One in SCP (if not already present)

Key Differences: Literal vs Object

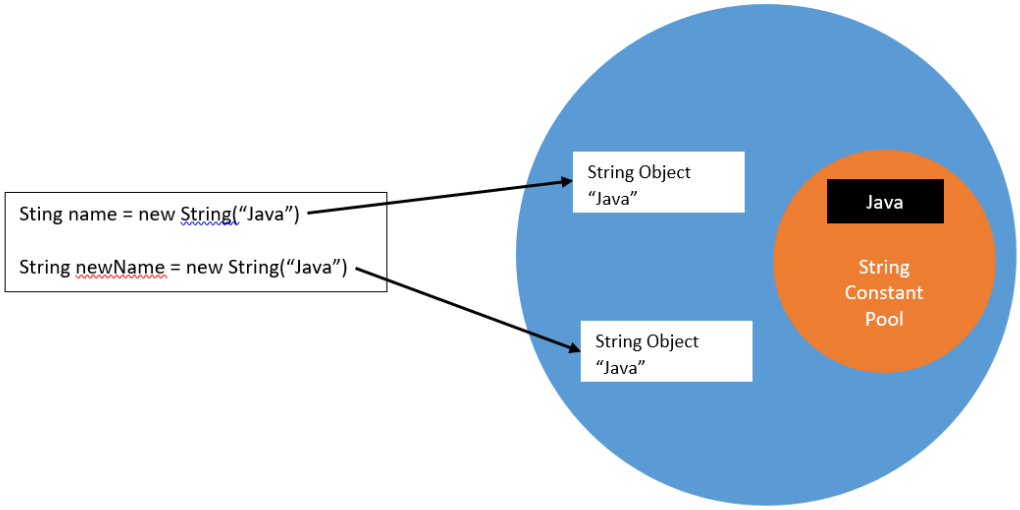
FEATURE	STRING LITERAL ("JAVA")	STRING OBJECT (NEW STRING("JAVA"))
MEMORY LOCATION	String Constant Pool (SCP)	Heap + reference to SCP
REUSE	Reused if already exists	Always a new object
EFFICIENCY	Memory efficient	Less efficient (creates duplicate)
COMPARISON USING ==	Might return true	Always returns false
EXAMPLE	String s = "Java";	String s = new String("Java");

Memory Diagram

String Literal



String Object



Comparison Example

```
public class StringMemoryDemo {
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = "Java";
        String s3 = new String("Java");

        System.out.println(s1 == s2);      // true (same SCP object)
        System.out.println(s1 == s3);      // false (heap vs SCP)
        System.out.println(s1.equals(s3)); // true (content comparison)
    }
}
```

Output:

```
true
false
true
```

Real-World Analogy

Imagine the **SCP as a library**:

- When you ask for a book titled "Java":
    - If it already exists on the shelf, you're **given the same copy** (literal).
    - If you say, "I want a brand new one" (using new), then the library **prints a fresh copy** and gives it to you.

Best Practices

- Use string **literals** when possible to save memory.
- ❌ Avoid unnecessary use of new String() unless you need a **separate object**.
- Always use .equals() to compare strings (not ==).

Bonus: intern() Method

You can force a string object to refer to SCP:

String s4 = new String("Java").intern();

Now s4 will point to "Java" in SCP — just like a literal.

Summary

TERM	MEANING
SCP	String Constant Pool — stores unique string literals
HEAP	General object storage area in memory
NEW STRING()	Always creates a new object in Heap
INTERN()	Moves or refers a string to the SCP
==	Compares reference (address)
EQUALS()	Compares content

2.11.3 StringBuffer in Java

1. Overview / Explanation

- StringBuffer is a **mutable** sequence of characters (unlike String, which is immutable).
- Part of java.lang package.
- Used when you need to **modify strings frequently** (e.g., appending, inserting, deleting).
- **Thread-safe** – methods are **synchronized**, so safe to use in multi-threaded environments.

Use Case: When building dynamic strings in a loop or multithreaded app – e.g., processing input, generating reports.

2. Declaration and Instantiation

```
StringBuffer sb1 = new StringBuffer();                // Empty buffer
StringBuffer sb2 = new StringBuffer("Hello");         // Initialized
StringBuffer sb3 = new StringBuffer(50);              // With capacity
```

3. Common Methods with Examples

append()

Adds text at the end.

```
sb1.append("Java");
System.out.println(sb1);  // Java
```

insert()

Inserts text at a specific index.

```
sb1.insert(4, " Programming");
System.out.println(sb1);  // Java Programming
```

replace()

Replaces part of the string between start and end index.

```
sb1.replace(0, 4, "Python");
System.out.println(sb1);  // Python Programming
```

delete()

Deletes characters between start and end index.

```
sb1.delete(0, 7);
System.out.println(sb1);  // Programming
```

reverse()

Reverses the entire content.

```
sb1.reverse();
System.out.println(sb1);  // gnimmargorP
```

length() and capacity()

```
System.out.println(sb1.length());   // No. of characters
System.out.println(sb1.capacity()); // Buffer capacity
```

charAt() and setCharAt()

```
char ch = sb1.charAt(0);
sb1.setCharAt(0, 'X');
```

4. Why Use StringBuffer Over String?

OPERATION	STRING	STRINGBUFFER
MUTABILITY	Immutable	Mutable
THREAD-SAFE	Not thread-safe	Yes
PERFORMANCE	Slower in loops	Faster in loops

5. StringBuffer vs StringBuilder

FEATURE	STRINGBUFFER	STRINGBUILDER
THREAD-SAFETY	Yes (synchronized)	No
PERFORMANCE	Slower	Faster (in single-thread)
USE CASE	Multithreaded apps	Single-thread apps

2.11.4 StringBuilder in Java

1. Overview / Explanation

- StringBuilder is a **mutable** sequence of characters, just like StringBuffer.
- **Not thread-safe**, but **faster** than StringBuffer in single-threaded applications.
- Part of java.lang package.
- Ideal when you're performing **lots of modifications to strings** in a **single-threaded** context.

Use Case: Building or modifying strings inside loops, parsing files, generating HTML reports, etc.

2. Declaration and Instantiation

```
StringBuilder sb1 = new StringBuilder();                // Empty buffer
StringBuilder sb2 = new StringBuilder("Hello");         // With initial value
StringBuilder sb3 = new StringBuilder(50);              // With specific capacity
```

3. Common Methods with Examples

append()

```
sb1.append("Java");
System.out.println(sb1);  // Java
```

insert()

```
sb1.insert(4, " World");
System.out.println(sb1);  // Java World
```

replace()

```
sb1.replace(0, 4, "Hello");
System.out.println(sb1);  // Hello World
```

delete()

```
sb1.delete(5, 11);
System.out.println(sb1);  // Hello
```

reverse()

```
sb1.reverse();
System.out.println(sb1);  // olleH
```

length() and capacity()

```
System.out.println(sb1.length());    // Number of characters
System.out.println(sb1.capacity());  // Total buffer size (default is 16 + initial
content length)
```

charAt() and setCharAt()

```
char ch = sb1.charAt(0);
sb1.setCharAt(0, 'M');
System.out.println(sb1);  // Ml...
```

4. StringBuilder vs String vs StringBuffer

FEATURE	STRING	STRINGBUILDER	STRINGBUFFER
MUTABILITY	❌ Immutable	Mutable	Mutable
THREAD-SAFE	❌ No	❌ No	Yes
PERFORMANCE	❌ Slower	Fastest	⚠ Slower (sync)
BEST FOR	Constant text	Fast updates (1 thread)	Multithreading

2.11.5 String vs StringBuffer vs StringBuilder

FEATURE	STRING	STRINGBUFFER	STRINGBUILDER
MUTABILITY	❌ Immutable	Mutable	Mutable
THREAD-SAFE	❌ No	Yes (all methods are synchronized)	❌ No
PERFORMANCE	❌ Slowest (new object per change)	⚠ Slower (due to thread-safety overhead)	Fastest (no sync overhead)
SYNCHRONIZATION	❌ Not applicable	Synchronized	❌ Not synchronized
USE CASE	Constant/fixed string content	Multi-threaded environment	Single-threaded environment
PACKAGE	java.lang	java.lang	java.lang
INTRODUCED IN	JDK 1.0	JDK 1.0	JDK 1.5
METHODS FOR CHANGE	N/A (strings can't be modified)	append(), insert(), delete(), replace()	append(), insert(), delete(), replace()
MEMORY EFFICIENT?	❌ No (creates many objects)	Yes	Yes

Example Comparison

```
// String (immutable)
String s = "Hello";
s = s + " World";  // Creates a new String object

// StringBuffer (mutable, thread-safe)
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");  // Modifies original object

// StringBuilder (mutable, not thread-safe)
StringBuilder sb2 = new StringBuilder("Hello");
sb2.append(" World");  // Modifies original object
```

When to Use What?

SITUATION	RECOMMENDED TYPE
SIMPLE, UNCHANGING TEXT	String
MANY STRING CHANGES IN MULTITHREADED CODE	StringBuffer
MANY STRING CHANGES IN SINGLE-THREADED CODE	StringBuilder

2.12.1 Reusability (in Java)

Definition:

Reusability is the ability to write code once and **use it multiple times** without rewriting it.

Real-life Analogy:

Like using the **same key for multiple locks** in your house — one key, many doors.

In Java:

- Achieved using:
    - **Methods**
    - **Classes**
    - **Inheritance**
    - **Packages**

Benefits:

- Reduces code duplication
- Saves development time
- Improves code quality and consistency
- Easier to debug and test

2.12.2 Modularity (in Java)

Definition:

Modularity is the process of dividing a large program into **independent, interchangeable modules**.

Real-life Analogy:

Like assembling a **car from different parts** (engine, wheels, seats) — each part (module) works independently but together they form a complete system.

In Java:

- Achieved using:
    - **Classes and methods**
    - **Packages**
    - **Modules (Java 9+)**

Benefits:

- Easier to understand and manage code
- Improves maintainability
- Encourages separation of concerns
- Simplifies teamwork and parallel development

Quick Comparison:

FEATURE	REUSABILITY	MODULARITY
PURPOSE	Use the same code again	Divide code into logical parts
REDUCES	Duplication	Complexity
ACHIEVED BY	Inheritance, methods, packages	Classes, packages, Java modules