SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES

(Autonomous) — Chittoor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COMPLETE LECTURE NOTES

UNITS I – V

23CSE232 — OBJECT ORIENTED PROGRAMMING THROUGH JAVA

II B.Tech — III Semester | L T P C: 3 0 0 3

(Common to CSE, CSM, CAI, CSD)

Prepared by: A. Ajitha & M.L.Madhuri
Department of Computer Science and Engineering

TABLE OF CONTENTS

Unit V — Strings, Multithreading, JDBC & JavaFX

UNIT I

Object Oriented Programming Concepts

23CSE232 — Object Oriented Programming through Java

1. Object Oriented Programming – Basic Concepts and Principles

1.1 Introduction

Object Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which contain data (fields/attributes) and code (methods) that operate on that data. Unlike procedure-oriented programming, where the focus is on writing functions/procedures, OOP focuses on creating objects that model real-world entities. Java is a purely object-oriented language (except for primitive data types) and every Java program is built around classes and objects.

1.2 Procedure-Oriented vs Object-Oriented Programming

- Procedure-Oriented Programming (POP) divides a program into functions; data is separate from functions and moves freely across the program, which is unsafe.
- Object-Oriented Programming (OOP) divides a program into objects; data and functions (methods) are bound together inside a class, providing better security through encapsulation.
- POP follows a top-down design approach, while OOP follows a bottom-up design approach.
- OOP supports code reusability through inheritance, whereas POP does not directly support this.

1.3 Basic Concepts / Principles of OOP

The four fundamental pillars (principles) of Object Oriented Programming are:

- Class and Object
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

1.3.1 Class

A class is a user-defined blueprint or template from which objects are created. It represents a set of properties (fields) and behaviours (methods) common to all objects of that type. A class itself does not occupy memory for data until an object is created.

1.3.2 Object

An object is a real-world entity that is an instance of a class. It has state (values of fields), behaviour (methods), and identity (unique memory address). Objects are created using the 'new' keyword in Java.

1.3.3 Encapsulation

Encapsulation is the technique of binding data (fields) and the methods that operate on that data into a single unit (class), and restricting direct access to some of the object's components using access modifiers such as private. It is achieved in Java using private fields with public getter and setter methods.

1.3.4 Abstraction

Abstraction means hiding the internal implementation details and showing only the essential features of an object to the user. In Java, abstraction is achieved using abstract classes and interfaces.

1.3.5 Inheritance

Inheritance is the mechanism by which one class (subclass/child class) acquires the properties and behaviour of another class (superclass/parent class), promoting code reusability. It is implemented in Java using the 'extends' keyword.

1.3.6 Polymorphism

Polymorphism means 'many forms'. It allows an object to behave differently based on the context. Java supports two types of polymorphism: compile-time (method overloading) and run-time (method overriding).

1.4 Benefits of Object Oriented Programming

- Improves software re-usability through inheritance.
- Provides data security through encapsulation and access control.
- Reduces code redundancy and improves maintainability.
- Makes it easier to model real-world problems.
- Improves productivity through modular design.

1.5 Comparison: Procedure-Oriented vs Object-Oriented Programming

Basis	Procedure-Oriented (POP)	Object-Oriented (OOP)
Approach	Top-down approach	Bottom-up approach
Division	Divided into functions	Divided into objects
Data Security	Data moves freely, less secure	Data is bound with methods, more secure
Overloading	Not possible	Functions/operators can be overloaded
Data Hiding	Not supported	Supported through encapsulation
Examples	C, Pascal, Fortran	Java, C++, Python

1.6 Detailed Example: Combining All OOP Principles

The following complete program models a Bank Account and demonstrates encapsulation, inheritance, abstraction, and polymorphism working together.

```
abstract class Account {
    protected String accNo;
    protected double balance;         // encapsulated (protected) data

    Account(String accNo, double balance) {
        this.accNo = accNo;
        this.balance = balance;
    }

    abstract double calculateInterest();  // abstraction

    void deposit(double amount) {
        balance += amount;
    }

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

class SavingsAccount extends Account {           // inheritance
    SavingsAccount(String accNo, double balance) {
        super(accNo, balance);
    }
```

```
    @Override
    double calculateInterest() {                  // polymorphism (overriding)
        return balance * 0.04;
    }
}

class CurrentAccount extends Account {
    CurrentAccount(String accNo, double balance) {
        super(accNo, balance);
    }

    @Override
    double calculateInterest() {
        return 0.0;   // current accounts earn no interest
    }
}

public class BankDemo {
    public static void main(String[] args) {
        Account a1 = new SavingsAccount("SB101", 10000);
        Account a2 = new CurrentAccount("CA202", 25000);

        a1.deposit(2000);
        a1.showBalance();
        System.out.println("Interest: " + a1.calculateInterest());

        a2.showBalance();
        System.out.println("Interest: " + a2.calculateInterest());
    }
}

/* Output:
Account: SB101 Balance: 12000.0
Interest: 480.0
Account: CA202 Balance: 25000.0
Interest: 0.0
*/
```

1.7 Constructors in Java

A constructor is a special method used to initialize objects at the time of creation. It has the same name as the class and no return type. Java provides default constructors automatically if none is defined, and also supports parameterized constructors and constructor overloading.

Default Constructor:

```
class Book {
    String title;

    Book() {                      // default (no-argument) constructor
        title = "Untitled";
        System.out.println("Default constructor called");
    }
}
```

Parameterized Constructor and Constructor Overloading:

```
class Book {
    String title;
    double price;

    Book() {                              // constructor 1 - no-arg
        this("Untitled", 0.0);             // calls constructor 3 using this()
    }

    Book(String title) {                   // constructor 2 - overloaded
        this(title, 0.0);
    }

    Book(String title, double price) {      // constructor 3 - overloaded
        this.title = title;
        this.price = price;
    }

    void display() {
        System.out.println(title + " - Rs." + price);
    }
}

public class ConstructorDemo {
    public static void main(String[] args) {
        Book b1 = new Book();
        Book b2 = new Book("Java Basics");
        Book b3 = new Book("OOP Concepts", 450.0);

        b1.display();
        b2.display();
        b3.display();
    }
}
```

```
/* Output:
Untitled - Rs.0.0
Java Basics - Rs.0.0
OOP Concepts - Rs.450.0
*/
```

1.8 Access Modifiers in Java

Access modifiers control the visibility (accessibility) of classes, fields, and methods. They are essential for implementing encapsulation properly.

Modifier	Same Class	Same Package	Subclass (Different Package)	Different Package
private	Yes	No	No	No
default (no modifier)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes
class Demo { private int secret = 10; // accessible only within this class int defaultVar = 20; // accessible within the same package protected int protectedVar = 30; // accessible in package and subclasses public int publicVar = 40; // accessible from anywhere }				

1.9 Class vs Object – Quick Comparison

Class	Object
A logical/blueprint entity	A physical/real-world entity

Class	Object
Declared once using the class keyword	Can be created multiple times using 'new'
Does not occupy memory (until instantiated)	Occupies memory when created
Defines properties and behaviours	Holds actual values for those properties

1.10 Abstract Class vs Interface

Both abstract classes and interfaces are used to achieve abstraction in Java, but they differ in several important ways:

Basis	Abstract Class	Interface
Methods	Can have both abstract and concrete (implemented) methods	Traditionally only abstract methods; default/static methods allowed from Java 8 onwards
Variables	Can have final, non-final, static, and non-static variables	Only public, static, and final variables (constants)
Inheritance	A class can extend only one abstract class	A class can implement multiple interfaces
Constructors	Can have constructors	Cannot have constructors
Keyword	abstract class, extends	interface, implements

1.11 Key Points to Remember

- A class is a blueprint; an object is a runtime instance of a class.
- Encapsulation is implemented using private fields with public getters/setters.
- Abstraction hides implementation and is achieved with abstract classes/interfaces.
- Inheritance promotes code reuse using the 'extends' keyword.
- Polymorphism allows one interface to be used for a general class of actions (overloading and overriding).

Review Questions

Short Answer Questions:

- Define Object Oriented Programming. How is it different from Procedure Oriented Programming?

- What is a class? What is an object? Give an example of each.
- Explain encapsulation with a suitable Java program.
- What is abstraction? How is it achieved in Java?
- Differentiate between method overloading and method overriding.

Long Answer / Programming Questions:

- Explain all the principles of OOP with neat Java example programs for each.
- Write a Java program to demonstrate encapsulation using a class 'Employee' with private salary field.
- Write a Java program using inheritance and polymorphism to model different types of vehicles.

2. Program Structure in Java

2.1 Introduction

Every Java program follows a well-defined structure. Understanding the structure is essential for writing syntactically correct programs. A Java program essentially consists of documentation, package statements, import statements, class/interface declarations, and the main method which acts as the entry point for execution.

2.2 General Structure of a Java Program

```
// 1. Documentation Section (optional comments)
/* Program to demonstrate Java structure */

// 2. Package Statement (optional)
package com.example.myapp;

// 3. Import Statements (optional)
import java.util.Scanner;

// 4. Class Definition
public class HelloWorld {

    // 5. Main Method - Program Execution starts here
    public static void main(String[] args) {
        // 6. Statements
        System.out.println("Hello, World!");
    }
}
```

2.2.1 Understanding JVM, JRE, and JDK

Java achieves platform independence through the concept of 'Write Once, Run Anywhere'. This is made possible by three key components:

Component	Full Form	Description
JVM	Java Virtual Machine	Executes Java bytecode (.class files) and provides platform independence. Each OS has its own JVM implementation.
JRE	Java Runtime Environment	Provides the libraries and JVM required to run Java applications, but not the tools to develop them.
JDK	Java Development Kit	A superset of JRE that includes the compiler (javac), debugger, and other development tools needed to write and compile Java programs.

Java Compilation and Execution Process:

```
Step 1: Write source code       -->  HelloWorld.java
Step 2: Compile using javac     -->  javac HelloWorld.java
                                      (produces HelloWorld.class - bytecode)
Step 3: Run using java          -->  java HelloWorld
                                      (JVM interprets/executes the bytecode)
```

2.2.2 Understanding the main() Method Signature

Every standalone Java application requires the main() method as its starting point. Each keyword in its signature has a specific meaning:

Keyword / Part	Meaning
public	Access modifier - main() must be accessible from outside the class by the JVM
static	Allows the JVM to call main() without creating an object of the class
void	main() does not return any value

Keyword / Part	Meaning
main	The fixed method name recognized by the JVM as the entry point
String[] args	An array of Strings used to receive command line arguments

2.3 Writing Simple Java Programs

A simple Java program to add two numbers and display the result:

```
public class AddNumbers {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int sum = a + b;
        System.out.println("Sum = " + sum);
    }
}

/* Output:
Sum = 30
*/
```

2.4 Elements or Tokens in Java Programs

The smallest individual units in a Java program are called tokens. The Java compiler divides the program text into tokens for compilation. The main types of tokens are:

- Keywords: reserved words with special meaning, e.g. class, public, static, void, int, if, else.
- Identifiers: names given to variables, methods, classes, e.g. sum, Student, main.
- Literals (Constants): fixed values such as 10, 3.14, 'A', "Hello", true.
- Operators: symbols that perform operations, e.g. +, -, *, /, =, ==.
- Separators/Punctuators: symbols used to separate code elements, e.g. ; , ( ) { } [ ].

2.5 Command Line Arguments

Command line arguments are values passed to the main() method at the time of running the program. They are stored in the String array parameter of main and can be used to supply input without using Scanner.

```
public class CommandLineDemo {
    public static void main(String[] args) {
        System.out.println("Number of arguments: " + args.length);
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + ": " + args[i]);
        }
    }
}

// Run using:
// javac CommandLineDemo.java
// java CommandLineDemo Java OOP Syllabus

/* Output:
Number of arguments: 3
Argument 0: Java
Argument 1: OOP
Argument 2: Syllabus
*/
```

2.6 Java Statements

A statement is a complete instruction that the Java compiler can execute. Java statements always end with a semicolon (;). Common types include declaration statements, expression statements, and control-flow statements.

```
int x = 5;                // declaration statement
x = x + 1;                // expression statement
System.out.println(x);    // method call statement
if (x > 0) {               // control statement
    System.out.println("Positive");
}
```

2.7 Escape Sequences

Escape sequences are special character combinations beginning with a backslash (\) used to represent characters that cannot be typed directly or have special meaning.

Escape Sequence	Meaning
\n	New line
\t	Horizontal tab

Escape Sequence	Meaning
\b	Backspace
\r	Carriage return
\\"	Double quote
\\'	Single quote
\\\\	Backslash

```
public class EscapeDemo {
    public static void main(String[] args) {
        System.out.println("Name:\tRavi");
        System.out.println("He said, \"Java is fun!\"");
        System.out.println("Line1\nLine2");
    }
}
```

2.8 Comments in Java

Comments are non-executable statements used to make code readable. Java supports three types of comments:

```
// 1. Single-line comment

/* 2. Multi-line
      comment */

/** 3. Documentation comment
 * used by javadoc tool
 * @author Student
 */
```

2.9 Programming Style

- Use meaningful and descriptive names for classes, methods, and variables.
- Follow camelCase for variables/methods (e.g. studentName) and PascalCase for class names (e.g. StudentRecord).
- Indent code consistently to show block structure.

- Use comments to explain complex logic.
- Keep one class per file with the file name matching the public class name.

2.9.1 Java Naming Conventions Summary

Identifier Type	Convention	Example
Class / Interface	PascalCase (first letter of each word capitalized)	StudentRecord, Runnable
Method / Variable	camelCase (first word lowercase, rest capitalized)	calculateTotal, studentName
Constant (final)	UPPER_CASE with underscores	MAX_VALUE, PI
Package	all lowercase	com.college.student

2.9.2 Compile-Time Errors vs Run-Time Errors

Basis	Compile-Time Error	Run-Time Error
When detected	During compilation (javac)	During program execution (java)
Cause	Syntax mistakes, missing semicolons, type mismatches	Logical issues that surface only when the code runs, e.g. division by zero
Example	int x = "hello"; // type mismatch	int r = 10 / 0; // ArithmeticException
Also known as	Syntax Error	Exception

2.10 User Input to Programs

Java provides several ways to accept input from the user at runtime. The most commonly used class is Scanner, available in the java.util package. BufferedReader is another option for reading input, typically used for reading strings efficiently.

Using the Scanner class:

```
import java.util.Scanner;

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

```
        System.out.print("Enter your name: ");
        String name = sc.nextLine();

        System.out.print("Enter your age: ");
        int age = sc.nextInt();

        System.out.print("Enter your CGPA: ");
        double cgpa = sc.nextDouble();

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("CGPA: " + cgpa);

        sc.close();
    }
}
```

Using BufferedReader:

```
import java.io.BufferedReader; import
java.io.InputStreamReader; import
java.io.IOException;   public class
BufferedReaderDemo {     public static void
main(String[] args) throws IOException {
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
System.out.print("Enter a line of text: ");
String line = br.readLine();
System.out.println("You entered: " + line);    } }
```

Scanner Method	Purpose
nextInt()	Reads an int value
nextLong()	Reads a long value
nextFloat()	Reads a float value
nextDouble()	Reads a double value
next()	Reads a single word (String, no spaces)
nextLine()	Reads an entire line including spaces
nextBoolean()	Reads a boolean value

2.10.1 Difference Between print(), println(), and printf()

Method	Behaviour
System.out.print()	Prints the given text without moving the cursor to a new line.
System.out.println()	Prints the given text and moves the cursor to a new line afterwards.
System.out.printf()	Prints formatted output using format specifiers such as %d, %s, %f, %c.
public class PrintMethodsDemo { public static void main(String[] args) { System.out.print("Hello "); System.out.print("World"); // same line System.out.println(); // move to new line System.out.println("Java"); // new line after System.out.printf("Score: %d%n", 95); } } /* Output: Hello World Java Score: 95 */	

2.10.2 Complete Annotated Program (Putting It All Together)

The following program brings together documentation, imports, tokens, statements, escape sequences, comments, and user input in a single complete example:

```
/* ========================================
   Program: Student Grade Calculator
   Description: Reads student marks and prints
   the grade using an annotated structure.
   ======================================== */

import java.util.Scanner;                // import statement

public class GradeCalculator {             // class declaration

    public static void main(String[] args) {  // main method - entry point

        Scanner sc = new Scanner(System.in);   // object creation
        int marks;                              // variable declaration

        System.out.print("Enter marks (0-100): \t"); // escape sequence \t
        marks = sc.nextInt();                    // input statement

        char grade;                              // local variable
```

```
        if (marks >= 90) {                          // control statement
            grade = 'A';
        } else if (marks >= 75) {
            grade = 'B';
        } else if (marks >= 50) {
            grade = 'C';
        } else {
            grade = 'F';
        }

        System.out.println("Grade: " + grade);   // output statement
        sc.close();
    }
}
```

2.11 Java Keywords

Keywords are reserved words that have a predefined meaning in Java and cannot be used as identifiers (names of variables, classes, or methods). Java has 53 reserved keywords, including two reserved literals (true, false) and the null literal.

Category	Keywords
Access Modifiers	public, private, protected
Class-related	class, interface, extends, implements, package, import
Primitive Types	byte, short, int, long, float, double, char, boolean
Control Flow	if, else, switch, case, default, for, while, do, break, continue
Exception Handling	try, catch, finally, throw, throws
Modifiers	static, final, abstract, synchronized, transient, volatile, native, strictfp
Others	new, this, super, return, void, instanceof, enum, assert, const*, goto*

(* const and goto are reserved but not used in Java.)

2.12 Rules for Naming Identifiers

- An identifier can contain letters, digits, underscore (_), and dollar sign ($).
- An identifier must not start with a digit.
- Java is case-sensitive: 'value' and 'Value' are different identifiers.
- Reserved keywords cannot be used as identifiers.
- No spaces or special symbols (like @, #, %) are allowed.

Valid Identifiers	Invalid Identifiers	Reason for Invalid
studentName	2ndYear	Cannot start with a digit
_count	class	'class' is a reserved keyword
total$Marks	my name	Spaces are not allowed
empId1	int	'int' is a reserved keyword

Review Questions

Short Answer Questions:

- What is a token in Java? Explain the different types of tokens with examples.
- What are command line arguments? Write a program to demonstrate their use.
- List and explain any five escape sequences used in Java with examples.
- Differentiate between Scanner class and BufferedReader class for taking user input.
- What are keywords? Can they be used as identifiers? Justify your answer.

Long Answer / Programming Questions:

- Explain the general structure of a Java program with a suitable example.
- Write a Java program that accepts a student's name, roll number, and marks from the user using the Scanner class and displays them.
- Explain the rules for naming identifiers in Java with valid and invalid examples.

3. Data Types, Variables, and Operators

3.1 Introduction

Data types specify the type and size of data that a variable can hold. Java is a strongly typed language, which means every variable must be declared with a data type before it is used.

3.2 Data Types in Java

Java data types are broadly classified into two categories:

- Primitive Data Types – byte, short, int, long, float, double, char, boolean
- Non-Primitive (Reference) Data Types – Class, Array, String, Interface

Data Type	Size	Default Value	Example
byte	1 byte	0	byte b = 10;
short	2 bytes	0	short s = 1000;
int	4 bytes	0	int x = 50000;
long	8 bytes	0L	long l = 123456789L;
float	4 bytes	0.0f	float f = 3.14f;
double	8 bytes	0.0d	double d = 3.14159;
char	2 bytes	'\u0000'	char c = 'A';
boolean	1 bit	false	boolean flag = true;

```
public class DataTypesDemo {
    public static void main(String[] args) {
        byte b = 10;
        short s = 2000;
        int i = 100000;
        long l = 10000000000L;
        float f = 3.14f;
        double d = 3.14159265;
        char c = 'J';
        boolean flag = true;

        System.out.println("byte: " + b);
        System.out.println("short: " + s);
        System.out.println("int: " + i);
        System.out.println("long: " + l);
        System.out.println("float: " + f);
        System.out.println("double: " + d);
        System.out.println("char: " + c);
        System.out.println("boolean: " + flag);
    }
}
```

3.2.1 Individual Data Type Example Programs

Program: Demonstrating char data type and character arithmetic.

```
public class CharDemo {
    public static void main(String[] args) {
        char ch = 'A';
        System.out.println("Character: " + ch);
        System.out.println("ASCII value: " + (int) ch);   // char to int casting

        char next = (char) (ch + 1);       // character arithmetic
        System.out.println("Next character: " + next);
    }
}

/* Output:
Character: A
ASCII value: 65
Next character: B
*/
```

Program: Demonstrating boolean data type and its default value.

```
public class BooleanDemo {
    static boolean flag;   // default value is false

    public static void main(String[] args) {
        System.out.println("Default value: " + flag);
        boolean isEligible = (18 >= 18);
        System.out.println("Is Eligible: " + isEligible);
    }
}

/* Output:
Default value: false
Is Eligible: true
*/
```

Program: Integer overflow behaviour when a value exceeds the type's range.

```
public class OverflowDemo {
    public static void main(String[] args) {
        byte b = 127;      // maximum value of byte
        b++;                // exceeds range, wraps around
        System.out.println("Overflowed byte: " + b);

        int maxInt = Integer.MAX_VALUE;
```

```
        System.out.println("Max int: " + maxInt);
        System.out.println("Max int + 1: " + (maxInt + 1));  // overflows to
negative
    }
}

/* Output:
Overflowed byte: -128
Max int: 2147483647
Max int + 1: -2147483648
*/
```

3.3 Declaration of Variables

A variable is a named memory location used to store a value that can change during program execution. Variables must be declared before use.

```
// Syntax
dataType variableName;
dataType variableName = value;

// Examples
int age;
age = 20;
int marks = 95;
double price = 499.50;

// Multiple variable declaration
int a = 1, b = 2, c = 3;
```

3.4 Type Casting

Type casting is the process of converting a variable from one data type to another. Java supports two kinds of casting:

- Widening (Implicit) Casting: Automatic conversion of a smaller type to a larger type (byte -> short -> int -> long -> float -> double).
- Narrowing (Explicit) Casting: Manual conversion of a larger type to a smaller type, which may cause loss of data.

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

```
        // Widening casting (implicit)
        int myInt = 9;
        double myDouble = myInt;   // int -> double
        System.out.println("Widening: " + myDouble);

        // Narrowing casting (explicit)
        double d = 9.78;
        int i = (int) d;           // double -> int
        System.out.println("Narrowing: " + i);
    }
}

/* Output:
Widening: 9.0
Narrowing: 9
*/
```

3.5 Scope of Variable Identifier

Based on scope, variables in Java are classified as:

- Local Variables: Declared inside a method/block; accessible only within that method/block.
- Instance Variables: Declared inside a class but outside methods; each object has its own copy.
- Static (Class) Variables: Declared with the 'static' keyword; shared across all objects of the class.


```
class ScopeDemo {
    int instanceVar = 10;          // instance variable
    static int staticVar = 20;     // static variable

    void display() {
        int localVar = 30;          // local variable
        System.out.println("Instance: " + instanceVar);
        System.out.println("Static: " + staticVar);
        System.out.println("Local: " + localVar);
    }
}
```

3.6 Literal Constants

A literal is a fixed value that appears directly in the code without being computed. Java supports integer, floating-point, character, string, and boolean literals.

```
int decimal = 100;         // decimal literal
int octal = 0144;          // octal literal (prefix 0)
int hex = 0x64;            // hexadecimal literal (prefix 0x)
int binary = 0b1100100;    // binary literal (prefix 0b)
double dbl = 3.14;         // floating point literal
char ch = 'A';             // character literal
String str = "Java";       // string literal
boolean flag = true;       // boolean literal
```

3.7 Symbolic Constants

A symbolic constant is a variable whose value cannot be changed once assigned. In Java, symbolic constants are declared using the 'final' keyword.

```
public class ConstantDemo {
    static final double PI = 3.14159;   // symbolic constant

    public static void main(String[] args) {
        double radius = 5.0;
        double area = PI * radius * radius;
        System.out.println("Area = " + area);
        // PI = 3.14; // Error: cannot assign a value to final variable PI
    }
}
```

3.8 Formatted Output with printf() Method

The printf() method is used to display formatted output using format specifiers such as %d (integer), %f (float/double), %s (string), and %c (character).

```
public class PrintfDemo {
    public static void main(String[] args) {
        String name = "Ravi";
        int age = 20;
        double marks = 89.756;

        System.out.printf("Name: %s%n", name);
        System.out.printf("Age: %d%n", age);
        System.out.printf("Marks: %.2f%n", marks);   // 2 decimal places
        System.out.printf("%-10s|%5d%n", name, age);  // width formatting
    }
}
```

```
/* Output:
Name: Ravi
Age: 20
Marks: 89.76
Ravi      |   20
*/
```

3.9 Static Variables and Methods

The 'static' keyword indicates that a member (variable or method) belongs to the class itself rather than to any specific object. Static members are shared across all instances and can be accessed without creating an object.

```
class Counter {
    static int count = 0;   // static variable

    Counter() {
        count++;             // incremented every time an object is created
    }

    static void showCount() {   // static method
        System.out.println("Total objects created: " + count);
    }
}

public class StaticDemo {
    public static void main(String[] args) {
        new Counter();
        new Counter();
        new Counter();
        Counter.showCount();   // called using class name, no object needed
    }
}

/* Output:
Total objects created: 3
*/
```

A static block is a special block of code that is executed exactly once, when the class is first loaded into memory, and is typically used to initialize static variables.

```
class Configuration {
```

```
    static String appName;

    static {                          // static block
        appName = "Student Portal";
        System.out.println("Static block executed - class loaded");
    }
}

public class StaticBlockDemo {
    public static void main(String[] args) {
        System.out.println("App Name: " + Configuration.appName);
    }
}

/* Output:
Static block executed - class loaded
App Name: Student Portal
*/
```

3.10 Attribute Final

The 'final' keyword can be applied to variables, methods, and classes: a final variable becomes a constant, a final method cannot be overridden, and a final class cannot be inherited.

```
final class MathConstants {          // final class - cannot be extended
    final double PI = 3.14159;        // final variable - cannot be reassigned

    final void showPI() {              // final method - cannot be overridden
        System.out.println("PI = " + PI);
    }
}
```

3.10.1 Primitive vs Reference (Non-Primitive) Data Types

Basis	Primitive Data Types	Reference (Non-Primitive) Data Types
Storage	Store the actual value directly	Store the memory address (reference) of the object
Size	Fixed size depending on type	Depends on the object created
Default Value	0, 0.0, false, or '\u0000'	null

Basis	Primitive Data Types	Reference (Non-Primitive) Data Types
Examples	byte, int, char, boolean, double	String, Array, Class, Interface

3.10.2 Wrapper Classes and Autoboxing (Preview)

Every primitive data type in Java has a corresponding Wrapper class in the java.lang package, which allows primitives to be treated as objects (needed for use in collections and generics, covered in detail in later units). Autoboxing is the automatic conversion of a primitive to its wrapper object, and unboxing is the reverse.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
public class AutoboxingDemo { public static void main(String[] args) { int primitiveInt = 10; Integer wrappedInt = primitiveInt; // autoboxing: int -> Integer Integer wrapperVal = 25; int unwrapped = wrapperVal; // unboxing: Integer -> int System.out.println("Wrapped: " + wrappedInt); System.out.println("Unwrapped: " + unwrapped); } }	

3.11 Practice Programs

Program 1: Swap two numbers using a temporary variable and without using a temporary variable.

```
public class SwapDemo {
```

```
    public static void main(String[] args) {
        int a = 5, b = 10, temp;

        // using temporary variable
        temp = a;
        a = b;
        b = temp;
        System.out.println("After swap (with temp): a=" + a + ", b=" + b);

        // without using temporary variable
        a = a + b;
        b = a - b;
        a = a - b;
        System.out.println("After swap (without temp): a=" + a + ", b=" + b);
    }
}
```

Program 2: Calculate the area and circumference of a circle.

```
public class CircleDemo {
    static final double PI = 3.14159;

    public static void main(String[] args) {
        double radius = 7.0;
        double area = PI * radius * radius;
        double circumference = 2 * PI * radius;

        System.out.printf("Area = %.2f%n", area);
        System.out.printf("Circumference = %.2f%n", circumference);
    }
}
```

Program 3: Calculate Simple Interest using variables and casting.

```
public class SimpleInterestDemo {
    public static void main(String[] args) {
        double principal = 15000;
        double rate = 8.5;
        int time = 3;

        double interest = (principal * rate * time) / 100;
        System.out.println("Simple Interest = " + interest);
    }
}
```

Program 4: A simple command-line based calculator combining variables, type casting, static methods and printf() formatting.

```
import java.util.Scanner;

public class SimpleCalculator {

    static double add(double a, double b) { return a + b; }
    static double subtract(double a, double b) { return a - b; }
    static double multiply(double a, double b) { return a * b; }
    static double divide(double a, double b) { return a / b; }

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

        System.out.print("Enter first number: ");
        double num1 = sc.nextDouble();

        System.out.print("Enter second number: ");
        double num2 = sc.nextDouble();

        System.out.printf("Sum = %.2f%n", add(num1, num2));
        System.out.printf("Difference = %.2f%n", subtract(num1, num2));
        System.out.printf("Product = %.2f%n", multiply(num1, num2));
        System.out.printf("Quotient = %.2f%n", divide(num1, num2));

        sc.close();
    }
}
```

3.12 Common Errors to Avoid

- Forgetting to initialize a local variable before use (causes a compile-time error).
- Assigning a floating-point literal to an int without casting (e.g. int x = 3.14; is invalid).
- Trying to modify a variable declared as final after initialization.
- Mixing up = (assignment) with == (comparison) in expressions.
- Losing data precision when narrowing a double/long to int without realizing truncation occurs.

Review Questions

Short Answer Questions:

- What are the different data types supported in Java? Explain with examples.
- Differentiate between implicit and explicit type casting with examples.
- What is the difference between local, instance, and static variables?
- What is a literal? Explain different types of literals in Java.
- What is the purpose of the final keyword? Give examples for final variable, method, and class.

Long Answer / Programming Questions:

- Explain all primitive data types in Java along with their size and range, with a program demonstrating each.
- Write a Java program to demonstrate widening and narrowing type casting with proper output.
- Write a Java program to calculate the simple interest and compound interest for given principal, rate and time using formatted output.

4. Introduction to Operators

4.1 Introduction

An operator is a special symbol that performs a specific operation on one, two, or three operands and produces a result. Java operators are broadly classified as: Arithmetic, Assignment, Relational, Logical, Bitwise, Unary, and Ternary operators.

4.2 Precedence and Associativity of Operators

Operator precedence determines the order in which operators are evaluated in an expression with multiple operators. Associativity determines the order of evaluation when operators of the same precedence appear together (left-to-right or right-to-left).

```
int result = 10 + 5 * 2;   // * has higher precedence than +
System.out.println(result); // Output: 20  (5*2=10, then 10+10=20)

int x = 20 - 5 - 3;         // left-to-right associativity for '-'
System.out.println(x);      // Output: 12
```

4.3 Assignment Operator (=)

The assignment operator assigns the value on the right-hand side to the variable on the left-hand side. Java also supports compound assignment operators.

```
int a = 10;      // simple assignment
a += 5;          // a = a + 5  -> 15
a -= 3;          // a = a - 3  -> 12
a *= 2;          // a = a * 2  -> 24
a /= 4;          // a = a / 4  -> 6
a %= 4;          // a = a % 4  -> 2
System.out.println(a);
```

4.4 Basic Arithmetic Operators

Operator	Meaning	Example (a=10,b=3)
+	Addition	a+b = 13
-	Subtraction	a-b = 7
*	Multiplication	a*b = 30
/	Division	a/b = 3
%	Modulus (remainder)	a%b = 1

```
public class ArithmeticDemo {
    public static void main(String[] args) {
        int a = 10, b = 3;
        System.out.println("Sum: " + (a + b));
        System.out.println("Difference: " + (a - b));
        System.out.println("Product: " + (a * b));
        System.out.println("Quotient: " + (a / b));
        System.out.println("Remainder: " + (a % b));
    }
}
```

4.5 Increment (++) and Decrement (--) Operators

These are unary operators used to increase or decrease the value of a variable by 1. They can be used in prefix (++a) or postfix (a++) form.

```
public class IncrementDemo {
    public static void main(String[] args) {
        int a = 5;
        System.out.println(a++);   // Postfix: prints 5, then a becomes 6
        System.out.println(a);     // prints 6
```

```
        System.out.println(++a);   // Prefix: a becomes 7, then prints 7

        int b = 5;
        System.out.println(b--);   // Postfix: prints 5, then b becomes 4
        System.out.println(--b);   // Prefix: b becomes 3, then prints 3
    }
}
```

4.6 Ternary Operator (?:)

The ternary operator is a shorthand for the if-else statement. It takes three operands and returns one of two values based on a condition.

```
// Syntax: condition ? value_if_true : value_if_false;
public class TernaryDemo {
    public static void main(String[] args) {
        int a = 10, b = 20;
        int max = (a > b) ? a : b;
        System.out.println("Maximum = " + max);
    }
}

/* Output:
Maximum = 20
*/
```

4.7 Relational Operators

Relational operators are used to compare two values and return a boolean result (true or false): ==, !=, >, <, >=, <=.

```
public class RelationalDemo {
    public static void main(String[] args) {
        int a = 10, b = 20;
        System.out.println(a == b);   // false
        System.out.println(a != b);   // true
        System.out.println(a > b);    // false
        System.out.println(a < b);    // true
        System.out.println(a >= 10);  // true
        System.out.println(b <= 15);  // false
    }
}
```

4.8 Boolean Logical Operators

Logical operators combine multiple boolean expressions: && (Logical AND), || (Logical OR), ! (Logical NOT).

```
public class LogicalDemo {
    public static void main(String[] args) {
        int age = 25;
        boolean hasID = true;

        System.out.println(age >= 18 && hasID);   // true  (AND)
        System.out.println(age < 18 || hasID);    // true  (OR)
        System.out.println(!hasID);                // false (NOT)
    }
}
```

4.9 Bitwise Logical Operators

Bitwise operators perform operations at the bit level: & (AND), | (OR), ^ (XOR), ~ (Complement), << (Left Shift), >> (Right Shift), >>> (Unsigned Right Shift).

```
public class BitwiseDemo {
    public static void main(String[] args) {
        int a = 12;   // 0000 1100
        int b = 10;   // 0000 1010

        System.out.println(a & b);    // 8   (AND)
        System.out.println(a | b);    // 14  (OR)
        System.out.println(a ^ b);    // 6   (XOR)
        System.out.println(~a);       // -13 (Complement)
        System.out.println(a << 2);   // 48  (Left Shift)
        System.out.println(a >> 2);   // 3   (Right Shift)
    }
}
```

4.8.1 Short-Circuit Evaluation of Logical Operators

The && and || operators use short-circuit evaluation: if the result can be determined from the first operand alone, the second operand is not evaluated at all. This is useful for avoiding errors such as division by zero or null pointer exceptions.

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

```
        int a = 0;

        // Second condition is never evaluated because the first is false
        if (a != 0 && (10 / a > 1)) {
            System.out.println("Division performed");
        } else {
            System.out.println("Skipped division safely using short-circuit &&");
        }

        // Second condition is never evaluated because the first is true
        if (a == 0 || (10 / a > 1)) {
            System.out.println("Condition true using short-circuit ||");
        }
    }
}

/* Output:
Skipped division safely using short-circuit &&
Condition true using short-circuit ||
*/
```

4.9.1 The instanceof Operator

The instanceof operator is a special relational operator used to test whether an object is an instance of a specific class or subclass. It returns a boolean value and is commonly used before performing a downcast.

```
class Animal { }
class Dog extends Animal { }

public class InstanceofDemo {
    public static void main(String[] args) {
        Animal a = new Dog();
        System.out.println(a instanceof Dog);      // true
        System.out.println(a instanceof Animal);    // true

        Animal a2 = new Animal();
        System.out.println(a2 instanceof Dog);       // false
    }
}
```

4.9.2 Operator Overloading in Java

Unlike C++, Java does not allow programmers to define custom behaviour for operators (no user-defined operator overloading). The only exception built into the language is the '+' operator, which is overloaded by Java itself to perform both numeric addition and String concatenation.

```
public class PlusOverloadDemo {
    public static void main(String[] args) {
        int sum = 5 + 3;                 // numeric addition
        String text = "Java" + "OOP";     // string concatenation
        String mixed = "Total: " + 10 + 20;  // left to right: "Total: 10" + 20

        System.out.println(sum);
        System.out.println(text);
        System.out.println(mixed);
    }
}

/* Output:
8
JavaOOP
Total: 1020
*/
```

4.10 Java Operator Precedence Table

The table below lists Java operators from highest to lowest precedence, along with their associativity.

Precedence	Operator(s)	Associativity
1 (Highest)	() [] . (postfix)	Left to Right
2	++ -- (unary) + - (unary) ! ~	Right to Left
3	* / %	Left to Right
4	+ -	Left to Right
5	<< >> >>>	Left to Right
6	< <= > >= instanceof	Left to Right
7	== !=	Left to Right
8	& (bitwise AND)	Left to Right
9	^ (bitwise XOR)	Left to Right
10	| (bitwise OR)	Left to Right

Precedence	Operator(s)	Associativity
11	&&	Left to Right
12	||	Left to Right
13	?:	Right to Left
14 (Lowest)	= += -= *= /= %= etc.	Right to Left

4.11 Step-by-Step Expression Evaluation Example

Consider the expression: int result = 10 + 20 * 2 - 5 % 3;

```
public class ExpressionDemo {
    public static void main(String[] args) {
        int result = 10 + 20 * 2 - 5 % 3;
        // Step 1: 20 * 2 = 40   (highest precedence among + - * %)
        // Step 2: 5 % 3  = 2
        // Step 3: 10 + 40 - 2  (left to right)
        // Step 4: 50 - 2 = 48
        System.out.println("Result = " + result);
    }
}

/* Output:
Result = 48
*/
```

4.11.1 More Worked Expression Evaluations

Expression	Step-by-Step Evaluation	Result
20 / 4 * 2	(20/4)=5, then 5*2 (left to right, same precedence)	10
7 + 3 > 5 && 2 < 4	(7+3)=10, 10>5 is true, 2<4 is true, true&&true	true
5 == 5 || 10 / 0 > 1	5==5 is true, short-circuit skips 10/0	true
(3 + 2) * (6 - 4)	(3+2)=5, (6-4)=2, 5*2 (parentheses first)	10

4.12 Practice Programs Using Operators

Program 1: Check if a number is even or odd using the bitwise AND operator.

```
public class EvenOddBitwise {
    public static void main(String[] args) {
        int num = 15;
        if ((num & 1) == 0) {
            System.out.println(num + " is Even");
        } else {
            System.out.println(num + " is Odd");
        }
    }
}
```

Program 2: Swap two numbers using the bitwise XOR operator (no temporary variable).

```
public class SwapXOR {
    public static void main(String[] args) {
        int a = 6, b = 9;
        a = a ^ b;
        b = a ^ b;
        a = a ^ b;
        System.out.println("a = " + a + ", b = " + b);
    }
}

/* Output:
a = 9, b = 6
*/
```

Review Questions

Short Answer Questions:

- What is operator precedence? Why is it important in evaluating expressions?
- Differentiate between prefix and postfix increment operators with an example.
- Explain the ternary operator with a suitable example.
- List and explain any four bitwise operators in Java with examples.
- Differentiate between logical AND (&&) and bitwise AND (&) operators.

Long Answer / Programming Questions:

- Explain all types of operators available in Java with example programs for each category.

- Write a Java program to demonstrate the use of relational and logical operators together to validate user eligibility (age and citizenship).
- Evaluate the expression 15 + 4 * 2 - 10 / 5 step by step and verify the result using a Java program.

5. Control Statements

5.1 Introduction

Control statements are used to control the flow of execution of a Java program based on certain conditions or to repeat a block of code multiple times. They are classified into decision-making (selection) statements, iteration (looping) statements, and jump statements.

5.2 if Statement

The if statement executes a block of code only if the given condition is true.

```
public class IfDemo {
    public static void main(String[] args) {
        int num = 15;
        if (num > 0) {
            System.out.println("Positive Number");
        }
    }
}
```

5.3 Nested if Statement

An if statement placed inside another if statement is called a nested if statement, used for checking multiple conditions.

```
public class NestedIfDemo {
    public static void main(String[] args) {
        int num = 0;
        if (num >= 0) {
            if (num == 0) {
                System.out.println("Number is Zero");
            } else {
                System.out.println("Positive Number");
            }
        } else {
            System.out.println("Negative Number");
        }
```

```
    }
}
```

5.4 if-else Statement

The if-else statement executes one block of code if the condition is true and another block if it is false.

```
public class IfElseDemo {
    public static void main(String[] args) {
        int num = 10;
        if (num % 2 == 0) {
            System.out.println(num + " is Even");
        } else {
            System.out.println(num + " is Odd");
        }
    }
}
```

5.5 Ternary Operator (?:) as a Conditional Expression

As covered earlier, the ternary operator provides a compact alternative to simple if-else statements.

```
public class TernaryCondDemo {
    public static void main(String[] args) {
        int num = 7;
        String result = (num % 2 == 0) ? "Even" : "Odd";
        System.out.println(result);
    }
}
```

5.6 Switch Statement

The switch statement is a multi-way decision-making statement that tests a variable against a list of values (cases).

```
public class SwitchDemo {
    public static void main(String[] args) {
        int day = 3;
        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
```

```
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            default:
                System.out.println("Invalid Day");
        }
    }
}

/* Output:
Wednesday
*/
```

5.7 Iteration Statements – while Loop

The while loop repeats a block of statements as long as the given condition remains true. The condition is checked before executing the loop body (entry-controlled loop).

```
public class WhileDemo {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 5) {
            System.out.println("Count: " + i);
            i++;
        }
    }
}
```

5.8 do-while Loop

The do-while loop is an exit-controlled loop; the condition is checked after the loop body executes, guaranteeing at least one execution.

```
public class DoWhileDemo {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println("Count: " + i);
            i++;
        } while (i <= 5);
```

```
    }
}
```

5.9 for Loop

The for loop combines initialization, condition checking, and increment/decrement in a single line, making it ideal when the number of iterations is known.

```
public class ForDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Count: " + i);
        }
    }
}
```

5.10 Nested for Loop

A for loop inside another for loop is called a nested for loop, commonly used for pattern printing and multi-dimensional processing.

```
public class NestedForDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}

/* Output:
*
* *
* * *
*/
```

5.11 For-Each Loop (Enhanced for Loop)

The for-each loop is used to iterate over arrays and collections without using an index variable.

```
public class ForEachDemo {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        for (int num : numbers) {
            System.out.println(num);
        }
    }
}
```

5.12 Break Statement

The break statement is used to terminate a loop or switch statement immediately, transferring control to the statement following the loop/switch.

```
public class BreakDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break;   // loop terminates when i equals 5
            }
            System.out.println(i);
        }
    }
}

/* Output: 1 2 3 4 */
```

5.13 Continue Statement

The continue statement skips the current iteration of a loop and moves control to the next iteration.

```
public class ContinueDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                continue;   // skips printing when i equals 3
            }
            System.out.println(i);
        }
    }
}

/* Output: 1 2 4 5 */
```

5.14 Comparison of Looping Statements

Loop	Condition Check	Minimum Executions	Best Used When
while	Before loop body	0 (may not execute)	Number of iterations is not known in advance
do-while	After loop body	1 (always executes once)	Loop body must execute at least once
for	Before loop body	0 (may not execute)	Number of iterations is known in advance

5.15 Pattern Printing Programs Using Nested Loops

Program 1: Print a number triangle pattern.

```
public class NumberTrianglePattern {
    public static void main(String[] args) {
        int rows = 5;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print(j + " ");
            }
            System.out.println();
        }
    }
}

/* Output:
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
*/
```

Program 2: Print a pyramid pattern of stars.

```
public class PyramidPattern {
    public static void main(String[] args) {
        int rows = 4;
        for (int i = 1; i <= rows; i++) {
            for (int sp = rows; sp > i; sp--) {
```

```
                System.out.print(" ");
            }
            for (int j = 1; j <= (2 * i - 1); j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}

/* Output:
   *
  ***
 *****
*******
*/
```

Program 3: Print the multiplication table of a number using a nested for loop.

```
public class MultiplicationTable {
    public static void main(String[] args) {
        int num = 5;
        for (int i = 1; i <= 10; i++) {
            System.out.println(num + " x " + i + " = " + (num * i));
        }
    }
}
```

5.16 More Practice Programs Using Loops

Program 4: Generate the Fibonacci series up to n terms using a while loop.

```
public class FibonacciDemo {
    public static void main(String[] args) {
        int n = 10, count = 0, a = 0, b = 1;
        System.out.print("Fibonacci Series: ");
        while (count < n) {
            System.out.print(a + " ");
            int next = a + b;
            a = b;
            b = next;
            count++;
        }
    }
}
```

```
/* Output:
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34
*/
```

Program 5: Check whether a given number is prime using a for loop.

```
public class PrimeCheckDemo {
    public static void main(String[] args) {
        int num = 29;
        boolean isPrime = true;

        if (num <= 1) {
            isPrime = false;
        } else {
            for (int i = 2; i <= Math.sqrt(num); i++) {
                if (num % i == 0) {
                    isPrime = false;
                    break;
                }
            }
        }

        System.out.println(num + (isPrime ? " is Prime" : " is Not Prime"));
    }
}
```

Program 6: Find the factorial of a number using a do-while loop.

```
public class FactorialDemo {
    public static void main(String[] args) {
        int num = 5, i = 1;
        long factorial = 1;

        do {
            factorial *= i;
            i++;
        } while (i <= num);

        System.out.println("Factorial of " + num + " = " + factorial);
    }
}

/* Output:
```

```
Factorial of 5 = 120
*/
```

Program 7: Find the sum of digits of a number using a while loop.

```
public class SumOfDigitsDemo {
    public static void main(String[] args) {
        int num = 12345, sum = 0;

        while (num != 0) {
            int digit = num % 10;
            sum += digit;
            num /= 10;
        }
        System.out.println("Sum of digits = " + sum);
    }
}

/* Output:
Sum of digits = 15
*/
```

Program 8: Check whether a year is a leap year using if-else.

```
public class LeapYearDemo {
    public static void main(String[] args) {
        int year = 2024;
        boolean isLeap;

        if (year % 400 == 0) {
            isLeap = true;
        } else if (year % 100 == 0) {
            isLeap = false;
        } else if (year % 4 == 0) {
            isLeap = true;
        } else {
            isLeap = false;
        }

        System.out.println(year + (isLeap ? " is a Leap Year" : " is Not a Leap
Year"));
    }
}
```

Program 9: Reverse a given number using a while loop.

```
public class ReverseNumberDemo {
    public static void main(String[] args) {
        int num = 12345, reversed = 0;

        while (num != 0) {
            int digit = num % 10;
            reversed = reversed * 10 + digit;
            num /= 10;
        }
        System.out.println("Reversed Number: " + reversed);
    }
}

/* Output:
Reversed Number: 54321
*/
```

Program 10: Check whether a number is an Armstrong number using a for loop.

```
public class ArmstrongDemo {
    public static void main(String[] args) {
        int num = 153, sum = 0, temp = num;

        while (temp != 0) {
            int digit = temp % 10;
            sum += digit * digit * digit;    // cube of each digit
            temp /= 10;
        }

        if (sum == num) {
            System.out.println(num + " is an Armstrong number");
        } else {
            System.out.println(num + " is not an Armstrong number");
        }
    }
}

/* Output:
153 is an Armstrong number
*/
```

Program 11: Find the largest of three numbers using nested if-else.

```
public class LargestOfThree {
    public static void main(String[] args) {
        int a = 25, b = 42, c = 37;
        int largest;

        if (a >= b && a >= c) {
            largest = a;
        } else if (b >= a && b >= c) {
            largest = b;
        } else {
            largest = c;
        }

        System.out.println("Largest number is: " + largest);
    }
}

/* Output:
Largest number is: 42
*/
```

5.17 Switch Statement Using Strings (Java 7+)

Since Java 7, the switch statement can also work with String values, in addition to int, char, byte, short, and enum types.

```
public class SwitchStringDemo {
    public static void main(String[] args) {
        String grade = "B";
        switch (grade) {
            case "A":
                System.out.println("Excellent");
                break;
            case "B":
                System.out.println("Good");
                break;
            case "C":
                System.out.println("Average");
                break;
            default:
                System.out.println("Invalid Grade");
        }
    }
}
```

```
/* Output:
Good
*/
```

5.18 Labeled Break and Continue in Nested Loops

A label can be attached to a loop so that break or continue can refer to an outer loop directly from within an inner loop, instead of only affecting the innermost loop.

```
public class LabeledLoopDemo {
    public static void main(String[] args) {
        outer:
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (j == 2) {
                    continue outer;   // skips to next iteration of outer loop
                }
                System.out.println("i=" + i + ", j=" + j);
            }
        }
    }
}

/* Output:
i=1, j=1
i=2, j=1
i=3, j=1
*/
```

5.19 Enhanced Switch Expression (Java 14+ Preview)

Modern Java versions support an enhanced switch expression using the arrow (->) syntax, which avoids fall-through and can directly return a value.

```
public class EnhancedSwitchDemo {
    public static void main(String[] args) {
        int day = 3;
        String dayName = switch (day) {
            case 1 -> "Monday";
            case 2 -> "Tuesday";
            case 3 -> "Wednesday";
            default -> "Invalid Day";
        };
```

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

/* Output:
Wednesday
*/
```

5.20 Additional Pattern Program: Diamond Pattern

```
public class DiamondPattern {
    public static void main(String[] args) {
        int rows = 4;

        // upper half (pyramid)
        for (int i = 1; i <= rows; i++) {
            for (int sp = rows; sp > i; sp--) System.out.print(" ");
            for (int j = 1; j <= (2 * i - 1); j++) System.out.print("*");
            System.out.println();
        }
        // lower half (inverted pyramid)
        for (int i = rows - 1; i >= 1; i--) {
            for (int sp = rows; sp > i; sp--) System.out.print(" ");
            for (int j = 1; j <= (2 * i - 1); j++) System.out.print("*");
            System.out.println();
        }
    }
}

/* Output:
   *
  ***
 *****
*******
 *****
  ***
   *
*/
```

5.21 Common Errors to Avoid in Control Statements

- Forgetting the 'break' statement in a switch case, causing unintended fall-through to the next case.

- Using '=' instead of '==' in an if condition, which is a common logical error.
- Creating an infinite loop by forgetting to update the loop control variable.
- Misplacing a semicolon after a for/while condition, e.g. for(int i=0;i<5;i++); which creates an empty loop body.
- Confusing break (exits the loop entirely) with continue (skips only the current iteration).

Review Questions

Short Answer Questions:

- What is the difference between if-else and switch statement? When would you prefer one over the other?
- Differentiate between while and do-while loop with examples.
- What is the difference between break and continue statements?
- Explain the for-each loop with an example. What are its limitations compared to a normal for loop?
- Write the syntax of a nested for loop and explain its use with a pattern-printing example.

Long Answer / Programming Questions:

- Write a Java program to check whether a given number is prime or not, using appropriate control statements.
- Write a Java program to print the Fibonacci series up to n terms using a while loop.
- Write a Java program to print a pyramid pattern of stars using nested for loops.
- Write a menu-driven Java program using switch-case to perform basic arithmetic operations (add, subtract, multiply, divide) based on user choice.

```
Note: All programs shown above are complete, compilable Java programs. Save each
file with the same name as the public class (e.g. HelloWorld.java), compile with
'javac FileName.java', and run with 'java FileName'.
```


Unit I – Glossary of Important Terms

Term	Meaning
Class	A blueprint/template that defines the properties and behaviours of objects.
Object	An instance of a class with its own state and identity.

Term	Meaning
Encapsulation	Binding data and methods together and restricting direct access to data.
Abstraction	Hiding implementation details and exposing only essential features.
Inheritance	Acquiring properties and behaviour of one class into another using 'extends'.
Polymorphism	Ability of an object/method to take many forms (overloading/overriding).
Token	The smallest individual unit in a program (keyword, identifier, literal, operator, separator).
Keyword	A reserved word with a predefined meaning in Java.
Identifier	A user-defined name for a variable, method, or class.
Literal	A fixed value written directly in source code.
Variable	A named memory location that can hold a value which may change.
Type Casting	Converting a value from one data type to another.
Operator	A symbol that performs an operation on one or more operands.
Operand	A value or variable on which an operator acts.
Control Statement	A statement that alters the normal sequential flow of execution.
Loop	A control structure that repeats a block of statements multiple times.
JVM	Java Virtual Machine - executes compiled Java bytecode.
Bytecode	Platform-independent intermediate code (.class file) produced by the Java compiler.

Unit I – Additional Practice Problems (For Self Practice)

Attempt to write, compile, and run the following Java programs on your own using the concepts covered in this unit, before checking the review question solutions.

- Write a Java program to find the sum of the first n natural numbers using a for loop.
- Write a Java program to check whether a given character is a vowel or a consonant.
- Write a Java program to convert a temperature from Celsius to Fahrenheit using variables and formatted output.
- Write a Java program to display the multiplication tables of numbers from 1 to 10 using nested for loops.
- Write a Java program that demonstrates the use of all arithmetic and relational operators on two user-input numbers.
- Write a Java program to count the number of digits in a given number using a while loop.
- Write a Java program to demonstrate constructor overloading for a class 'Rectangle' that computes area and perimeter.
- Write a Java program using switch-case to display the number of days in a given month.
- Write a Java program to demonstrate the difference between static and instance variables using a class 'Employee'.
- Write a Java program to print all Armstrong numbers between 1 and 1000 using nested loops.

Unit I – Summary

This unit introduced the fundamentals of Object Oriented Programming and the Java language. It began with the core OOP principles - class, object, encapsulation, abstraction, inheritance, and polymorphism - which form the foundation for designing Java applications. It then covered the structure of a Java program, its tokens, comments, and command-line/user input mechanisms. The unit further explained Java's data types, variable declaration, type casting, scope, constants, and formatted output, followed by a detailed study of arithmetic, relational, logical, and bitwise operators along with operator precedence. Finally, it covered all control statements in Java - conditional (if, if-else, switch) and iterative (while, do-while, for, for-each) - along with jump statements (break, continue) that control the flow of program execution.

Key Takeaways

- OOP models real-world entities as objects that combine data and behaviour.
- Every Java program must have at least one class and a main() method as its entry point.
- Java is strongly and statically typed - every variable has a fixed data type known at compile time.

- Operators in Java follow a strict precedence and associativity that determines expression evaluation order.
- Control statements (selection, iteration, and jump) allow a program to make decisions and repeat tasks efficiently.

Unit I – Model Question Bank

Part A: Two-Mark Questions

- Define class and object.
- What is encapsulation?
- List any four primitive data types in Java.
- What is type casting?
- Define an operator. Give two examples.
- What is the use of the final keyword?
- What is a token in Java?
- Differentiate between = and == operators.
- What is the purpose of the break statement?
- Define a loop. Name the three types of loops in Java.

Part B: Five-Mark Questions

- Explain the four main principles of OOP with examples.
- Explain the different data types in Java with their sizes and ranges.
- Explain any five operators in Java with example programs.
- Differentiate between while, do-while, and for loops with examples.
- Explain the switch statement with a suitable example program.

Part C: Ten-Mark Questions

- Explain all the OOP principles in detail with a complete Java program demonstrating each principle.
- Explain the structure of a Java program in detail, including tokens, comments, and command line arguments, with suitable examples.68
- Explain all types of operators available in Java in detail, along with the operator precedence table and example programs.

- Explain all control statements available in Java (selection, iteration, and jump statements) with suitable example programs for each.
- Write Java programs to: (a) check if a number is prime, (b) generate the Fibonacci series, and (c) print a pyramid pattern using nested loops.

UNIT II

Classes, Objects and Methods

23CSE232 — Object Oriented Programming through Java

TOPIC 1: CLASSES AND OBJECTS

A class in Java is a template that defines the data (fields) and the actions (methods) that objects created from the class can perform. An object is a runtime instance of a class, having its own copy of instance variables and access to methods defined in the class. Together, classes and objects form the core mechanism of object-oriented programming — encapsulating data and behaviour into a single logical unit.

1.1 Introduction to Classes and Objects

In real life, we deal with objects such as a car, a book, or a student. Every such object has certain characteristics (state) and certain actions it can perform (behaviour). For example, a Student object has a name, roll number and marks (state), and can perform actions such as calculating percentage or displaying details (behaviour). Object-oriented programming models real-world entities in this same way using classes and objects.

A class is simply a logical construct; it does not occupy any memory until an object is created from it. When an object is created, memory is allocated for the instance variables defined in the class, and the object can then access the methods defined in that class. Multiple objects can be created from a single class, and each object maintains its own independent copy of instance variables, although all objects share the same method code.

The general structure of a Java class consists of the class declaration, fields (also called instance variables or data members), constructors, and methods. This is illustrated in the syntax below.

Syntax:

```
class ClassName {
// fields (instance variables)
dataType field1;
dataType field2;
// constructor
ClassName(parameters) {
// initialization code
}
// methods
returnType methodName(parameters) {
// method body
}
}
```

Example Program:

The following program demonstrates a simple class 'Student' with fields and a method, and shows how an object is created and used in the main method.

```
class Student {
String name;
int rollNo;
int marks;
void display() {
System.out.println("Name: " + name);
System.out.println("Roll No: " + rollNo);
System.out.println("Marks: " + marks);
}
}
public class Main {
public static void main(String[] args) {
Student s1 = new Student();   // object creation
s1.name = "Ramesh";
s1.rollNo = 101;
s1.marks = 89;
s1.display();
}
}
```

Output:

```
Name: Ramesh
Roll No: 101
Marks: 89
```

Key Points to Remember:

• A class is a logical/user-defined data type; an object is a physical, run-time entity created from a class.

• A class occupies no memory until an object is instantiated from it using the 'new' keyword.

• Each object has its own copy of instance variables but shares the same method code with other objects of the class.

• Objects communicate with each other by invoking each other's methods, similar to real-world objects interacting.

• The process of creating an object from a class is called instantiation.

1.2 Class Declaration and Modifiers

A class is declared using the keyword 'class' followed by the class name and a class body enclosed in curly braces. The class name should follow Java identifier naming rules and, by convention, begins with an uppercase letter.

Java allows access modifiers and non-access modifiers to be applied to a class declaration to control its visibility and behaviour:

public — the class is visible to all other classes everywhere.

default (no modifier) — the class is visible only within its own package.

final — the class cannot be extended (no subclass can be created from it).

abstract — the class cannot be instantiated directly; it may contain abstract methods that must be implemented by subclasses.

strictfp — restricts floating point calculations to ensure portability across platforms.

Syntax:

```
[access_modifier] [non_access_modifier] class ClassName {
// class body
}
// Examples:
public class Employee { }
final class Constants { }
abstract class Shape { }
```

Example Program:

The following example shows a public class declaration with fields and a method.

```
public class Employee {
String empName;
double salary;
void showDetails() {
System.out.println(empName + " earns Rs. " + salary);
}
public static void main(String[] args) {
Employee e = new Employee();
e.empName = "Priya";
e.salary = 45000.0;
e.showDetails();
}
```

```
}
```

Output:

```
Priya earns Rs. 45000.0
```

Additional Example:

This program demonstrates an abstract class, which cannot be instantiated directly but can be extended by a subclass that implements its abstract method.

```
abstract class Shape {
abstract double area();     // abstract method - no body
void describe() {
System.out.println("Area = " + area());
}
}
class Square extends Shape {
double side;
Square(double s) { side = s; }
double area() {              // must implement abstract method
return side * side;
}
}
public class Main {
public static void main(String[] args) {
// Shape s = new Shape();   // ERROR: cannot instantiate abstract class
Shape sq = new Square(5);
sq.describe();
}
}
```

Output:

```
Area = 25.0
```

Key Points to Remember:

• A top-level class can only be declared 'public' or 'default' (package-private) — not 'private' or 'protected'.

• A 'final' class prevents inheritance and is often used for security or to maintain immutability, e.g., java.lang.String.

• An 'abstract' class may contain both implemented and unimplemented (abstract) methods and cannot be instantiated using 'new'.

• Only one public top-level class is allowed per .java source file, and the file name must match that class name.

• Modifiers can be combined logically, e.g., 'public final class' can be declared but 'final' and 'abstract' cannot be combined on the same class.

1.3 Class Members

The members of a class are broadly classified into two categories: (1) Data members (instance variables/fields) that hold the state of an object, and (2) Member methods (or member functions) that define the behaviour of the class.

Instance variables are declared inside the class but outside any method, constructor or block. Each object created from the class gets its own separate copy of instance variables, so changes made to one object's variables do not affect another object's variables.

In addition to instance variables and instance methods, a class can also have static members (class variables and class methods) that belong to the class itself rather than to any individual object, and are shared by all objects of the class.

A class can also contain constructors, blocks (static and instance initializer blocks), and nested classes as its members.

Example Program:

This program shows both instance members and a static member of a class.

```
class Counter {
int id;                 // instance variable
static int count = 0;   // static (class) variable
Counter() {
count++;
id = count;
}
void display() {
System.out.println("Object id: " + id);
}
}
public class Main {
public static void main(String[] args) {
Counter c1 = new Counter();
Counter c2 = new Counter();
```

```
Counter c3 = new Counter();
c1.display();
c2.display();
c3.display();
System.out.println("Total objects created: " + Counter.count);
}
}
```

Output:

```
Object id: 1
Object id: 2
Object id: 3
Total objects created: 3
```

Additional Example:

This program further illustrates how each object maintains an independent copy of instance variables while sharing the class definition.

```
class Bank {
String owner;
double balance;
}
public class Main {
public static void main(String[] args) {
Bank acc1 = new Bank();
acc1.owner = "Arjun";
acc1.balance = 1000;
Bank acc2 = new Bank();
acc2.owner = "Meena";
acc2.balance = 2500;
System.out.println(acc1.owner + " balance: " + acc1.balance);
System.out.println(acc2.owner + " balance: " + acc2.balance);
}
}
```

Output:

```
Arjun balance: 1000.0
Meena balance: 2500.0
```

Key Points to Remember:

- Instance variables represent the state of an object and are created afresh for every new object.
- Static variables belong to the class as a whole and are shared across all instances of that class.
- Instance methods operate on instance data and require an object to be invoked.
- Static methods operate at the class level, do not require an object, and cannot access instance members directly.
- A class may also include instance/static initializer blocks that run when an object is created or when the class is loaded.

1.4 Declaration of Class Objects

Creating an object of a class involves two steps: declaration and instantiation. Declaration creates a reference variable of the class type, and instantiation uses the 'new' keyword to allocate memory on the heap for the object and invoke a constructor to initialize it.

The general form is 'ClassName referenceVariable = new ClassName(arguments);'. Here, 'ClassName' on the left specifies the type of the reference, and 'new ClassName(arguments)' creates the actual object and returns a reference to it, which is stored in the reference variable.

It is important to note that in Java, object variables are reference variables — they do not hold the object itself but hold the memory address (reference) of the object stored in heap memory.

Syntax:

```
ClassName obj;                  // declaration
obj = new ClassName();           // instantiation
// OR in a single statement
ClassName obj = new ClassName();
```

Example Program:

This program demonstrates declaring and instantiating multiple objects of the same class.

```
class Rectangle {
int length, width;
int area() {
return length * width;
}
}
public class Main {
public static void main(String[] args) {
Rectangle r1;              // declaration
```

```
r1 = new Rectangle();      // instantiation
r1.length = 10;
r1.width = 5;
Rectangle r2 = new Rectangle();  // declaration + instantiation
r2.length = 8;
r2.width = 6;
System.out.println("Area of r1: " + r1.area());
System.out.println("Area of r2: " + r2.area());
}
}
```

Output:

```
Area of r1: 50
Area of r2: 48
```

Additional Example:

This program demonstrates creating an array of objects, where each array element is a separate reference to its own Student object.

```
class Student {
String name;
int marks;
Student(String n, int m) {
name = n;
marks = m;
}
}
public class Main {
public static void main(String[] args) {
Student[] students = new Student[3];   // array of references
students[0] = new Student("Asha", 88);
students[1] = new Student("Kiran", 76);
students[2] = new Student("Ravi", 95);
for (Student s : students) {
System.out.println(s.name + " scored " + s.marks);
}
}
}
```

Output:

```
Asha scored 88
```

```
Kiran scored 76
Ravi scored 95
```

Key Points to Remember:

• The 'new' operator allocates memory for the object dynamically on the heap and returns its reference.

• A reference variable declared but not instantiated holds the value 'null' and does not point to any object.

• Attempting to access members through a null reference results in a NullPointerException at runtime.

• An array of objects can also be created, where each element is itself a separate reference to an object.

• The default constructor is invoked automatically if no other constructor is defined by the programmer.

1.5 Assigning One Object to Another

When one object reference is assigned to another using the '=' operator, Java does not create a new copy of the object. Instead, both reference variables point to the same object in heap memory. This is known as a shallow (reference) assignment.

As a consequence, any change made to the object's fields through one reference variable is reflected when the object is accessed through the other reference variable, since both refer to the same memory location.

To create an independent copy of an object (a separate object with the same field values), programmers typically create a new object explicitly and copy each field value, or implement the Cloneable interface and override the clone() method.

Example Program:

This program illustrates that assigning one object reference to another does not create a new object — both variables refer to the same object.

```
class Box {
int width;
}
public class Main {
```

```
public static void main(String[] args) {
Box b1 = new Box();
b1.width = 20;
Box b2 = b1;      // b2 now refers to the SAME object as b1
b2.width = 50;    // modifies the same object
System.out.println("b1.width = " + b1.width);
System.out.println("b2.width = " + b2.width);
}
}
```

Output:

```
b1.width = 50
b2.width = 50
```

Key Points to Remember:

• Object assignment copies the reference (address), not the actual data — this is called a shallow assignment.

• Two reference variables that refer to the same object are said to be 'aliases' of each other.

• To create an independent duplicate of an object, use the clone() method (deep or shallow copy) or manually copy each field.

• The '==' operator compares object references (addresses), while the equals() method (if overridden) compares object content.

• Reassigning one reference variable to a new object does not affect the object referred to by the other variable.

1.6 Access Control for Class Members

Access control (also called encapsulation support) determines which parts of a program can access the members of a class. Java provides four levels of access control that can be applied to fields, methods and constructors:

private — accessible only within the same class. This is the most restrictive level and is used to hide implementation details.

default (no modifier, also called package-private) — accessible only within classes of the same package.

protected — accessible within the same package, and also in subclasses located in other packages.

public — accessible from any class in any package. This is the least restrictive level.

Proper use of access control is central to encapsulation — a key principle of OOP — since it hides the internal representation of an object and exposes only what is necessary through public methods.

Syntax:

```
class Demo {
private int a;      // accessible only within Demo
int b;              // default - accessible within package
protected int c;    // accessible in package + subclasses
public int d;       // accessible everywhere
}
```

Example Program:

This program demonstrates the use of a private field that is accessed only through public methods of the same class.

```
class Account {
private double balance;   // private - hidden from outside
void deposit(double amount) {
balance += amount;
}
double getBalance() {
return balance;
}
}
public class Main {
public static void main(String[] args) {
Account a = new Account();
a.deposit(5000);
// a.balance = 10000;   // ERROR: balance has private access
System.out.println("Balance: " + a.getBalance());
}
}
```

Output:

```
Balance: 5000.0
```

Additional Example:

This program shows default (package-private) and protected access using two classes in the same file to illustrate visibility within a package.

```
class Machine {
int power;              // default access - visible within package
protected String type;  // protected - visible in package & subclasses
Machine(int p, String t) {
power = p;
type = t;
}
}
class Generator extends Machine {
Generator(int p, String t) {
super(p, t);
}
void show() {
System.out.println("Type: " + type + ", Power: " + power + " KW");
}
}
public class Main {
public static void main(String[] args) {
Generator g = new Generator(500, "Diesel");
g.show();
}
}
```

Output:

```
Type: Diesel, Power: 500 KW
```

Key Points to Remember:

- private members provide the strongest encapsulation and are hidden from all other classes, including subclasses.
- default (package-private) members are visible only to classes within the same package.
- protected members extend visibility to subclasses even if they reside in a different package.
- public members form the class's exposed interface, visible from anywhere in the application.
- Good OOP design typically keeps fields private and exposes controlled access through public getter/setter methods.

1.7 Accessing Private Members of a Class

Private members of a class cannot be accessed directly from outside the class — not even by objects of that class in another class. Any attempt to access a private field or call a private method from outside the declaring class results in a compile-time error.

To allow controlled access to private data from outside the class, the standard practice is to provide public getter and setter methods. A getter method returns the value of a private field, and a setter method allows the value to be modified, typically after performing validation. This approach preserves encapsulation while still allowing controlled interaction with the object's state.

Example Program:

This program shows getter and setter methods used to access and validate a private field from outside the class.

```
class Person {
private int age;   // private field
// setter with validation
void setAge(int age) {
if (age > 0)
this.age = age;
else
System.out.println("Invalid age!");
}
// getter
int getAge() {
return age;
}
}
public class Main {
public static void main(String[] args) {
Person p = new Person();
p.setAge(25);
System.out.println("Age: " + p.getAge());
p.setAge(-5);
}
}
```

Output:

```
Age: 25
Invalid age!
```

Key Points to Remember:

- Getter and setter methods provide a controlled, validated way to read and modify private data.

• Validation logic (e.g., range checks) can be embedded inside setter methods to maintain object integrity.

• Read-only behaviour can be achieved by providing only a getter method and no setter for a field.

• Encapsulating fields as private and exposing them via methods allows internal implementation to change without affecting external code.

• This technique is fundamental to the JavaBeans convention widely used in enterprise Java applications.

1.8 Constructor Methods for a Class

A constructor is a special method that is automatically invoked at the time an object is created, and is used to initialize the instance variables of the object. A constructor has the same name as the class and has no return type — not even 'void'.

If a class does not define any constructor, Java automatically provides a default constructor that initializes fields with default values (0, null, false, etc.). However, once any constructor is explicitly defined, the default constructor is no longer provided automatically.

Constructors can accept parameters, allowing an object to be initialized with specific values at the time of creation. A constructor with parameters is called a parameterized constructor, while one with no parameters is called a default (no-argument) constructor.

Syntax:

```
class ClassName {
ClassName() {                 // no-argument constructor
// default initialization
}
ClassName(dataType param1, ...) {   // parameterized constructor
// initialization using parameters
}
}
```

Example Program:

This program demonstrates both a no-argument constructor and a parameterized constructor.

```
class Book {
String title;
double price;
Book() {                       // no-argument constructor
```

```
title = "Unknown";
price = 0.0;
}
Book(String t, double p) {      // parameterized constructor
title = t;
price = p;
}
void display() {
System.out.println(title + " - Rs." + price);
}
}
public class Main {
public static void main(String[] args) {
Book b1 = new Book();
Book b2 = new Book("Java Basics", 350.0);
b1.display();
b2.display();
}
}
```

Output:

```
Unknown - Rs.0.0
Java Basics - Rs.350.0
```

Additional Example:

This program shows a private constructor used to restrict object creation from outside the class, allowing only a static method within the class to create the object (the Singleton pattern).

```
class Singleton {
private static Singleton instance;
private int value;
private Singleton() {     // private constructor
value = 100;
}
static Singleton getInstance() {
if (instance == null)
instance = new Singleton();
return instance;
}
int getValue() { return value; }
}
public class Main {
public static void main(String[] args) {
```

```
Singleton s1 = Singleton.getInstance();
Singleton s2 = Singleton.getInstance();
System.out.println("s1 == s2 : " + (s1 == s2));
System.out.println("Value: " + s1.getValue());
}
}
```

Output:

```
s1 == s2 : true
Value: 100
```

Key Points to Remember:

- A constructor has the same name as its class and no return type, not even void.
- The compiler supplies a no-argument default constructor only if the class defines no constructor at all.
- Constructors are invoked automatically and exactly once per object, immediately after memory allocation.
- Constructors can be overloaded, made private (to restrict object creation, e.g., in Singleton design), or chained using 'this()'.
- Unlike normal methods, constructors cannot be declared 'static', 'final', or 'abstract'.

1.9 Overloaded Constructor Methods

Just as regular methods can be overloaded, a class can have multiple constructors with the same name but different parameter lists (different number, type or order of parameters). This is called constructor overloading.

Constructor overloading allows objects of the same class to be initialized in different ways depending on the information available at the time of object creation, providing flexibility to the programmer using the class.

The Java compiler determines which constructor to invoke based on the number and types of arguments passed during object creation — this is an example of compile-time (static) polymorphism.

Example Program:

This program shows a class with three overloaded constructors used to create Box objects in different ways.

```
class Box {
double length, width, height;
Box() {                                    // no-arg
length = width = height = 0;
}
Box(double side) {                          // cube
length = width = height = side;
}
Box(double l, double w, double h) {         // full dimensions
length = l; width = w; height = h;
}
double volume() {
return length * width * height;
}
}
public class Main {
public static void main(String[] args) {
Box b1 = new Box();
Box b2 = new Box(4);
Box b3 = new Box(2, 3, 4);
System.out.println("Volume b1: " + b1.volume());
System.out.println("Volume b2: " + b2.volume());
System.out.println("Volume b3: " + b3.volume());
}
}
```

Output:

```
Volume b1: 0.0
Volume b2: 64.0
Volume b3: 24.0
```

Key Points to Remember:

- Overloaded constructors differ in the number, type, or order of their parameters.
- They provide multiple convenient ways to initialize objects depending on the data available at creation time.
- The correct constructor to invoke is decided by the compiler at compile time based on argument types (static binding).
- Constructor overloading is often combined with 'this()' chaining to avoid duplicating initialization code.

- This is a direct application of compile-time polymorphism within the context of object creation.

Comparison: Constructor vs. Method

Basis	Constructor	Method
Name	Same as the class name	Any valid identifier (usually a verb)
Return type	No return type, not even void	Must have a return type (or void)
Invocation	Invoked automatically when an object is created using 'new'	Invoked explicitly by the programmer using dot notation
Purpose	Initializes the state of a newly created object	Defines a reusable unit of behaviour/computation
Inheritance	Not inherited by subclasses	Inherited by subclasses (unless private/static/final)
Modifiers	Cannot be static, final or abstract	Can be static, final, abstract, synchronized, etc.

1.10 Nested Classes

A class defined within another class is called a nested class. The class that contains the nested class is called the outer (enclosing) class. Nested classes are used to logically group classes that are used in only one place, increase encapsulation, and lead to more readable and maintainable code.

Nested classes are categorized as: (1) Static nested classes — declared with the 'static' modifier, they behave like a static member of the outer class and can be instantiated without an instance of the outer class. (2) Inner classes (non-static) — associated with an instance of the outer class and can directly access the instance members of the outer class.

Inner classes are further categorized into member inner classes, local inner classes (defined inside a method) and anonymous inner classes (without a name, typically used to override a method of a class/interface on the fly).

Syntax:

```
class Outer {
static class StaticNested { }   // static nested class
class Inner { }                  // non-static inner class
}
// Instantiation
Outer.StaticNested sn = new Outer.StaticNested();
Outer outer = new Outer();
```

```
Outer.Inner in = outer.new Inner();
```

Example Program:

This program shows a static nested class and a non-static inner class within an outer class.

```
class Outer {
int outerData = 10;
static class StaticNested {
void show() {
System.out.println("Inside static nested class");
}
}
class Inner {
void show() {
System.out.println("Outer data = " + outerData);
}
}
}
public class Main {
public static void main(String[] args) {
Outer.StaticNested sn = new Outer.StaticNested();
sn.show();
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
inner.show();
}
}
```

Output:

```
Inside static nested class
Outer data = 10
```

Additional Example:

This program demonstrates an anonymous inner class that provides a one-time implementation of an interface without creating a separately named class.

```
interface Greeting {
void sayHello();
}
public class Main {
public static void main(String[] args) {
```

```
Greeting g = new Greeting() {     // anonymous inner class
public void sayHello() {
System.out.println("Hello from an anonymous class!");
}
};
g.sayHello();
}
}
```

Output:

```
Hello from an anonymous class!
```

Key Points to Remember:

- Nested classes improve encapsulation by hiding helper classes that are relevant to only one enclosing class.
- A static nested class does not have access to the instance members of the outer class directly.
- A non-static inner class holds an implicit reference to its enclosing object and can access all its members, including private ones.
- Local inner classes are defined within a method body and are visible only within that method.
- Anonymous inner classes have no name and are typically used to provide a one-time implementation of an interface or abstract class.

1.11 Final Class and Methods

The keyword 'final' can be applied to classes, methods and variables to restrict how they can be used further:

final class — a class declared as final cannot be extended (subclassed). This is used when the design of a class should not be altered, e.g. the String class in Java is final.

final method — a method declared as final cannot be overridden by a subclass. This is used to prevent a subclass from changing the essential behaviour of a method.

final variable — a variable declared as final can be assigned a value only once; once initialized, its value cannot be changed. A final variable behaves like a constant.

Syntax:

```
final class ClassName { }        // cannot be extended
class ClassName {
final void methodName() { }  // cannot be overridden
final int CONSTANT = 100;    // cannot be reassigned
}
```

Example Program:

This program demonstrates a final variable and a final method.

```
class Vehicle {
final int wheels = 4;   // final variable
final void start() {    // final method - cannot be overridden
System.out.println("Vehicle is starting...");
}
}
public class Main {
public static void main(String[] args) {
Vehicle v = new Vehicle();
v.start();
System.out.println("Number of wheels: " + v.wheels);
// v.wheels = 6;   // ERROR: cannot assign a value to final variable
}
}
```

Output:

```
Vehicle is starting...
Number of wheels: 4
```

Additional Example:

This program shows an attempt to override a final method, which results in a compile-time error (shown here as a comment since it would prevent compilation).

```
class Parent {
final void display() {
System.out.println("This is a final method in Parent");
}
}
class Child extends Parent {
// void display() {              // ERROR: cannot override final method
//     System.out.println("Trying to override");
// }
```

```
void newMethod() {
System.out.println("Child has its own new method");
}
}
public class Main {
public static void main(String[] args) {
Child c = new Child();
c.display();      // inherited final method
c.newMethod();
}
}
```

Output:

```
This is a final method in Parent
Child has its own new method
```

Key Points to Remember:

• 'final' applied to a class prevents any further subclassing, preserving the exact behaviour of the class.

• 'final' applied to a method locks its implementation, preventing subclasses from overriding it.

• 'final' applied to a variable enforces single-assignment (constant) semantics, improving code safety.

• Making a class immutable typically involves declaring the class final and all its fields private and final.

• The Java Virtual Machine can perform certain compile-time optimizations on final methods since they cannot change at runtime.

1.12 Passing Arguments by Value and by Reference

In Java, all arguments are passed to methods strictly by value — this means a copy of the argument's value is passed to the method, not the original variable itself. However, the behaviour differs for primitive types and object (reference) types, which is a frequent source of confusion.

Pass by value for primitives — when a primitive type (int, char, double, etc.) is passed to a method, a copy of its value is passed. Changes made to the parameter inside the method do not affect the original variable in the calling method.

Pass by value of the reference for objects — when an object is passed to a method, a copy of the reference (memory address) is passed, not the object itself. Since this copied reference still points to

the same object, changes made to the object's fields inside the method are reflected in the original object. However, reassigning the parameter itself to a new object inside the method does not affect the original reference in the caller.

Example Program:

This program demonstrates the difference between passing a primitive and passing an object reference to a method.

```
class Data {
int value;
}
public class Main {
static void modifyPrimitive(int x) {
x = x + 10;             // does not affect the caller's variable
}
static void modifyObject(Data d) {
d.value = d.value + 10; // affects the original object
}
public static void main(String[] args) {
int num = 5;
modifyPrimitive(num);
System.out.println("num after method call: " + num);
Data obj = new Data();
obj.value = 5;
modifyObject(obj);
System.out.println("obj.value after method call: " + obj.value);
}
}
```

Output:

```
num after method call: 5
obj.value after method call: 15
```

Key Points to Remember:

- Java always uses pass-by-value; there is no true pass-by-reference for either primitives or objects.
- For primitives, a copy of the actual value is passed, so the original variable remains unaffected.
- For objects, a copy of the reference (address) is passed, so changes to the object's fields are visible to the caller.

• Reassigning an object parameter inside a method to point to a new object does not change what the caller's variable refers to.

• Understanding this distinction is essential for predicting the outcome of method calls involving objects.

1.13 Keyword 'this'

The keyword 'this' is a reference variable that refers to the current object — the object whose method or constructor is being executed. It is commonly used in the following situations:

1. To resolve naming conflicts between instance variables and parameters/local variables that have the same name (e.g., this.name = name;).

2. To invoke another constructor of the same class from within a constructor — this is called constructor chaining, and is written as 'this(arguments);' as the first statement of a constructor.

3. To pass the current object as an argument to another method or constructor.

4. To return the current object from a method, which is useful for method chaining.

Syntax:

```
class ClassName {
int x;
ClassName(int x) {
this.x = x;      // 'this' resolves the naming conflict
}
ClassName() {
this(0);          // constructor chaining
}
}
```

Example Program:

This program demonstrates 'this' used to resolve a naming conflict and for constructor chaining.

```
class Student {
String name;
int age;
Student() {
this("Not Assigned", 0);   // constructor chaining
}
Student(String name, int age) {
this.name = name;   // 'this.name' is instance variable,
```

```
this.age = age;     // 'name'/'age' are parameters
}
void display() {
System.out.println(name + " - " + age + " years");
}
}
public class Main {
public static void main(String[] args) {
Student s1 = new Student();
Student s2 = new Student("Kavya", 20);
s1.display();
s2.display();
}
}
```

Output:

```
Not Assigned - 0 years
Kavya - 20 years
```

Additional Example:

This program demonstrates method chaining by returning 'this' from a method, allowing multiple method calls to be linked in a single statement.

```
class TextBuilder {
String text = "";
TextBuilder append(String s) {
text = text + s;
return this;    // returns current object for chaining
}
void show() {
System.out.println(text);
}
}
public class Main {
public static void main(String[] args) {
TextBuilder tb = new TextBuilder();
tb.append("Java ").append("is ").append("fun!").show();
}
}
```

Output:

```
Java is fun!
```

Key Points to Remember:

• 'this' always refers to the object on which the currently executing method or constructor was invoked.

• 'this.fieldName' distinguishes an instance variable from a parameter or local variable of the same name.

• 'this(...)' used as the first statement of a constructor calls another constructor of the same class (constructor chaining).

• 'this' can be passed as an argument to another method that requires a reference to the current object.

• Returning 'this' from a method allows multiple method calls to be chained together (method chaining/fluent style).

2: METHODS

A method is a block of code that performs a specific task and is executed when it is called (invoked). Methods define the behaviour of a class and enable code reuse, modularity and better organisation of a program. Every method has a name, a return type, an optional list of parameters, and a body containing the statements to be executed.

Methods are central to two of the four pillars of object-oriented programming — abstraction (a method hides how a task is performed and exposes only what it does) and polymorphism (through overloading and overriding). Mastering how methods are declared, invoked, overloaded, and overridden is therefore essential before proceeding to inheritance and interfaces in later units.

2.1 Introduction to Methods

Methods are used to break a program into smaller, manageable, and reusable pieces of code. Instead of writing the same code repeatedly, a method can be defined once and called (invoked) as many times as required, from anywhere within the program where it is accessible.

Using methods provides several advantages: it improves code reusability, makes programs easier to read and maintain, allows a complex problem to be divided into smaller sub-problems (modularity), and facilitates debugging since errors can be isolated to specific methods.

A method can accept input values (parameters/arguments), process them, and optionally return a result to the caller using the 'return' statement.

Key Points to Remember:

- Every Java method belongs to some class and defines a specific unit of behaviour.
- Methods promote the DRY (Don't Repeat Yourself) principle by allowing reusable blocks of logic.
- A method's signature consists of its name and parameter list (not the return type).
- Method invocation transfers control to the method body; control returns to the caller once execution completes or a 'return' statement executes.
- The main() method is itself a special static method that serves as the entry point of a Java application.

2.2 Defining Methods

A method definition consists of a method header and a method body. The method header specifies the access modifier, return type, method name, and parameter list, while the method body contains the actual statements enclosed in curly braces.

If a method does not return any value, its return type is declared as 'void'. If it returns a value, the return type must match the type of value returned using the 'return' statement, and the method must return a value along every possible execution path.

Syntax:

```
accessModifier returnType methodName(parameterList) {
// method body
return value;   // only if returnType is not void
}
```

Example Program:

This program defines and calls a method that returns the sum of two integers, and another that returns nothing.

```
public class Main {
// method with return type
static int add(int a, int b) {
return a + b;
}
// method with void return type
static void greet(String name) {
System.out.println("Hello, " + name + "!");
}
```

```
public static void main(String[] args) {
greet("Anitha");
int result = add(15, 25);
System.out.println("Sum = " + result);
}
}
```

Output:

```
Hello, Anitha!
Sum = 40
```

Additional Example:

This program defines a method that returns a boolean value to check whether a number is prime.

```
public class Main {
static boolean isPrime(int n) {
if (n < 2) return false;
for (int i = 2; i <= Math.sqrt(n); i++) {
if (n % i == 0) return false;
}
return true;
}
public static void main(String[] args) {
int[] numbers = {2, 15, 17, 20, 23};
for (int n : numbers) {
System.out.println(n + " is prime: " + isPrime(n));
}
}
}
```

Output:

```
2 is prime: true
15 is prime: false
17 is prime: true
20 is prime: false
23 is prime: true
```

Key Points to Remember:

- The method header specifies modifiers, return type, method name and formal parameter list.

• Parameters listed in the method definition are called formal parameters; values supplied during a call are actual parameters (arguments).

• A 'void' method may still use a bare 'return;' statement to exit early.

• Local variables declared inside a method exist only for the duration of that method's execution.

• Methods can call other methods of the same class directly, or methods of other classes through an object reference.

2.3 Overloaded Methods

Method overloading is a feature that allows a class to have more than one method with the same name, provided their parameter lists differ in the number, type, or order of parameters. The return type alone is not sufficient to overload a method.

Method overloading is an example of compile-time polymorphism, since the compiler decides which overloaded method to invoke based on the arguments supplied at the call site, at compile time itself.

Overloading improves the readability of a program by allowing logically related operations to share the same method name, e.g., different versions of a method to add two integers, two doubles, or three integers.

Example Program:

This program demonstrates method overloading with different parameter lists.

```
public class Main {
static int add(int a, int b) {
return a + b;
}
static double add(double a, double b) {
return a + b;
}
static int add(int a, int b, int c) {
return a + b + c;
}
public static void main(String[] args) {
System.out.println("add(2,3) = " + add(2, 3));
System.out.println("add(2.5,3.5) = " + add(2.5, 3.5));
System.out.println("add(1,2,3) = " + add(1, 2, 3));
}
}
```

Output:

```
add(2,3) = 5
add(2.5,3.5) = 6.0
add(1,2,3) = 6
```

Key Points to Remember:

• Overloading is resolved entirely at compile time based on the number and types of arguments — this is compile-time (static) polymorphism.

• Changing only the return type of a method, while keeping the same name and parameters, does NOT constitute valid overloading.

• Overloading can also occur through automatic type promotion, e.g., calling add(int,int) when only add(double,double) is defined.

• Constructors can be overloaded in exactly the same way as normal methods.

• Overloading improves readability by letting semantically similar operations share one intuitive method name.

2.4 Class Objects as Parameters in Methods

Just like primitive data types, objects of a class can also be passed as arguments to a method. When an object is passed, a copy of the reference to the object is passed, allowing the method to access and modify the fields of the actual object through that reference.

Passing objects as parameters is useful when a method needs to operate on the data contained in an object, or when comparing two objects, or when one object needs to use the state of another object to compute a result.

Example Program:

This program demonstrates passing an object of class Rectangle as a parameter to a method that compares two rectangle objects.

```
class Rectangle {
int length, width;
Rectangle(int l, int w) {
length = l;
width = w;
}
int area() {
return length * width;
}
```

```
}
public class Main {
static void compare(Rectangle r1, Rectangle r2) {
if (r1.area() > r2.area())
System.out.println("First rectangle is bigger");
else if (r1.area() < r2.area())
System.out.println("Second rectangle is bigger");
else
System.out.println("Both rectangles are equal");
}
public static void main(String[] args) {
Rectangle r1 = new Rectangle(4, 5);
Rectangle r2 = new Rectangle(3, 6);
compare(r1, r2);
}
}
```

Output:

```
Second rectangle is bigger
```

Key Points to Remember:

• Passing an object to a method allows the method to read and modify the object's fields via the copied reference.

• This technique is frequently used to compare, combine, or transfer data between two or more objects.

• Arrays of objects can also be passed to methods for batch processing.

• A method can also accept 'this' (the current object) as an implicit argument-like reference.

• Care must be taken since modifications made inside the method persist even after the method returns, unlike primitive parameters.

2.5 Access Control (for Methods)

Similar to fields, methods can be declared with the four access modifiers — private, default, protected, and public — to control from where they can be invoked.

A private method can be called only from within the same class, and is often used for internal helper operations that should not be exposed outside the class. A public method forms part of the class's

interface and can be called from any other class. Default and protected methods provide intermediate levels of visibility restricted to the same package, or the same package plus subclasses, respectively.

Choosing appropriate access control for methods is an important design decision that supports encapsulation by exposing only the necessary operations of a class to the outside world while hiding internal implementation details.

Example Program:

This program shows a private helper method that is only used internally, and a public method that exposes the functionality.

```
class Calculator {
private int square(int x) {   // private helper method
return x * x;
}
public int sumOfSquares(int a, int b) {   // public method
return square(a) + square(b);
}
}
public class Main {
public static void main(String[] args) {
Calculator c = new Calculator();
System.out.println("Sum of squares: " + c.sumOfSquares(3, 4));
// c.square(5);   // ERROR: square() has private access
}
}
```

Output:

```
Sum of squares: 25
```

Additional Example:

This program shows a protected method being accessed by a subclass in the same package, illustrating intermediate access control.

```
class Base {
protected void greet() {
System.out.println("Hello from Base class");
}
}
class Derived extends Base {
void callGreet() {
```

```
greet();     // protected method accessible in subclass
}
}
public class Main {
public static void main(String[] args) {
Derived d = new Derived();
d.callGreet();
}
}
```

Output:

```
Hello from Base class
```

Key Points to Remember:

- private methods are typically used as internal helper routines not meant to be part of the public API.
- public methods define the external interface through which other classes interact with an object.
- protected methods are commonly used in class hierarchies designed for extension through inheritance.
- Overriding a method in a subclass cannot reduce its visibility (e.g., a public method cannot be overridden as private).
- Choosing the narrowest possible access level for each method is considered good encapsulation practice.

2.6 Recursive Methods

A method that calls itself, directly or indirectly, is called a recursive method, and the technique is called recursion. Recursion is a powerful alternative to iteration (loops) for solving problems that can be broken down into smaller sub-problems of the same type, such as calculating factorial, Fibonacci series, or traversing tree structures.

Every recursive method must have a base case — a condition under which the method returns a result without making a further recursive call — otherwise the recursion will continue indefinitely, eventually causing a StackOverflowError.

Each recursive call is placed on the Java call stack until the base case is reached; the results are then combined as the calls return, unwinding the stack.

Syntax:

```
returnType methodName(parameters) {
if (baseCondition) {
return baseResult;      // base case
} else {
return methodName(modifiedParameters);   // recursive call
}
}
```

Example Program:

This program calculates the factorial of a number using a recursive method.

```
public class Main {
static int factorial(int n) {
if (n == 0 || n == 1)
return 1;               // base case
else
return n * factorial(n - 1);  // recursive call
}
public static void main(String[] args) {
int num = 5;
System.out.println(num + "! = " + factorial(num));
}
}
```

Output:

```
5! = 120
```

Additional Example:

This program computes the nth Fibonacci number using recursion.

```
public class Main {
static int fibonacci(int n) {
if (n == 0) return 0;      // base case 1
if (n == 1) return 1;      // base case 2
return fibonacci(n - 1) + fibonacci(n - 2);   // recursive call
}
public static void main(String[] args) {
for (int i = 0; i < 8; i++) {
System.out.print(fibonacci(i) + " ");
}
```

```
}
}
```

Output:

```
0 1 1 2 3 5 8 13
```

Key Points to Remember:

• Every valid recursive solution must include at least one base case that terminates the recursive calls.

• Recursive solutions are often more readable for problems with a naturally recursive structure (trees, divide-and-conquer algorithms).

• Excessive recursion depth without reaching a base case leads to a StackOverflowError.

• Recursive methods generally use more memory than equivalent iterative solutions due to call-stack overhead.

• Common examples of recursion include factorial computation, Fibonacci series generation, and binary search.

2.7 Nesting of Methods

Nesting of methods refers to calling one method from within the body of another method of the same class. When a method 'A' calls method 'B', which in turn may call method 'C', this is referred to as method nesting or a call chain.

Method nesting promotes code reuse and allows complex tasks to be broken down into a hierarchy of smaller, well-defined methods, where higher-level methods coordinate the work performed by lower-level methods.

Example Program:

This program demonstrates nested method calls where the main method calls display(), which internally calls calculateArea().

```
class Circle {
double radius;
Circle(double r) {
radius = r;
}
double calculateArea() {
```

```
return Math.PI * radius * radius;
}
void display() {
double area = calculateArea();   // nested method call
System.out.println("Radius: " + radius);
System.out.println("Area: " + area);
}
}
public class Main {
public static void main(String[] args) {
Circle c = new Circle(7);
c.display();
}
}
```

Output:

```
Radius: 7.0
Area: 153.93804002589985
```

Key Points to Remember:

- Method nesting allows a task to be broken into a hierarchy of smaller, single-purpose methods.
- Each method in the chain should ideally perform one clearly-defined task (single responsibility principle).
- Nested calls can occur within the same class or across different classes via object references.
- Excessive or deeply nested method calls can make debugging harder, so meaningful naming and documentation are important.
- Method nesting is different from nested classes — it refers to the sequence of method invocations, not class structure.

2.8 Overriding Methods

Method overriding occurs when a subclass provides its own specific implementation of a method that is already defined in its superclass, with the same name, same return type (or a covariant type), and the same parameter list. Overriding is used to achieve runtime polymorphism.

Overriding differs from overloading: overriding involves a superclass–subclass relationship and identical method signatures, whereas overloading occurs within the same class using different

signatures. Overriding is resolved at runtime (dynamic method dispatch), whereas overloading is resolved at compile time.

The '@Override' annotation, though optional, is recommended when overriding a method, as it lets the compiler verify that a method is actually overriding a superclass method and catch mistakes such as typographical errors in the method signature.

A method declared as 'final', 'static', or 'private' in a superclass cannot be overridden by a subclass.

Example Program:

This program demonstrates method overriding, where the subclass Dog overrides the sound() method of superclass Animal.

```
class Animal {
void sound() {
System.out.println("Animal makes a sound");
}
}
class Dog extends Animal {
@Override
void sound() {
System.out.println("Dog barks");
}
}
public class Main {
public static void main(String[] args) {
Animal a = new Animal();
a.sound();
Animal d = new Dog();   // upcasting
d.sound();              // calls Dog's overridden method (dynamic dispatch)
}
}
```

Output:

```
Animal makes a sound
Dog barks
```

Additional Example:

This program demonstrates using the 'super' keyword inside an overriding method to call the superclass version before adding new behaviour.

```
class Employee {
void showDetails() {
System.out.println("Employee details being shown");
}
}
class Manager extends Employee {
@Override
void showDetails() {
super.showDetails();    // call superclass version first
System.out.println("Manager also handles a team");
}
}
public class Main {
public static void main(String[] args) {
Manager m = new Manager();
m.showDetails();
}
}
```

Output:

```
Employee details being shown
Manager also handles a team
```

Key Points to Remember:

• Overriding requires the same method name, same parameter list, and a compatible (same or covariant) return type as the superclass method.

• The access modifier of the overriding method cannot be more restrictive than that of the overridden method.

• Dynamic method dispatch means the JVM decides, at runtime, which overridden version to execute based on the actual object type.

• The 'super' keyword can be used inside an overriding method to explicitly call the superclass's version of the method.

• Overriding is the mechanism through which runtime polymorphism is achieved in Java.

Comparison: Method Overloading vs. Method Overriding

Basis	Method Overloading	Method Overriding
Definition	Multiple methods with the same name but different parameter lists in the same class	Subclass redefines a method with the same signature as in its superclass
Inheritance involved?	No	Yes (requires a superclass-subclass relationship)
Parameter list	Must differ	Must be exactly the same
Return type	Can be same or different	Must be same or a covariant type
Binding	Compile-time (static) binding	Runtime (dynamic) binding
Polymorphism type	Compile-time polymorphism	Runtime polymorphism
Access modifier	Can be changed freely	Cannot be more restrictive than the overridden method

2.9 Attributes Final and Static

final attribute — When applied to a field, 'final' makes that field a constant whose value must be assigned exactly once, either at declaration or inside a constructor, and cannot be changed afterwards. By convention, final constants are named using uppercase letters with underscores, e.g., MAX_VALUE.

static attribute — When applied to a field, 'static' makes it a class variable rather than an instance variable. A static field is shared by all objects of the class; it is created once when the class is loaded into memory and is accessed using the class name (ClassName.fieldName), although it can also be accessed through an object reference.

static methods — Similarly, a method declared 'static' belongs to the class rather than to any object, can be called without creating an object of the class, and can only directly access other static members of the class (it cannot directly access instance variables or call non-static methods without an object reference).

A field can also be declared both 'static' and 'final' together to create a class-wide constant, e.g., 'static final double PI = 3.14159;'.

Syntax:

```
class ClassName {
static int count;                // static (class) variable
```

```
final int id;                     // final (constant) instance field
static final double PI = 3.14159; // static final constant
static void staticMethod() { }    // static method
}
```

Example Program:

This program demonstrates static and final fields, and a static method used to track the number of objects created.

```
class Product {
static int totalProducts = 0;     // static variable - shared
static final double TAX_RATE = 0.18;   // static final constant
final int productId;              // final instance field
String name;
Product(String name) {
this.name = name;
totalProducts++;
productId = totalProducts;     // assigned once
}
static void showTotal() {          // static method
System.out.println("Total products: " + totalProducts);
}
}
public class Main {
public static void main(String[] args) {
Product p1 = new Product("Laptop");
Product p2 = new Product("Mouse");
System.out.println(p1.name + " ID: " + p1.productId);
System.out.println(p2.name + " ID: " + p2.productId);
System.out.println("Tax rate: " + Product.TAX_RATE);
Product.showTotal();
}
}
```

Output:

```
Laptop ID: 1
Mouse ID: 2
Tax rate: 0.18
Total products: 2
```

Additional Example:

This program demonstrates method hiding, where a static method in a subclass with the same signature as a static method in the superclass hides (rather than overrides) it, resolved using the reference type at compile time.

```
class Parent {
static void show() {
System.out.println("Static method in Parent");
}
}
class Child extends Parent {
static void show() {           // hides Parent's static method
System.out.println("Static method in Child");
}
}
public class Main {
public static void main(String[] args) {
Parent p = new Child();
p.show();     // calls Parent's version - resolved by reference type
Child.show();
}
}
```

Output:

```
Static method in Parent
Static method in Child
```

Key Points to Remember:

• A 'static' variable is initialized only once, when the class is first loaded, and retains its value across all objects.

• A 'final' variable must be definitely assigned before it is used, and only once during the lifetime of the object or class.

• 'static final' fields are the standard way to define named constants in Java, e.g., Math.PI.

• Static methods cannot be overridden in the traditional sense — a static method in a subclass with the same signature 'hides' rather than overrides the superclass version.

• Excessive use of static members can reduce flexibility and testability, so they should be used judiciously — primarily for constants and utility methods.

REVIEW QUESTIONS

Topic 1: Classes and Objects

Short Answer Questions:

1. **Define a class. How is it different from an object?**
2. **What is the difference between an instance variable and a static variable?**
3. **List the four access modifiers available in Java and their scope.**
4. **What is a default constructor? When is it provided automatically by Java?**
5. **Differentiate between constructor overloading and method overloading.**
6. **What happens when you assign one object reference to another using '='?**
7. **Why can a private member not be accessed directly from outside its class?**
8. **What is the purpose of the keyword 'this' inside a constructor?**
9. **Can a final class be extended? Justify your answer.**
10. **What is a nested class? Name its two broad categories.**

Long Answer / Essay Questions:

1. **Explain, with a suitable Java program, how classes and objects are declared, created and used.**
2. **Explain the various access specifiers in Java with an example program showing each one.**
3. **What are constructors? Explain default constructors, parameterized constructors and constructor overloading with example programs.**
4. **Explain static and non-static nested classes in Java with example programs.**
5. **Explain how arguments are passed to methods in Java. Illustrate the difference in behaviour for primitive and object arguments using a program.**
6. **Discuss the significance of the keyword 'this' with suitable example programs covering all of its uses.**
7. **Explain the concept of 'final' in the context of classes, methods and variables with appropriate examples.**

Topic 2: Methods

Short Answer Questions:

1. **What is a method? What are its essential parts?**

2. Differentiate between formal parameters and actual parameters.

3. What is method overloading? State the rules for overloading a method.

4. Can two methods be overloaded if they differ only in return type? Justify.

5. What is recursion? What is a base case, and why is it necessary?

6. Differentiate between method overloading and method overriding.

7. What is a static method? List two restrictions on static methods.

8. Why can a static method not directly access instance variables?

9. What is the purpose of the @Override annotation?

10. Define nesting of methods with a simple example.

Long Answer / Essay Questions:

1. Explain how methods are defined and invoked in Java with a suitable example program.

2. What is method overloading? Explain with a program that overloads a method in three different ways.

3. Explain how objects can be passed as parameters to methods with a suitable example program.

4. Explain recursion in Java. Write and explain a program to find the factorial of a number using recursion.

5. Distinguish between method overloading and method overriding with suitable example programs for each.

6. Explain the use of 'static' and 'final' attributes for class members with example programs.

7. Explain dynamic method dispatch and how it enables runtime polymorphism through method overriding.

PROGRAMMING EXERCISES

Attempt the following programs to strengthen your practical understanding of classes, objects and methods covered in this unit.

1. Write a Java program to define a class 'Employee' with fields empId, name and salary, along with a parameterized constructor and a method to display employee details. Create and display at least three Employee objects.

2. Write a Java program to demonstrate constructor overloading in a class 'Box' that computes the volume of a cube, a cuboid, and a box with default dimensions.

3. Write a Java program that uses private fields with public getter and setter methods to implement a simple 'BankAccount' class supporting deposit and withdrawal, with validation to prevent a negative balance.

4. Write a Java program to demonstrate the difference between passing a primitive type and an object to a method.

5. Write a Java program to demonstrate method overloading by writing three versions of a method 'area' for a square, rectangle and circle.

6. Write a Java program that uses a recursive method to compute the sum of the first N natural numbers, and compare it with an iterative version.

7. Write a Java program demonstrating a static nested class and a non-static inner class within an outer class 'Company' with fields for department and employee count.

8. Write a Java program to demonstrate method overriding where a superclass 'Shape' defines a method area() and subclasses 'Circle' and 'Square' override it appropriately.

9. Write a Java program that uses 'this' both to resolve a naming conflict and to chain two constructors of the same class.

10. Write a Java program that uses a static variable to count and display the total number of objects created from a class 'Product'.

UNIT SUMMARY

This unit covered the two most fundamental building blocks of object-oriented programming in Java: classes and objects, and methods. We studied how classes are declared with appropriate modifiers, how class members (fields and methods) are organised and access-controlled, and how objects are created, assigned and passed around a program.

We examined constructors in depth — including overloaded and chained constructors using 'this' — as well as nested classes and the role of 'final' in restricting classes, methods and variables. We also studied the crucial distinction between pass-by-value semantics for primitives and reference types.

In the second half of the unit, we studied methods in detail: how they are defined and invoked, how overloading achieves compile-time polymorphism, how methods can accept objects as parameters, how access control applies to methods, and how recursion and method nesting allow complex problems to be decomposed into simpler steps. Finally, we examined method overriding as the basis of runtime polymorphism, and the use of 'static' and 'final' attributes to create class-level and constant members.

A firm grasp of these concepts is essential, as they form the foundation for the more advanced object-oriented concepts of inheritance, interfaces, and polymorphism covered in the subsequent units.

UNIT III

Arrays

23CSE232 — Object Oriented Programming through Java

3.1 Introduction to Arrays

An array is a collection of a fixed number of elements of the same data type that are stored in contiguous memory locations and are referred to by a common name. Instead of declaring individual variables such as mark1, mark2, mark3 and so on for storing the marks of thirty students, a programmer can declare one array variable such as marks and use marks[0], marks[1], marks[2], and so on to represent individual values. This makes it possible to process large collections of related data efficiently using loops rather than writing repetitive code for every value.

In Java, an array is treated as an object. Every array has an associated class type, and the array object is created dynamically using the 'new' operator, even when the elements themselves are of a primitive type such as int, char, or double. Because arrays are objects, an array variable is actually a reference to the array object stored on the heap, not the array itself. Java arrays are strongly typed, which means an int array can only hold integer values and a String array can only hold String objects. Each array also has a public final field called length, which stores the number of elements the array can hold; unlike the length() method of the String class, the length field of an array is accessed without parentheses.

Arrays offer several advantages over using a set of unrelated variables. They allow related data to be grouped under a single identifier, they allow random access to elements using an index, they allow iteration through all elements using loops, and they allow the array itself to be passed to methods as a single argument. The primary limitation of an array is that its size is fixed once it is created; Java does not permit the size of an already-created array to be changed, although a new, larger array can be created and the old data copied into it, which is discussed later in this unit.

Characteristics of Arrays in Java

- Arrays are homogeneous — all elements must be of the same declared type.
- Arrays are objects and are always created on the heap using the 'new' keyword (or an array initializer).
- Array elements are stored in contiguous memory locations, which allows fast, constant-time random access using an index.
- The index of an array always starts at 0 and goes up to length − 1.
- The size of an array is fixed at the time of creation and cannot be changed later.
- Arrays can hold primitive data types (int, char, double, boolean, etc.) or reference types (objects, Strings, other arrays).

Why Use Arrays? — A Motivating Comparison

To appreciate why arrays are essential, consider the alternative of using separate variables to store the marks of five students: int mark1, mark2, mark3, mark4, mark5. Computing the average would require writing out every variable name explicitly, and the program would need to be rewritten if the number of students changed. If the same task is written in terms of an array, a single loop can process any number of students without modifying the logic of the program at all, which is one of the most important motivations for using arrays in real programs.

Example Program:

```
public class WithoutVsWithArray {
    public static void main(String[] args) {
        // Without an array — repetitive and does not scale
        int mark1 = 78, mark2 = 88, mark3 = 92, mark4 = 65, mark5 = 74;
        int totalWithoutArray = mark1 + mark2 + mark3 + mark4 + mark5;
        System.out.println("Total (without array): " + totalWithoutArray);

        // With an array — concise and scales to any number of students
        int[] marks = {78, 88, 92, 65, 74};
        int totalWithArray = 0;
        for (int i = 0; i < marks.length; i++) {
            totalWithArray += marks[i];
        }
        System.out.println("Total (with array): " + totalWithArray);
        System.out.println("Average: " + ((double) totalWithArray /
marks.length));
    }
}
```

Output:

```
Total (without array): 397
Total (with array): 397
Average: 79.4
```

Key Points to Remember

- An array groups multiple values of the same type under a single variable name for efficient, loop-based processing.
- Every array in Java is an object; the array variable itself only stores a reference to that object.
- The number of elements an array can hold is fixed permanently at the time the array is created with 'new'.

- Arrays support random access — any element can be reached directly using its index without traversing preceding elements.

3.2 Declaration and Initialization of Arrays

Using an array in a Java program involves two distinct steps: declaring the array variable and creating (instantiating) the array object. Declaration merely tells the compiler the name and type of the variable that will refer to an array; it does not allocate any memory for elements. Creation, performed with the 'new' operator, actually allocates memory on the heap for the specified number of elements and produces a reference that is stored in the array variable.

Declaring an Array

An array variable can be declared by placing square brackets either after the data type or after the variable name. Placing the brackets after the type, as in 'int[] marks;', is the preferred Java style because it visually groups the array nature with the type. The C-style form 'int marks[];' is also legal but less commonly used in modern Java code.

Syntax:

```
dataType[] arrayName;      // preferred style
dataType arrayName[];      // C-style, also legal
```

Creating (Instantiating) an Array

Once declared, an array must be created using the 'new' operator, specifying the number of elements the array will hold. When an array is created this way, Java automatically initializes every element to the default value for that type: 0 for numeric types, '\u0000' for char, false for boolean, and null for reference types.

Syntax:

```
arrayName = new dataType[size];

// Declaration and creation combined:
dataType[] arrayName = new dataType[size];
```

Array Initializers (Shorthand Initialization)

Java also allows an array to be declared and initialized in a single statement using an array initializer — a comma-separated list of values enclosed in curly braces. When this form is used, the size of the array is automatically determined from the number of values supplied, and the 'new' keyword may be omitted.

Syntax:

```
dataType[] arrayName = {value1, value2, value3, ...};

// equivalent long form
dataType[] arrayName = new dataType[]{value1, value2, value3, ...};
```

Example Program: Declaring and Initializing Arrays

The following program illustrates the three ways of declaring and initializing arrays discussed above, and prints their contents and lengths.

Example Program:

```
public class ArrayDeclarationDemo {
    public static void main(String[] args) {
        // 1. Declare, then create with 'new' (default values)
        int[] scores = new int[5];
        System.out.println("Default values of 'scores' array:");
        for (int i = 0; i < scores.length; i++) {
            System.out.print(scores[i] + " ");
        }
        System.out.println();

        // 2. Create and assign values individually
        int[] marks = new int[3];
        marks[0] = 85;
        marks[1] = 90;
        marks[2] = 78;
        System.out.println("Marks array (assigned individually):");
        for (int m : marks) {
            System.out.print(m + " ");
        }
        System.out.println();

        // 3. Array initializer shorthand
        double[] prices = {199.99, 349.50, 25.75};
        System.out.println("Prices array (initializer shorthand):");
        for (double p : prices) {
```

```
            System.out.print(p + " ");
        }
        System.out.println();
        System.out.println("Length of prices array: " + prices.length);
    }
}
```

Output:

```
Default values of 'scores' array:
0 0 0 0 0
Marks array (assigned individually):
85 90 78
Prices array (initializer shorthand):
199.99 349.5 25.75
Length of prices array: 3
```


Key Points to Remember

- Declaration only introduces the array variable name and type; it does not allocate memory for elements.
- The 'new' operator is what actually allocates memory on the heap for the array's elements.
- When created with 'new' and no initializer, numeric elements default to 0, boolean defaults to false, and object references default to null.
- The array-initializer shorthand '{...}' can only be used at the point of declaration, not in a later assignment statement.

3.3 Storage of Array in Computer Memory

Understanding how arrays are stored in memory helps explain why array access is so efficient and why array variables behave the way they do when assigned or passed to methods. In Java, memory is broadly divided into two regions relevant here: the stack, which stores method call frames and local reference variables, and the heap, which stores all objects, including arrays.

When an array is declared, only a reference variable is created on the stack; this reference initially holds the value null until the array is created with 'new'. When the array is created, Java allocates a contiguous block of memory on the heap large enough to hold all the elements, and the reference variable on the stack is updated to store the memory address (reference) of this block. Because the elements of the array occupy contiguous memory, the address of any element can be computed

directly from the base address of the array and the index of the element, using the formula: address of element[i] = base address + (i × size of each element). This is what allows array access to occur in constant time, O(1), regardless of the size of the array or the index being accessed.

It is important to note that when an array of objects (a reference type array, such as String[] or a custom class array) is created, the array itself stores references to the objects, not the objects themselves. The objects are separately allocated on the heap, and the array holds pointers to them. This distinction becomes especially significant when arrays are assigned to one another or passed to methods, which is discussed in Section 3.6.

Conceptual Memory Layout

Consider the declaration 'int[] arr = {10, 20, 30, 40};'. Conceptually, the stack holds the variable 'arr', which stores a reference (address) pointing to a block on the heap. That heap block stores the four integer values contiguously, along with the array's length metadata.

Memory Representation:

```
Stack:  arr  --------> (reference to heap block)
Heap:   [length = 4] [10][20][30][40]
        index:         0    1    2    3
```

Example Program: Demonstrating Contiguous Access

Although Java does not expose raw memory addresses to the programmer, the identity hash code of an array object (obtained via the default toString() or hashCode()) confirms that the entire array is a single object on the heap, and the constant-time access behavior confirms contiguous storage.

Example Program:

```
public class ArrayMemoryDemo {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40};

        // The array reference itself (identity of the single heap object)
        System.out.println("Array object reference: " + arr);

        // Accessing elements is O(1) regardless of index
        System.out.println("Element at index 0: " + arr[0]);
        System.out.println("Element at index 3: " + arr[3]);

        // Two variables referring to the SAME array (same heap block)
```

```
        int[] ref2 = arr;
        System.out.println("arr == ref2 (same heap object)? " + (arr == ref2));
    }
}
```

Output:

```
Array object reference: [I@1b6d3586
Element at index 0: 10
Element at index 3: 40
arr == ref2 (same heap object)? true
```

Memory Considerations for Multi-Dimensional Arrays

For a two-dimensional array, Java does not allocate one single contiguous block for the entire grid the way languages like C do. Instead, a 2-D array is an array of row-arrays: the outer array stores references to separate row arrays, and each row array is itself a contiguous block of memory. This means rows can even be of different lengths (a jagged array, discussed in Section 3.12), and it also means that accessing arr[i][j] involves two steps internally — first following the reference to row i, and then indexing into that row at position j.

Key Points to Remember

- An array variable on the stack stores a reference (address); the actual elements live in a contiguous block on the heap.
- Element access is O(1) because the address of any element can be computed directly from the base address and index.
- Two-dimensional arrays are arrays of row-array references, not one single contiguous 2-D block, allowing rows of unequal length.
- Reference-type arrays store references to objects, not the objects themselves, inside the array's memory block.

3.4 Accessing Elements of Arrays

Individual elements of an array are accessed using the index (subscript) operator '[]', placed after the array name, with the index specifying the position of the element. Array indices in Java are zero-based, meaning the first element is at index 0 and the last element of an array of length n is at index n − 1. Accessing an array element with an index outside the valid range (less than 0 or greater than

or equal to length) does not cause a compile-time error; instead, it throws an ArrayIndexOutOfBoundsException at run time, which the programmer must guard against using proper loop bounds or exception handling.

Accessing Elements Using the Index Operator

Syntax:

```
value = arrayName[index];      // read
arrayName[index] = value;      // write
```

Traversing Arrays with Loops

The most common way to access every element of an array is with a 'for' loop that runs from 0 to length − 1. Java also provides the enhanced for loop (for-each loop), which iterates over every element of the array in order without requiring an explicit index variable, but it can only be used to read elements, not to modify the original array by index.

Syntax:

```
// standard for loop
for (int i = 0; i < arrayName.length; i++) {
    // use arrayName[i]
}

// enhanced for-each loop
for (dataType element : arrayName) {
    // use element (read-only view)
}
```

Example Program: Accessing Array Elements and Handling Invalid Index

Example Program:

```
public class ArrayAccessDemo {
    public static void main(String[] args) {
        int[] temperatures = {30, 32, 28, 35, 31};

        // Access using standard for loop
        System.out.println("Using standard for loop:");
        for (int i = 0; i < temperatures.length; i++) {
            System.out.println("Day " + (i + 1) + ": " + temperatures[i] + " C");
```

```
        }

        // Access using enhanced for-each loop
        System.out.println("\nUsing for-each loop:");
        for (int t : temperatures) {
            System.out.print(t + " ");
        }
        System.out.println();

        // Demonstrate ArrayIndexOutOfBoundsException
        try {
            System.out.println(temperatures[10]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("\nError caught: " + e);
        }
    }
}
```

Output:

```
Using standard for loop:
Day 1: 30 C
Day 2: 32 C
Day 3: 28 C
Day 4: 35 C
Day 5: 31 C

Using for-each loop:
30 32 28 35 31

Error caught: java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds
for length 5
```

Key Points to Remember

- Valid indices for an array of length n range from 0 to n − 1; index n and beyond are invalid.
- Accessing an index outside the valid range compiles successfully but throws ArrayIndexOutOfBoundsException at run time.
- The for-each loop is convenient for reading elements in order but cannot be used to modify elements by index or to traverse in reverse.
- The standard for loop remains necessary whenever the current index value itself is required inside the loop body.

3.5 Operations on Array Elements

Beyond simple access, several standard operations are commonly performed on arrays: traversal (visiting every element), insertion (placing a new value, typically requiring shifting of elements), deletion (removing a value, again requiring shifting), updation (modifying an existing element), and aggregate operations such as finding the sum, average, maximum, or minimum of the elements. Because Java arrays are of fixed size, insertion and deletion in the middle of an array require manually shifting the subsequent elements, since there is no built-in 'insert' or 'remove' operation as with dynamic structures like ArrayList.

Traversal, Sum, Average, Maximum and Minimum

These operations are performed by iterating once through the array and accumulating the required result in a variable, comparing each element as needed.

Insertion and Deletion by Shifting Elements

To insert a value at a given position, all elements from that position onward must first be shifted one place to the right (starting from the last element and moving backward, to avoid overwriting data), and then the new value is placed at the freed position. This requires the underlying array to have at least one unused slot, or a larger array must be created. To delete a value at a given position, all elements after that position are shifted one place to the left, overwriting the deleted value, and the logical size of the used portion of the array is reduced by one.

Example Program: Insertion, Deletion and Aggregate Operations

Example Program:

```
import java.util.Arrays;

public class ArrayOperationsDemo {
    public static void main(String[] args) {
        int[] arr = new int[10];
        int[] initial = {12, 45, 23, 67, 8};
        int n = initial.length;
        System.arraycopy(initial, 0, arr, 0, n);

        System.out.println("Original array: " +
Arrays.toString(Arrays.copyOf(arr, n)));

        // --- Insertion: insert 50 at index 2 ---
        int insertPos = 2, insertValue = 50;
```

```
        for (int i = n; i > insertPos; i--) {
            arr[i] = arr[i - 1];
        }
        arr[insertPos] = insertValue;
        n++;
        System.out.println("After inserting 50 at index 2: " +
Arrays.toString(Arrays.copyOf(arr, n)));

        // --- Deletion: delete element at index 4 ---
        int deletePos = 4;
        for (int i = deletePos; i < n - 1; i++) {
            arr[i] = arr[i + 1];
        }
        n--;
        System.out.println("After deleting element at index 4: " +
Arrays.toString(Arrays.copyOf(arr, n)));

        // --- Updation: update index 0 to 99 ---
        arr[0] = 99;
        System.out.println("After updating index 0 to 99: " +
Arrays.toString(Arrays.copyOf(arr, n)));

        // --- Aggregate operations ---
        int sum = 0, max = arr[0], min = arr[0];
        for (int i = 0; i < n; i++) {
            sum += arr[i];
            if (arr[i] > max) max = arr[i];
            if (arr[i] < min) min = arr[i];
        }
        double average = (double) sum / n;
        System.out.println("Sum = " + sum + ", Average = " + average + ", Max = "
+ max + ", Min = " + min);
    }
}
```

Output:

```
Original array: [12, 45, 23, 67, 8]
After inserting 50 at index 2: [12, 45, 50, 23, 67, 8]
After deleting element at index 4: [12, 45, 50, 23, 8]
After updating index 0 to 99: [99, 45, 50, 23, 8]
Sum = 225, Average = 45.0, Max = 99, Min = 8
```

Example Program: Finding the Largest and Second-Largest Elements

A frequently asked variant of the maximum-finding operation is to locate both the largest and the second-largest values in a single traversal, which is achieved by maintaining two tracking variables and updating them carefully as the array is scanned.

Example Program:

```
public class SecondLargestDemo {
    public static void main(String[] args) {
        int[] arr = {45, 89, 23, 67, 89, 12, 76};
        int largest = Integer.MIN_VALUE;
        int secondLargest = Integer.MIN_VALUE;

        for (int value : arr) {
            if (value > largest) {
                secondLargest = largest;
                largest = value;
            } else if (value > secondLargest && value != largest) {
                secondLargest = value;
            }
        }

        System.out.println("Largest element: " + largest);
        System.out.println("Second largest element: " + secondLargest);
    }
}
```

Output:

```
Largest element: 89
Second largest element: 76
```

Key Points to Remember

- Insertion and deletion in an array require manually shifting elements, since arrays do not support these operations natively.
- Insertion shifts elements rightward starting from the end; deletion shifts elements leftward starting from the deletion point.
- Aggregate operations (sum, average, max, min) require only a single linear traversal of the array.
- Finding the second-largest value needs careful handling of duplicate values, as shown by the 'value != largest' check above.

3.6 Assigning One Array to Another

Because array variables in Java hold references and not the actual data, the statement 'arr2 = arr1;' does not copy the elements of arr1 into a new array; instead, it makes arr2 point to the very same heap object as arr1. After such an assignment, both arr1 and arr2 refer to one array, and any modification made through either variable is visible through the other, since there is only one underlying array in memory. This behavior is often a source of subtle bugs for beginners who expect a duplicate array to be created.

To create a genuinely independent copy of an array — sometimes called a deep or shallow copy depending on the element type — Java provides several approaches: writing an explicit loop that copies each element, using the static method System.arraycopy(), using the convenience methods Arrays.copyOf() and Arrays.copyOfRange() from java.util.Arrays, or using the clone() method that every array object inherits from Object.

Reference Assignment vs. True Copying

Syntax:

```
// Reference assignment (NOT a copy — both names point to same array)
int[] arr2 = arr1;

// True copy using a loop
int[] copy = new int[arr1.length];
for (int i = 0; i < arr1.length; i++) copy[i] = arr1[i];

// True copy using System.arraycopy
System.arraycopy(arr1, 0, copy, 0, arr1.length);

// True copy using Arrays.copyOf
int[] copy = java.util.Arrays.copyOf(arr1, arr1.length);

// True copy using clone()
int[] copy = arr1.clone();
```

Example Program: Reference Assignment vs. clone()

Example Program:

```
import java.util.Arrays;

public class ArrayAssignmentDemo {
```

```
    public static void main(String[] args) {
        int[] original = {1, 2, 3, 4, 5};

        // Reference assignment
        int[] refCopy = original;
        refCopy[0] = 100;   // modifies the SAME array
        System.out.println("original after modifying refCopy: " +
Arrays.toString(original));

        // True copy using clone()
        int[] original2 = {1, 2, 3, 4, 5};
        int[] trueCopy = original2.clone();
        trueCopy[0] = 999;  // does NOT affect original2
        System.out.println("original2 after modifying trueCopy: " +
Arrays.toString(original2));
        System.out.println("trueCopy: " + Arrays.toString(trueCopy));

        System.out.println("original == refCopy (same object)? " + (original ==
refCopy));
        System.out.println("original2 == trueCopy (same object)? " + (original2
== trueCopy));
    }
}
```

Output:

```
original after modifying refCopy: [100, 2, 3, 4, 5]
original2 after modifying trueCopy: [1, 2, 3, 4, 5]
trueCopy: [999, 2, 3, 4, 5]
original == refCopy (same object)? true
original2 == trueCopy (same object)? false
```

Key Points to Remember

- 'arr2 = arr1;' copies only the reference — both variable names then point to one shared array object.
- clone(), Arrays.copyOf(), Arrays.copyOfRange(), and System.arraycopy() all create a genuinely new, independent array.
- For arrays of primitive types, clone() performs a full (deep) copy of the values themselves.
- For arrays of objects, clone() performs only a shallow copy — the new array holds copies of the same object references, so the referenced objects themselves are still shared.

3.7 Dynamic Change of Array Size

A defining characteristic of arrays in Java is that once created, their size is fixed and cannot be altered. There is no operator or method that literally extends or shrinks an existing array object in place. However, a program can simulate a resizable array by creating a new array of the desired new size and copying the required elements from the old array into the new one, after which the reference variable is made to point to the new array; the old array, no longer referenced, is eventually reclaimed by the garbage collector.

The java.util.Arrays class provides a convenient method, copyOf(originalArray, newLength), that performs exactly this task: it creates a new array of the specified length, copies as many elements as will fit from the original array, and pads any additional slots with default values (0, false, or null) if the new length is larger. In practice, when a program needs a data structure whose size can grow or shrink freely at run time, it is generally preferable to use java.util.ArrayList or java.util.Vector, both of which are built on top of an internal array that is automatically resized behind the scenes.

Resizing an Array Using Arrays.copyOf()

Syntax:

```
dataType[] newArray = Arrays.copyOf(oldArray, newLength);
```

Example Program: Growing and Shrinking an Array

Example Program:

```
import java.util.Arrays;

public class ArrayResizeDemo {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30};
        System.out.println("Original array: " + Arrays.toString(arr));

        // Grow the array to size 6
        arr = Arrays.copyOf(arr, 6);
        arr[3] = 40;
        arr[4] = 50;
        arr[5] = 60;
        System.out.println("After growing to size 6: " + Arrays.toString(arr));

        // Shrink the array to size 4
        arr = Arrays.copyOf(arr, 4);
```

```
        System.out.println("After shrinking to size 4: " + Arrays.toString(arr));

        // Equivalent dynamic behaviour using ArrayList for comparison
        java.util.ArrayList<Integer> dynamicList = new java.util.ArrayList<>();
        dynamicList.add(10);
        dynamicList.add(20);
        dynamicList.add(30);
        dynamicList.add(40);   // grows automatically
        dynamicList.remove(0); // shrinks automatically
        System.out.println("ArrayList after add/remove: " + dynamicList);
    }
}
```

Output:

```
Original array: [10, 20, 30]
After growing to size 6: [10, 20, 30, 40, 50, 60]
After shrinking to size 4: [10, 20, 30, 40]
ArrayList after add/remove: [20, 30, 40]
```

Key Points to Remember

- Java arrays cannot be resized in place — 'resizing' always means creating a new array and copying the relevant elements over.
- Arrays.copyOf() is the simplest built-in way to obtain a larger or smaller copy of an existing array.
- When growing an array, any newly added slots beyond the original length are filled with default values.
- For frequently changing collections, ArrayList or Vector is usually a better choice than manually resizing arrays.

3.8 Sorting of Arrays

Sorting refers to arranging the elements of an array in a particular order, usually ascending or descending. Sorting can be performed manually by implementing a sorting algorithm such as bubble sort, selection sort, or insertion sort, or it can be performed conveniently using the built-in static method Arrays.sort() from java.util.Arrays, which uses a highly optimized dual-pivot quicksort for primitive types and a stable mergesort/Timsort variant for object arrays.

Manual Sorting: Bubble Sort

Bubble sort repeatedly steps through the array, compares each pair of adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no more swaps are needed, at which point the array is sorted. Bubble sort has a worst-case and average time complexity of $O(n^2)$, making it inefficient for large arrays but simple to understand for teaching purposes.

Example Program:

```
public class BubbleSortDemo {
    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }

        System.out.print("Sorted array (bubble sort): ");
        for (int val : arr) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

Output:

```
Sorted array (bubble sort): 11 12 22 25 64
```

Manual Sorting: Selection Sort

Selection sort divides the array into a sorted part and an unsorted part. In each pass, it finds the minimum element from the unsorted part and swaps it with the first element of the unsorted part, thereby growing the sorted part by one element. Like bubble sort, selection sort has $O(n^2)$ time complexity, but it performs fewer swaps, which can matter when swapping is expensive.

Example Program:

```
public class SelectionSortDemo {
    public static void main(String[] args) {
        int[] arr = {29, 10, 14, 37, 13};
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }

        System.out.print("Sorted array (selection sort): ");
        for (int val : arr) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

Output:

```
Sorted array (selection sort): 10 13 14 29 37
```

Manual Sorting: Insertion Sort

Insertion sort builds the sorted array one element at a time by taking each element from the unsorted part and inserting it into its correct position within the already-sorted part, shifting larger elements rightward to make room. Insertion sort performs well on small or nearly-sorted arrays and, like the other manual methods, has a worst-case time complexity of $O(n^2)$.

Example Program:

```
public class InsertionSortDemo {
    public static void main(String[] args) {
        int[] arr = {12, 11, 13, 5, 6};
```

```
        int n = arr.length;

        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }

        System.out.print("Sorted array (insertion sort): ");
        for (int val : arr) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

Output:

```
Sorted array (insertion sort): 5 6 11 12 13
```

Sorting Using Arrays.sort()

Syntax:

```
Arrays.sort(arrayName);                        // ascending order, entire array
Arrays.sort(arrayName, fromIndex, toIndex);    // ascending order, sub-range
Arrays.sort(arrayName, Collections.reverseOrder()); // descending, object arrays
only
```

Example Program:

```
import java.util.Arrays;

public class ArraysSortDemo {
    public static void main(String[] args) {
        int[] numbers = {64, 25, 12, 22, 11};
        Arrays.sort(numbers);
        System.out.println("Sorted (ascending): " + Arrays.toString(numbers));
```

```
        String[] names = {"Charlie", "Alice", "Bob"};
        Arrays.sort(names);
        System.out.println("Sorted names: " + Arrays.toString(names));

        // Sort only a sub-range: indices 1 (inclusive) to 4 (exclusive)
        int[] partial = {5, 3, 8, 1, 9, 2};
        Arrays.sort(partial, 1, 4);
        System.out.println("Partially sorted (index 1 to 3): " +
Arrays.toString(partial));
    }
}
```

Output:

```
Sorted (ascending): [11, 12, 22, 25, 64]
Sorted names: [Alice, Bob, Charlie]
Partially sorted (index 1 to 3): [5, 1, 3, 8, 9, 2]
```


Key Points to Remember

- Bubble sort repeatedly swaps adjacent out-of-order elements; simple to code but inefficient for large arrays — $O(n^2)$.
- Selection sort repeatedly selects the minimum of the unsorted part and swaps it into place — fewer swaps than bubble sort.
- Insertion sort builds the sorted portion incrementally by inserting each new element into its correct position.
- Arrays.sort() is the preferred choice in real programs — it is highly optimized and far faster than manual $O(n^2)$ methods for large data.

3.9 Search for Values in Arrays

Searching refers to finding whether a particular value is present in an array, and if so, at what index. Two classical algorithms are used: linear search, which checks every element one by one and works on both sorted and unsorted arrays but takes O(n) time in the worst case; and binary search, which repeatedly divides a sorted array in half and works only on sorted arrays, but is much faster, taking only O(log n) time in the worst case.

Linear Search

Example Program:

```
public class LinearSearchDemo {
    public static void main(String[] args) {
        int[] arr = {34, 12, 67, 8, 45, 23};
        int key = 45;
        int foundIndex = -1;

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == key) {
                foundIndex = i;
                break;
            }
        }

        if (foundIndex != -1) {
            System.out.println("Value " + key + " found at index " + foundIndex);
        } else {
            System.out.println("Value " + key + " not found in the array");
        }
    }
}
```

Output:

```
Value 45 found at index 4
```

Binary Search (Manual Implementation and Arrays.binarySearch)

Binary search requires the array to be sorted first. It compares the search key with the middle element: if they match, the search ends; if the key is smaller, the search continues in the left half; if the key is larger, the search continues in the right half. This halving continues until the key is found or the search range becomes empty. The java.util.Arrays class provides a ready-made binarySearch() method that implements this algorithm.

Example Program:

```
import java.util.Arrays;

public class BinarySearchDemo {
    public static void main(String[] args) {
        int[] arr = {8, 12, 23, 34, 45, 67};   // must be sorted
```

```
        int key = 34;

        // Manual binary search
        int low = 0, high = arr.length - 1, result = -1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == key) {
                result = mid;
                break;
            } else if (arr[mid] < key) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        System.out.println("Manual binary search: value " + key + " found at
index " + result);

        // Using built-in Arrays.binarySearch()
        int index = Arrays.binarySearch(arr, key);
        System.out.println("Arrays.binarySearch(): value " + key + " found at
index " + index);
    }
}
```

Output:

```
Manual binary search: value 34 found at index 3
Arrays.binarySearch(): value 34 found at index 3
```

Key Points to Remember

- Linear search checks every element sequentially; it works on unsorted data but has O(n) worst-case time.
- Binary search requires the array to be sorted first, but achieves a much faster O(log n) worst-case time by repeatedly halving the search range.
- Arrays.binarySearch() returns a negative value if the key is not found, which encodes the insertion point; it should not simply be treated as 'not found equals −1'.
- Choosing between linear and binary search is a trade-off between the cost of sorting the data once and the cost of each individual search.

3.10 The Arrays Class (java.util.Arrays)

The java.util.Arrays class is a utility class that contains only static methods for performing common operations on arrays, such as sorting, searching, filling, comparing, and converting arrays to strings or lists. Because all its methods are static, they are called directly on the class name (Arrays.methodName(...)) rather than on an object instance, and the class itself cannot be instantiated.

Commonly Used Methods of the Arrays Class

- Arrays.sort(arr) — sorts the array in ascending order.
- Arrays.binarySearch(arr, key) — searches a sorted array for a key using binary search.
- Arrays.fill(arr, value) — fills every element of the array with the given value.
- Arrays.equals(arr1, arr2) — returns true if both arrays have the same length and identical elements in the same order.
- Arrays.copyOf(arr, newLength) — returns a new array of the given length, copied from the original.
- Arrays.copyOfRange(arr, from, to) — returns a new array containing a specified range of the original.
- Arrays.toString(arr) — returns a readable String representation of a one-dimensional array.
- Arrays.deepToString(arr) — returns a readable String representation of a multi-dimensional array.
- Arrays.asList(arr) — returns a fixed-size List backed by the given array.

Example Program: Using Methods of the Arrays Class

Example Program:

```
import java.util.Arrays;
import java.util.List;

public class ArraysClassDemo {
    public static void main(String[] args) {
        int[] a = {5, 2, 9, 1, 7};
        int[] b = {5, 2, 9, 1, 7};

        Arrays.sort(a);
        System.out.println("Sorted a: " + Arrays.toString(a));

        System.out.println("a.equals(b) before sorting b? " + Arrays.equals(a,
b));
```

```
        int[] filled = new int[5];
        Arrays.fill(filled, 7);
        System.out.println("Filled array: " + Arrays.toString(filled));

        int[] range = Arrays.copyOfRange(a, 1, 4);
        System.out.println("copyOfRange(a, 1, 4): " + Arrays.toString(range));

        List<Integer> list = Arrays.asList(1, 2, 3, 4);
        System.out.println("Arrays.asList: " + list);

        int[][] grid = {{1, 2}, {3, 4}};
        System.out.println("deepToString(grid): " + Arrays.deepToString(grid));
    }
}
```

Output:

```
Sorted a: [1, 2, 5, 7, 9]
a.equals(b) before sorting b? false
Filled array: [7, 7, 7, 7, 7]
copyOfRange(a, 1, 4): [2, 5, 7]
Arrays.asList: [1, 2, 3, 4]
deepToString(grid): [[1, 2], [3, 4]]
```

Key Points to Remember

- All methods of the Arrays class are static, so they are invoked as Arrays.methodName(...), never on an array instance.
- Arrays.toString() should be used instead of println(arrayName) directly, since the latter prints an unreadable object reference such as [I@1b6d3586.
- Arrays.deepToString() must be used instead of Arrays.toString() for multi-dimensional arrays to correctly display nested contents.
- Arrays.asList() returns a fixed-size list view backed by the array — elements can be changed, but the list cannot be resized.

3.11 Two-Dimensional Arrays

A two-dimensional array is an array of arrays, commonly used to represent data in a tabular or matrix form with rows and columns. In Java, a two-dimensional array is declared using two pairs of square

brackets and created by specifying the number of rows and columns. Internally, a 2-D array such as int[][] table is really a one-dimensional array of length equal to the number of rows, in which each element is itself a reference to a one-dimensional array (a row) of the specified number of columns.

An element of a two-dimensional array is accessed using two indices: the first for the row and the second for the column, in the form arrayName[row][column]. Two-dimensional arrays can be traversed using nested loops, with the outer loop iterating over rows and the inner loop iterating over columns.

Declaration, Creation and Initialization

Syntax:

```
dataType[][] arrayName = new dataType[rows][columns];

// with an initializer
dataType[][] arrayName = {
    {row0col0, row0col1, ...},
    {row1col0, row1col1, ...}
};
```

Example Program: Matrix Addition Using Two-Dimensional Arrays

Example Program:

```
public class TwoDArrayDemo {
    public static void main(String[] args) {
        int[][] matrixA = {
            {1, 2, 3},
            {4, 5, 6}
        };
        int[][] matrixB = {
            {7, 8, 9},
            {10, 11, 12}
        };
        int rows = matrixA.length;
        int cols = matrixA[0].length;
        int[][] result = new int[rows][cols];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = matrixA[i][j] + matrixB[i][j];
            }
        }
```

```
        System.out.println("Sum of matrices:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
Sum of matrices:
8 10 12
14 16 18
```

Example Program: Transpose of a Matrix

The transpose of a matrix is obtained by interchanging its rows and columns, so that the element at position [i][j] in the original matrix moves to position [j][i] in the transposed matrix. If the original matrix has r rows and c columns, the transposed matrix has c rows and r columns.

Example Program:

```
public class MatrixTransposeDemo {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6}
        };
        int rows = matrix.length;
        int cols = matrix[0].length;
        int[][] transpose = new int[cols][rows];

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                transpose[j][i] = matrix[i][j];
            }
        }

        System.out.println("Transpose of the matrix:");
        for (int i = 0; i < cols; i++) {
```

```
            for (int j = 0; j < rows; j++) {
                System.out.print(transpose[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
Transpose of the matrix:
1 4
2 5
3 6
```

Key Points to Remember

- A two-dimensional array in Java is an array of one-dimensional row arrays, accessed as arrayName[row][column].
- matrix.length gives the number of rows, and matrix[0].length gives the number of columns of the first row.
- Two nested loops are required to traverse a two-dimensional array — the outer loop for rows, the inner loop for columns.
- Matrix operations such as addition, subtraction, and transpose all require dimension compatibility to be checked before processing.

3.12 Arrays of Varying Lengths (Jagged Arrays)

Because a two-dimensional array in Java is actually an array of one-dimensional array references, there is no requirement that every row have the same number of columns. An array in which different rows have different lengths is called a jagged array (or a ragged array), since its shape resembles jagged, uneven edges rather than a uniform rectangular grid. Jagged arrays are useful when the amount of data per row genuinely differs, such as storing the list of students in each of several classes of different sizes, which avoids wasting memory that a rectangular array would otherwise reserve for unused slots.

Declaring a Jagged Array

Syntax:

```
dataType[][] arrayName = new dataType[numberOfRows][];   // columns left
unspecified
arrayName[0] = new dataType[size0];
arrayName[1] = new dataType[size1];
// ... each row created with its own length
```

Example Program: Jagged Array

Example Program:

```
public class JaggedArrayDemo {
    public static void main(String[] args) {
        int[][] jagged = new int[4][];
        jagged[0] = new int[]{1};
        jagged[1] = new int[]{1, 2};
        jagged[2] = new int[]{1, 2, 3};
        jagged[3] = new int[]{1, 2, 3, 4};

        for (int i = 0; i < jagged.length; i++) {
            System.out.print("Row " + i + " (length " + jagged[i].length + "):
");
            for (int j = 0; j < jagged[i].length; j++) {
                System.out.print(jagged[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
Row 0 (length 1): 1
Row 1 (length 2): 1 2
Row 2 (length 3): 1 2 3
Row 3 (length 4): 1 2 3 4
```

Key Points to Remember

- A jagged array is a 2-D array in which each row can have a different number of columns.
- Jagged arrays are declared with only the number of rows specified initially: 'new dataType[rows][]'; each row is then created individually.

- Jagged arrays save memory compared to a rectangular array when row sizes genuinely differ, since no unused slots are reserved.
- The length of an individual row i in a jagged array is obtained with arrayName[i].length, not arrayName.length.

3.13 Three-Dimensional Arrays

A three-dimensional array extends the same idea one level further and is essentially an array of two-dimensional arrays. It is declared using three pairs of square brackets and is useful for representing data that naturally has three dimensions, such as a set of grids, a small volumetric dataset, or a collection of matrices (for example, marks of students across multiple sections and multiple subjects). An element is accessed using three indices in the form arrayName[i][j][k], and traversal requires three nested loops.

Declaration, Creation and Access

Syntax:

```
dataType[][][] arrayName = new dataType[size1][size2][size3];
value = arrayName[i][j][k];
```

Example Program: Three-Dimensional Array

Example Program:

```
public class ThreeDArrayDemo {
    public static void main(String[] args) {
        // 2 groups x 2 rows x 3 columns
        int[][][] cube = new int[2][2][3];
        int value = 1;

        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 3; k++) {
                    cube[i][j][k] = value++;
                }
            }
        }

        for (int i = 0; i < 2; i++) {
            System.out.println("Group " + i + ":");
            for (int j = 0; j < 2; j++) {
```

```
                for (int k = 0; k < 3; k++) {
                    System.out.print(cube[i][j][k] + " ");
                }
                System.out.println();
            }
        }
    }
}
```

Output:

```
Group 0:
1 2 3
4 5 6
Group 1:
7 8 9
10 11 12
```


Key Points to Remember

- A three-dimensional array is an array of two-dimensional arrays, declared with three pairs of square brackets.
- An element is accessed with three indices, arrayName[i][j][k], representing the group/plane, row, and column respectively.
- Traversing a three-dimensional array fully requires three nested loops.
- Three-dimensional arrays are useful for representing stacks of matrices or simple volumetric/grid data.

3.14 Arrays as Vectors

The term 'vector' is used in two related but distinct senses in the context of this unit. In the mathematical sense, a one-dimensional array is a natural way to represent a vector — an ordered list of numbers on which operations such as addition, scalar multiplication, and the dot product can be performed, exactly as with arrays of numbers. In the Java class-library sense, java.util.Vector is a legacy dynamic-array class (part of the Java Collections Framework, predating ArrayList) that automatically grows and shrinks as elements are added or removed, and whose methods are synchronized, making it suitable for use in multi-threaded programs where thread safety is required.

Unlike a plain array, a Vector does not require its size to be fixed in advance and does not require the programmer to manually copy elements into a larger array when it needs to grow — this growth is handled internally by the Vector class itself. Vector stores elements as Objects internally (using generics in modern Java for type safety), and provides methods such as add(), get(), set(), remove(), size(), and contains() for manipulating its contents.

Mathematical Vector Operations Using an Array

Example Program:

```
public class ArrayAsVectorDemo {
    public static void main(String[] args) {
        double[] vectorA = {2, 4, 6};
        double[] vectorB = {1, 3, 5};

        // Vector addition
        double[] sum = new double[3];
        for (int i = 0; i < 3; i++) {
            sum[i] = vectorA[i] + vectorB[i];
        }
        System.out.print("Vector sum: ");
        for (double v : sum) System.out.print(v + " ");
        System.out.println();

        // Scalar multiplication
        double scalar = 2.0;
        double[] scaled = new double[3];
        for (int i = 0; i < 3; i++) {
            scaled[i] = vectorA[i] * scalar;
        }
        System.out.print("vectorA scaled by 2: ");
        for (double v : scaled) System.out.print(v + " ");
        System.out.println();

        // Dot product
        double dotProduct = 0;
        for (int i = 0; i < 3; i++) {
            dotProduct += vectorA[i] * vectorB[i];
        }
        System.out.println("Dot product of vectorA and vectorB: " + dotProduct);
    }
}
```

Output:

```
Vector sum: 3.0 7.0 11.0
vectorA scaled by 2: 4.0 8.0 12.0
Dot product of vectorA and vectorB: 44.0
```

Example Program: java.util.Vector as a Dynamic Array

Example Program:

```
import java.util.Vector;

public class VectorClassDemo {
    public static void main(String[] args) {
        Vector<Integer> vector = new Vector<>();

        vector.add(10);
        vector.add(20);
        vector.add(30);
        System.out.println("Vector after adding elements: " + vector);
        System.out.println("Size of vector: " + vector.size());

        vector.add(1, 15);   // insert at index 1
        System.out.println("After inserting 15 at index 1: " + vector);

        vector.remove(Integer.valueOf(30));   // remove by value
        System.out.println("After removing value 30: " + vector);

        System.out.println("Element at index 0: " + vector.get(0));
        System.out.println("Does vector contain 15? " + vector.contains(15));
    }
}
```

Output:

```
Vector after adding elements: [10, 20, 30]
Size of vector: 3
After inserting 15 at index 1: [10, 15, 20, 30]
After removing value 30: [10, 15, 20]
Element at index 0: 10
Does vector contain 15? true
```

Array vs. Vector — Key Differences

- An array has a fixed size decided at creation time; a Vector grows and shrinks dynamically as elements are added or removed.
- An array can hold primitive types directly; a Vector can only hold objects (primitives are auto-boxed, e.g. int becomes Integer).
- Array element access with [] is slightly faster since it involves no method call overhead; Vector access uses method calls such as get() and set().
- Vector's methods are synchronized (thread-safe) by default, which arrays are not — array thread safety must be handled manually by the programmer.
- Vector provides many convenience methods (add, remove, contains, indexOf, etc.) that must be written manually for arrays.

Key Points to Remember

- A one-dimensional array can directly represent a mathematical vector and support operations such as addition and dot product.
- java.util.Vector is a legacy, synchronized, dynamically-growing array-backed class in the Java Collections Framework.
- Unlike arrays, a Vector automatically manages its own capacity and never needs to be manually resized by the programmer.
- Vector should be preferred over a plain array only when thread safety or dynamic resizing is genuinely required, since it carries extra overhead.

3.15 Solved Programming Examples

This section brings together several additional worked programs that are frequently asked in examinations and that reinforce the concepts covered throughout this unit — array traversal, conditional logic, and multi-dimensional arrays.

Program 1: Reverse an Array In-Place

Reversing an array in place is achieved by swapping elements from both ends of the array, moving the two index pointers toward the centre until they meet.

Example Program:

```
public class ReverseArrayDemo {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int start = 0, end = arr.length - 1;
```

```
        while (start < end) {
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }

        System.out.print("Reversed array: ");
        for (int val : arr) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

Output:

```
Reversed array: 50 40 30 20 10
```

Program 2: Check Whether an Array Is a Palindrome

An array is said to be a palindrome if it reads the same from both ends — that is, element i equals element (n − 1 − i) for every valid index i.

Example Program:

```
public class ArrayPalindromeDemo {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 1};
        boolean isPalindrome = true;

        for (int i = 0; i < arr.length / 2; i++) {
            if (arr[i] != arr[arr.length - 1 - i]) {
                isPalindrome = false;
                break;
            }
        }

        if (isPalindrome) {
            System.out.println("The array is a palindrome.");
        } else {
```

```
            System.out.println("The array is NOT a palindrome.");
        }
    }
}
```

Output:

```
The array is a palindrome.
```

Program 3: Merge Two Sorted Arrays into One Sorted Array

Two already-sorted arrays can be merged into a single sorted array efficiently in O(m + n) time by comparing the front elements of both arrays and repeatedly picking the smaller one, without needing to re-sort the combined result.

Example Program:

```
public class MergeSortedArraysDemo {
    public static void main(String[] args) {
        int[] a = {1, 3, 5, 7};
        int[] b = {2, 4, 6, 8, 10};
        int[] merged = new int[a.length + b.length];

        int i = 0, j = 0, k = 0;
        while (i < a.length && j < b.length) {
            if (a[i] <= b[j]) {
                merged[k++] = a[i++];
            } else {
                merged[k++] = b[j++];
            }
        }
        while (i < a.length) merged[k++] = a[i++];
        while (j < b.length) merged[k++] = b[j++];

        System.out.print("Merged sorted array: ");
        for (int val : merged) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

Output:

```
Merged sorted array: 1 2 3 4 5 6 7 8 10
```

Program 4: Student Marks Using a Two-Dimensional Array with Row-wise Totals

A very common practical use of two-dimensional arrays in academic settings is to store the marks obtained by several students across several subjects, with rows representing students and columns representing subjects, and to compute each student's total and average.

Example Program:

```
public class StudentMarksDemo {
    public static void main(String[] args) {
        String[] students = {"Ravi", "Priya", "Karthik"};
        int[][] marks = {
            {78, 85, 90},   // Ravi: Subject1, Subject2, Subject3
            {88, 76, 95},   // Priya
            {65, 70, 80}    // Karthik
        };

        for (int i = 0; i < marks.length; i++) {
            int total = 0;
            for (int j = 0; j < marks[i].length; j++) {
                total += marks[i][j];
            }
            double average = (double) total / marks[i].length;
            System.out.println(students[i] + " -> Total: " + total + ", Average:
" + average);
        }
    }
}
```

Output:

```
Ravi -> Total: 253, Average: 84.33333333333333
Priya -> Total: 259, Average: 86.33333333333333
Karthik -> Total: 215, Average: 71.66666666666667
```

Program 5: Count the Frequency of Each Element in an Array

Frequency counting is a common array-processing task in which the program determines how many times each distinct value occurs, which is useful in applications such as finding duplicates or computing a histogram of the data.

Example Program:

```
public class FrequencyCountDemo {
    public static void main(String[] args) {
        int[] arr = {4, 2, 4, 5, 2, 4, 1};
        boolean[] visited = new boolean[arr.length];

        for (int i = 0; i < arr.length; i++) {
            if (visited[i]) continue;
            int count = 1;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j]) {
                    visited[j] = true;
                    count++;
                }
            }
            System.out.println(arr[i] + " occurs " + count + " time(s)");
        }
    }
}
```

Output:

```
4 occurs 3 time(s)
2 occurs 2 time(s)
5 occurs 1 time(s)
1 occurs 1 time(s)
```

Program 6: Multiply Two Matrices Using Two-Dimensional Arrays

Matrix multiplication is a classic two-dimensional array problem. For two matrices A (of size m x n) and B (of size n x p) to be multiplied, the number of columns of A must equal the number of rows of B, and the resulting matrix C has size m x p, where each element C[i][j] is computed as the sum of products of the corresponding row of A and column of B.

Example Program:

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

```
        int[][] a = {
            {1, 2},
            {3, 4}
        };
        int[][] b = {
            {5, 6},
            {7, 8}
        };
        int m = a.length, n = a[0].length, p = b[0].length;
        int[][] product = new int[m][p];

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < p; j++) {
                int sum = 0;
                for (int k = 0; k < n; k++) {
                    sum += a[i][k] * b[k][j];
                }
                product[i][j] = sum;
            }
        }

        System.out.println("Product of the two matrices:");
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < p; j++) {
                System.out.print(product[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
Product of the two matrices:
19 22
43 50
```

Program 7: Sum of Diagonal Elements of a Square Matrix

For a square matrix (equal number of rows and columns), the primary diagonal consists of elements where the row index equals the column index (arr[i][i]), and the secondary diagonal consists of elements where the column index equals (n − 1 − i). Summing these diagonals is a common examination problem that reinforces indexing skills in two-dimensional arrays.

Example Program:

```
public class MatrixDiagonalSumDemo {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };
        int n = matrix.length;
        int primaryDiagonalSum = 0, secondaryDiagonalSum = 0;

        for (int i = 0; i < n; i++) {
            primaryDiagonalSum += matrix[i][i];
            secondaryDiagonalSum += matrix[i][n - 1 - i];
        }

        System.out.println("Sum of primary diagonal: " + primaryDiagonalSum);
        System.out.println("Sum of secondary diagonal: " + secondaryDiagonalSum);
    }
}
```

Output:

```
Sum of primary diagonal: 15
Sum of secondary diagonal: 15
```

Unit III — Summary

An array is Java's fundamental fixed-size, homogeneous data structure for storing multiple values of the same type under a single variable name, with elements stored contiguously on the heap and accessed in constant time through zero-based indexing. Arrays must be declared and then created with the 'new' operator (or initialized directly with an initializer list), and every array carries a public 'length' field describing its capacity. Because an array variable holds only a reference, assigning one array variable to another shares the same underlying data; independent copies require clone(), System.arraycopy(), or the convenience methods of the java.util.Arrays class.

Since array size is fixed once created, 'resizing' is simulated by creating a new array of the desired size and copying the required elements across, most conveniently with Arrays.copyOf(). Standard algorithms for organizing and locating data — bubble sort, selection sort, insertion sort, linear search, and binary search — were studied in both hand-written and library form via Arrays.sort() and

Arrays.binarySearch(). Multi-dimensional arrays (two-dimensional and three-dimensional) extend the same principles to represent tabular and volumetric data, while jagged arrays relax the requirement that every row have equal length. Finally, the array was connected to the mathematical notion of a vector, and contrasted with java.util.Vector, a legacy synchronized class that automatically manages its own resizing.

Glossary of Key Terms

Array — A fixed-size collection of elements of the same data type, stored contiguously and accessed by index.

Index — The position of an element within an array, starting from 0 and ending at length − 1.

ArrayIndexOutOfBoundsException — A run-time exception thrown when an array is accessed using an invalid (out-of-range) index.

Shallow Copy — A copy in which the new array holds copies of the same references as the original, so referenced objects remain shared.

Deep Copy — A copy in which both the array and the objects it references are duplicated, producing a fully independent structure.

Jagged Array — A multi-dimensional array whose rows can have differing lengths.

Linear Search — An algorithm that checks each array element sequentially until the target value is found or the array ends.

Binary Search — An algorithm that repeatedly halves a sorted array's search range to locate a target value in O(log n) time.

Vector — A legacy, synchronized, dynamically resizable array-backed class in java.util.

Unit III — Question Bank

Part A — Short Answer Questions (2 Marks)

- 1. Define an array. Why are arrays needed in programming?
- 2. Differentiate between array declaration and array creation in Java.
- 3. What is the default value stored in an int array element that has not been explicitly initialized?
- 4. What is an ArrayIndexOutOfBoundsException? When does it occur?

- 5. What does the 'length' field of an array represent? How is it different from the length() method of String?
- 6. What is the difference between assigning one array to another and cloning an array?
- 7. Can the size of an array be changed after it is created in Java? Justify your answer.
- 8. What is a jagged array? Give one real-life example where a jagged array is useful.
- 9. State the time complexity of linear search and binary search.
- 10. What is the java.util.Arrays class used for? Name any two of its methods.

Part B — Medium Answer Questions (5 Marks)

- 1. Explain, with a suitable example program, the different ways of declaring and initializing an array in Java.
- 2. Describe how arrays are stored in computer memory in Java, and explain why array element access is a constant-time operation.
- 3. Write a Java program to insert and delete an element at a given position in an array, shifting the remaining elements appropriately.
- 4. Explain the difference between reference assignment and true copying of arrays in Java, with a program illustrating both.
- 5. Write a Java program to implement the bubble sort algorithm to sort an array of integers in ascending order.
- 6. Write a Java program to search for a given value in a sorted array using the binary search algorithm.
- 7. Explain any five commonly used methods of the java.util.Arrays class with suitable examples.
- 8. Write a Java program to add two matrices using two-dimensional arrays.

Part C — Long Answer / Programming Questions (10 Marks)

- 1. What is an array? Explain in detail the declaration, creation, and initialization of one-dimensional arrays in Java with example programs and outputs. Also explain how memory is allocated for arrays.
- 2. Explain the various operations that can be performed on array elements (traversal, insertion, deletion, updation and aggregate operations) with a complete Java program demonstrating each operation and its output.

- 3. Discuss how the size of an array can be dynamically changed in Java despite arrays having a fixed size. Compare this approach with using ArrayList/Vector, supporting your answer with example programs.
- 4. Explain two-dimensional and three-dimensional arrays in Java with their declaration, initialization, and element access. Write a Java program to demonstrate matrix addition using two-dimensional arrays.
- 5. What are jagged arrays? Explain how they differ from regular rectangular two-dimensional arrays. Write a Java program to create and display a jagged array.
- 6. Explain the concept of representing a vector using a one-dimensional array. Write Java programs to perform vector addition, scalar multiplication and dot product using arrays, and also demonstrate the use of the java.util.Vector class as a dynamic array.
- 7. Compare and contrast arrays and the java.util.Vector class in Java under the headings: size, type of elements stored, performance, thread-safety, and available methods. Support your explanation with example code for both.

Reference Textbooks

- Anitha Seth, B.L. Juneja — Java: One Step Ahead, Oxford University Press.
- Debasis Samanta, Monalisa Sarma — Joy with Java: Fundamentals of Object Oriented Programming, Cambridge, 2023.
- Paul Deitel, Harvey Deitel — Java for Programmers, 4th Edition, Pearson.
- Herbert Schildt — The Complete Reference Java, 11th Edition, TMH.
- Y. Daniel Liang — Introduction to Java Programming, 7th Edition, Pearson.

UNIT IV

Packages, Exception Handling & Java I/O

23CSE232 — Object Oriented Programming through Java

4.1 Introduction to Packages

A package in Java is a namespace mechanism used to group related classes, interfaces, and sub-packages together under a single umbrella name. Packages serve two main purposes: they help organize a large project into logical, manageable units, and they prevent naming conflicts by allowing two classes with the same simple name to coexist as long as they belong to different packages, since a class is uniquely identified by its fully qualified name (package name plus class name).

Java packages also provide a mechanism for controlling access. Members of a class that are declared without an access modifier (the 'default' or package-private access level) are visible only to other classes within the same package, which allows related classes to cooperate closely while still hiding implementation details from unrelated code elsewhere in the application. Packages are conceptually similar to folders in a file system, and in fact, the standard Java compiler expects the directory structure on disk to mirror the package hierarchy declared in the source code.

Java packages are broadly of two kinds: built-in packages, which are supplied by the Java platform itself (such as java.lang, java.util, java.io, and java.time), and user-defined packages, which a programmer creates to organize their own classes. Built-in packages are collectively referred to as the Java class library or the Java API, and a working knowledge of the most commonly used built-in packages is essential for productive Java programming.

Advantages of Using Packages

- Namespace management — avoids naming collisions between classes written by different programmers or organizations.
- Access protection — package-private members are accessible only within the same package, supporting encapsulation.
- Better organization — related classes and interfaces are grouped logically, making large codebases easier to navigate and maintain.
- Reusability — a well-designed package can be reused across multiple projects simply by importing it.

Key Points to Remember

- A package is Java's mechanism for namespace management, access control, and logical organization of related types.
- Built-in packages (java.lang, java.util, java.io, java.time, etc.) collectively form the Java class library / Java API.

- A class's fully qualified name (package + class name) is what guarantees uniqueness, not the simple class name alone.
- Package-private (default) access allows classes in the same package to cooperate closely while still hiding details from outside code.

4.2 Defining and Creating a Package

A package is created in Java using the 'package' keyword, which must appear as the very first non-comment statement in a source file. Every class declared in that file then automatically becomes a member of the named package. If a source file does not contain a package statement, all its classes belong to a nameless default package, which is acceptable only for small or trivial programs and should be avoided in real projects.

By convention, package names are written entirely in lowercase, and to guarantee global uniqueness across organizations, multi-word package names are often formed by reversing the organization's internet domain name, for example com.sitams.university. On disk, the directories must exactly mirror the package hierarchy — a class declared as 'package com.sitams.arrays;' must reside inside a directory path com/sitams/arrays/ relative to the project's source root, and this correspondence is mandatory for the compiler and JVM to locate the class correctly.

Syntax for Declaring a Package

Syntax:

```
package packageName;                // simple package
package org.company.moduleName;     // hierarchical (dotted) package name

// Must be the first statement in the source file (before imports and class
declarations)
```

Example Program: Creating and Using a User-Defined Package

The following example creates a simple package named 'shapes' containing a class Circle, and then accesses that class from another class located in the default package by using its fully qualified name.

Example Program:

```
// File: shapes/Circle.java
package shapes;
```

```
public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }
}

// File: PackageDemo.java  (default package)
public class PackageDemo {
    public static void main(String[] args) {
        shapes.Circle c = new shapes.Circle(5.0);
        System.out.println("Area of circle: " + c.area());
    }
}

/* Compilation and execution from the project root:
   javac shapes/Circle.java
   javac PackageDemo.java
   java PackageDemo
*/
```

Output:

```
Area of circle: 78.53981633974483
```

Sub-packages and Package Hierarchies

Java packages can be organized hierarchically using dots, forming sub-packages — for example, com.sitams.cse.arrays is a sub-package of com.sitams.cse, which is itself a sub-package of com.sitams. It is important to understand that Java treats each level of this hierarchy as a completely separate package for access-control purposes: importing a parent package (or using the wildcard 'com.sitams.*') does NOT automatically import its sub-packages, since sub-packages have no special inheritance relationship with their parent beyond the shared name prefix and matching directory structure.

Example Program:

```
// File: com/sitams/cse/util/Greeter.java
package com.sitams.cse.util;

public class Greeter {
    public static void greet(String name) {
        System.out.println("Hello, " + name + "! Welcome to SITAMS CSE.");
    }
}

// File: SubPackageDemo.java (default package)
import com.sitams.cse.util.Greeter;

public class SubPackageDemo {
    public static void main(String[] args) {
        Greeter.greet("Ajitha");
    }
}
```

Output:

```
Hello, Ajitha! Welcome to SITAMS CSE.
```

Key Points to Remember

- The 'package' statement must be the first non-comment line in a source file, and only one is allowed per file.
- The directory structure on disk must exactly mirror the declared package hierarchy for the compiler and JVM to locate classes.
- Classes with no explicit package statement belong to the unnamed default package, which should be avoided in real, multi-file projects.
- A sub-package (e.g., com.sitams.cse.util) is treated as an entirely distinct package from its parent (com.sitams.cse) for import and access purposes.

4.3 Importing Packages and Classes; Path and Class Path; Access Control

To use a class from a package without writing its fully qualified name every time, Java provides the 'import' statement, which must appear after the package statement (if any) but before any class declarations. A single class can be imported explicitly, or an entire package's public top-level classes

can be imported at once using the wildcard '*', although importing only the specific classes actually used is generally considered better practice for code clarity.

For the compiler and the Java Virtual Machine to locate classes and packages, two environment settings are important: the PATH variable, which tells the operating system where to find the executable programs javac and java themselves, and the CLASSPATH variable, which tells the JVM and compiler where to search for compiled .class files and packages that are not part of the standard library, such as user-defined packages or third-party libraries (typically supplied via JAR files). The classpath can be set through the CLASSPATH environment variable, or more commonly and more reliably specified per invocation with the -cp or -classpath command-line option.

Import Statement Syntax

Syntax:

```
import packageName.ClassName;      // import a single class
import packageName.*;              // import all public classes of a package
import static packageName.ClassName.memberName;  // static import of a member
```

Access Control Levels in Java

Java defines four access levels that control the visibility of classes, fields, methods, and constructors across packages: private (accessible only within the declaring class), default/package-private (accessible within the same package, used when no modifier is specified), protected (accessible within the same package, and also by subclasses in other packages), and public (accessible from anywhere).

Memory Representation:

```
Modifier     Same Class   Same Package   Subclass(other pkg)   Everywhere
private         Yes           No                No                No
(default)       Yes           Yes               No                No
protected       Yes           Yes               Yes               No
public          Yes           Yes               Yes               Yes
```

Example Program: Using Import Statements and Access Modifiers

Example Program:

```
import java.util.ArrayList;   // import a single class
import java.util.List;
```

```
public class ImportAccessDemo {
    public static void main(String[] args) {
        List<String> cities = new ArrayList<>();
        cities.add("Chittoor");
        cities.add("Tirupati");
        cities.add("Chennai");
        System.out.println("Cities: " + cities);
    }
}
```

Output:

```
Cities: [Chittoor, Tirupati, Chennai]
```

Key Points to Remember

- The package statement, if present, must be the first line of a source file; only one package statement is permitted per file.
- Wildcard imports (packageName.*) import only the classes of that exact package, not those of its sub-packages.
- CLASSPATH tells the JVM/compiler where to find compiled classes and JARs; PATH tells the OS where to find the java/javac executables.
- protected access is the only level that grants visibility to subclasses residing in a different package.

4.4 The java.lang Package: Object, Math, and Wrapper Classes

The java.lang package is automatically imported into every Java program without an explicit import statement, since it contains the classes most fundamental to the language itself, including Object, String, Math, System, Thread, and the wrapper classes for the primitive types. Understanding this package well is essential because nearly every Java program relies on it, directly or indirectly.

The Object Class

Object is the root of the Java class hierarchy — every class in Java, whether or not it explicitly extends anything, implicitly extends Object. This means every object automatically inherits several useful methods from Object, including toString() (returns a String representation), equals() (compares objects for equality), hashCode() (returns an integer hash code used by hash-based collections), and

getClass() (returns runtime type information). Classes commonly override toString() and equals() to provide meaningful, class-specific behavior instead of the default identity-based implementations.

Example Program:

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

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

    @Override
    public String toString() {
        return "Student{name='" + name + "', rollNo=" + rollNo + "}";
    }
}

public class ObjectClassDemo {
    public static void main(String[] args) {
        Student s1 = new Student("Ajitha", 101);
        System.out.println(s1);                         // calls overridden
toString()
        System.out.println("Class: " + s1.getClass().getName());
    }
}
```

Output:

```
Student{name='Ajitha', rollNo=101}
Class: Student
```

The Math Class

The Math class provides static methods and constants for performing common mathematical operations such as computing powers, square roots, trigonometric functions, absolute values, and rounding, along with the constants Math.PI and Math.E. Since all Math methods are static, they are called directly on the class name without creating an object.

Example Program:

```
public class MathClassDemo {
    public static void main(String[] args) {
        System.out.println("Square root of 81: " + Math.sqrt(81));
        System.out.println("2 raised to the power 10: " + Math.pow(2, 10));
        System.out.println("Absolute value of -25: " + Math.abs(-25));
        System.out.println("Maximum of 45 and 78: " + Math.max(45, 78));
        System.out.println("Rounded value of 4.6: " + Math.round(4.6));
        System.out.println("Value of PI: " + Math.PI);
    }
}
```

Output:

```
Square root of 81: 9.0
2 raised to the power 10: 1024.0
Absolute value of -25: 25
Maximum of 45 and 78: 78
Rounded value of 4.6: 5
Value of PI: 3.141592653589793
```

Wrapper Classes

Since Java's primitive types (int, char, double, boolean, etc.) are not objects, the language provides a corresponding wrapper class for each primitive type — Integer, Character, Double, Boolean, and so on — so that primitive values can be treated as objects when required, such as when storing them in collections like ArrayList, which can only hold objects. Wrapper classes also provide many useful static utility methods, such as Integer.parseInt() to convert a String to an int, and Integer.MAX_VALUE / Integer.MIN_VALUE to obtain the range of the type.

Syntax:

```
Primitive    Wrapper Class
int          Integer
char         Character
double       Double
boolean      Boolean
long         Long
float        Float
byte         Byte
short        Short
```

Auto-boxing and Auto-unboxing

Auto-boxing is the automatic conversion that the Java compiler performs when a primitive value is used where an object of the corresponding wrapper class is expected, for example when adding an int to a collection that stores Integer objects. Auto-unboxing is the reverse process — automatically converting a wrapper object back to its corresponding primitive value when needed, for instance when a wrapper object is used in an arithmetic expression. This feature, introduced in Java 5, removes the need for programmers to manually call methods like Integer.valueOf() and intValue() in most common situations.

Example Program:

```
import java.util.ArrayList;

public class AutoboxingDemo {
    public static void main(String[] args) {
        // Auto-boxing: int literal automatically becomes an Integer object
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(10);     // auto-boxing: int -> Integer
        numbers.add(20);
        numbers.add(30);

        // Auto-unboxing: Integer object automatically becomes an int
        int total = 0;
        for (int n : numbers) {   // auto-unboxing happens here
            total += n;
        }
        System.out.println("Numbers: " + numbers);
        System.out.println("Total: " + total);

        Integer boxedValue = 55;      // auto-boxing
        int unboxedValue = boxedValue; // auto-unboxing
        System.out.println("Boxed: " + boxedValue + ", Unboxed: " +
unboxedValue);
    }
}
```

Output:

```
Numbers: [10, 20, 30]
Total: 60
Boxed: 55, Unboxed: 55
```

Key Points to Remember

- java.lang is imported implicitly into every Java source file; explicit import is never required for its classes.
- Every class in Java implicitly extends Object and inherits toString(), equals(), hashCode(), and getClass().
- All Math class members are static and are always called as Math.methodName(...), never on an instance.
- Auto-boxing/unboxing allows primitives and their wrapper objects to be used almost interchangeably, but excessive unboxing of null wrapper references throws a NullPointerException.

4.5 The java.util Package: Enumeration, Random, and Formatter Classes

The java.util package is one of the richest packages in the Java class library, containing the Collections Framework (List, Set, Map and their implementations), utility classes for random number generation, date/calendar handling (largely superseded by java.time), string formatting, and the legacy Enumeration interface for traversing collections of elements.

The Enumeration Interface

Enumeration is a legacy interface (predating the modern Iterator interface) that defines two methods, hasMoreElements() and nextElement(), used to step through the elements of an older collection class such as Vector or Hashtable one at a time. Although largely replaced by the more capable Iterator interface in modern code, Enumeration is still encountered when working with legacy APIs such as Vector.

Example Program:

```
import java.util.Enumeration;
import java.util.Vector;

public class EnumerationDemo {
    public static void main(String[] args) {
        Vector<String> fruits = new Vector<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");

        Enumeration<String> e = fruits.elements();
```

```
        while (e.hasMoreElements()) {
            System.out.println(e.nextElement());
        }
    }
}
```

Output:

```
Apple
Banana
Mango
```

The Random Class

The Random class provides methods to generate pseudo-random numbers of various primitive types, such as nextInt(), nextDouble(), and nextBoolean(). An optional seed value may be supplied to the constructor to obtain a reproducible sequence of 'random' numbers, which is particularly useful for testing and debugging.

Example Program:

```
import java.util.Random;

public class RandomClassDemo {
    public static void main(String[] args) {
        Random random = new Random(42);   // fixed seed for reproducible output

        System.out.println("Random int (0-99): " + random.nextInt(100));
        System.out.println("Random double: " + random.nextDouble());
        System.out.println("Random boolean: " + random.nextBoolean());
    }
}
```

Output:

```
Random int (0-99): 63
Random double: 0.7275636800328681
Random boolean: false
```

The Formatter Class and Formatted Output

The Formatter class supports C-style formatted output through a format string containing conversion specifiers such as %d for integers, %f for floating-point numbers, and %s for strings. In practice, most programmers use the equivalent convenience methods String.format() or System.out.printf(), which internally use a Formatter, rather than instantiating a Formatter object directly.

Example Program:

```
public class FormatterDemo {
    public static void main(String[] args) {
        String name = "Ajitha";
        double salary = 55234.567;

        System.out.printf("Name: %s%n", name);
        System.out.printf("Salary: %.2f%n", salary);
        System.out.printf("Padded integer: [%5d]%n", 42);

        String formatted = String.format("%-10s|%10s", "Left", "Right");
        System.out.println(formatted);
    }
}
```

Output:

```
Name: Ajitha
Salary: 55234.57
Padded integer: [   42]
Left      |     Right
```

The StringTokenizer Class

java.util.StringTokenizer is a legacy utility class used to break a String into individual tokens based on a set of delimiter characters (whitespace, by default). Although the split() method of the String class (which uses regular expressions) and the Scanner class are generally preferred in modern code for their greater flexibility, StringTokenizer remains a simple, lightweight, and slightly faster option for straightforward tokenizing tasks and is still occasionally seen in legacy code and examinations.

Example Program:

```
import java.util.StringTokenizer;

public class StringTokenizerDemo {
```

```
    public static void main(String[] args) {
        String data = "Ajitha,CSE,SITAMS,Chittoor";
        StringTokenizer tokenizer = new StringTokenizer(data, ",");

        System.out.println("Number of tokens: " + tokenizer.countTokens());
        while (tokenizer.hasMoreTokens()) {
            System.out.println("Token: " + tokenizer.nextToken());
        }
    }
}
```

Output:

```
Number of tokens: 4
Token: Ajitha
Token: CSE
Token: SITAMS
Token: Chittoor
```


Key Points to Remember

- Enumeration is a legacy traversal interface predating Iterator; modern code generally prefers Iterator or the for-each loop.
- Random with a fixed seed always produces the same sequence of pseudo-random numbers, which is useful for reproducible testing.
- %d, %f, %s, and %n are among the most commonly used format conversion specifiers in printf-style formatting.
- printf() and String.format() both rely on the same underlying formatting rules used by the Formatter class.

4.6 Date and Time Handling: The java.time Package

Introduced in Java 8, the java.time package (often called the 'new Date and Time API') provides a modern, immutable, and thread-safe replacement for the older, error-prone java.util.Date and java.util.Calendar classes. Its key classes include LocalDate (a date without time or time zone), LocalTime (a time without date), LocalDateTime (a combination of both), and Instant (a single instantaneous point on the timeline, measured in seconds and nanoseconds from the epoch of 1 January 1970 UTC), along with the DateTimeFormatter class for formatting and parsing, and the TemporalAdjusters utility class for calculating dates such as 'the first Monday of next month.'

The Instant Class

java.time.Instant represents a machine-readable timestamp — an exact point on the UTC timeline — and is most suitable for recording event timestamps or measuring elapsed time, rather than for human-facing date display, since Instant has no concept of time zones or calendar fields like 'day of month'.

Example Program:

```
import java.time.Instant;
import java.time.Duration;

public class InstantDemo {
    public static void main(String[] args) throws InterruptedException {
        Instant start = Instant.now();
        Thread.sleep(50);          // simulate some work
        Instant end = Instant.now();

        System.out.println("Start instant: " + start);
        Duration elapsed = Duration.between(start, end);
        System.out.println("Elapsed time (ms): " + elapsed.toMillis());
    }
}
```

Output:

```
Start instant: 2026-01-14T05:12:41.203145Z
Elapsed time (ms): 52
```

Temporal Adjusters Class

The TemporalAdjusters utility class (in java.time.temporal) provides ready-made adjuster objects for common date calculations that would otherwise require manual, error-prone arithmetic — for example, finding the first day of the next month, the last day of the current month, or the next occurrence of a particular day of the week.

Example Program:

```
import java.time.LocalDate;
import java.time.DayOfWeek;
import java.time.temporal.TemporalAdjusters;

public class TemporalAdjustersDemo {
```

```
    public static void main(String[] args) {
        LocalDate today = LocalDate.of(2026, 7, 13);   // a fixed reference date

        LocalDate firstDayNextMonth =
today.with(TemporalAdjusters.firstDayOfNextMonth());
        LocalDate lastDayOfMonth =
today.with(TemporalAdjusters.lastDayOfMonth());
        LocalDate nextMonday =
today.with(TemporalAdjusters.next(DayOfWeek.MONDAY));

        System.out.println("Today: " + today);
        System.out.println("First day of next month: " + firstDayNextMonth);
        System.out.println("Last day of this month: " + lastDayOfMonth);
        System.out.println("Next Monday: " + nextMonday);
    }
}
```

Output:

```
Today: 2026-07-13
First day of next month: 2026-08-01
Last day of this month: 2026-07-31
Next Monday: 2026-07-20
```

Formatting Date and Time with DateTimeFormatter

The DateTimeFormatter class converts date/time objects to and from human-readable String representations using a pattern of letters, such as 'dd-MM-yyyy' or 'HH:mm:ss', broadly analogous to (but distinct from) the older SimpleDateFormat pattern syntax used with java.util.Date.

Example Program:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateTimeFormatDemo {
    public static void main(String[] args) {
        LocalDateTime dateTime = LocalDateTime.of(2026, 7, 13, 14, 30, 0);

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy
HH:mm:ss");
        String formatted = dateTime.format(formatter);
```

```
        System.out.println("Default toString(): " + dateTime);
        System.out.println("Custom formatted: " + formatted);
    }
}
```

Output:

```
Default toString(): 2026-07-13T14:30
Custom formatted: 13-07-2026 14:30:00
```

Key Points to Remember

- java.time classes (LocalDate, LocalTime, LocalDateTime, Instant) are all immutable and thread-safe, unlike the legacy Date/Calendar classes.
- Instant represents a machine timestamp with no calendar or time-zone context; use LocalDate/LocalDateTime for human-facing dates.
- TemporalAdjusters removes the need to write manual arithmetic for common date calculations like 'next Monday' or 'last day of month'.
- DateTimeFormatter.ofPattern() defines a custom textual layout for converting date/time objects to and from Strings.

4.7 Introduction to Exception Handling

An exception is an event that disrupts the normal flow of a program's instructions during execution, typically arising from an abnormal condition such as invalid user input, division by zero, an attempt to access an array with an invalid index, or a file that cannot be found. Without a mechanism to handle such conditions, an unexpected error would cause the entire program to terminate abruptly, which is unacceptable for any real-world, reliable application.

Java's exception-handling mechanism allows a program to detect an abnormal condition, transfer control to a dedicated block of code designed to deal with that condition, and — where possible — resume execution afterward in a controlled manner, rather than crashing. This is accomplished through five keywords: try (marks a block of code to be monitored for exceptions), catch (defines a handler for a specific type of exception), finally (defines a block that always executes, whether or not an exception occurred), throw (used to explicitly raise an exception), and throws (used in a method signature to declare that the method may propagate a particular exception to its caller).

Why Exception Handling Is Important

- It separates error-handling code from the regular business logic, improving readability and maintainability.
- It allows a program to recover gracefully from an error condition instead of terminating abruptly.
- It allows different types of errors to be handled differently, using multiple catch blocks.
- It supports propagating an error up through a chain of method calls to a point where it can be meaningfully handled.

Example Program: Program Behaviour With and Without Exception Handling

The following program illustrates the difference in behaviour between code that does not handle a potential error and code that does. Without handling, an unexpected condition (dividing by zero) would abruptly terminate the program; with handling, the program detects the condition, reports it in a controlled way, and continues running normally afterward.

Example Program:

```
public class WithoutVsWithHandlingDemo {
    public static void main(String[] args) {
        int[] denominators = {2, 0, 5};

        for (int d : denominators) {
            try {
                int result = 100 / d;
                System.out.println("100 / " + d + " = " + result);
            } catch (ArithmeticException e) {
                System.out.println("Skipped 100 / " + d + " — " +
e.getMessage());
            }
        }
        System.out.println("Program completed all iterations without crashing.");
    }
}
```

Output:

```
100 / 2 = 50
Skipped 100 / 0 — / by zero
100 / 5 = 20
Program completed all iterations without crashing.
```

Key Points to Remember

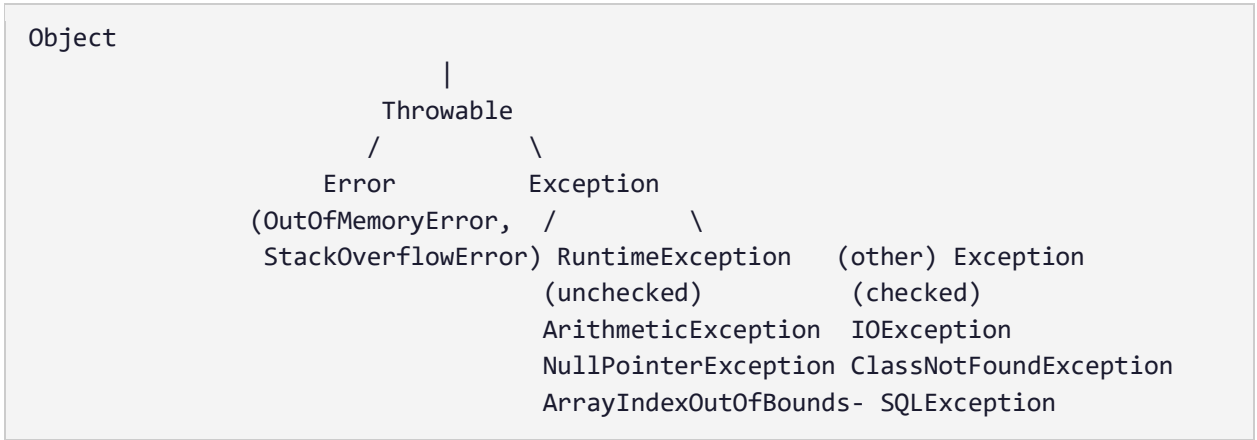
- An exception is an abnormal event that disrupts a program's normal instruction flow, typically caused by invalid data, resource unavailability, or logic errors.
- The five exception-handling keywords are try, catch, finally, throw, and throws, each with a distinct, well-defined role.
- Handling an exception allows a program to continue running after an error, rather than terminating the entire application abruptly.
- Good exception handling separates error-recovery logic from the main business logic, improving both readability and reliability.

4.8 Hierarchy of Standard Exception Classes; the Throwable Class

All exception and error types in Java are represented by classes, and all of these classes are ultimately descended from a single class named Throwable, which is itself a direct subclass of Object. Throwable has two major direct subclasses: Error, which represents serious problems that a reasonable application should generally not attempt to catch (such as OutOfMemoryError or StackOverflowError, usually caused by conditions outside the program's control), and Exception, which represents conditions that a well-written application should anticipate and handle. The Exception branch itself splits further into checked exceptions (subclasses of Exception other than RuntimeException, which the compiler forces the programmer to either catch or declare) and unchecked exceptions (subclasses of RuntimeException, which the compiler does not force the programmer to handle explicitly, though they usually indicate programming bugs such as invalid array indices or null references).

Exception Class Hierarchy Diagram

Memory Representation:



```
                                        Exception
                                    NumberFormatException
```

Common Methods Defined by the Throwable Class

- getMessage() — returns the detail message String describing the exception.
- toString() — returns a short description that includes the class name and detail message.
- printStackTrace() — prints the exception type, message, and the full call stack at the point the exception was thrown, which is invaluable for debugging.
- getCause() — returns the underlying cause of this exception, if it was triggered by another exception.

Example Program: Exploring the Throwable Hierarchy

Example Program:

```
public class ThrowableHierarchyDemo {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
            arr[5] = 10;   // throws ArrayIndexOutOfBoundsException
        } catch (Exception e) {
            System.out.println("Caught by type: " + e.getClass().getName());
            System.out.println("Is it a RuntimeException? " + (e instanceof
RuntimeException));
            System.out.println("Message: " + e.getMessage());
        }
    }
}
```

Output:

```
Caught by type: java.lang.ArrayIndexOutOfBoundsException
Is it a RuntimeException? true
Message: Index 5 out of bounds for length 3
```

Key Points to Remember

- Throwable is the superclass of all errors and exceptions; its two direct subclasses are Error and Exception.

- Error indicates serious problems outside normal application control and is generally not meant to be caught.
- RuntimeException and its subclasses are unchecked — the compiler does not force them to be caught or declared.
- All other subclasses of Exception are checked — the compiler forces the programmer to either catch them or declare them with 'throws'.

4.9 try, catch, and finally Blocks; Multiple Catch Clauses

The core of Java's exception-handling mechanism is the try-catch-finally construct. Code that might throw an exception is placed inside a try block. Immediately following the try block, one or more catch blocks may be provided, each specifying the type of exception it is prepared to handle; when an exception occurs inside the try block, control transfers to the first catch block whose parameter type matches (or is a superclass of) the thrown exception's type. An optional finally block, placed after all catch blocks, always executes — whether or not an exception was thrown, and even if a catch block itself returns or re-throws — making it the ideal place to release resources such as open files or network connections.

Syntax of try-catch-finally

Syntax:

```
try {
    // code that might throw an exception
} catch (ExceptionType1 e1) {
    // handle ExceptionType1
} catch (ExceptionType2 e2) {
    // handle ExceptionType2
} finally {
    // always executes (cleanup code)
}
```

Multiple Catch Clauses and Multi-Catch Syntax

A single try block can be followed by several catch blocks, each catching a different exception type, allowing different kinds of errors to be handled differently. Catch blocks are checked in the order they are written, so a more specific exception type must always be caught before a more general one (such as its superclass); placing a general catch block first would make the later, more specific catch blocks unreachable and cause a compile-time error. Since Java 7, the multi-catch syntax allows a

single catch block to handle several unrelated exception types at once, separated by the pipe symbol '|', when they should be handled identically.

Example Program: Multiple Catch Blocks and finally

Example Program:

```
public class MultiCatchDemo {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30};

        try {
            System.out.println("Result: " + (numbers[1] / 0));   //
ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error: " + e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index error: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed — cleanup complete.");
        }

        // Multi-catch syntax (Java 7+)
        try {
            String text = null;
            System.out.println(text.length());   // NullPointerException
        } catch (NullPointerException | ArithmeticException e) {
            System.out.println("Caught with multi-catch: " +
e.getClass().getSimpleName());
        }
    }
}
```

Output:

```
Arithmetic error: / by zero
Finally block executed — cleanup complete.
Caught with multi-catch: NullPointerException
```

Nested try Blocks and Guaranteed finally Execution

try blocks can be nested inside one another, and finally blocks are guaranteed to run for every enclosing try, in the order from innermost to outermost, even when the exception is not caught at the

inner level and propagates outward, which makes finally reliable for releasing resources allocated at each nesting level.

Example Program:

```
public class NestedTryDemo {
    public static void main(String[] args) {
        try {
            try {
                int result = 10 / 0;
            } finally {
                System.out.println("Inner finally executed.");
            }
        } catch (ArithmeticException e) {
            System.out.println("Outer catch handled: " + e.getMessage());
        } finally {
            System.out.println("Outer finally executed.");
        }
    }
}
```

Output:

```
Inner finally executed.
Outer catch handled: / by zero
Outer finally executed.
```

The try-with-resources Statement

Introduced in Java 7, try-with-resources is a special form of the try statement that automatically closes one or more resources when the block finishes, whether normally or due to an exception, eliminating the need to write an explicit finally block just to call close(). Any resource used in this way must implement the AutoCloseable (or Closeable) interface, which almost all of Java's I/O classes — including FileReader, FileWriter, Scanner, and the byte-stream classes — already do. Multiple resources can be declared in the same try statement, separated by semicolons, and they are closed automatically in the reverse order of their declaration.

Example Program:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
```

```
import java.io.IOException;

public class TryWithResourcesDemo {
    public static void main(String[] args) {
        String fileName = "twr_demo.txt";

        // Writing with a single resource
        try (FileWriter writer = new FileWriter(fileName)) {
            writer.write("Managed automatically by try-with-resources.");
        } catch (IOException e) {
            System.out.println("Write error: " + e.getMessage());
        }

        // Reading with a single resource — closed automatically, no explicit
finally needed
        try (BufferedReader reader = new BufferedReader(new
FileReader(fileName))) {
            System.out.println("File contents: " + reader.readLine());
        } catch (IOException e) {
            System.out.println("Read error: " + e.getMessage());
        }
        System.out.println("Both resources were closed automatically.");
    }
}
```

Output:

```
File contents: Managed automatically by try-with-resources.
Both resources were closed automatically.
```

Key Points to Remember

- A try block must be followed by at least one catch block, a finally block, or both.
- catch blocks are evaluated top-to-bottom; the more specific exception subclass must be caught before its superclass.
- finally always executes, making it the standard place to close files, network connections, or database handles.
- Multi-catch (ExceptionA | ExceptionB) reduces code duplication when several exception types require identical handling.

4.10 The throw and throws Keywords

While most exceptions in a program are thrown automatically by the Java runtime when an abnormal condition is detected (such as division by zero), a programmer can also deliberately raise (throw) an exception using the 'throw' keyword, typically to signal that a method's precondition has been violated or that some application-specific error condition has occurred. The 'throws' keyword, in contrast, is used in a method's signature — not inside its body — to declare that the method might propagate one or more checked exceptions to its caller, without handling them itself; this is a contractual obligation that forces every calling method to either catch the declared exception or declare it further up the call chain.

Syntax of throw and throws

Syntax:

```
// throw – used inside a method body to raise an exception
throw new ExceptionType("error message");

// throws – used in a method signature to declare possible checked exceptions
returnType methodName(parameters) throws ExceptionType1, ExceptionType2 {
    // method body
}
```

Creating and Using Custom (User-Defined) Exceptions

Beyond the exception classes built into the Java library, a programmer can define a custom exception class by extending Exception (for a checked exception) or RuntimeException (for an unchecked exception), which allows an application to represent domain-specific error conditions with a meaningful, self-documenting exception type rather than reusing a generic one.

Example Program:

```
// Custom checked exception
class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}

class BankAccount {
    private double balance;

    BankAccount(double balance) {
        this.balance = balance;
```

```
    }

    void withdraw(double amount) throws InsufficientBalanceException {
        if (amount > balance) {
            throw new InsufficientBalanceException(
                "Cannot withdraw " + amount + "; available balance is only " +
balance);
        }
        balance -= amount;
        System.out.println("Withdrawal successful. Remaining balance: " +
balance);
    }
}

public class CustomExceptionDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(5000.0);
        try {
            account.withdraw(2000.0);
            account.withdraw(4000.0);   // triggers custom exception
        } catch (InsufficientBalanceException e) {
            System.out.println("Transaction failed: " + e.getMessage());
        }
    }
}
```

Output:

```
Withdrawal successful. Remaining balance: 3000.0
Transaction failed: Cannot withdraw 4000.0; available balance is only 3000.0
```

Key Points to Remember

- 'throw' raises a single, specific exception instance at the exact point in the code where the error condition is detected.
- 'throws' appears in a method signature and only declares the possibility of a checked exception; it does not itself raise anything.
- A method that calls another method declaring 'throws' must either catch the declared exception or re-declare it with its own 'throws' clause.
- Custom exceptions should extend Exception for checked behavior or RuntimeException for unchecked behavior, depending on whether callers must be forced to handle them.

4.11 Checked Exceptions vs. Unchecked Exceptions

Java exceptions are classified into two categories based on whether the compiler enforces handling at compile time. Checked exceptions are conditions that a well-written program should anticipate and recover from, such as attempting to open a file that does not exist (FileNotFoundException) or a class that cannot be located at runtime (ClassNotFoundException); the compiler forces every method that might throw a checked exception to either catch it or declare it in a 'throws' clause, and code that fails to do so will not compile. Unchecked exceptions are subclasses of RuntimeException, such as ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, and NumberFormatException; these typically indicate a programming error, and the compiler does not require them to be caught or declared, although catching them where appropriate is still considered good practice for producing robust, user-friendly applications.

Checked vs. Unchecked Exceptions — Comparison

Memory Representation:

```
Aspect            Checked Exception          Unchecked Exception
Superclass        Exception (not Runtime-     RuntimeException
                  Exception)
Compiler check    Must be caught or declared  Not enforced by compiler
Typical cause     External conditions         Programming logic errors
                  (missing file, network)     (null reference, bad index)
Examples          IOException,                NullPointerException,
                  SQLException,               ArithmeticException,
                  ClassNotFoundException      ArrayIndexOutOfBoundsException
```

Example Program: Checked Exception Requiring Explicit Handling

The following program attempts to read a file that may not exist, which raises a checked FileNotFoundException that must be either caught or declared, unlike the unchecked exceptions seen in earlier examples.

Example Program:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

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

```
            File file = new File("nonexistent_data.txt");
            Scanner fileReader = new Scanner(file);   // may throw
FileNotFoundException
            while (fileReader.hasNextLine()) {
                System.out.println(fileReader.nextLine());
            }
            fileReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("Checked exception caught: " + e.getMessage());
        }
    }
}
```

Output:

```
Checked exception caught: nonexistent_data.txt (No such file or directory)
```


Key Points to Remember

- Checked exceptions must be handled or declared at compile time; forgetting to do so is a compilation error, not a run-time surprise.
- Unchecked exceptions (RuntimeException and its subclasses) are not checked by the compiler and usually signal a bug in the code.
- Error and its subclasses are neither checked nor typically caught, since they represent conditions the application usually cannot recover from.
- A good rule of thumb: use checked exceptions for recoverable external conditions and unchecked exceptions (or assertions) for programming errors.

4.12 Java I/O API and Byte Streams

The java.io package provides Java's classic input/output framework, built around the abstract concept of a stream — an ordered sequence of data flowing from a source (such as a file, the keyboard, or a network connection) to a destination (such as a file, the screen, or a network connection). Streams are broadly divided into two families based on the unit of data they handle: byte streams, which process raw binary data one byte (or block of bytes) at a time and are suitable for any kind of data including images and executable files; and character streams, discussed in the next section, which process text data as 16-bit Unicode characters and automatically handle character-encoding conversions.

The byte-stream classes are rooted at two abstract base classes: InputStream, for reading bytes, and OutputStream, for writing bytes. Commonly used concrete subclasses include FileInputStream and FileOutputStream (for reading from and writing to files), BufferedInputStream and BufferedOutputStream (which wrap another stream to add an internal buffer, reducing the number of expensive physical read/write operations), and ByteArrayInputStream / ByteArrayOutputStream (for reading from or writing to an in-memory byte array).

Core Methods of InputStream and OutputStream

- int read() — reads a single byte and returns it as an int (0–255), or −1 if the end of the stream has been reached.
- int read(byte[] buffer) — reads multiple bytes into the given array and returns the number of bytes actually read.
- void write(int b) — writes a single byte to the output stream.
- void write(byte[] buffer) — writes an entire array of bytes to the output stream.
- void close() — releases any system resources (such as file handles) associated with the stream; should always be called when done.
- void flush() — forces any buffered output data to be written out immediately.

Example Program: Writing to and Reading from a File Using Byte Streams

Example Program:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamDemo {
    public static void main(String[] args) {
        String fileName = "sample_bytes.txt";
        String content = "Hello, Byte Streams!";

        // Writing bytes to a file
        try (FileOutputStream fos = new FileOutputStream(fileName)) {
            fos.write(content.getBytes());
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            System.out.println("Error writing file: " + e.getMessage());
        }

        // Reading bytes from the file
```

```
        try (FileInputStream fis = new FileInputStream(fileName)) {
            int byteData;
            StringBuilder sb = new StringBuilder();
            while ((byteData = fis.read()) != -1) {
                sb.append((char) byteData);
            }
            System.out.println("Data read from file: " + sb.toString());
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Output:

```
Data written successfully.
Data read from file: Hello, Byte Streams!
```

The Standard Streams: System.out, System.err, and System.in

Every Java program has access to three pre-connected standard streams through the System class: System.out (a PrintStream connected to the standard output, normally the console, used by System.out.println()), System.err (a separate PrintStream connected to the standard error output, conventionally used for error and diagnostic messages so they can be redirected independently of normal output), and System.in (an InputStream connected to standard input, normally the keyboard, commonly wrapped in a Scanner for convenient reading).

Example Program:

```
public class StandardStreamsDemo {
    public static void main(String[] args) {
        System.out.println("This is normal output (System.out).");
        System.err.println("This is an error message (System.err).");

        // System.in is typically wrapped by a Scanner for convenient reading
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.println("Enter a number: ");
        // int n = sc.nextInt();  // would read from the keyboard in an
interactive run
    }
}
```

Output:

```
This is normal output (System.out).
This is an error message (System.err).
Enter a number:
```

Key Points to Remember

- Byte streams (InputStream/OutputStream family) handle raw binary data and are suitable for any file type, not just text.
- The try-with-resources construct (used above) automatically closes a stream when the try block finishes, even if an exception occurs.
- Buffered stream wrappers (BufferedInputStream/BufferedOutputStream) significantly improve performance by reducing physical I/O operations.
- Both InputStream.read() and reading past the end of a file return −1 to signal that no more data is available.

4.13 Character Streams and the Scanner Class

Character streams are designed specifically for handling text data, automatically translating between the internal 16-bit Unicode representation Java uses for characters and whatever byte-based encoding (such as UTF-8) is used by the external file or device. The character-stream classes are rooted at two abstract classes, Reader (for reading characters) and Writer (for writing characters), with common concrete subclasses including FileReader and FileWriter (for text files), BufferedReader and BufferedWriter (which add buffering and, in the case of BufferedReader, a convenient readLine() method), and InputStreamReader / OutputStreamWriter, which act as a bridge between byte streams and character streams.

Example Program: Reading and Writing Text Using Character Streams

Example Program:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CharacterStreamDemo {
```

```
    public static void main(String[] args) {
        String fileName = "sample_text.txt";

        // Writing text using BufferedWriter
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(fileName))) {
            writer.write("Line 1: Character streams in Java.");
            writer.newLine();
            writer.write("Line 2: Handled through Reader and Writer classes.");
        } catch (IOException e) {
            System.out.println("Error writing file: " + e.getMessage());
        }

        // Reading text using BufferedReader
        try (BufferedReader reader = new BufferedReader(new
FileReader(fileName))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Output:

```
Line 1: Character streams in Java.
Line 2: Handled through Reader and Writer classes.
```

The Scanner Class for Console and File Input

java.util.Scanner is a widely used convenience class for reading and parsing text input, whether from the keyboard (System.in), a file, or a String, and provides typed methods such as nextInt(), nextDouble(), and next() (for a single token) as well as nextLine() (for an entire line), in addition to the hasNext... family of methods used to check whether more input of a given type is available before attempting to read it.

Syntax:

```
Scanner sc = new Scanner(System.in);   // read from the keyboard
Scanner sc = new Scanner(new File("data.txt"));  // read from a file
```

```
int n = sc.nextInt();
double d = sc.nextDouble();
String word = sc.next();
String line = sc.nextLine();
boolean more = sc.hasNextInt();
```

Example Program:

```
import java.util.Scanner;

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

        System.out.print("Enter your name: ");
        String name = sc.nextLine();

        System.out.print("Enter your age: ");
        int age = sc.nextInt();

        System.out.println("Hello " + name + ", you are " + age + " years old.");
        sc.close();
    }
}
/* Sample input:
Ajitha
34
*/
```

Output:

```
Enter your name: Enter your age: Hello Ajitha, you are 34 years old.
```

Key Points to Remember

- Character streams (Reader/Writer family) are the correct choice for text data, since they handle character-encoding conversion automatically.
- BufferedReader.readLine() returns null when the end of the stream is reached, which is the standard way to detect end-of-file while reading lines.

- Scanner is convenient for parsing typed tokens from console, file, or String input, but mixing nextInt()/nextDouble() with nextLine() requires care, since the numeric methods do not consume the trailing newline character.
- Always close streams and Scanner objects (or use try-with-resources) to release underlying system resources promptly.

4.14 Files in Java

The java.io.File class represents an abstract, platform-independent path to a file or directory on the file system, and provides methods for querying and manipulating the file system entry itself — such as checking whether it exists, whether it is a file or a directory, its size, and its last-modified time — as well as for creating, deleting, and renaming files and directories. It is important to note that a File object represents a path name, not the actual content of the file; reading or writing the content of a file still requires an appropriate stream class such as FileReader, FileWriter, FileInputStream, or FileOutputStream layered on top of it.

Commonly Used Methods of the File Class

- boolean exists() — returns true if the file or directory denoted by this path actually exists.
- boolean createNewFile() — creates a new, empty file at this path, if one does not already exist.
- boolean delete() — deletes the file or directory denoted by this path.
- boolean mkdir() / mkdirs() — creates a directory, or an entire chain of parent directories, respectively.
- long length() — returns the size of the file, in bytes.
- String[] list() / File[] listFiles() — returns the names (or File objects) of the entries contained in a directory.
- boolean isDirectory() / boolean isFile() — tests whether this path denotes a directory or a regular file.

Example Program: Working with the File Class

Example Program:

```
import java.io.File;
import java.io.IOException;

public class FileClassDemo {
    public static void main(String[] args) throws IOException {
```

```
        File file = new File("demo_notes.txt");

        if (file.createNewFile()) {
            System.out.println("File created: " + file.getName());
        } else {
            System.out.println("File already exists.");
        }

        System.out.println("Absolute path: " + file.getAbsolutePath());
        System.out.println("Exists? " + file.exists());
        System.out.println("Is a file? " + file.isFile());
        System.out.println("Is a directory? " + file.isDirectory());
        System.out.println("Size (bytes): " + file.length());

        File directory = new File("demo_folder");
        if (directory.mkdir()) {
            System.out.println("Directory created: " + directory.getName());
        }

        // Clean up
        file.delete();
        directory.delete();
        System.out.println("Cleanup complete.");
    }
}
```

Output:

```
File created: demo_notes.txt
Absolute path: /home/user/project/demo_notes.txt
Exists? true
Is a file? true
Is a directory? false
Size (bytes): 0
Directory created: demo_folder
Cleanup complete.
```

Modern File I/O with java.nio.file (Brief Overview)

Since Java 7, the java.nio.file package (NIO.2) offers a more powerful and flexible alternative to java.io.File, centered on the Path interface and the utility class Files, which provides convenient static methods such as Files.readAllLines(), Files.write(), Files.copy(), and Files.exists() that often accomplish in a single line what previously required several lines of stream-handling code. While

java.io.File remains widely used and fully supported, many modern Java codebases prefer java.nio.file for new file-handling code.

Example Program:

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.List;
import java.io.IOException;

public class NioFileDemo {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("nio_demo.txt");

        Files.write(path, List.of("First line via NIO", "Second line via NIO"),
                StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);

        List<String> lines = Files.readAllLines(path);
        lines.forEach(System.out::println);

        System.out.println("File size: " + Files.size(path) + " bytes");
        Files.delete(path);
    }
}
```

Output:

```
First line via NIO
Second line via NIO
File size: 45 bytes
```

Key Points to Remember

- A File object represents only a path name on the file system, not the file's content; content access requires a stream class.
- createNewFile(), mkdir(), and delete() all return a boolean indicating success, and should generally be checked rather than ignored.
- mkdirs() (with an 's') creates all necessary but nonexistent parent directories, while mkdir() creates only the final directory and fails if a parent is missing.

- The java.nio.file package (Path and Files) is the modern, recommended alternative to java.io.File for new code.

4.15 Solved Programming Examples

This section brings together several additional worked programs that combine packages, exception handling, and file I/O concepts studied in this unit, reflecting the kind of integrated problems frequently asked in examinations.

Program 1: Validating User Input with NumberFormatException

Converting a String to a number using methods such as Integer.parseInt() throws a NumberFormatException (an unchecked exception) when the String does not represent a valid number, which is a very common real-world validation scenario.

Example Program:

```
public class NumberFormatValidationDemo {
    public static void main(String[] args) {
        String[] inputs = {"42", "hello", "17", "3.14x"};

        for (String input : inputs) {
            try {
                int value = Integer.parseInt(input);
                System.out.println("Valid integer: " + value);
            } catch (NumberFormatException e) {
                System.out.println("Invalid integer input: \"" + input + "\"");
            }
        }
    }
}
```

Output:

```
Valid integer: 42
Invalid integer input: "hello"
Valid integer: 17
Invalid integer input: "3.14x"
```

Program 2: Copying a File's Contents Using Byte Streams

A common practical file-handling task is copying the contents of one file into another, which can be done efficiently by reading a block (buffer) of bytes at a time rather than one byte at a time.

Example Program:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopyDemo {
    public static void main(String[] args) {
        String source = "source_file.txt";
        String destination = "destination_file.txt";

        try (FileOutputStream fos = new FileOutputStream(source)) {
            fos.write("Data to be copied across files.".getBytes());
        } catch (IOException e) {
            System.out.println("Setup error: " + e.getMessage());
        }

        try (FileInputStream fis = new FileInputStream(source);
             FileOutputStream fos = new FileOutputStream(destination)) {
            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = fis.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);
            }
            System.out.println("File copied successfully.");
        } catch (IOException e) {
            System.out.println("Copy error: " + e.getMessage());
        }
    }
}
```

Output:

```
File copied successfully.
```

Program 3: Word Count in a Text File Using Scanner

Scanner's token-based reading makes it convenient to count the number of words in a text file, simply by repeatedly calling next() until no more tokens remain, using hasNext() to detect the end of input.

Example Program:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class WordCountDemo {
    public static void main(String[] args) throws IOException {
        String fileName = "wordcount_demo.txt";
        try (FileWriter fw = new FileWriter(fileName)) {
            fw.write("Java exception handling makes programs more robust and
reliable");
        }

        int wordCount = 0;
        try (Scanner sc = new Scanner(new File(fileName))) {
            while (sc.hasNext()) {
                sc.next();
                wordCount++;
            }
        }
        System.out.println("Total word count: " + wordCount);
    }
}
```

Output:

```
Total word count: 9
```

Program 4: Chained Exceptions — Wrapping a Lower-Level Exception

It is often good practice for a higher-level method to catch a low-level, technical exception and re-throw it wrapped inside a more meaningful, application-specific exception, while preserving the original exception as the 'cause' using the constructor that accepts a Throwable, so that no diagnostic information is lost.

Example Program:

```
class DataProcessingException extends Exception {
    public DataProcessingException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
public class ChainedExceptionDemo {
    static void processValue(String raw) throws DataProcessingException {
        try {
            int value = Integer.parseInt(raw);
            System.out.println("Processed value: " + value);
        } catch (NumberFormatException e) {
            throw new DataProcessingException("Failed to process input: " + raw,
e);
        }
    }

    public static void main(String[] args) {
        try {
            processValue("abc");
        } catch (DataProcessingException e) {
            System.out.println("Error: " + e.getMessage());
            System.out.println("Root cause: " + e.getCause());
        }
    }
}
```

Output:

```
Error: Failed to process input: abc
Root cause: java.lang.NumberFormatException: For input string: "abc"
```

Program 5: Reading Configuration Data with the Properties Class

java.util.Properties is a specialized Hashtable-based class (in java.util, working closely with java.io) commonly used to load and store simple key-value configuration data, often from a .properties file, using the convenient load() and store() methods together with a Reader or InputStream.

Example Program:

```
import java.io.FileWriter;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

public class PropertiesDemo {
    public static void main(String[] args) throws IOException {
        String fileName = "config.properties";
```

```
        // Writing configuration data
        try (FileWriter fw = new FileWriter(fileName)) {
            fw.write("college=SITAMS\ncourse=23CSE232\nsemester=3\n");
        }

        // Reading configuration data
        Properties props = new Properties();
        try (FileReader fr = new FileReader(fileName)) {
            props.load(fr);
        }

        System.out.println("College: " + props.getProperty("college"));
        System.out.println("Course: " + props.getProperty("course"));
        System.out.println("Semester: " + props.getProperty("semester"));
    }
}
```

Output:

```
College: SITAMS
Course: 23CSE232
Semester: 3
```

Unit IV — Summary

A package groups related classes and interfaces under a common namespace, prevents naming collisions through fully qualified names, and works together with Java's four access modifiers (private, default, protected, public) to control visibility across classes and packages. The java.lang package, imported implicitly everywhere, supplies the Object root class, the static Math utility class, and the wrapper classes that bridge primitives and objects through auto-boxing and auto-unboxing. The java.util package adds the legacy Enumeration interface, the Random class, and formatted-output support, while the modern java.time package (Instant, LocalDate/LocalDateTime, TemporalAdjusters, DateTimeFormatter) provides an immutable, thread-safe replacement for the older Date/Calendar classes.

Exception handling allows a Java program to detect and recover from abnormal run-time conditions using try, catch, and finally blocks, with throw used to raise an exception explicitly and throws used in a method signature to declare a checked exception that the method does not itself handle. Every exception and error descends from Throwable, which splits into Error (serious, generally unrecoverable problems) and Exception (conditions an application should anticipate), the latter

further dividing into checked exceptions (enforced by the compiler) and unchecked RuntimeException subclasses (not enforced). Finally, Java's I/O framework is built on the idea of streams: byte streams (InputStream/OutputStream) for binary data, character streams (Reader/Writer) for text data with automatic encoding conversion, the Scanner class for convenient typed parsing of console or file input, and the File class (along with the more modern java.nio.file package) for interacting with the underlying file system.

Glossary of Key Terms

Package — A namespace mechanism that groups related classes and interfaces and helps prevent naming conflicts.

Classpath — The set of locations the JVM and compiler search to locate compiled classes and packages not built into the platform.

Auto-boxing — The automatic conversion of a primitive value into an instance of its corresponding wrapper class.

Throwable — The root superclass of all exception and error types in Java.

Checked Exception — An exception type the compiler forces the programmer to either catch or declare using 'throws'.

Unchecked Exception — A subclass of RuntimeException that the compiler does not force the programmer to catch or declare.

Stream — An ordered sequence of data flowing between a source and a destination, forming the basis of Java's I/O model.

Byte Stream — A stream that reads or writes raw binary data one byte (or block of bytes) at a time.

Character Stream — A stream that reads or writes text data as Unicode characters, handling encoding conversion automatically.

Unit IV — Question Bank

Part A — Short Answer Questions (2 Marks)

- 1. Define a package. What is the syntax for declaring a package in Java?
- 2. Differentiate between the PATH and CLASSPATH environment variables.
- 3. List the four access modifiers in Java and state one context in which each is used.
- 4. What is auto-boxing? Give a short example.

- 5. What is the Throwable class? Name its two direct subclasses.
- 6. Differentiate between checked and unchecked exceptions with one example of each.
- 7. What is the purpose of the 'finally' block? Does it always execute?
- 8. Differentiate between the 'throw' and 'throws' keywords.
- 9. What is the difference between a byte stream and a character stream?
- 10. What does the File class represent? Does it give access to file content directly?

Part B — Medium Answer Questions (5 Marks)

- 1. Explain how to create and use a user-defined package in Java with a suitable example program.
- 2. Explain the Object class and describe any three methods it provides that are inherited by every Java class.
- 3. Write a Java program that demonstrates auto-boxing and auto-unboxing using an ArrayList of Integer.
- 4. Explain the standard Java exception class hierarchy with a neat diagram.
- 5. Write a Java program that uses multiple catch blocks to handle two different exception types.
- 6. Explain how a custom (user-defined) exception class is created and used, with an example program.
- 7. Write a Java program to write text to a file and read it back using character streams (BufferedReader/BufferedWriter).
- 8. Explain the commonly used methods of the java.io.File class with an example program.

Part C — Long Answer / Programming Questions (10 Marks)

- 1. What are packages in Java? Explain in detail how packages are created, imported, and how access control interacts with packages, supported by example programs and outputs.
- 2. Explain the java.lang package in detail, covering the Object class, the Math class, wrapper classes, and auto-boxing/auto-unboxing, with example programs for each.
- 3. Explain Java's exception-handling mechanism in detail, covering try, catch, finally, multiple catch clauses, and the throw/throws keywords, with complete example programs and outputs for each construct.
- 4. Discuss the difference between checked and unchecked exceptions in Java. Explain how to design and use a custom exception class, illustrating with a complete banking or similar example program.

- 5. Explain Java's I/O stream model in detail, covering byte streams and character streams, and write example programs demonstrating file reading and writing using both FileInputStream/FileOutputStream and BufferedReader/BufferedWriter.
- 6. Explain the Scanner class and the File class in Java with suitable example programs, and discuss how the modern java.nio.file package (Path and Files) compares with the traditional java.io.File approach.
- 7. Explain the modern java.time date and time API, covering the Instant class, TemporalAdjusters, and DateTimeFormatter, with example programs and outputs illustrating each.

Reference Textbooks

- Anitha Seth, B.L. Juneja — Java: One Step Ahead, Oxford University Press.
- Debasis Samanta, Monalisa Sarma — Joy with Java: Fundamentals of Object Oriented Programming, Cambridge, 2023.
- Paul Deitel, Harvey Deitel — Java for Programmers, 4th Edition, Pearson.
- Herbert Schildt — The Complete Reference Java, 11th Edition, TMH.
- Y. Daniel Liang — Introduction to Java Programming, 7th Edition, Pearson.

UNIT V

Strings, Multithreading, JDBC & JavaFX

23CSE232 — Object Oriented Programming through Java

5.1 Introduction to String Handling and the CharSequence Interface

Strings are among the most frequently used data types in any Java program, representing sequences of characters used for text such as names, messages, file paths, and user input. Unlike C, where a string is simply an array of characters terminated by a null byte, Java treats a string as a full-fledged object of the class java.lang.String, which brings with it a large set of built-in methods for creating, comparing, searching, and transforming text.

A defining characteristic of the String class in Java is that String objects are immutable — once a String object is created, its character content can never be changed. Any operation that appears to 'modify' a String, such as concatenation or converting it to uppercase, actually creates and returns a brand-new String object, leaving the original object untouched. This design choice enables important optimizations such as string pooling (interning), makes Strings inherently safe to share across multiple threads without synchronization, and allows Strings to be safely used as keys in hash-based collections such as HashMap, since their hash code never changes after creation.

The CharSequence Interface

java.lang.CharSequence is a common interface implemented by String, StringBuilder, and StringBuffer, representing a readable sequence of characters regardless of the underlying implementation. It declares a small set of fundamental methods — length(), charAt(int index), subSequence(int start, int end), and toString() — that any class representing textual data can implement, which allows methods to be written generically to accept any kind of character sequence rather than being tied to the concrete String class alone.

Syntax:

```
public interface CharSequence {
    int length();
    char charAt(int index);
    CharSequence subSequence(int start, int end);
    public String toString();
}
```

Example Program: CharSequence as a Common Type

Example Program:

```
public class CharSequenceDemo {
    static void printInfo(CharSequence cs) {
```

```
        System.out.println("Value: " + cs + " | Length: " + cs.length()
                + " | Char at 0: " + cs.charAt(0));
    }

    public static void main(String[] args) {
        String str = "Java";
        StringBuilder sb = new StringBuilder("Programming");

        printInfo(str);   // String implements CharSequence
        printInfo(sb);    // StringBuilder implements CharSequence too
    }
}
```

Output:

```
Value: Java | Length: 4 | Char at 0: J
Value: Programming | Length: 11 | Char at 0: P
```


Key Points to Remember

- A String in Java is a full object of the String class, not a raw character array as in C.
- String objects are immutable — every apparent modification actually produces a brand-new String object.
- CharSequence is a common interface implemented by String, StringBuilder, and StringBuffer, enabling generic text-processing code.
- Immutability makes Strings inherently thread-safe and suitable as keys in hash-based collections.

5.2 The String Class: Creation and Immutability

Strings in Java can be created in two principal ways: using a string literal enclosed in double quotes, or using the 'new' keyword with the String constructor. These two approaches behave differently with respect to memory: string literals are stored in a special memory region called the String Constant Pool (part of the heap since Java 7), where the JVM automatically reuses an existing String object if an identical literal already exists, whereas 'new String(...)' always forces the creation of a distinct new object on the heap, even if an identical value already exists in the pool.

Ways of Creating Strings

Syntax:

```
String s1 = "Hello";                // string literal – may reuse a pooled
object
String s2 = new String("Hello");    // always creates a new object on the heap
String s3 = new String(charArray);  // from a char[] array
String s4 = String.valueOf(123);    // converting other types to a String
```

Example Program: String Literals, the String Pool, and 'new String()'

Example Program:

```
public class StringCreationDemo {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "Hello";                // reuses the same pooled object as
s1
        String s3 = new String("Hello");    // forces a new, separate object

        System.out.println("s1 == s2 (same pooled object)? " + (s1 == s2));
        System.out.println("s1 == s3 (new String() object)? " + (s1 == s3));
        System.out.println("s1.equals(s3) (content comparison)? " +
s1.equals(s3));

        // Demonstrating immutability
        String original = "Java";
        String upper = original.toUpperCase();
        System.out.println("Original unchanged: " + original);
        System.out.println("New object returned: " + upper);
    }
}
```

Output:

```
s1 == s2 (same pooled object)? true
s1 == s3 (new String() object)? false
s1.equals(s3) (content comparison)? true
Original unchanged: Java
New object returned: JAVA
```

Key Points to Remember

- String literals are stored in the string constant pool, so two identical literals share the same object reference.

- 'new String(...)' always creates a distinct object, even when its content duplicates an existing pooled string.
- '==' compares object references (identity); '.equals()' compares actual character content — always prefer equals() for content comparison.
- Because Strings are immutable, methods like toUpperCase(), concat(), and replace() all return a new String rather than modifying the original.

5.3 Extracting Characters from Strings

The String class provides several methods for extracting individual characters or sub-portions of a string's content, which is fundamental to almost any text-processing task such as parsing, validation, or formatting.

Key Extraction Methods

- char charAt(int index) — returns the character at the specified zero-based index.
- char[] toCharArray() — converts the entire String into a new character array.
- void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) — copies a range of characters into a supplied character array.
- String substring(int beginIndex) / String substring(int beginIndex, int endIndex) — extracts a portion of the string as a new String.

Example Program: Extracting Characters and Substrings

Example Program:

```
public class CharacterExtractionDemo {
    public static void main(String[] args) {
        String text = "SITAMS Chittoor";

        System.out.println("Character at index 0: " + text.charAt(0));
        System.out.println("Character at index 7: " + text.charAt(7));

        char[] chars = text.toCharArray();
        System.out.print("As char array: ");
        for (char c : chars) {
            System.out.print(c);
        }
        System.out.println();

        String sub1 = text.substring(7);          // from index 7 to end
```

```
        String sub2 = text.substring(0, 6);        // index 0 (inclusive) to 6
(exclusive)
        System.out.println("substring(7): " + sub1);
        System.out.println("substring(0, 6): " + sub2);
    }
}
```

Output:

```
Character at index 0: S
Character at index 7: C
As char array: SITAMS Chittoor
substring(7): Chittoor
substring(0, 6): SITAMS
```

Key Points to Remember

- charAt() uses zero-based indexing; the valid range is 0 to length() − 1, otherwise a StringIndexOutOfBoundsException is thrown.
- substring(begin, end) includes the character at 'begin' but excludes the character at 'end'.
- toCharArray() is useful whenever a method or algorithm requires direct character-by-character array access.
- All extraction methods return new objects or arrays; none of them modify the original String.

5.4 Comparing Strings

Because '==' compares references rather than content, Java provides a family of dedicated methods on the String class for comparing the actual textual content of two strings, along with methods for comparing them in a case-insensitive manner or determining their relative lexicographic (dictionary) order.

Common String Comparison Methods

- boolean equals(Object other) — returns true if both strings have identical content, character for character.
- boolean equalsIgnoreCase(String other) — like equals(), but ignores uppercase/lowercase differences.
- int compareTo(String other) — returns a negative number, zero, or a positive number depending on the strings' lexicographic order.

- int compareToIgnoreCase(String other) — like compareTo(), ignoring case differences.
- boolean startsWith(String prefix) / boolean endsWith(String suffix) — check whether a string begins or ends with a given substring.

Example Program: String Comparison Methods

Example Program:

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

        System.out.println("s1.equals(s2): " + s1.equals(s2));
        System.out.println("s1.equalsIgnoreCase(s2): " +
s1.equalsIgnoreCase(s2));
        System.out.println("s1.compareTo(\"Python\"): " +
s1.compareTo("Python"));
        System.out.println("s3.startsWith(\"Java\"): " + s3.startsWith("Java"));
        System.out.println("s1.endsWith(\"va\"): " + s1.endsWith("va"));
    }
}
```

Output:

```
s1.equals(s2): false
s1.equalsIgnoreCase(s2): true
s1.compareTo("Python"): -6
s3.startsWith("Java"): true
s1.endsWith("va"): true
```

Key Points to Remember

- '==' checks reference identity, not content; always use equals() to compare string content, even for literals.
- compareTo() returns a value based on the difference between the Unicode values of the first differing characters, not simply −1/0/1.
- equalsIgnoreCase() and compareToIgnoreCase() are useful for case-insensitive matching such as validating user input.

- startsWith() and endsWith() are frequently used for simple pattern checks such as validating file extensions or URL prefixes.

5.5 Modifying and Transforming Strings

Although String objects themselves are immutable, the String class offers a rich set of methods that compute and return a new, transformed String based on the original — covering concatenation, case conversion, whitespace trimming, character/substring replacement, and padding — which together cover the vast majority of everyday text-manipulation needs.

Common String Transformation Methods

- String concat(String other) — appends the given string, returning a new combined String (the '+' operator has the same effect).
- String toUpperCase() / String toLowerCase() — returns a new String converted to the specified case.
- String trim() / String strip() — returns a new String with leading and trailing whitespace removed.
- String replace(char oldChar, char newChar) — returns a new String with every occurrence of oldChar replaced by newChar.
- String replace(CharSequence target, CharSequence replacement) — replaces every occurrence of a substring.
- static String join(CharSequence delimiter, CharSequence... elements) — joins multiple strings with a given delimiter.

Example Program: Modifying and Transforming Strings

Example Program:

```
public class StringModificationDemo {
    public static void main(String[] args) {
        String greeting = "  Hello, World!  ";

        System.out.println("Trimmed: [" + greeting.trim() + "]");
        System.out.println("Uppercase: " + greeting.trim().toUpperCase());
        System.out.println("Lowercase: " + greeting.trim().toLowerCase());

        String replaced = greeting.trim().replace("World", "SITAMS");
        System.out.println("Replaced: " + replaced);
```

```
        String concatenated = "Object".concat("Oriented").concat("Programming");
        System.out.println("Concatenated: " + concatenated);

        String joined = String.join("-", "2026", "07", "13");
        System.out.println("Joined: " + joined);
    }
}
```

Output:

```
Trimmed: [Hello, World!]
Uppercase: HELLO, WORLD!
Lowercase: hello, world!
Replaced: Hello, SITAMS!
Concatenated: ObjectOrientedProgramming
Joined: 2026-07-13
```


Key Points to Remember

- Every String transformation method returns a new object; the original String is never altered in place.
- trim() removes ASCII whitespace only; the newer strip() (Java 11+) is Unicode-aware and handles a broader range of whitespace characters.
- Chaining method calls (e.g., greeting.trim().toUpperCase()) is common and efficient since each call simply operates on the previous result.
- String.join() is a convenient static method for combining multiple strings with a common separator, avoiding manual loop-based concatenation.

5.6 Searching within Strings

Locating a character or substring within a larger string is a frequent requirement, and the String class provides several searching methods that return either the position of a match or a boolean indicating whether a match exists at all.

Common String Searching Methods

- int indexOf(String str) — returns the index of the first occurrence of str, or −1 if not found.
- int indexOf(String str, int fromIndex) — searches starting from a given index.
- int lastIndexOf(String str) — returns the index of the last occurrence of str.

- boolean contains(CharSequence sequence) — returns true if the string contains the given sequence anywhere.
- boolean matches(String regex) — returns true if the entire string matches the given regular expression.

Example Program: Searching within Strings

Example Program:

```
public class StringSearchDemo {
    public static void main(String[] args) {
        String sentence = "Java is powerful. Java is popular.";

        System.out.println("First 'Java' at index: " + sentence.indexOf("Java"));
        System.out.println("Last 'Java' at index: " +
sentence.lastIndexOf("Java"));
        System.out.println("Second 'Java' search from index 5: "
                + sentence.indexOf("Java", 5));
        System.out.println("Contains 'powerful'? " +
sentence.contains("powerful"));
        System.out.println("Contains 'weak'? " + sentence.contains("weak"));

        String email = "student@sitams.ac.in";
        System.out.println("Looks like a simple email? "
                + email.matches("^[\\w.]+@[\\w.]+\\.[a-z]+$"));
    }
}
```

Output:

```
First 'Java' at index: 0
Last 'Java' at index: 18
Second 'Java' search from index 5: 18
Contains 'powerful'? true
Contains 'weak'? false
Looks like a simple email? true
```

Key Points to Remember

- indexOf() and lastIndexOf() return −1 when no match is found; this must always be checked before using the result as an index.
- contains() is a convenient shortcut equivalent to checking whether indexOf() returns a value other than −1.

- matches() checks the entire string against a regular expression, unlike contains(), which checks for a match anywhere within the string.
- Search methods are case-sensitive by default; the string (and/or the search target) must be normalized with toLowerCase()/toUpperCase() for case-insensitive searching.

5.7 The StringBuffer Class

Since String objects are immutable, repeatedly modifying text using String concatenation in a loop is inefficient, because every '+' operation silently creates a brand-new String object and discards the previous one, wasting both time and memory. For situations that require frequent, in-place modification of text — such as building up a large string incrementally — Java provides the StringBuffer class (and its non-synchronized, generally faster counterpart StringBuilder, introduced in Java 5), both of which represent a mutable sequence of characters that can be efficiently appended to, inserted into, deleted from, and reversed without creating a new object for every change.

StringBuffer is synchronized, meaning its methods are thread-safe and can be safely called from multiple threads concurrently, at the cost of some performance overhead; StringBuilder offers an identical API but without synchronization, making it faster in single-threaded contexts, which is the vast majority of everyday use. In modern Java code, StringBuilder is generally preferred unless the mutable string object genuinely needs to be shared safely across multiple threads.

Common Methods of StringBuffer / StringBuilder

- StringBuffer append(...) — adds the given value to the end of the character sequence and returns the same object (enabling method chaining).
- StringBuffer insert(int offset, ...) — inserts the given value at the specified position.
- StringBuffer delete(int start, int end) / deleteCharAt(int index) — removes a range of characters or a single character.
- StringBuffer replace(int start, int end, String str) — replaces a range of characters with a given string.
- StringBuffer reverse() — reverses the order of characters in place.
- int capacity() / void ensureCapacity(int minimum) — query or grow the internal buffer's capacity.

Example Program: Building and Modifying Text with StringBuffer

Example Program:

```
public class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Java");

        sb.append(" Programming");                 // "Java Programming"
        sb.insert(4, " OOP");                       // "Java OOP Programming"
        System.out.println("After append and insert: " + sb);

        sb.replace(5, 8, "Object-Oriented");        // replaces "OOP"
        System.out.println("After replace: " + sb);

        sb.delete(0, 5);                            // removes "Java "
        System.out.println("After delete: " + sb);

        StringBuffer reversed = new StringBuffer("SITAMS").reverse();
        System.out.println("Reversed: " + reversed);

        System.out.println("Current length: " + sb.length());
        System.out.println("Current capacity: " + sb.capacity());
    }
}
```

Output:

```
After append and insert: Java OOP Programming
After replace: Java Object-Oriented Programming
After delete: Object-Oriented Programming
Reversed: SMATIS
Current length: 27
Current capacity: 21
```

String vs. StringBuffer vs. StringBuilder

Memory Representation:

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread safety	Safe (immutable)	Synchronized (safe)	Not synchronized
Performance	Slow for repeated modification	Slower than StringBuilder	Fastest for single-threaded use
Introduced in	JDK 1.0	JDK 1.0	JDK 5

Key Points to Remember

- StringBuffer (and StringBuilder) represent a mutable, resizable sequence of characters, unlike immutable String objects.
- Most StringBuffer methods return 'this', enabling fluent method chaining such as sb.append(...).insert(...).reverse().
- StringBuffer is synchronized (thread-safe); StringBuilder has an identical API but is not synchronized and is generally faster.
- For building strings inside loops, StringBuffer/StringBuilder should always be preferred over repeated String concatenation with '+'.

5.8 Introduction to Multithreaded Programming

A thread is the smallest independent unit of execution within a program — a single sequential flow of control that can run concurrently with other such flows inside the same process. Multithreading is the capability of a program to run several threads at the same time, sharing the same memory space (unlike separate processes, which have their own isolated memory), which makes threads much lighter-weight to create and switch between than full operating-system processes.

Java has supported multithreading as a core language feature from its very first release, reflecting its early design goal of being suitable for interactive, networked, and graphical applications where different tasks — such as responding to user input, downloading data, and updating a display — naturally need to happen concurrently rather than strictly one after another.

Why Multiple Threads Are Needed

- Responsiveness — a GUI application can continue responding to user actions while a lengthy operation (such as a file download) runs in a background thread.
- Better resource utilization — while one thread waits for a slow I/O operation, the CPU can be used productively by another thread.
- Simplifying certain designs — some problems (such as a server handling many simultaneous client connections) are naturally expressed as multiple concurrent tasks.
- Improved throughput on multi-core processors — independent tasks can be executed genuinely simultaneously, rather than one at a time.

Multithreading and Multi-core Processors

On a single-core processor, the operating system creates the illusion of concurrency by rapidly switching between threads (a technique called time-slicing or context switching), so that only one

thread actually executes at any given instant. On a modern multi-core (or multi-processor) system, however, genuinely separate threads can execute simultaneously on different cores, allowing a well-designed multithreaded Java program to achieve true parallelism and make full use of all the available processing power, which is one of the primary motivations for writing multithreaded code on today's hardware.

Key Points to Remember

- A thread is a lightweight, independent path of execution within a program, sharing memory with other threads in the same process.
- Multithreading improves responsiveness, resource utilization, and (on multi-core hardware) genuine parallel throughput.
- On a single core, threads share the CPU through time-slicing; on multiple cores, threads can run truly simultaneously.
- Java has built-in language and library support for threads, reflecting its original design for interactive and networked applications.

5.9 The Thread Class and the Main Thread; Creating New Threads

Every running Java program has at least one thread of execution automatically created and started by the JVM, called the main thread, which is the thread that executes the statements inside the main() method. Additional threads can be created explicitly by the programmer, either by extending the java.lang.Thread class and overriding its run() method, or — generally the preferred approach — by implementing the java.lang.Runnable functional interface and passing an instance to a Thread object's constructor. The second approach is preferred because Java supports only single inheritance of classes, so extending Thread directly uses up a class's one opportunity to extend another class, whereas implementing Runnable leaves that option free and also cleanly separates 'the task to be run' from 'the mechanism that runs it.'

Method 1: Creating a Thread by Extending the Thread Class

Example Program:

```
class MyThread extends Thread {
    private String taskName;

    MyThread(String taskName) {
        this.taskName = taskName;
    }
```

```
    @Override
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(taskName + " - iteration " + i
                    + " (Thread: " + Thread.currentThread().getName() + ")");
        }
    }
}

public class ThreadExtendsDemo {
    public static void main(String[] args) {
        System.out.println("Main thread: " + Thread.currentThread().getName());

        MyThread t1 = new MyThread("Task-A");
        MyThread t2 = new MyThread("Task-B");
        t1.start();   // start(), NOT run() — start() creates a new call stack
        t2.start();
    }
}
```

Output:

```
Main thread: main
Task-A - iteration 1 (Thread: Thread-0)
Task-B - iteration 1 (Thread: Thread-1)
Task-A - iteration 2 (Thread: Thread-0)
Task-B - iteration 2 (Thread: Thread-1)
Task-A - iteration 3 (Thread: Thread-0)
Task-B - iteration 3 (Thread: Thread-1)
```

Method 2: Creating a Thread by Implementing the Runnable Interface

This approach is generally preferred because it does not consume the class's single inheritance slot and cleanly separates the task's logic from Java's threading mechanism.

Example Program:

```
class MyTask implements Runnable {
    private String taskName;

    MyTask(String taskName) {
        this.taskName = taskName;
```

```
    }

    @Override
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(taskName + " - step " + i);
        }
    }
}

public class RunnableDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyTask("Runnable-A"));
        Thread t2 = new Thread(new MyTask("Runnable-B"));
        t1.start();
        t2.start();

        // Java 8+: Runnable can also be supplied as a lambda expression
        Thread t3 = new Thread(() -> System.out.println("Lambda-based thread
running"));
        t3.start();
    }
}
```

Output:

```
Runnable-A - step 1
Runnable-B - step 1
Lambda-based thread running
Runnable-A - step 2
Runnable-B - step 2
Runnable-A - step 3
Runnable-B - step 3
```

Key Points to Remember

- The main() method itself always executes on a JVM-created thread known as the main thread.
- Calling start() creates a new call stack and eventually invokes run() on a separate thread; calling run() directly executes the code on the current thread, with no concurrency at all.
- Implementing Runnable (rather than extending Thread) is generally preferred, since it preserves the class's ability to extend another class.

- The exact interleaving/order of output from multiple started threads is not guaranteed and can vary between runs, since it depends on the operating system's thread scheduler.

5.10 Thread Life Cycle (States) and Thread Priority

A thread in Java moves through a well-defined sequence of states during its lifetime, represented by the enum Thread.State: NEW (created but not yet started), RUNNABLE (eligible to run, either currently executing or waiting for CPU time from the scheduler), BLOCKED (waiting to acquire a lock held by another thread), WAITING / TIMED_WAITING (paused, waiting indefinitely or for a specified time for another thread to perform a particular action, such as via wait() or sleep()), and TERMINATED (the thread has completed execution, whether normally or due to an uncaught exception).

Each thread also has an integer priority, ranging from Thread.MIN_PRIORITY (1) through the default Thread.NORM_PRIORITY (5) to Thread.MAX_PRIORITY (10), which serves as a hint to the thread scheduler about the relative importance of threads competing for CPU time. It is important to understand that thread priority is only a scheduling hint, not a guarantee — the exact effect of priority is platform-dependent, and a well-designed multithreaded program should never rely on priority alone to guarantee a particular execution order.

Thread Life Cycle Diagram

Memory Representation:

```
NEW  --start()-->  RUNNABLE  <---->  BLOCKED / WAITING / TIMED_WAITING
                        |
                   run() completes
                        |
                        v
                   TERMINATED
```

Example Program: Thread States and Priority

Example Program:

```
public class ThreadStatePriorityDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
```

```
                Thread.currentThread().interrupt();
            }
        }, "WorkerThread");

        System.out.println("State before start(): " + worker.getState());   //
NEW

        worker.setPriority(Thread.MAX_PRIORITY);
        System.out.println("Priority set to: " + worker.getPriority());

        worker.start();
        System.out.println("State just after start(): " + worker.getState()); //
RUNNABLE or TIMED_WAITING

        worker.join();   // wait for worker to finish before continuing
        System.out.println("State after completion: " + worker.getState());  //
TERMINATED
    }
}
```

Output:

```
State before start(): NEW
Priority set to: 10
State just after start(): TIMED_WAITING
State after completion: TERMINATED
```

Key Points to Remember

- The five key thread states are NEW, RUNNABLE, BLOCKED, WAITING/TIMED_WAITING, and TERMINATED.
- join() causes the calling thread to pause until the target thread has finished executing, which is useful for coordinating results between threads.
- Thread priority (1–10) is only a scheduling hint to the JVM/OS and does not guarantee any specific execution order.
- A thread cannot be restarted once it has reached the TERMINATED state; calling start() again on it throws an IllegalThreadStateException.

5.11 Thread Synchronization; Race Conditions and Deadlock

When multiple threads access and modify shared data concurrently without coordination, the final result can become unpredictable and incorrect — a problem known as a race condition, since the outcome depends on the unpredictable 'race' between threads to read and write the shared data. Java addresses this problem through synchronization, which ensures that only one thread at a time can execute a particular block of code or method on a given object, using the 'synchronized' keyword applied either to an entire method or to a specific block of code guarded by a chosen lock object (also called a monitor).

While synchronization solves race conditions, using it carelessly can introduce a different problem called deadlock, which occurs when two or more threads are each waiting indefinitely for a lock held by another thread in the group, so that none of them can ever proceed — for example, thread A holds lock 1 and waits for lock 2, while thread B holds lock 2 and waits for lock 1. Deadlock is typically avoided by always acquiring multiple locks in a fixed, consistent global order across all threads.

Example Program: Race Condition Without Synchronization

Example Program:

```
class Counter {
    private int count = 0;

    void increment() {              // NOT synchronized — vulnerable to a race
condition
        count++;
    }

    int getCount() {
        return count;
    }
}

public class RaceConditionDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 10000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
```

```
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Expected: 20000, Actual (often less due to race
condition): "
                + counter.getCount());
    }
}
```

Output:

```
Expected: 20000, Actual (often less due to race condition): 17364
```

Example Program: Fixing the Race Condition with 'synchronized'

Example Program:

```
class SafeCounter {
    private int count = 0;

    synchronized void increment() {   // synchronized – only one thread at a time
        count++;
    }

    int getCount() {
        return count;
    }
}

public class SynchronizedCounterDemo {
    public static void main(String[] args) throws InterruptedException {
        SafeCounter counter = new SafeCounter();
        Runnable task = () -> {
            for (int i = 0; i < 10000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
```

```
        t1.join();
        t2.join();

        System.out.println("Expected: 20000, Actual (correct with
synchronization): "
                + counter.getCount());
    }
}
```

Output:

```
Expected: 20000, Actual (correct with synchronization): 20000
```

Key Points to Remember

- A race condition occurs when the correctness of a result depends on the unpredictable timing/interleaving of multiple threads accessing shared data.
- The 'synchronized' keyword ensures that only one thread at a time can execute a guarded method or block on a given lock object.
- Deadlock occurs when two or more threads wait forever for locks held by each other, forming a circular waiting chain.
- A common strategy to avoid deadlock is to always acquire multiple locks in the same, fixed order across every thread in the program.

5.12 Inter-thread Communication: wait(), notify(), and Thread Control

Beyond simple mutual exclusion, threads sometimes need to actively coordinate with one another — for example, a 'consumer' thread may need to pause until a 'producer' thread has data ready for it. Java supports this kind of inter-thread communication through three methods defined on the Object class (available to every object, since locking is per-object): wait() (causes the current thread to release the lock and pause until another thread calls notify()/notifyAll() on the same object), notify() (wakes up one waiting thread), and notifyAll() (wakes up all threads waiting on that object). These methods may only be called from within a block of code synchronized on the object in question, otherwise an IllegalMonitorStateException is thrown at run time.

Older versions of the Thread class also defined suspend(), resume(), and stop() methods intended for directly pausing, resuming, and terminating a thread from outside; however, all three have been

formally deprecated since Java 1.2 because they are inherently unsafe — they can leave shared data in an inconsistent state or cause deadlocks by suspending a thread while it still holds a lock. Modern Java code achieves the same effects safely using flags checked cooperatively by the thread itself, combined with wait()/notify() or higher-level concurrency utilities from the java.util.concurrent package.

Example Program: Producer–Consumer Using wait() and notify()

Example Program:

```
class SharedBuffer {
    private int data;
    private boolean available = false;

    synchronized void produce(int value) throws InterruptedException {
        while (available) {
            wait();                 // wait until the consumer has taken the
previous value
        }
        data = value;
        available = true;
        System.out.println("Produced: " + value);
        notify();                   // wake up the waiting consumer
    }

    synchronized int consume() throws InterruptedException {
        while (!available) {
            wait();                 // wait until the producer has data ready
        }
        available = false;
        System.out.println("Consumed: " + data);
        notify();                   // wake up the waiting producer
        return data;
    }
}

public class ProducerConsumerDemo {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 3; i++) buffer.produce(i);
            } catch (InterruptedException e) {
Thread.currentThread().interrupt(); }
```

```
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 3; i++) buffer.consume();
            } catch (InterruptedException e) {
Thread.currentThread().interrupt(); }
        });

        producer.start();
        consumer.start();
    }
}
```

Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
```

Safely Stopping a Thread Using a Cooperative Flag

Since Thread.stop() is deprecated and unsafe, the recommended pattern for stopping a thread is to use a shared boolean 'flag' (often declared volatile so that updates are immediately visible to all threads) that the thread's run() method checks periodically, exiting its loop gracefully when the flag is set.

Example Program:

```
public class SafeStopDemo {
    private static volatile boolean running = true;

    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            int count = 0;
            while (running) {
                count++;
            }
            System.out.println("Worker stopped gracefully after " + count + "
iterations (approx).");
        });
```

```
        worker.start();
        Thread.sleep(50);     // let the worker run briefly
        running = false;      // cooperative signal to stop
        worker.join();
        System.out.println("Main thread confirms worker has terminated.");
    }
}
```

Output:

```
Worker stopped gracefully after 5482913 iterations (approx).
Main thread confirms worker has terminated.
```

Key Points to Remember

- wait(), notify(), and notifyAll() must always be called from within a synchronized block/method on the object being waited upon.
- wait() releases the object's lock while paused, allowing other threads to acquire it — unlike sleep(), which does not release any lock.
- Thread.suspend(), resume(), and stop() are deprecated and unsafe; a cooperative volatile flag is the recommended modern replacement.
- The 'while' loop (not 'if') around a wait() call guards against spurious wake-ups, ensuring the awaited condition is re-checked before proceeding.

5.13 Introduction to JDBC and JDBC Architecture

JDBC (Java Database Connectivity) is a standard Java API that provides a uniform way for a Java application to connect to, query, and update a relational database, regardless of which specific database product (MySQL, Oracle, PostgreSQL, SQL Server, etc.) is being used. This is made possible because JDBC defines a set of interfaces (in the java.sql and javax.sql packages) — such as Connection, Statement, and ResultSet — that every database vendor's own JDBC driver implements, allowing application code written against the standard JDBC interfaces to work with any compliant database simply by swapping the underlying driver.

JDBC Architecture

JDBC follows a layered, driver-based architecture. The Java application communicates only with the standard JDBC API (interfaces such as Connection, Statement, PreparedStatement, and ResultSet).

Underneath this API sits the JDBC Driver Manager, which is responsible for selecting and loading the correct vendor-specific JDBC driver at run time based on the connection URL supplied by the application. The driver itself then translates the standard JDBC calls into the specific network protocol or API required by the target database server, and translates the database's responses back into the standard JDBC objects that the application expects.

Memory Representation:

```
Java Application
       |
  JDBC API (Connection, Statement, ResultSet, ...)
       |
 JDBC DriverManager
       |
 Vendor-specific JDBC Driver  (e.g. MySQL Connector/J)
       |
   Database Server (e.g. MySQL)
```


Key JDBC Interfaces and Classes

- DriverManager — manages the list of registered JDBC drivers and creates Connection objects using getConnection().
- Connection — represents an active session/connection to a specific database.
- Statement — used to execute simple, static SQL statements against the database.
- PreparedStatement — a precompiled SQL statement supporting parameterized queries, offering better performance and protection against SQL injection.
- ResultSet — represents the tabular result of a SQL query, allowing row-by-row iteration over the returned data.

Key Points to Remember

- JDBC provides a database-independent API; switching databases mainly requires changing the driver and connection URL, not the application logic.
- DriverManager is responsible for selecting the correct driver and establishing a Connection to the target database.
- PreparedStatement is generally preferred over Statement for queries involving user-supplied values, both for performance and SQL-injection safety.
- ResultSet provides cursor-based, row-by-row access to query results, moved forward using its next() method.

5.14 JDBC Environment Setup, Establishing Connections, and the ResultSet Interface

Setting up a JDBC-based project to work with MySQL requires three ingredients: a running MySQL database server, the MySQL Connector/J driver JAR file added to the project's classpath (available for download from MySQL's official site or via a build tool such as Maven/Gradle), and Java code that loads the driver, establishes a Connection using a JDBC URL of the general form 'jdbc:mysql://hostname:port/databaseName', and supplies valid database credentials.

Steps to Set Up and Use JDBC with MySQL

- 1. Install and start a MySQL server, and create the target database and any required tables.
- 2. Download the MySQL Connector/J JAR file and add it to the project's classpath (or declare it as a Maven/Gradle dependency).
- 3. Load/register the JDBC driver (automatic in modern JDBC 4.0+ via service auto-discovery, but Class.forName("com.mysql.cj.jdbc.Driver") is still sometimes used explicitly).
- 4. Establish a Connection using DriverManager.getConnection(url, username, password).
- 5. Create a Statement or PreparedStatement, execute the required SQL, and process the returned ResultSet (for queries) or update count (for inserts/updates/deletes).
- 6. Close the ResultSet, Statement, and Connection (or use try-with-resources) to release database resources promptly.

Example Program: Connecting to MySQL and Querying Data

Example Program:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JdbcConnectionDemo {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/college_db";
        String username = "root";
        String password = "password";

        String sql = "SELECT roll_no, name, marks FROM students WHERE marks >=
?";
```

```
        try (Connection conn = DriverManager.getConnection(url, username,
password);
             PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setInt(1, 75);   // parameter: minimum marks
            try (ResultSet rs = stmt.executeQuery()) {
                while (rs.next()) {
                    int rollNo = rs.getInt("roll_no");
                    String name = rs.getString("name");
                    int marks = rs.getInt("marks");
                    System.out.println(rollNo + " | " + name + " | " + marks);
                }
            }
        } catch (SQLException e) {
            System.out.println("Database error: " + e.getMessage());
        }
    }
}
```

Output:

```
101 | Ajitha | 88
104 | Karthik | 79
107 | Priya | 91
```

Performing Insert and Update Operations with executeUpdate()

While executeQuery() is used for SELECT statements that return a ResultSet, data-modifying statements — INSERT, UPDATE, and DELETE — are executed with executeUpdate(), which instead returns an int representing the number of rows affected by the operation, allowing the program to confirm whether the intended change actually took place.

Example Program:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JdbcUpdateDemo {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/college_db";
        String username = "root";
```

```
        String password = "password";

        String insertSql = "INSERT INTO students (roll_no, name, marks) VALUES
(?, ?, ?)";
        String updateSql = "UPDATE students SET marks = ? WHERE roll_no = ?";

        try (Connection conn = DriverManager.getConnection(url, username,
password)) {

            try (PreparedStatement insertStmt = conn.prepareStatement(insertSql))
{
                insertStmt.setInt(1, 110);
                insertStmt.setString(2, "Sneha");
                insertStmt.setInt(3, 85);
                int rowsInserted = insertStmt.executeUpdate();
                System.out.println("Rows inserted: " + rowsInserted);
            }

            try (PreparedStatement updateStmt = conn.prepareStatement(updateSql))
{
                updateStmt.setInt(1, 90);
                updateStmt.setInt(2, 110);
                int rowsUpdated = updateStmt.executeUpdate();
                System.out.println("Rows updated: " + rowsUpdated);
            }
        } catch (SQLException e) {
            System.out.println("Database error: " + e.getMessage());
        }
    }
}
```

Output:

```
Rows inserted: 1
Rows updated: 1
```

The ResultSet Interface

A ResultSet represents the table of results returned by a SQL SELECT query and maintains a cursor that initially points just before the first row. The next() method advances the cursor to the following row and returns false once there are no more rows, which is why it is the standard loop condition for iterating over query results. Individual column values in the current row are retrieved using typed

getter methods such as getInt(), getString(), getDouble(), and getDate(), each of which can be called either with the column's name or its 1-based column index.

Key Points to Remember

- The JDBC connection URL format is 'jdbc:<subprotocol>://<host>:<port>/<databaseName>', with the subprotocol identifying the target database type.
- PreparedStatement parameters are set with typed setter methods (setInt(), setString(), etc.) using 1-based positional indices matching the '?' placeholders.
- ResultSet.next() must be called before accessing the first row, and returns false once the cursor moves past the last row.
- Closing Connection, Statement, and ResultSet objects (ideally via try-with-resources) is essential to avoid exhausting database connection resources.

5.15 JavaFX GUI Programming

JavaFX is a modern, modular framework for building rich graphical user interface (GUI) applications in Java, intended as the successor to the older Swing and AWT toolkits. A JavaFX application is structured around a Stage (the top-level application window), which contains a Scene (the container for all visual content shown at one time), which in turn holds a hierarchical tree of Node objects called the scene graph — including layout containers (such as VBox, HBox, and BorderPane) and individual UI controls (such as Button, Label, TextField, and ImageView).

JavaFX applications can be built purely in Java code, or their layout can be designed visually using Scene Builder, a separate visual design tool that generates an FXML file (an XML-based description of the scene graph) which can then be loaded into a Java application at run time using the FXMLLoader class; this separation allows UI design and application logic to be developed somewhat independently, similar in spirit to how HTML separates page structure from JavaScript logic.

Structure of a JavaFX Application

Every JavaFX application extends the abstract class javafx.application.Application and overrides its start(Stage primaryStage) method, which the JavaFX runtime calls automatically after performing internal initialization. Inside start(), the programmer builds the scene graph, wraps it in a Scene, attaches that Scene to the primaryStage, and finally calls primaryStage.show() to make the window visible.

Syntax:

```
public class MyApp extends Application {
    @Override
    public void start(Stage primaryStage) {
        // build UI nodes, create a Scene, attach it to primaryStage, then show()
    }

    public static void main(String[] args) {
        launch(args);   // bootstraps the JavaFX runtime and calls start()
    }
}
```

Example Program: Displaying Text and an Image with Basic Layout

Example Program:

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TextImageDemo extends Application {
    @Override
    public void start(Stage primaryStage) {
        Label heading = new Label("Welcome to SITAMS");
        heading.setStyle("-fx-font-size: 20px; -fx-font-weight: bold;");

        ImageView imageView = new ImageView(new
Image("https://example.com/logo.png"));
        imageView.setFitWidth(150);
        imageView.setPreserveRatio(true);

        VBox root = new VBox(15, heading, imageView);   // 15px spacing between
nodes
        root.setAlignment(Pos.CENTER);

        Scene scene = new Scene(root, 400, 300);
        primaryStage.setTitle("JavaFX Text and Image Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

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

Output:

```
(Launches a 400x300 window titled "JavaFX Text and Image Demo", showing a bold
heading label above a centered image.)
```

Event Handling and Mouse Events

JavaFX controls raise events (such as a button click or a mouse movement) that a program responds to by registering an event handler — commonly a lambda expression implementing the appropriate functional interface, such as EventHandler<ActionEvent> for a button's setOnAction(), or EventHandler<MouseEvent> for setOnMouseClicked(), setOnMouseEntered(), and setOnMouseExited(). This event-driven model allows the application to react to user interaction asynchronously, without blocking the rest of the interface.

Example Program:

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class EventHandlingDemo extends Application {
    @Override
    public void start(Stage primaryStage) {
        Label statusLabel = new Label("Click the button below");
        Button clickButton = new Button("Click Me");

        // Action event (button click)
        clickButton.setOnAction(event -> statusLabel.setText("Button was
clicked!"));

        // Mouse events
        clickButton.setOnMouseEntered(e -> clickButton.setStyle("-fx-background-
color: lightblue;"));
```

```
        clickButton.setOnMouseExited(e -> clickButton.setStyle(""));

        VBox root = new VBox(20, statusLabel, clickButton);
        root.setAlignment(Pos.CENTER);

        Scene scene = new Scene(root, 300, 200);
        primaryStage.setTitle("Event Handling Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Output:

```
(Displays a window with a label and button; clicking the button updates the label
text, and hovering highlights the button background.)
```

Laying Out Nodes in the Scene Graph

JavaFX provides several built-in layout container classes that automatically arrange their child nodes according to a particular strategy: VBox (arranges children vertically), HBox (arranges children horizontally), BorderPane (divides space into top, bottom, left, right, and center regions), GridPane (arranges children in a configurable grid of rows and columns), and StackPane (stacks children on top of one another, centered by default). Choosing the right layout container greatly simplifies building a responsive, well-organized interface compared to manually positioning every node with fixed pixel coordinates.

Example Program: A Simple Login Form Using GridPane

GridPane is especially well-suited to form-style layouts, such as a login screen, where labels and input fields need to align neatly into rows and columns.

Example Program:

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
```

```
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class LoginFormDemo extends Application {
    @Override
    public void start(Stage primaryStage) {
        GridPane grid = new GridPane();
        grid.setPadding(new Insets(20));
        grid.setHgap(10);
        grid.setVgap(10);

        Label userLabel = new Label("Username:");
        TextField userField = new TextField();
        Label passLabel = new Label("Password:");
        PasswordField passField = new PasswordField();
        Button loginButton = new Button("Login");

        grid.add(userLabel, 0, 0);
        grid.add(userField, 1, 0);
        grid.add(passLabel, 0, 1);
        grid.add(passField, 1, 1);
        grid.add(loginButton, 1, 2);

        loginButton.setOnAction(e ->
            System.out.println("Login attempted for user: " +
userField.getText()));

        Scene scene = new Scene(grid, 320, 180);
        primaryStage.setTitle("Login Form (GridPane)");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Output:

```
(Displays a neatly aligned login form with username/password fields and a login
button; clicking Login prints the entered username to the console.)
```

JavaFX Scene Builder

Scene Builder is a standalone visual, drag-and-drop design tool for laying out a JavaFX scene graph without writing layout code by hand; it saves the resulting layout as an FXML file. This FXML file is then loaded at run time in the Java application's start() method using FXMLLoader.load(getClass().getResource("layout.fxml")), and any interactive behavior is typically implemented in a separate 'controller' Java class linked to the FXML via the fx:controller attribute, keeping the visual layout and the application logic cleanly separated.

Key Points to Remember

- A JavaFX application extends Application and overrides start(Stage), which is invoked by the JavaFX runtime after launch() bootstraps it.
- The containment hierarchy is Stage (window) → Scene (content container) → scene graph of Node objects (layouts and controls).
- Event handlers are typically supplied as lambda expressions matching the required functional interface, such as EventHandler<ActionEvent> or EventHandler<MouseEvent>.
- Scene Builder produces FXML layout files that are loaded at run time via FXMLLoader, separating visual design from application logic through a controller class.

5.16 Solved Programming Examples

This section brings together several additional worked programs that combine the string-handling and multithreading concepts studied in this unit, reflecting the kind of integrated problems frequently asked in examinations.

Program 1: Check Whether a String Is a Palindrome

A string is a palindrome if it reads the same forwards and backwards. This is conveniently checked in Java by reversing the string using StringBuffer/StringBuilder and comparing it against the original.

Example Program:

```
public class StringPalindromeDemo {
    public static void main(String[] args) {
        String input = "madam";
```

```
        String reversed = new StringBuilder(input).reverse().toString();

        if (input.equalsIgnoreCase(reversed)) {
            System.out.println("\"" + input + "\" is a palindrome.");
        } else {
            System.out.println("\"" + input + "\" is NOT a palindrome.");
        }
    }
}
```

Output:

```
"madam" is a palindrome.
```

Program 2: Count the Number of Words, Vowels, and Consonants in a String

Example Program:

```
public class StringAnalysisDemo {
    public static void main(String[] args) {
        String text = "Java is an object oriented programming language";
        String[] words = text.split("\\s+");
        int vowelCount = 0, consonantCount = 0;

        for (char c : text.toLowerCase().toCharArray()) {
            if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
                vowelCount++;
            } else if (c >= 'a' && c <= 'z') {
                consonantCount++;
            }
        }

        System.out.println("Word count: " + words.length);
        System.out.println("Vowel count: " + vowelCount);
        System.out.println("Consonant count: " + consonantCount);
    }
}
```

Output:

```
Word count: 7
Vowel count: 15
```

```
Consonant count: 26
```

Program 3: Multithreaded Even/Odd Number Printer Using Synchronization

This program demonstrates two threads cooperating using wait()/notify() to print alternating even and odd numbers in the correct order, reinforcing inter-thread communication concepts.

Example Program:

```
class NumberPrinter {
    private int number = 1;
    private static final int LIMIT = 6;

    synchronized void printOdd() throws InterruptedException {
        while (number <= LIMIT) {
            while (number % 2 == 0) wait();
            System.out.println("Odd: " + number++);
            notify();
        }
    }

    synchronized void printEven() throws InterruptedException {
        while (number <= LIMIT) {
            while (number % 2 != 0) wait();
            System.out.println("Even: " + number++);
            notify();
        }
    }
}

public class EvenOddThreadDemo {
    public static void main(String[] args) {
        NumberPrinter printer = new NumberPrinter();

        Thread oddThread = new Thread(() -> {
            try { printer.printOdd(); } catch (InterruptedException e) { }
        });
        Thread evenThread = new Thread(() -> {
            try { printer.printEven(); } catch (InterruptedException e) { }
        });

        oddThread.start();
        evenThread.start();
    }
}
```

Output:

```
Odd: 1
Even: 2
Odd: 3
Even: 4
Odd: 5
Even: 6
```

Program 4: Formatting a Report Line Using StringBuilder and String.format

Example Program:

```
public class ReportFormattingDemo {
    public static void main(String[] args) {
        String[] names = {"Ajitha", "Karthik", "Priya"};
        int[] marks = {88, 79, 91};

        StringBuilder report = new StringBuilder();
        report.append(String.format("%-10s %s%n", "Name", "Marks"));
        report.append("-".repeat(20)).append(System.lineSeparator());

        for (int i = 0; i < names.length; i++) {
            report.append(String.format("%-10s %5d%n", names[i], marks[i]));
        }

        System.out.print(report);
    }
}
```

Output:

```
Name       Marks
--------------------
Ajitha        88
Karthik       79
Priya         91
```

Program 5: Reverse Each Word in a Sentence While Preserving Word Order

This program splits a sentence into words, reverses each word individually using StringBuilder, and reassembles the sentence, keeping the original word order but reversing the characters within each word.

Example Program:

```
public class ReverseWordsDemo {
    public static void main(String[] args) {
        String sentence = "Java is fun to learn";
        String[] words = sentence.split(" ");
        StringBuilder result = new StringBuilder();

        for (String word : words) {
            String reversedWord = new StringBuilder(word).reverse().toString();
            result.append(reversedWord).append(" ");
        }

        System.out.println("Original: " + sentence);
        System.out.println("Each word reversed: " + result.toString().trim());
    }
}
```

Output:

```
Original: Java is fun to learn
Each word reversed: avaJ si nuf ot nrael
```

Program 6: Simulating Parallel Downloads with Multiple Threads

This program simulates several independent 'file downloads' proceeding concurrently using separate threads, each reporting its own progress, illustrating how multithreading allows unrelated tasks to make progress without waiting for one another to finish.

Example Program:

```
class DownloadTask implements Runnable {
    private String fileName;

    DownloadTask(String fileName) {
        this.fileName = fileName;
    }
```

```
    @Override
    public void run() {
        for (int progress = 25; progress <= 100; progress += 25) {
            System.out.println(fileName + " download progress: " + progress +
"%");
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println(fileName + " download complete.");
    }
}

public class ParallelDownloadDemo {
    public static void main(String[] args) {
        Thread download1 = new Thread(new DownloadTask("notes.pdf"));
        Thread download2 = new Thread(new DownloadTask("slides.pptx"));

        download1.start();
        download2.start();
    }
}
```

Output:

```
notes.pdf download progress: 25%
slides.pptx download progress: 25%
notes.pdf download progress: 50%
slides.pptx download progress: 50%
notes.pdf download progress: 75%
slides.pptx download progress: 75%
notes.pdf download progress: 100%
notes.pdf download complete.
slides.pptx download progress: 100%
slides.pptx download complete.
```

Unit V — Summary

Java represents text using the immutable String class, which implements the CharSequence interface shared with the mutable StringBuffer and StringBuilder classes. String provides a comprehensive set of methods for extracting characters, comparing content, transforming case and structure, and

searching, while StringBuffer/StringBuilder should be preferred whenever text needs to be built or modified repeatedly, since they avoid the overhead of creating a new object on every change. Multithreaded programming allows a Java application to run several independent flows of execution concurrently, created either by extending Thread or, preferably, by implementing Runnable; threads progress through a well-defined life cycle of states and can be coordinated safely using the synchronized keyword and the wait()/notify() mechanism, which together prevent race conditions while avoiding the unsafe, deprecated suspend()/resume()/stop() methods.

JDBC provides a standard, database-independent API for Java applications to connect to and interact with relational databases such as MySQL, built around DriverManager (which establishes a Connection), Statement/PreparedStatement (which execute SQL), and ResultSet (which exposes query results through a row-by-row cursor). Finally, JavaFX offers a modern framework for building graphical user interfaces around the Stage-Scene-Node containment model, supporting text and image display, a rich set of layout containers (VBox, HBox, BorderPane, GridPane), an event-driven programming model for handling button clicks and mouse actions, and the Scene Builder visual design tool for producing FXML layouts that can be loaded and controlled from Java code.

Glossary of Key Terms

Immutable — A property of an object (such as String) whose state cannot be changed after it is created.

CharSequence — A common interface implemented by String, StringBuffer, and StringBuilder representing a readable sequence of characters.

Thread — The smallest independent unit of execution within a program, capable of running concurrently with other threads.

Race Condition — An error condition where a program's correctness depends on the unpredictable timing of concurrent thread execution.

Deadlock — A situation where two or more threads wait indefinitely for locks held by each other, so none can proceed.

Synchronization — Java's mechanism (the 'synchronized' keyword) for ensuring only one thread executes a guarded section of code at a time.

JDBC — Java Database Connectivity — the standard Java API for connecting to and interacting with relational databases.

ResultSet — A JDBC interface representing the tabular results of a SQL query, accessed through a forward-moving cursor.

Scene Graph — The hierarchical tree of Node objects (layouts and controls) that make up a JavaFX Scene.

Unit V — Question Bank

Part A — Short Answer Questions (2 Marks)

- 1. Why are String objects in Java said to be immutable? What advantage does this provide?
- 2. What is the CharSequence interface? Name any two classes that implement it.
- 3. Differentiate between the '==' operator and the equals() method when comparing Strings.
- 4. What is the difference between String and StringBuffer?
- 5. Define a thread. How is it different from a process?
- 6. List the five key states in the life cycle of a Java thread.
- 7. What is a race condition? Give one example scenario.
- 8. What is the purpose of the 'synchronized' keyword?
- 9. What is JDBC? Name the JDBC interface used to represent query results.
- 10. What is the role of the Stage and Scene classes in a JavaFX application?

Part B — Medium Answer Questions (5 Marks)

- 1. Explain any five commonly used methods of the String class for extracting, comparing, and searching text, with example code.
- 2. Explain the StringBuffer class and its commonly used methods with a suitable example program.
- 3. Explain the two ways of creating a thread in Java (extending Thread vs. implementing Runnable), with example programs for both.
- 4. Explain the life cycle of a thread in Java with a neat diagram covering all thread states.
- 5. Write a Java program to demonstrate a race condition and show how the 'synchronized' keyword fixes it.
- 6. Explain the JDBC architecture and the steps involved in connecting a Java application to a MySQL database.
- 7. Explain the ResultSet interface and how query results are processed, with an example program.

- 8. Explain the structure of a JavaFX application, including the Stage, Scene, and scene graph, with a simple example.

Part C — Long Answer / Programming Questions (10 Marks)

- 1. Explain String handling in Java in detail, covering the CharSequence interface, string creation and immutability, character extraction, comparison, modification, and searching, supported by example programs and outputs for each.
- 2. Explain multithreaded programming in Java in detail — the need for multiple threads, creating threads, the thread life cycle, and thread priority — with complete example programs and outputs.
- 3. Discuss thread synchronization in Java in detail, covering race conditions, the synchronized keyword, deadlock, and inter-thread communication using wait()/notify(), with complete example programs illustrating each concept.
- 4. Explain JDBC architecture in detail and write a complete Java program that connects to a MySQL database, executes a parameterized query using PreparedStatement, and processes the results using ResultSet.
- 5. Explain JavaFX GUI programming in detail, covering the Stage/Scene/Node model, commonly used layout containers, event handling, and mouse events, with complete example programs and expected behavior.
- 6. Compare String, StringBuffer, and StringBuilder in terms of mutability, thread-safety, and performance, and write Java programs demonstrating typical usage of each.
- 7. Discuss the dangers of using the deprecated Thread.suspend()/resume()/stop() methods, and explain the modern, safe alternatives for pausing, resuming, and stopping a thread, with example programs.

Reference Textbooks

- Anitha Seth, B.L. Juneja — Java: One Step Ahead, Oxford University Press.
- Debasis Samanta, Monalisa Sarma — Joy with Java: Fundamentals of Object Oriented Programming, Cambridge, 2023.
- Paul Deitel, Harvey Deitel — Java for Programmers, 4th Edition, Pearson.
- Herbert Schildt — The Complete Reference Java, 11th Edition, TMH.
- Y. Daniel Liang — Introduction to Java Programming, 7th Edition, Pearson.