Module 6 CORE JAVA

1. Introduction to Java

o History of Java

- Developed by: James Gosling and his team at Sun Microsystems in 1995.
- Original name: Oak (later renamed to Java in 1995 due to trademark issues).
- Purpose: Initially designed for embedded systems but evolved into a general-purpose programming language.
- Acquisition: Sun Microsystems was acquired by Oracle Corporation in 2010, and Oracle continues to maintain Java.

o Features of Java (Platform Independent, Object-Oriented, etc.)

1. Platform Independent
   - Java programs are compiled into bytecode which can run on any system with the Java Virtual Machine (JVM), making it *write once, run anywhere*.
2. Object-Oriented
   - Java follows the OOP paradigm with concepts like classes, objects, inheritance, polymorphism, encapsulation, and abstraction.
3. Simple and Familiar
   - Syntax is similar to C/C++, but with simpler memory management and no pointers.
4. Secure
   - Java has a strong security model with features like bytecode verification, sandboxing, and runtime security policies.
5. Robust
   - Built-in exception handling and garbage collection help create reliable applications.

6. Multithreaded
    - Java supports multithreading, allowing multiple threads to run concurrently for better performance.
7. High Performance
    - Though interpreted, Java's Just-In-Time (JIT) compiler helps improve performance.
8. Distributed
    - Java has networking capabilities built into the language, making it suitable for distributed computing.

Understanding JVM, JRE, and

Term	Description
JVM (Java Virtual Machine)	Interprets and runs Java bytecode. It's platform-dependent.
JRE (Java Runtime Environment)	Contains JVM and libraries required to run Java applications.
JDK (Java Development Kit)	Contains JRE and development tools like compiler (`javac`), debugger, etc. Needed to develop Java programs.

o Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)

Steps:

1. Install JDK:
    - Download from Oracle or use OpenJDK.
2. Set Environment Variables:
    - Add the `bin` folder of the JDK to the system `PATH`.
    - Set `JAVA_HOME` to point to the JDK installation directory.
3. Choose an IDE:

- **Eclipse:** Feature-rich, popular among beginners.
- IntelliJ IDEA: Advanced features, preferred by professionals.
- NetBeans: Oracle-supported IDE with GUI builder.
- VS Code (with extensions): Lightweight alternative.

o Java Program Structure (Packages, Classes, Methods)

```
package mypackage;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

- Package: Organizes classes into namespaces.
- Class: Blueprint for creating objects.
- Method: Function within a class. `main()` is the entry point of any Java application

2. Data Types, Variables, and Operators

o Primitive Data Types in Java (int, float, char, etc.)
Java has 8 primitive data types, which are the most basic types of data:

Data Type	Size	Description	Example
`byte`	1 byte	Small integer values (-128 to 127)	`byte a = 100;`
`short`	2 bytes	Larger integers than byte	`short b = 10000;`
`int`	4 bytes	Commonly used for integers	`int c = 100000;`
`long`	8 bytes	Large integer values	`long d = 100000L;`

`float`	4 bytes	Single-precision decimal	`float e = 5.75f;`
`double`	8 bytes	Double-precision decimal	`double f = 19.99;`
`char`	2 bytes	Single Unicode character	`char g = 'A';`
`boolean`	1 bit	True or false values	`boolean h = true;`

o Variable Declaration and Initialization

Declaration: Tells the compiler to reserve memory.

```
int age;
float salary;
```

Initialization: Assigns an initial value.

```
age = 25;
salary = 35000.50f;
```

Combined Declaration and Initialization:

```
int score = 95;
char grade = 'A';
```

Variable Naming Rules:

- Must begin with a letter, `_`, or `$`.
- Cannot use keywords (e.g., `int`, `class`).
- Java is case-sensitive (`Age` ≠ `age`).

o Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise

Java provides several types of operators to perform operations on variables and values:

1. Arithmetic Operators

Operator	Description	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`

*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

2. Relational (Comparison) Operators

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

3. Logical Operators

Operator	Description	Example
&&	Logical AND	a > 5 && b < 10
`		`
!	Logical NOT	!(a > b)

4. Assignment Operators

Operator	Description	Example
=	Assigns value	a = 10;
+=	a = a + b	a += b;
-=	a = a - b	a -= b;
*=	a = a * b	a *= b;

/=	a = a / b	a /= b;
%=	a = a % b	a %= b;

5. Unary Operators

Operator	Description	Example
+	Unary plus	+a
-	Unary minus	-a
++	Increment	a++ or ++a
--	Decrement	a-- or --a
!	Logical complement	!true (is false)

6. Bitwise Operators

Operator	Description	Example
&	Bitwise AND	a & b
`	`	Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise Complement	~a
<<	Left shift	a << 2
>>	Right shift	a >> 2

o Type Conversion and Type Casting

Aspect	Type Conversion (Widening)	Type Casting (Narrowing)
What it is	Automatic conversion	Manual conversion
From → To	Smaller to larger data type	Larger to smaller data type
Data loss	No	Possible
Syntax	Done automatically	(targetType) value

```
}
```

```
// Type Conversion
int a = 10;
double b = a;  // int → double

// Type Casting
double x = 9.8;
int y = (int) x;  // double → int (y = 9)
```

3. Control Flow Statements

Theory:

Control flow statements determine the order in which instructions are executed in a program.

o If-Else Statements

Used to execute certain code only when a condition is true.

Syntax:

```
if (condition) {

    // code runs if condition is true
} else {
    // code runs if condition is false
}
```

o Switch Case Statements

Used to select one of many code blocks to execute, based on the value of a variable.

Syntax:

```
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code
}
```

o Loops (For, While, Do-While)

Loops are used to repeat a block of code multiple times.

a) For Loop

Used when the number of iterations is known.

Syntax:

c
CopyEdit

```
for (initialization; condition; increment) {
    // code
}
```

b) While Loop

Executes code as long as the condition is true.

Syntax:

```
c
CopyEdit
while (condition) {
    // code
}
```

c) Do-While Loop

Similar to while, but executes at least once.

Syntax:

```
do {
    // code
} while (condition);
```

o Break and Continue Keywords

- **Break**: Exits the loop or switch statement immediately.
- **Continue**: Skips the current iteration and moves to the next one.

Example (Break):

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    printf("%d\n", i);
}
```

Example (Continue):

```
for (int i = 0; i < 5; i++) {
    if (i == 2) continue;
    printf("%d\n", i);
}
```

4. Classes and Objects

Theory:

o Defining a Class and Object in Java

- A class is a blueprint for creating objects.
- An object is an instance of a class.

Syntax – **Class Definition:**

```
class Car {
    String color;
    int speed;

    void drive() {
        System.out.println("Car is driving");
    }
}
```

Syntax – **Creating an Object:**

```
Car myCar = new Car();  // Object creation
myCar.color = "Red";    // Accessing members
myCar.drive();          // Calling method
```

o Constructors and Overloading

A **constructor** is a special method that initializes objects. It has the **same name as the class** and **no return type**.

Types of Constructors:

- **Default Constructor:** No parameters
- **Parameterized Constructor:** Takes parameters
- **Constructor Overloading:** Multiple constructors with different parameters

Example:

```
class Car {

    String color;
    int speed;

    // Default constructor
    Car() {
        color = "White";
        speed = 0;
    }

    // Parameterized constructor
    Car(String c, int s) {
        color = c;
        speed = s;
    }
}
```

Usage:

```
Car c1 = new Car();               // Calls default constructor
Car c2 = new Car("Blue", 120);   // Calls parameterized constructor
```

o Object Creation, Accessing Members of the Class

Object Creation:

```
Car myCar = new Car("Black", 100);
```

Accessing Members:

```
System.out.println(myCar.color);  // Accessing field
myCar.drive();                    // Calling method
```

o this Keyword

The `this` keyword refers to the current object inside a class. It helps resolve naming conflicts and refers to instance variables.

Example:

```
class Car {
    String color;

    Car(String color) {
        this.color = color;  // 'this.color' refers to the instance 
variable
    }
}
```

Without `this`, Java might confuse the parameter `color` with the instance variable `color`.

5. Methods in Java

A method in Java is a block of code that performs a specific task. It helps in modularizing code and reusing functionality.

Theory:
o Defining Methods

Syntax:

```
returnType methodName(parameters) {

    // method body
}
```

`void`: Return type (no return value)

`greet`: Method name

`()` : No parameters

o Method Parameters and Return Types

Parameters: Allow you to pass values to methods.
Return Type: Specifies what the method returns (e.g., `int`, `String`, `void`, etc.)

Example:

```
int add(int a, int b) {
    return a + b;
}
```

Calling the method:

```
int result = add(5, 3);  // result = 8
```

o Method Overloading

Method overloading means defining multiple methods with the same name but different parameter lists.

Example:

```
class MathOps {

    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Java chooses the correct method based on the arguments passed.

o Static Methods and Variables

- `static` methods/variables belong to the class, not to instances (objects).
- You can call a static method without creating an object.

Static Method Example:

```
class Utility {
    static void sayHi() {
        System.out.println("Hi from static method");
    }
}
```

Usage:

```
Utility.sayHi();  // No object needed
```

6. Object-Oriented Programming (OOPs) Concepts

Theory:

o Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction

a. Encapsulation

- Wrapping data (variables) and code (methods) into a single unit (class).
- Helps protect data using access modifiers (e.g., `private`, `public`).

Example:

```
class Person {
    private String name;

    public void setName(String n) {
        name = n;
    }

    public String getName() {
        return name;
    }
}
```

b. Inheritance

- One class (child) inherits fields and methods from another (parent).
- Promotes code reusability.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

c. Polymorphism

- Ability of an object to take many forms.
- Two types: Compile-time (method overloading) and Runtime (method overriding).

d. Abstraction

- Hiding internal details and showing only essential features.
- Achieved using abstract classes or interfaces.

Example (abstract class):

```
abstract class Shape {

    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }
}
```

o Inheritance: Single, Multilevel, Hierarchical

a. Single Inheritance

One child class inherits from one parent class.

```
class A { }
class B extends A { }
```

b. Multilevel Inheritance

A child inherits from a parent, and another child inherits from that child.

```
class A { }
class B extends A { }
class C extends B { }
```

c. Hierarchical Inheritance

Multiple classes inherit from the same parent.

```
class A { }
class B extends A { }
class C extends A { }
```

> ⚠️ Java does not support multiple inheritance with classes (to avoid ambiguity), but it supports it through interfaces.

o Method Overriding and Dynamic Method Dispatch

Method Overriding

- Redefining a parent class method in the child class using the same method name and parameters.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Dynamic Method Dispatch

- Runtime decision of which method to call (from parent or child).
- Achieved via method overriding and reference of parent class pointing to child object.

Example:

```
Animal a = new Dog();
a.sound();  // Outputs: Dog barks
```

7. Constructors and Destructors

In Java, **constructors** initialize objects when they're created. Java doesn't support traditional **destructors** (like C++); instead, it uses **garbage collection** to manage object destruction.

Theory:
o Constructor Types (Default, Parameterized)

A constructor is a special method that:

- Has the same name as the class
- Has no return type
- Is called automatically when an object is created

a. Default Constructor

- Takes no parameters.
- If no constructor is defined, Java provides a default one.

Example:

```
class Student {
    Student() {
        System.out.println("Default constructor called");
    }
}
```

b. Parameterized Constructor

- Takes arguments to initialize fields.

Example:

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

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

Usage:

```
Student s = new Student("Alice", 20);
```

o Copy Constructor (Emulated in Java)

Java does not have built-in copy constructors like C++, but you can create one manually.

Example:

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

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

    // Copy constructor
    Student(Student s) {
        name = s.name;
        age = s.age;
    }
}
```

Usage:

```
Student s1 = new Student("John", 22);
Student s2 = new Student(s1);  // Copy of s1
```

o Constructor Overloading

Constructor overloading means having multiple constructors with different parameter lists in the same class.

Example:

```
class Box {

    int length, width;

    Box() {
        length = width = 0;
    }

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

o Object Life Cycle and Garbage Collection

Object Life Cycle:

1. Creation → via constructor
2. Use → methods and fields are accessed
3. Destruction → handled automatically by Garbage Collector

Garbage Collection:

- Java uses automatic memory management.
- When there are no references to an object, it becomes eligible for garbage collection.

You can suggest garbage collection using:

```
System.gc();
```

Note: This does not guarantee immediate collection.

finalize() method (Deprecated):

- Used to perform cleanup before object is collected.

- Now considered outdated and unreliable.

Example:

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

8. Arrays and Strings

Theory:
o One-Dimensional and Multidimensional Arrays

1. One-Dimensional and Multidimensional Arrays

a. One-Dimensional Array

- A list of values of the same type stored in a single row.

Syntax:

```
int[] numbers = new int[5];  // declaration + memory allocation
numbers[0] = 10;             // initialization
```

Example:

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(arr[2]);  // Output: 3
```

b. Multidimensional Array

- An array of arrays (most commonly a **2D array**).

Syntax:

```
int[][] matrix = new int[2][3];
matrix[0][0] = 1;
```

Example:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println(matrix[1][2]);  // Output: 6
```

o String Handling in Java: String Class, StringBuffer, StringBuilder

Java provides three main classes for string handling:

a. String Class (Immutable)

- Once created, **cannot be changed**.

```
String s = "Hello";
String t = s.concat(" World");
System.out.println(s);  // Hello (unchanged)
```

b. StringBuffer (Mutable, Thread-Safe)

- Can be modified and is **safe for multi-threaded programs**.

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb);  // Hello World
```

c. StringBuilder (Mutable, Not Thread-Safe)

- Like `StringBuffer`, but **faster** in single-threaded applications.

```
StringBuilder sb = new StringBuilder("Hi");
sb.append(" Java");
System.out.println(sb);  // Hi Java
```

o Array of Objects

- You can create an array that holds multiple objects of a class.

Example:

```
class Student {
    String name;
    Student(String n) {
        name = n;
    }
}

public class Main {
    public static void main(String[] args) {
        Student[] students = new Student[3];
        students[0] = new Student("Alice");
        students[1] = new Student("Bob");
        students[2] = new Student("Charlie");

        for (Student s : students) {
            System.out.println(s.name);
        }
    }
}
```

o String Methods (length, charAt, substring, etc.)

Common String Methods

Method	Description	Example
`length()`	Returns length of string	`"Java".length()` → 4
`charAt(int)`	Returns character at specified index	`"Java".charAt(1)` → `'a'`
`substring(a, b)`	Returns substring from index `a` to `b-1`	`"Hello".substring(1, 4)` → `ell`
`equals()`	Compares two strings (case-sensitive)	`"Java".equals("java")` → `false`
`equalsIgnoreCase()`	Compares ignoring case	`"Java".equalsIgnoreCase("java")` → `true`
`toUpperCase()`	Converts to uppercase	`"java".toUpperCase()` → `JAVA`
`toLowerCase()`	Converts to lowercase	`"JAVA".toLowerCase()` → `java`
`contains()`	Checks if string contains a sequence	`"Hello".contains("el")` → `true`

`replace(a, b)`	Replaces character `a` with `b`	`"Java".replace('a','o')` → `Jovo`
`indexOf()`	Returns index of first occurrence	`"banana".indexOf('a')` → 1

9. Inheritance and Polymorphism

Theory:

o Inheritance Types and Benefits

Inheritance allows a class (subclass/child) to inherit the fields and methods of another class (superclass/parent).

Type	Description	Example
Single	One subclass inherits one superclass	`class B extends A`
Multilevel	A class inherits from a class, which inherits another	`C extends B extends A`
Hierarchical	Multiple subclasses inherit the same superclass	`class B extends A`, `class C extends A`

🔸 **Note:** Java **does not support multiple inheritance with classes** to avoid ambiguity (the *Diamond Problem*). It is supported via **interfaces**.

Benefits of Inheritance:

- **Code Reusability**: Share code between classes.
- **Maintainability**: Easier to manage and update code.
- **Extensibility**: Add new features by extending existing classes.

- **Runtime Polymorphism** support via method overriding.

o Method Overriding

- When a **subclass** provides a specific implementation of a method that is already defined in its **superclass**.
- Requires same method **name, return type, and parameters**.

Rules:

- Method in subclass must have the **same signature**.
- Use `@Override` annotation (optional but recommended).
- Access level must not be more restrictive.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

o Dynamic Binding (Run-Time Polymorphism)

- Also called **late binding**.
- When a **parent class reference refers to a child class object**, and **overridden methods are resolved at runtime**.

Example:

```
Animal a = new Dog();
a.sound();  // Output: Dog barks
```

- This enables **flexibility and extensibility** in code.

o Super Keyword and Method Hiding

super Keyword

Used to:

- Call the **parent class constructor**
- Access **parent class methods/variables**

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void sound() {
        super.sound();  // Calls Animal's sound()
        System.out.println("Dog barks");
    }
}
```

Method Hiding

Occurs when a **static method** in the subclass has the **same name and signature** as one in the superclass.

- Unlike method overriding, this is **compile-time resolution**.
- The method that gets called depends on the reference type.

Example:

```
class A {
    static void show() {
        System.out.println("Static method in A");
    }
}

class B extends A {
    static void show() {
        System.out.println("Static method in B");
    }
}

public class Main {
    public static void main(String[] args) {
        A obj = new B();
        obj.show();  // Output: Static method in A (method hiding)
    }
}
```

10. Interfaces and Abstract Classes

o Abstract Classes and Methods

Abstract Class:

- A class that **cannot be instantiated**.
- May contain **abstract methods** (without a body) and **concrete methods** (with a body).
- Used when you want to provide **base functionality** with the option for subclasses to override behavior.

Syntax:

```
abstract class Animal {
    abstract void sound();         // abstract method
    void eat() {                   // concrete method
        System.out.println("Eating...");
    }
}
```

Subclass Example:

```
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

o Interfaces: Multiple Inheritance in Java

Interface:

- A **completely abstract class**.
- Contains only **abstract methods** (until Java 7), and from Java 8 onward, can include:
    - `default` methods (with body)
    - `static` methods
- All fields are implicitly `public static final`.

Syntax:

```
interface Animal {
    void sound();  // implicitly public and abstract
}
```

Class implementing the interface:

```
class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

Multiple Inheritance with Interfaces

- A class can **implement multiple interfaces**.
- This is how Java **supports multiple inheritance** without the ambiguity of class-based multiple inheritance.

Example:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class C implements A, B {
    public void methodA() {
        System.out.println("Method A");
    }

    public void methodB() {
        System.out.println("Method B");
    }
}
```

o Implementing Multiple Interfaces

Feature	Abstract Class	Interface
Methods	Can have abstract & concrete	Only abstract (till Java 7), can have `default` & `static` (from Java 8)
Fields	Can have variables (any type)	`public static final` only
Inheritance	Single class inheritance only	Multiple interfaces allowed
Constructor	Can have constructors	Cannot have constructors
Access Modifiers	Can use any	All methods are `public` by default
Use Case	Partial abstraction	Full abstraction or multiple inheritance

11. Packages and Access Modifiers

o Java Packages: Built-in and User-Defined Packages

A **package** in Java is a namespace that organizes related classes and interfaces.

a. Types of Packages

✅ Built-in Packages

- Provided by Java SDK.
- Example:
    - `java.util` (e.g., `ArrayList`, `Date`)
    - `java.io` (e.g., `File`, `BufferedReader`)
    - `java.lang` (e.g., `String`, `Math`) → *Automatically imported*

✅ User-Defined Packages

- Created by developers to group related classes.

Creating a Package:

```
package mypackage;

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

Using the Package:

```
import mypackage.MyClass;

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

o Access Modifiers: Private, Default, Protected, Public

Access modifiers define the **visibility/scope** of classes, methods, and variables.

Modifier	Same Class	Same Package	Subclass	Other Packages
`private`	✅	❌	❌	❌
default (no keyword)	✅	✅	❌	❌
`protected`	✅	✅	✅	❌ (unless via inheritance)
`public`	✅	✅	✅	✅

a. private

- Visible **only within the same class**.

```
class A {
    private int data = 10;
}
```

b. default (no keyword)

- Visible **within the same package**.

```
class A {
    int data = 10;  // default access
}
```

c. protected

- Visible in:
    - Same package
    - Subclasses (even in different packages via inheritance)

```
class A {
    protected int data = 10;
}
```

d. public

- Accessible **from anywhere**.

```
public class A {
    public int data = 10;
}
```

o Importing Packages and Classpath

a. Importing Packages

To use a class from another package:

```
import packageName.ClassName;
import packageName.*;  // imports all classes from the package
```

b. Setting the Classpath

The **classpath** is the path where Java looks for classes and packages.

Command line example:

```
javac -cp .;mypackage MyProgram.java
java -cp .;mypackage MyProgram
```

12. Exception Handling

o Types of Exceptions: Checked and Unchecked

a. Checked Exceptions

- Checked at **compile-time**
- Must be **handled explicitly** using `try-catch` or `throws`
- Examples:
    - `IOException`

    - `SQLException`
    - `FileNotFoundException`

```
import java.io.*;

public class Example {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
FileReader("file.txt")); // Checked
    }
}
```

b. Unchecked Exceptions

- Checked at **runtime**
- Caused by **programming errors**
- Examples:
    - `NullPointerException`
    - `ArithmeticException`
    - `ArrayIndexOutOfBoundsException`

```
public class Example {
    public static void main(String[] args) {
        int a = 10 / 0;  // ArithmeticException
    }
}
```

o try, catch, finally, throw, throws

✅ `try`

- Code that might throw an exception is placed inside a `try` block.

✅ `catch`

- Used to **handle the exception**.

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
}
```

✅ `finally`

- **Always executed**, whether or not an exception occurs.
- Commonly used to **release resources**.

```
try {
    int data = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Exception caught");
} finally {
    System.out.println("Finally block executed");
}
```

✅ `throw`

- Used to **explicitly throw an exception**.

```
throw new ArithmeticException("Cannot divide by zero");
```

✅ `throws`

- Declares that a method **may throw an exception**.

```
void readFile() throws IOException {
    FileReader fr = new FileReader("file.txt");
}
```

o Custom Exception Classes

You can define your own exceptions by **extending** the `Exception` or `RuntimeException` class.

Checked Custom Exception Example:

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

public class Test {
    static void checkAge(int age) throws MyException {
        if (age < 18)
            throw new MyException("Age must be 18 or above");
    }

    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (MyException e) {
            System.out.println("Caught Exception: " +
e.getMessage());
        }
    }
}
```

Unchecked Custom Exception Example:

```
class MyRuntimeException extends RuntimeException {
    public MyRuntimeException(String msg) {
        super(msg);
    }
}
```

13. Multithreading

o Introduction to Threads

A **thread** is the smallest unit of a process that can run independently.

Java supports multithreading using the `Thread` class and `Runnable` interface.

Benefits:

- Efficient CPU usage

- Simpler program structure for asynchronous tasks
- Enables real-time behavior (e.g., games, servers)

o Creating Threads by Extending Thread Class or Implementing Runnable Interface

✅ a. By Extending `Thread` Class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // starts the thread
    }
}
```

✅ b. By Implementing `Runnable` Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread running...");
    }
}

public class Test {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

o Thread Life Cycle

State	Description
New	Thread object is created but not started

Runnable	start() has been called, ready to run
Running	Thread is executing run() method
Blocked/Waiting	Thread is paused for a resource/time/event
Terminated	Thread has completed or exited

```
NEW --> RUNNABLE --> RUNNING --> TERMINATED
             ↑           ↓
         WAITING <----- BLOCKED
```

o Synchronization and Inter-thread Communication

✅ a. Synchronization

- Ensures that only **one thread accesses a critical section** at a time.
- Prevents **race conditions** when multiple threads share data.

Synchronized method:

```
class Table {
    synchronized void printTable(int n) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
        }
    }
}
```

Synchronized block:

```
synchronized(obj) {
    // critical section
}
```

✅ b. Inter-thread Communication

- Used to **coordinate** actions between threads using:
    - `wait()`
    - `notify()`
    - `notifyAll()`

Example:

```
class Shared {
    synchronized void produce() throws InterruptedException {
        System.out.println("Producing...");
        wait();
        System.out.println("Resumed");
    }

    synchronized void consume() {
        System.out.println("Consuming...");
        notify();
    }
}
```

Method	Description
`start()`	Starts the thread
`run()`	Contains the thread's code
`sleep(ms)`	Pauses thread for specified time
`join()`	Waits for a thread to die
`yield()`	Temporarily pauses current thread
`setPriority()`	Sets thread priority (1 to 10)

14. File Handling

Theory:

o Introduction to File I/O in Java (java.io package)

- File I/O allows **persistent storage** of data (unlike RAM).
- Java provides **character streams** and **byte streams** for file operations.

Common I/O Classes:

Class	Purpose
`File`	Represents file/directory
`FileReader`	Reads character files
`FileWriter`	Writes character files
`BufferedReader`	Efficient reading
`BufferedWriter`	Efficient writing
`ObjectInputStream`, `ObjectOutputStream`	For serialization

o FileReader and FileWriter Classes

These classes are used for **reading and writing text files** (character-by-character).

✅ FileReader Example:

```
import java.io.*;

public class ReadFile {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("example.txt");
        int ch;
        while ((ch = fr.read()) != -1) {
            System.out.print((char) ch);
        }
        fr.close();
    }
}
```

✅ FileWriter Example:

```
import java.io.*;

public class WriteFile {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("example.txt");
        fw.write("Hello, Java FileWriter!");
        fw.close();
    }
}
```

o BufferedReader and BufferedWriter

These are **wrapper classes** that improve performance by reducing the number of I/O operations (using a buffer).

✅ BufferedReader Example:

```
import java.io.*;

public class BufferedRead {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
FileReader("example.txt"));
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
        br.close();
    }
}
```

✅ BufferedWriter Example:

```
import java.io.*;

public class BufferedWrite {
    public static void main(String[] args) throws IOException {
        BufferedWriter bw = new BufferedWriter(new
FileWriter("example.txt"));
        bw.write("Buffered writing is efficient!");
        bw.newLine();
        bw.close();
    }
}
```

o Serialization and Deserialization

Serialization is the process of **converting an object into a byte stream** to store or transfer it.
Deserialization is the **reconstruction of the object** from the byte stream.

✅ Requirements:

- The class must **implement `Serializable` interface**
- Fields not to be serialized should be marked `transient`

✅ Serialization Example:

```
import java.io.*;

class Student implements Serializable {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

public class SerializeDemo {
    public static void main(String[] args) throws Exception {
        Student s1 = new Student(101, "Alice");

        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("student.ser"));
        out.writeObject(s1);
        out.close();
        System.out.println("Object serialized");
    }
}
```

✅ Deserialization Example:

```java
import java.io.*;

public class DeserializeDemo {
    public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(new
FileInputStream("student.ser"));
        Student s = (Student) in.readObject();
        in.close();

        System.out.println("ID: " + s.id + ", Name: " + s.name);
    }
}
```

15. Collections Framework

Theory: The Java Collections Framework (in `java.util`) provides a set of classes and interfaces for storing and manipulating groups of objects.

o Introduction to Collections Framework

- A **Collection** is a group of individual objects represented as a **single unit**.
- Java Collections provide **data structures** (like lists, sets, maps) and **algorithms** to work with them efficiently.
- All collections are part of the `java.util` package.

Key Benefits:

- Reusability
- Type safety (with Generics)
- Efficiency
- Built-in sorting and searching

o List, Set, Map, and Queue Interfaces

Interface	Description
List	Ordered collection, allows duplicates

Set	Unordered collection, no duplicates
Queue	Elements are processed in a specific order (FIFO)
Map	Key-value pairs, no duplicate keys

o ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

✅ List Implementations

Class	Features
`ArrayList`	Resizable array, fast for search
`LinkedList`	Doubly linked list, fast insert/delete

Example:

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
```

✅ Set Implementations

Class	Features
`HashSet`	Unordered, uses hashing
`TreeSet`	Sorted in natural order (uses TreeMap internally)

Example:

```
Set<Integer> set = new HashSet<>();
set.add(10);
set.add(20);
```

✅ Map Implementations

Class	Features
`HashMap`	Key-value pairs, no order

`TreeMap`	Sorted by keys

Example:

```
Map<String, Integer> map = new HashMap<>();
map.put("Math", 90);
map.put("English", 85);
```

✅ Queue Implementations

Class	Features
`LinkedList`	Implements `Queue` interface
`PriorityQueue`	Elements ordered by priority

Example:

```
Queue<String> queue = new LinkedList<>();
queue.add("Task1");
queue.add("Task2");
```

🔹 Iterators and ListIterators

✅ Iterator

- Used to **traverse elements** one-by-one.
- Applicable to `List`, `Set`, and `Queue`.

Methods:

- `hasNext()`
- `next()`
- `remove()`

Example:

```
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

✅ ListIterator

- More powerful: can **traverse forward and backward**
- Only available for **List** types

Methods:

- `hasNext()`, `next()`
- `hasPrevious()`, `previous()`
- `add()`, `set()`, `remove()`

Example:

```
ListIterator<String> litr = list.listIterator();
while (litr.hasNext()) {
    System.out.println(litr.next());
}
```

✅ **Summary Table**

Interface	**Allows Duplicates**	**Maintains Order**	**Example Classes**
List	Yes	Yes	`ArrayList`, `LinkedList`
Set	No	No (Sorted in `TreeSet`)	`HashSet`, `TreeSet`
Map	Keys: No, Values: Yes	Keys unordered (Sorted in `TreeMap`)	`HashMap`, `TreeMap`
Queue	Yes	Yes (FIFO or priority)	`LinkedList`, `PriorityQueue`

16. Java Input/Output (I/O)

Theory: Java I/O provides input and output streams to read data from and write data to different sources such as files, memory, keyboard, etc.

o Streams in Java (InputStream, OutputStream)

A **stream** is a sequence of data. Java uses **streams** to perform I/O operations on data.

✅ Types of Streams:

a. Byte Streams (for binary data)

- Classes:
  - `InputStream` (abstract)
  - `OutputStream` (abstract)
  - Subclasses: `FileInputStream`, `FileOutputStream`

b. Character Streams (for text data)

- Classes:
  - `Reader` (abstract)
  - `Writer` (abstract)
  - Subclasses: `FileReader`, `FileWriter`

o Reading and Writing Data Using Streams

✅ Reading Using `InputStream`

```
import java.io.*;

public class ByteReadExample {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("input.txt");
        int data;
        while ((data = fis.read()) != -1) {
            System.out.print((char) data);
        }
        fis.close();
    }
}
```

✅ Writing Using `OutputStream`

```
import java.io.*;

public class ByteWriteExample {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("output.txt");
        String text = "Hello from OutputStream!";
        fos.write(text.getBytes());
        fos.close();
    }
}
```

✅ Reading Using `Reader` (Character Stream)

```
import java.io.*;

public class CharReadExample {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("input.txt");
        int ch;
        while ((ch = fr.read()) != -1) {
            System.out.print((char) ch);
        }
        fr.close();
    }
}
```

✅ Writing Using `Writer`

```
import java.io.*;

public class CharWriteExample {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("output.txt");
        fw.write("Writing with FileWriter!");
        fw.close();
    }
}
```

o Handling File I/O Operations

Java provides the **`File` class** in `java.io` to handle files and directories.

✅ Creating and Checking File

```
import java.io.*;

public class FileExample {
    public static void main(String[] args) {
        File file = new File("test.txt");
        try {
            if (file.createNewFile()) {
                System.out.println("File created.");
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ Common File Methods

Method	Description
`exists()`	Checks if file exists
`getName()`	Returns file name
`length()`	Returns file size in bytes
`delete()`	Deletes the file
`mkdir()` / `mkdirs()`	Creates directories
`listFiles()`	Lists contents of a directory

Lab Exercise:1

o Install JDK and set up environment variables.

Download JDK from Oracle/OpenJDK website.

Install JDK.

Set Environment Variable:

- `JAVA_HOME` → Path to JDK
- Add `JAVA_HOME/bin` to `PATH`.

o Write a simple "Hello World" Java program.
o Compile and run the program using command-line tools (javac, java).

Lab Exercise:2
o Write a program to demonstrate the use of different data types.
o Create a calculator using arithmetic and relational operators.
o Demonstrate type casting (explicit and implicit).

Lab Exercise:3
o Write a program to find if a number is even or odd using an if-else statement.
o Implement a simple menu-driven program using a switch-case.
o Write a program to display the Fibonacci series using a loop

Lab Exercise: 4
o Create a class Student with attributes (name, age) and a method to display the details.
o Create multiple constructors in a class and demonstrate constructor overloading.
o Implement a simple class with getters and setters for encapsulation.

Lab Exercise:5
o Write a program to find the maximum of three numbers using a method.
o Implement method overloading by creating methods for different data types.
o Create a class with static variables and methods to demonstrate their use.

Lab Exercise:6
o Write a program demonstrating single inheritance.
o Create a class hierarchy and demonstrate multilevel inheritance.
o Implement method overriding to show polymorphism in action.

Lab Exercise: 7
o Write a program to create and initialize an object using a parameterized constructor.
o Demonstrate constructor overloading by passing different types of parameters.

Lab Exercise:8
o Write a program to perform matrix addition and subtraction using 2D arrays.
o Create a program to reverse a string and check for palindromes.
o Implement string comparison using equals() and compareTo() methods

Lab Exercise:9
o Write a program that demonstrates inheritance using extends keyword.
o Implement runtime polymorphism by overriding methods in the child class.
o Use the super keyword to call the parent class constructor and methods.

Lab Exercise: 10
o Create an abstract class and implement its methods in a subclass.
o Write a program that implements multiple interfaces in a single class.
o Implement an interface for a real-world example,such as a payment gateway.

Lab Exercise: 11
o Create a user-defined package and import it into another program.
o Demonstrate the use of different access modifiers within the same package and across different packages.

Lab Exercise: 12
o Write a program to demonstrate exception handling using try-catch-finally.
o Implement multiple catch blocks for different types of exceptions.
o Create a custom exception class and use it in your program.

Lab Exercise: 13
o Write a program to create and run multiple threads using the Thread class.
o Implement thread synchronization using synchronized blocks or methods.
o Use inter-thread communication methods like wait(), notify(), and notifyAll().

Lab Exercise: 14
o Write a program to read and write content to a file using FileReader and FileWriter.
o Implement a program that reads a file line by line using BufferedReader.
o Create a program that demonstrates object serialization and deserialization.

Lab Exercise: 15
o Write a program that demonstrates the use of an ArrayList and LinkedList.
o Implement a program using HashSet to remove duplicate elements from a list.

o Create a HashMap to store and retrieve key-value pairs.

Lab Exercise:16

o Write a program to read input from the console using Scanner.
o Implement a file copy program using FileInputStream and FileOutputStream.
o Create a program that reads from one file and writes the content to another file.