III Semester

Course 5: Object Oriented Programming using Java

Unit - 1

OOPs Concepts and Java Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which can contain data and methods. Java is one of the most popular OOP languages.

Key OOP Concepts in Java

1. **Encapsulation**
   - Wrapping data (variables) and methods into a single unit (class).
   - Access control using **private**, **public**, **protected** modifiers.
   - Example:
   - ```
     class Person {
         private String name;
         public void setName(String name) { this.name = name; }
         public String getName() { return name; }
     }
     ```
2. **Inheritance**
   - Allows one class to acquire the properties of another.
   - Uses `extends` keyword.
   - Example:
   - ```
     class Animal {
         void makeSound() { System.out.println("Some sound..."); }
     }
     class Dog extends Animal {
         void bark() { System.out.println("Woof! Woof!"); }
     }
     ```
3. **Polymorphism**
   - Ability to take multiple forms (Method Overloading & Method Overriding).
   - **Method Overloading** (Compile-time Polymorphism):
   - ```
     class MathOperations {
         int add(int a, int b) { return a + b; }
         double add(double a, double b) { return a + b; }
     }
     ```
   - **Method Overriding** (Runtime Polymorphism):
   - ```
     class Parent {
         void show() { System.out.println("Parent method"); }
     }
     class Child extends Parent {
         void show() { System.out.println("Child method"); }
     }
     ```
4. **Abstraction**
   - Hiding implementation details and showing only necessary features.
   - Achieved through **abstract classes** and **interfaces**.

- **Abstract Class Example**:
- `abstract class Vehicle {`
- `    abstract void start();`
- `}`
- `class Car extends Vehicle {`
- `    void start() { System.out.println("Car starts with a key"); }`
- `}`
- **Interface Example**:
- `interface Animal {`
- `    void makeSound();`
- `}`
- `class Dog implements Animal {`
- `    public void makeSound() { System.out.println("Bark"); }`
- `}`

Java Features Supporting OOP

- **Class & Object**: Blueprint and instance of a class.
- **Constructors**: Special methods to initialize objects.
- **Static & Final Keywords**: Static for shared resources, Final to prevent modification.
- **Exception Handling**: Using `try-catch-finally` to handle errors.

Introduction to Object-Oriented Concepts

Object-Oriented Programming (OOP) is a programming paradigm that organizes code using **objects** rather than just functions and logic. It helps in designing scalable, maintainable, and reusable software.

Why OOP?

Before OOP, **procedural programming** was widely used, where code was written in a step-by-step manner. However, as software systems grew larger, procedural code became hard to maintain. OOP provides a structured way to organize and manage code efficiently.

Core OOP Concepts

1. **Objects and Classes**
   - **Object**: A real-world entity with **state** (attributes) and **behavior** (methods).
   - **Class**: A blueprint/template for creating objects.

   Example in Java:

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

    void accelerate() {
        speed += 10;
        System.out.println(brand + " is running at " + speed + "
km/h.");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Object creation
        myCar.brand = "Toyota";
        myCar.speed = 0;
        myCar.accelerate();
    }
}
```

---

2. **Encapsulation** *(Data Hiding)*
   - Wrapping data and methods into a single unit (class).
   - Restricting direct access to class fields using **private** access modifiers.
   - Providing controlled access using **getters** and **setters**.

Example:

```
class BankAccount {
    private double balance;

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

    public double getBalance() {
        return balance;
    }
}
```

---

3. **Inheritance** *(Code Reusability)*
   - Enables a child class to inherit properties and behaviors from a parent class.
   - Uses the `extends` keyword in Java.

Example:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Inherited method
```

```
        myDog.bark();
    }
}
```

---

4. **Polymorphism** *(Flexibility & Extensibility)*
    - **Method Overloading (Compile-time Polymorphism)**: Multiple methods with the same name but different parameters.
    - **Method Overriding (Runtime Polymorphism)**: A subclass provides its own implementation of a method from the parent class.

Overloading Example:

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

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

Overriding Example:

```
class Parent {
    void show() {
        System.out.println("Parent's method");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("Child's method");
    }
}
```

---

5. **Abstraction** *(Hiding Implementation Details)*
    - Hides complex implementation and shows only necessary details.
    - Achieved using **abstract classes** and **interfaces**.

Abstract Class Example:

```
abstract class Vehicle {
    abstract void start(); // Abstract method (no body)
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with a key.");
    }
}
```

Interface Example:

```
interface Animal {
    void makeSound();
}

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

Benefits of OOP

- **Code Reusability** (Inheritance)
- **Better Data Security** (Encapsulation)
- **Scalability & Maintainability** (Abstraction & Polymorphism)
- **Easier to Debug and Modify**

Procedural vs. Object-Oriented Programming (OOP) Paradigms

Both **Procedural Programming** and **Object-Oriented Programming (OOP)** are widely used programming paradigms, but they differ in their approach to organizing and structuring code.

---

1. Procedural Programming

Procedural programming is based on a **sequence of procedures (functions)** that operate on data. The program is structured as a set of procedures that manipulate variables and execute operations step by step.

Key Features:

- **Linear and structured**: Follows a top-down approach where code execution follows a sequential flow.
- **Functions (procedures)**: Code is divided into functions that perform specific tasks.
- **Global vs. Local variables**: Data is mostly stored in variables and passed between functions.
- **Focus on process**: The emphasis is on functions and their execution order.

Examples of Procedural Languages:

- **C**
- **Pascal**
- **Fortran**

Example (C):

```
#include <stdio.h>

// Function to add two numbers
int add(int a, int b) {
```

```
    return a + b;
}

int main() {
    int result = add(5, 10);
    printf("Sum: %d\n", result);
    return 0;
}
```

---

2. Object-Oriented Programming (OOP)

OOP is based on the concept of **objects**, which are instances of **classes** that encapsulate data and behavior.

Key Features:

- **Encapsulation**: Data and methods that operate on that data are grouped within classes.
- **Abstraction**: Hides implementation details and exposes only necessary functionality.
- **Inheritance**: Enables new classes to inherit properties and behaviors from existing classes.
- **Polymorphism**: Allows objects to take multiple forms (e.g., method overriding and overloading).

Examples of OOP Languages:

- **Java**
- **C++**
- **Python**
- **C#**

Example (Python - OOP)

```
class Calculator:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self):
        return self.a + self.b

# Creating an object of the class
calc = Calculator(5, 10)
print("Sum:", calc.add())
```

Key Differences Between Procedural and OOP

Feature	Procedural Programming	Object-Oriented Programming
Approach	Focuses on functions and procedures	Focuses on objects and classes
Data Handling	Data is separate from functions	Data and behavior are encapsulated in objects
Code Reusability	Limited code reuse	High reusability using inheritance
Complexity Handling	Difficult to manage large programs	Easier to manage large programs
Security	Less secure (data is accessible globally)	More secure (data is hidden inside objects)

When to Use Which Paradigm?

✅ **Use Procedural Programming** when:

- The problem is simple and sequential.
- Performance is a priority (e.g., embedded systems, system-level programming).
- Code needs to be small and efficient.

✅ **Use Object-Oriented Programming** when:

- The project is large and complex.
- Code needs to be modular and reusable.
- Security, maintainability, and scalability are required.

Java Programming Overview

Java is a **high-level, object-oriented, platform-independent** programming language developed by **Sun Microsystems** (now owned by Oracle). It is widely used for web applications, mobile applications (Android), enterprise software, and more.

---

1. Key Features of Java

1. **Object-Oriented** – Based on classes and objects.
2. **Platform Independent** – "Write Once, Run Anywhere" (WORA) due to the Java Virtual Machine (JVM).
3. **Robust and Secure** – Strong memory management and built-in security.
4. **Multithreading Support** – Enables concurrent programming.
5. **Automatic Garbage Collection** – Manages memory automatically.
6. **Rich API** – Extensive built-in libraries (Java Standard API).
7. **Portable** – Code can run on any system with a JVM.

---

2. Basic Java Syntax

Hello, World! in Java

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

Explanation:

- `public class HelloWorld` → Declares a class.
- `public static void main(String[] args)` → Main method (entry point).
- `System.out.println("Hello, World!");` → Prints text to the console.

---

3. Java Data Types

Java has two categories of data types:

Primitive Data Types:

Data Type	Size	Example
byte	1 byte	`byte b = 127;`
short	2 bytes	`short s = 32000;`
int	4 bytes	`int num = 100;`
long	8 bytes	`long bigNum = 100000L;`
float	4 bytes	`float pi = 3.14f;`
double	8 bytes	`double precise = 3.14159;`
char	2 bytes	`char letter = 'A';`
boolean	1 bit	`boolean isJavaFun = true;`

Non-Primitive Data Types:

- Strings (`String name = "Java";`)
- Arrays (`int[] numbers = {1, 2, 3};`)
- Classes, Interfaces, Objects

---

4. Java Control Statements

Conditional Statements:

```
int num = 10;
if (num > 5) {
    System.out.println("Number is greater than 5");
} else {
    System.out.println("Number is 5 or less");
}
```

Switch Case:

```
int day = 2;
switch(day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    default: System.out.println("Other day");
}
```

Loops in Java:

For Loop

```
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration: " + i);
}
```

While Loop

```
int i = 0;
```

```
while (i < 5) {
    System.out.println("Count: " + i);
    i++;
}
```

Do-While Loop

```
int i = 0;
do {
    System.out.println("Iteration: " + i);
    i++;
} while (i < 5);
```

---

5. Object-Oriented Programming (OOP) in Java

Defining a Class and Object

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

    void display() {
        System.out.println("Car Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.brand = "Toyota";
        myCar.speed = 120;
        myCar.display();
    }
}
```

OOP Principles in Java

1. **Encapsulation** – Data hiding using `private` access modifiers.
2. **Inheritance** – A subclass inherits properties from a superclass.
3. **Polymorphism** – Methods behave differently based on objects.
4. **Abstraction** – Hides implementation details using abstract classes or interfaces.

---

6. Exception Handling in Java

```
try {
    int result = 10 / 0;  // Division by zero
} catch (ArithmeticException e) {
    System.out.println("Error: Division by zero!");
} finally {
    System.out.println("This will always execute.");
}
```

---

7. Java Collections Framework (JCF)

Java provides powerful **collections** like **ArrayList, HashMap, HashSet, LinkedList**.

Example: Using ArrayList

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");

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

---

8. Java Multithreading

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

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

---

9. Java File Handling

```
import java.io.File;
import java.io.IOException;

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

---

10. Java GUI (Swing Example)

```
import javax.swing.*;

public class SimpleGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My First GUI");
```

```
        JButton button = new JButton("Click Me");
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

An Overview of Java

Introduction

Java is a **high-level, object-oriented, and platform-independent** programming language developed by **Sun Microsystems** (now owned by Oracle) in **1995**. It is widely used for **web development, mobile applications (Android), enterprise applications, and more**. Java follows the **"Write Once, Run Anywhere" (WORA)** principle, meaning compiled Java code can run on any platform with a **Java Virtual Machine (JVM)**.

---

1. Key Features of Java

1. **Object-Oriented** – Java is based on the concepts of classes and objects.
2. **Platform-Independent** – Java code runs on any system with a JVM.
3. **Robust and Secure** – Strong memory management and built-in security features.
4. **Multithreading Support** – Java can run multiple threads concurrently.
5. **Garbage Collection** – Automatic memory management through the JVM.
6. **Rich API** – Java provides a vast standard library (Java API).
7. **Portable** – Java programs can run on different operating systems without modification.

---

2. Java Architecture

Java Platform Components:

1. **Java Development Kit (JDK)** – Includes the compiler, libraries, and tools for developing Java applications.
2. **Java Runtime Environment (JRE)** – Provides libraries and JVM to run Java programs.
3. **Java Virtual Machine (JVM)** – Translates Java bytecode into machine code for execution.

Compilation Process in Java

1. **Write Java code** (`.java` file).
2. **Compile it** using `javac` (Java compiler), which generates **bytecode** (`.class` file).
3. **Run the bytecode** on the JVM, which interprets it for the specific OS.

---

3. Java Syntax and Basic Concepts

Hello, World! Program

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

Basic Java Data Types

Data Type	Size	Example
`int`	4 bytes	`int num = 10;`
`double`	8 bytes	`double pi = 3.14;`
`char`	2 bytes	`char letter = 'A';`
`boolean`	1 bit	`boolean isJavaFun = true;`

---

4. Object-Oriented Programming (OOP) in Java

Java is built around four OOP principles:

1. **Encapsulation** – Wrapping data and methods into a single unit (class).
2. **Inheritance** – A class can inherit properties from another class.
3. **Polymorphism** – A method can have different behaviors.
4. **Abstraction** – Hiding complex details and exposing only necessary functionality.

Example of a Java Class

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

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.brand = "Toyota";
        myCar.speed = 120;
        myCar.display();
    }
}
```

---

5. Control Structures in Java

Conditional Statements

```
if (age >= 18) {
    System.out.println("Adult");
} else {
    System.out.println("Minor");
}
```

Loops

For Loop

```
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration: " + i);
}
```

While Loop

```
int i = 0;
while (i < 5) {
    System.out.println("Count: " + i);
    i++;
}
```

---

6. Exception Handling in Java

```
try {
    int result = 10 / 0;  // Division by zero
} catch (ArithmeticException e) {
    System.out.println("Error: Division by zero!");
} finally {
    System.out.println("Execution complete.");
}
```

---

7. Java Collections Framework (JCF)

Java provides built-in **data structures** like `ArrayList`, `HashMap`, `LinkedList`, and `HashSet`.

Example: Using ArrayList

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");

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

---

8. Java Multithreading

Java supports **multithreading**, allowing concurrent execution of tasks.

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

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

---

9. Java File Handling

Java allows reading and writing files using the `java.io` package.

```
import java.io.File;
import java.io.IOException;

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

---

10. Java GUI Development

Java provides GUI libraries like **Swing** and **JavaFX** for desktop applications.

Simple GUI with Swing

```
import javax.swing.*;

public class SimpleGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My First GUI");
        JButton button = new JButton("Click Me");
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

---

11. Java Frameworks and Applications

Java is widely used in various domains with popular frameworks:

- **Spring Boot** (Enterprise Applications)
- **Hibernate** (Database ORM)
- **Android Development** (Mobile Apps)
- **Apache Kafka** (Real-time Data Processing)
- **JUnit** (Testing Framework)

Conclusion

Java is a powerful, **versatile**, and **widely used** programming language suitable for **desktop, web, mobile, and enterprise applications**. It remains a top choice due to its **platform independence, robustness, security, and extensive libraries**.

Java Environment Overview

The **Java Environment** is the setup required to develop, compile, and run Java programs. It includes various components such as the **Java Development Kit (JDK), Java Runtime Environment (JRE), and Java Virtual Machine (JVM)**.

---

1. Components of the Java Environment

The Java environment consists of three main components:

1.1 Java Development Kit (JDK)

The **JDK** is a software development kit that provides tools for developing Java applications.

✅ **Includes**:

- **Java Compiler (javac)** – Converts Java code into bytecode.
- **Java Runtime Environment (JRE)** – Contains libraries and JVM for execution.
- **Debugger (jdb)** – Helps debug Java programs.
- **JavaDoc Tool** – Generates documentation from Java code.
- **Additional Libraries** – Java Standard Edition (SE) libraries.

💡 **Download JDK from**: Oracle Java

1.2 Java Runtime Environment (JRE)

The **JRE** is required to run Java applications. It includes the necessary libraries and the **JVM** to execute Java programs.

✅ **Includes**:

- **JVM** – Runs Java bytecode.
- **Java Libraries** – Standard Java APIs (e.g., `java.util`, `java.io`).
- **Other Runtime Components** – Supports execution of Java applications.

💡 **JRE vs. JDK**:

- JDK **= JRE + Development Tools**
- If you only need to run Java programs, install the **JRE**.
- If you need to **develop** Java applications, install the **JDK**.

1.3 Java Virtual Machine (JVM)

The **JVM** is an engine that converts **Java bytecode** into machine code for execution.

✅ **Key Functions of JVM**:

1. **Loads** Java bytecode.
2. **Verifies** bytecode for security.
3. **Interprets or compiles** bytecode into native machine code.
4. **Manages memory** with **Garbage Collection**.

💡 **JVM is platform-dependent** but **Java bytecode is platform-independent**, allowing Java to run on any operating system.

---

2. Java Compilation and Execution Process

Step-by-Step Java Program Execution

1️⃣ **Write Java Code** (`.java` file)
2️⃣ **Compile Code** using `javac` → Generates **bytecode** (`.class` file)
3️⃣ **Run Program** using `java` command → JVM executes bytecode

Example

Java Code (`HelloWorld.java`)

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

Compilation

```
javac HelloWorld.java  # Compiles to HelloWorld.class
```

Execution

```
java HelloWorld  # Runs the program
```

---

3. Setting Up Java Environment

To develop and run Java programs, you need to set up the Java environment on your system.

3.1 Install Java (JDK)

Windows

1. Download JDK from Oracle's website.
2. Install it and note the installation path (e.g., `C:\Program Files\Java\jdk-XX`).
3. Set up **Environment Variables**:
    - Add `JAVA_HOME = C:\Program Files\Java\jdk-XX`
    - Add `C:\Program Files\Java\jdk-XX\bin` to the `PATH` variable.

Mac/Linux

1. Install JDK using a package manager:
    - **Mac** (Homebrew):
    - `brew install openjdk`
    - **Linux (Ubuntu/Debian)**:
    - `sudo apt update`
    - `sudo apt install default-jdk`
2. Verify installation:
3. `java -version`
4. `javac -version`

---

4. Java Development Tools

4.1 Integrated Development Environments (IDEs)

IDEs provide advanced tools for Java development.

✅ Popular Java IDEs:

- **Eclipse** – Powerful and free IDE.
- **IntelliJ IDEA** – Feature-rich (Community & Ultimate editions).
- **NetBeans** – Oracle-supported Java IDE.
- **VS Code** – Supports Java via extensions.

4.2 Build Tools

Build tools help manage dependencies and automate tasks.

✅ Popular Build Tools:

- **Maven** – Dependency management and project build tool.
- **Gradle** – Modern and faster alternative to Maven.
- **Ant** – Older build automation tool.

---

5. Java Environment Variables

Setting up environment variables ensures Java works correctly from the command line.

Windows (Setting JAVA_HOME)

1. Open **System Properties → Advanced → Environment Variables**.
2. Click **New** under **System Variables**.
3. Set:
    - **Variable Name:** `JAVA_HOME`
    - **Variable Value:** `C:\Program Files\Java\jdk-XX`
4. Add `%JAVA_HOME%\bin` to the `Path` variable.

Linux/macOS

Edit `.bashrc` or `.zshrc`:

```
export JAVA_HOME=/usr/lib/jvm/java-XX-openjdk
export PATH=$JAVA_HOME/bin:$PATH
```

Apply changes:

```
source ~/.bashrc  # or source ~/.zshrc
```

---

6. Checking Java Installation

After installing Java, verify the installation:

```
java -version
javac -version
```

✅ Expected Output (Example):

```
java version "17.0.1"  (or latest installed version)
javac 17.0.1
```

---

7. Running Java Programs in Different Ways

7.1 Using Command Line

```
javac MyProgram.java  # Compile
java MyProgram  # Run
```

7.2 Using an IDE

1. Open your IDE (Eclipse, IntelliJ, etc.).
2. Create a **New Java Project**.
3. Write Java code in a **class**.
4. Click **Run** or **Execute**.

---

8. Java Versions & Updates

Java has two main types of releases:

1. **LTS (Long-Term Support) Versions** – Stable and recommended for enterprises.
2. **Non-LTS Versions** – Frequent updates with new features.

✅ **Latest Java Versions (as of 2025)**:

- **Java 8 (LTS)** – Still widely used.
- **Java 11 (LTS)** – Common in production.
- **Java 17 (LTS)** – Recommended for new projects.
- **Java 21 (LTS)** – Latest long-term support version.

💡 Use `java -version` to check your installed version.

Conclusion

The Java environment consists of the **JDK, JRE, and JVM**, enabling Java development and execution. Properly setting up Java ensures smooth development and execution of Java programs.

Java Data Types

In Java, **data types** specify the size and type of values that variables can store. Java has two main categories of data types:

1. **Primitive Data Types** (Basic types like int, double, char, etc.)
2. **Non-Primitive Data Types** (Objects, Arrays, Strings, etc.)

---

1. Primitive Data Types

Java has **8 primitive data types**, which are built into the language and optimized for performance.

Data Type	Size	Default Value	Example	Description
byte	1 byte	0	byte b = 100;	Stores small integers (-128 to 127).
short	2 bytes	0	short s = 5000;	Stores medium-range integers (-32,768 to 32,767).
int	4 bytes	0	int num = 100000;	Stores large integers (-2^31 to 2^31-1).
long	8 bytes	0L	long bigNum = 1000000000L;	Stores very large integers (-2^63 to 2^63-1).
float	4 bytes	0.0f	float pi = 3.14f;	Stores decimal numbers with single precision.
double	8 bytes	0.0d	double e = 2.71828;	Stores decimal numbers with double precision.
char	2 bytes	'\u0000'	char letter = 'A';	Stores a single character (Unicode).
boolean	1 bit	false	boolean isJavaFun = true;	Stores true or false values.

Example: Using Primitive Data Types

```
public class DataTypesExample {
    public static void main(String[] args) {
        byte smallNumber = 100;
        int age = 25;
        long bigNumber = 10000000000L;
        float pi = 3.14f;
        double preciseNumber = 2.7182818284;
        char grade = 'A';
        boolean isJavaFun = true;

        System.out.println("Byte: " + smallNumber);
        System.out.println("Integer: " + age);
        System.out.println("Long: " + bigNumber);
        System.out.println("Float: " + pi);
        System.out.println("Double: " + preciseNumber);
        System.out.println("Char: " + grade);
        System.out.println("Boolean: " + isJavaFun);
    }
}
```

---

2. Non-Primitive Data Types

Non-primitive data types are more complex and are derived from primitive types.

Common Non-Primitive Data Types:

1. **String** – A sequence of characters.
2. **Array** – A collection of elements of the same type.
3. **Class** – A blueprint for creating objects.
4. **Interface** – Defines methods that a class must implement.

2.1 String Data Type

The `String` type is used to store sequences of characters.

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

2.2 Array Data Type

An **array** stores multiple values of the same type in a single variable.

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        System.out.println("First element: " + numbers[0]);
    }
}
```

2.3 Class and Object Data Type

Java follows an **object-oriented programming (OOP)** model, where **classes** define blueprints for objects.

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

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.brand = "Toyota";
```

```
        myCar.speed = 120;
        myCar.display();
    }
}
```

---

3. Type Conversion in Java

Java supports **implicit** and **explicit** type conversion.

3.1 Implicit Type Conversion (Widening)

Automatically converts a smaller data type to a larger data type.

```
int num = 10;
double converted = num;  // Implicit conversion (int → double)
System.out.println(converted); // Output: 10.0
```

3.2 Explicit Type Conversion (Narrowing)

Manually converts a larger data type to a smaller data type using **casting**.

```
double price = 9.99;
int roundedPrice = (int) price; // Explicit conversion (double → int)
System.out.println(roundedPrice); // Output: 9
```

---

4. Wrapper Classes (Primitive to Object)

Java provides **wrapper classes** to convert primitive types into objects.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Example: Using Wrapper Classes

```
public class WrapperExample {
```

```
    public static void main(String[] args) {
        Integer num = 100;  // Autoboxing (int → Integer)
        int value = num;     // Unboxing (Integer → int)
        System.out.println("Wrapper Class: " + num);
        System.out.println("Primitive Value: " + value);
    }
}
```

Conclusion

- Java provides **8 primitive data types** for efficiency.
- **Non-primitive types** (Strings, Arrays, Classes) are more complex.
- **Type conversion** allows data to be transformed from one type to another.
- **Wrapper classes** convert primitives into objects.

Java Variables

In Java, **variables** are used to store data in memory. A variable has a **name**, a **data type**, and a **value**. Java is a **strongly typed language**, meaning each variable must be explicitly declared with a type.

---

1. Types of Variables in Java

Java has **three main types of variables**:

1. **Local Variables** – Declared inside methods or blocks.
2. **Instance Variables (Non-Static Fields)** – Belong to an object and are stored in memory for each instance.
3. **Static Variables (Class Variables)** – Belong to the class and shared among all instances.

---

2. Declaring Variables in Java

A variable declaration follows this syntax:

```
dataType variableName = value;
```

Example

```
int age = 25;         // Integer variable
double price = 99.99; // Decimal value
char grade = 'A';     // Character
boolean isJavaFun = true; // Boolean
```

---

3. Types of Variables in Detail

3.1 Local Variables

- Declared **inside a method, constructor, or block**.
- Accessible **only within the method/block**.
- **Must be initialized** before use.

Example:

```
public class Example {
    public static void main(String[] args) {
        int x = 10; // Local variable
        System.out.println("Value of x: " + x);
    }
}
```

✅ **Output:** `Value of x: 10`

3.2 Instance Variables

- Declared **inside a class but outside a method**.
- **Each object gets its own copy** of instance variables.
- **Has a default value** if not initialized.

Example:

```
class Car {
    String brand; // Instance variable
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.brand = "Toyota"; // Assigning value
        System.out.println("Car brand: " + myCar.brand);
    }
}
```

✅ **Output:** `Car brand: Toyota`

3.3 Static Variables (Class Variables)

- Declared using the **static** keyword.
- Shared by **all instances** of a class.
- **Stored in memory only once** (in the class area).

Example:

```
class Employee {
    static String company = "TechCorp"; // Static variable
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Employee.company);
    }
}
```

✅ **Output:** `TechCorp`

---

4. Variable Naming Rules

✅ Valid Variable Names

- Must start with a **letter, $, or _**.
- Cannot be a **Java keyword**.
- Can use **camelCase** for readability.

❌ Invalid Variable Names

```
int 1num = 10;  // ❌ Cannot start with a number
int class = 5;  // ❌ "class" is a keyword
```

✅ Correct Naming

```
int num1 = 10;
String firstName = "John";
```

---

5. Variable Scope

5.1 Local Scope

Local variables exist **only inside the method or block**.

```
public class ScopeExample {
    public static void main(String[] args) {
        int a = 10; // Local scope
        System.out.println(a);
    }
    // System.out.println(a); ❌ Error: Variable "a" is not accessible here
}
```

5.2 Instance Scope

Instance variables belong to **objects** and exist **as long as the object exists**.

```
class Person {
    String name; // Instance variable
```

```
}
```

5.3 Class Scope

Static variables belong to the **class** and are accessible **throughout the program**.

```
class Example {
    static int count = 100;
}
```

---

6. Variable Initialization

Java variables **must be initialized** before use.

6.1 Default Values of Variables

Data Type	Default Value
`byte, short, int, long`	`0`
`float, double`	`0.0`
`char`	`'\u0000'` (null character)
`boolean`	`false`
`String` (objects)	`null`

Example:

```
class Example {
    int number; // Default: 0
    boolean flag; // Default: false

    public static void main(String[] args) {
        Example obj = new Example();
        System.out.println("Number: " + obj.number);
        System.out.println("Boolean: " + obj.flag);
    }
}
```

✅ **Output:**

```
Number: 0
Boolean: false
```

---

7. Final Variables (Constants)

Use the `final` keyword to create **constants** (unchangeable variables).

```
final double PI = 3.14159;
PI = 3.14; // ✕ Error: Cannot change a final variable
```

---

8. Type Conversion in Variables

8.1 Implicit Type Conversion (Widening)

Automatically converts **smaller to larger data types**.

```
int num = 100;
double d = num; // int → double (Widening)
System.out.println(d); // Output: 100.0
```

8.2 Explicit Type Conversion (Narrowing)

Manually converts **larger to smaller data types**.

```
double price = 9.99;
int roundedPrice = (int) price; // double → int (Narrowing)
System.out.println(roundedPrice); // Output: 9
```

Conclusion

- Java variables **store values** in memory.
- Three main types: **Local, Instance, and Static**.
- Variables have **scope** and **default values**.
- `final` variables are **constants**.
- Type conversion allows **widening** and **narrowing**.

Constants in Java

A **constant** in Java is a variable whose value **cannot be changed** once assigned. Java provides different ways to define constants, mainly using the **`final`** keyword.

---

1. Declaring Constants Using `final`

The `final` keyword makes a variable **unchangeable**.

Example: Declaring a Constant

```
public class ConstantsExample {
    public static void main(String[] args) {
        final double PI = 3.14159;  // Constant variable
        System.out.println("Value of PI: " + PI);

        // PI = 3.14; // ✕ Error: Cannot change a final variable
    }
}
```

✅ Output:

```
Value of PI: 3.14159
```

🔷 Rules for Constants:

- Must be initialized **when declared**.
- Cannot be **modified** later.

---

2. Naming Convention for Constants

By convention, constants are written in **uppercase with underscores**.

```
final int MAX_SPEED = 120;
final double TAX_RATE = 0.18;
```

---

3. Constants in Classes (Static Constants)

- If a constant belongs to a **class** rather than an instance, use **`static final`**.
- These constants are shared by **all objects** of the class.

Example: Static Constant

```
class MathConstants {
    static final double PI = 3.14159;
    static final double E = 2.71828;
}

public class Main {
    public static void main(String[] args) {
        System.out.println("PI: " + MathConstants.PI);
        System.out.println("Euler's Number: " + MathConstants.E);
    }
}
```

✅ Output:

```
PI: 3.14159
Euler's Number: 2.71828
```

🔷 Why Use `static`?

- Without `static`, every object would have its own copy.
- With `static`, the value is stored **once** in memory.

---

4. Constants in Interfaces

In **interfaces**, all fields are **implicitly** **`public static final`**, meaning:

- They are **public** (accessible everywhere).
- They are **static** (shared across all instances).
- They are **final** (cannot be modified).

Example: Constants in an Interface

```
interface Config {
    int MAX_USERS = 1000;  // public static final by default
    String APP_NAME = "MyApp";
}

public class Main {
    public static void main(String[] args) {
        System.out.println("App Name: " + Config.APP_NAME);
        System.out.println("Max Users: " + Config.MAX_USERS);
    }
}
```

✅ **Output:**

```
App Name: MyApp
Max Users: 1000
```

---

5. Constant Values in Enums

For **fixed sets of constants**, use **enums** instead of `final`.

Example: Using Enums for Constants

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}

public class Main {
    public static void main(String[] args) {
        Day today = Day.WEDNESDAY;
        System.out.println("Today is: " + today);
    }
}
```

✅ **Output:**

```
Today is: WEDNESDAY
```

---

6. Why Use Constants?

✅ **Improves Code Readability**
Instead of using `"admin"` multiple times:

```
if (userRole.equals("admin")) { }
```

Use a constant:

```
final String ADMIN_ROLE = "admin";
if (userRole.equals(ADMIN_ROLE)) { }
```

✅ **Prevents Accidental Changes**
Using `final` ensures that the value **cannot be modified**, preventing bugs.

✅ **Enhances Performance**
Constants **may be optimized** by the compiler since their values never change.

---

7. Difference Between `final`, `static final`, and `enum`

Type	Modifiable?	Instance or Class Level?	Best Use Case
`final`	❌ No	Instance or Local	Prevent variable modification
`static final`	❌ No	Class Level (Shared)	Global constants
`enum`	❌ No	Class Level	Fixed set of values

Conclusion

- Use **`final`** for immutable variables.
- Use **`static final`** for shared constants.
- Use **interfaces** for global constants.
- Use **enums** for predefined constant sets.

Scope and Lifetime of Variables in Java

In Java, **variable scope** determines **where** a variable can be accessed, and **lifetime** defines **how long** the variable exists in memory.

---

1. Types of Variable Scope in Java

Java has **four types of variable scope**:

1. **Local Scope (Method-Level Variables)**
2. **Instance Scope (Object-Level Variables)**
3. **Static Scope (Class-Level Variables)**
4. **Block Scope (Loop/Conditional Variables)**

---

2. Local Variables (Method-Level Scope)

Scope:

- Declared **inside a method, constructor, or block**.
- Can be accessed **only within** the method or block where it is defined.
- **Cannot be used outside** that method/block.

Lifetime:

- Created **when the method starts execution**.
- Destroyed **when the method finishes execution**.

Example:

```
public class LocalVariableExample {
    public void display() {
        int x = 10; // Local variable
        System.out.println("Local variable: " + x);
    }

    public static void main(String[] args) {
        LocalVariableExample obj = new LocalVariableExample();
        obj.display();

        // System.out.println(x); // ✕ ERROR: x is not accessible here
    }
}
```

✅ Output:

```
Local variable: 10
```

---

3. Instance Variables (Object-Level Scope)

Scope:

- Declared **inside a class but outside a method**.
- Each **object** has its own copy of instance variables.
- Can be accessed **throughout the class**.

Lifetime:

- Created **when an object is instantiated** (`new` keyword).
- Exists **until the object is garbage collected**.

Example:

```
class Car {
    String brand; // Instance variable

    void setBrand(String b) {
        brand = b;
    }
```

```
    void displayBrand() {
        System.out.println("Car brand: " + brand);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car(); // New object
        car1.setBrand("Toyota");
        car1.displayBrand();

        Car car2 = new Car(); // Another object
        car2.setBrand("Honda");
        car2.displayBrand();
    }
}
```

✅ **Output:**

```
Car brand: Toyota
Car brand: Honda
```

🔹 **Each object (`car1`, `car2`) has its own separate `brand` variable.**

---

4. Static Variables (Class-Level Scope)

Scope:

- Declared **inside a class** using the `static` keyword.
- Shared by **all objects of the class**.
- Accessed **directly using the class name**.

Lifetime:

- Created **when the class is loaded** in memory.
- Exists **until the program terminates**.

Example:

```
class Company {
    static String companyName = "TechCorp"; // Static variable

    void showCompany() {
        System.out.println("Company: " + companyName);
    }
}

public class Main {
    public static void main(String[] args) {
        Company emp1 = new Company();
        emp1.showCompany();

        Company emp2 = new Company();
```

```
        emp2.showCompany();

        // Accessing static variable directly using class name
        System.out.println("Company: " + Company.companyName);
    }
}
```

✅ **Output:**

```
Company: TechCorp
Company: TechCorp
Company: TechCorp
```

🔷 **All objects share the same `companyName` variable.**

---

5. Block Scope (Loop/Conditional Scope)

Scope:

- Declared inside **loops, conditional blocks, or curly braces `{}`**.
- Accessible **only within** the block `{}`.

Lifetime:

- Created **when the block starts** execution.
- Destroyed **when the block ends**.

Example:

```
public class BlockScopeExample {
    public static void main(String[] args) {
        if (true) {
            int num = 50; // Block scope
            System.out.println("Inside block: " + num);
        }

        // System.out.println(num); // ❌ ERROR: num is not accessible here
    }
}
```

✅ **Output:**

```
Inside block: 50
```

🔷 **The variable `num` exists only inside the `if` block.**

---

6. Lifetime of Variables Summary

Variable Type	Created When	Destroyed When
Local Variable	Method starts	Method ends
Instance Variable	Object is created	Object is garbage collected
Static Variable	Class is loaded	Program ends
Block Variable	Block starts	Block ends

---

7. Example Showing All Types of Variable Scope

```
class Example {
    static int staticVar = 100; // Static variable (Class-Level)
    int instanceVar = 50;       // Instance variable (Object-Level)

    void method() {
        int localVar = 20; // Local variable (Method-Level)

        for (int i = 0; i < 2; i++) { // Block variable
            int blockVar = i * 10;
            System.out.println("Block variable: " + blockVar);
        }

        // System.out.println(blockVar); // ✕ ERROR: blockVar is out of
scope
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.method();

        System.out.println("Instance variable: " + obj.instanceVar);
        System.out.println("Static variable: " + Example.staticVar);
    }
}
```

✅ Output:

```
Block variable: 0
Block variable: 10
Instance variable: 50
Static variable: 100
```

---

8. Key Takeaways

- **Local Variables:** Exist only inside a method and must be initialized before use.
- **Instance Variables:** Belong to an object and exist as long as the object exists.
- **Static Variables:** Belong to the class and remain throughout the program.
- **Block Variables:** Exist only inside `{}` blocks like loops and conditionals.

Operators in Java

Operators in Java are symbols that perform operations on variables and values. Java supports different types of operators, including **arithmetic, relational, logical, bitwise, assignment, and ternary operators**.

---

1. Types of Operators in Java

Operator Type	Example	Description
Arithmetic	`+, -, *, /, %`	Performs basic mathematical operations
Relational	`==, !=, >, <, >=, <=`	Compares values and returns `true` or `false`
Logical	`&&,` `	
Bitwise	`&,` `	`, ^, ~, <<, >>, >>>` `
Assignment	`=, +=, -=, *=, /=, %=`	Assigns values to variables
Ternary	`condition ? value1 : value2`	Shortens `if-else` statements
Unary	`+, -, ++, --`	Works with a single operand
Instanceof	`obj instanceof ClassName`	Checks if an object belongs to a class

---

2. Arithmetic Operators

Used for basic mathematical calculations.

Operator	Description	Example (`a = 10, b = 5`)	Result
`+`	Addition	`a + b`	15
`-`	Subtraction	`a - b`	5
`*`	Multiplication	`a * b`	50
`/`	Division	`a / b`	2
`%`	Modulus (Remainder)	`a % b`	0

Example Code

```
public class ArithmeticExample {
```

```
    public static void main(String[] args) {
        int a = 10, b = 5;
        System.out.println("Addition: " + (a + b)); // 15
        System.out.println("Subtraction: " + (a - b)); // 5
        System.out.println("Multiplication: " + (a * b)); // 50
        System.out.println("Division: " + (a / b)); // 2
        System.out.println("Modulus: " + (a % b)); // 0
    }
}
```

---

3. Relational (Comparison) Operators

Used to compare two values and return a boolean (`true` or `false`).

Operator	Description	Example (`a = 10, b = 5`)	Result
`==`	Equal to	`a == b`	`false`
`!=`	Not equal to	`a != b`	`true`
`>`	Greater than	`a > b`	`true`
`<`	Less than	`a < b`	`false`
`>=`	Greater or equal	`a >= b`	`true`
`<=`	Less or equal	`a <= b`	`false`

Example Code

```
public class RelationalExample {
    public static void main(String[] args) {
        int a = 10, b = 5;
        System.out.println(a > b); // true
        System.out.println(a < b); // false
        System.out.println(a == b); // false
        System.out.println(a != b); // true
    }
}
```

---

4. Logical Operators

Used to perform logical operations on boolean values.

Operator	Description	Example (`x = true, y = false`)	Result
`&&`	AND	`x && y`	`false`
`` ` ``		`` ` ``	OR

Operator	Description	Example (`x = true, y = false`)	Result
`!`	NOT	`!x`	false

Example Code

```
public class LogicalExample {
    public static void main(String[] args) {
        boolean x = true, y = false;
        System.out.println(x && y); // false
        System.out.println(x || y); // true
        System.out.println(!x); // false
    }
}
```

---

5. Bitwise Operators

Used to perform operations at the **bit level**.

Operator	Description	Example (`a = 5 (0101 in binary), b = 3 (0011)`)	Result (Binary/Decimal)
`&`	Bitwise AND	`a & b` (0101 & 0011)	`0001 (1)`
`` ` ``	`` ` ``	Bitwise OR	`` `a ``
`^`	Bitwise XOR	`a ^ b` (0101 ^ 0011)	`0110 (6)`
`~`	Bitwise NOT	`~a` (~0101)	`1010 (-6)`
`<<`	Left Shift	`a << 1` (0101 << 1)	`1010 (10)`
`>>`	Right Shift	`a >> 1` (0101 >> 1)	`0010 (2)`

Example Code

```
public class BitwiseExample {
    public static void main(String[] args) {
        int a = 5, b = 3;
        System.out.println(a & b); // 1
        System.out.println(a | b); // 7
        System.out.println(a ^ b); // 6
        System.out.println(~a); // -6
        System.out.println(a << 1); // 10
        System.out.println(a >> 1); // 2
    }
}
```

---

6. Assignment Operators

Used to assign values to variables.

Operator	Example (`a = 10`)	Equivalent To
`=`	`a = 10`	`a = 10`
`+=`	`a += 5`	`a = a + 5`
`-=`	`a -= 5`	`a = a - 5`
`*=`	`a *= 5`	`a = a * 5`
`/=`	`a /= 5`	`a = a / 5`
`%=`	`a %= 5`	`a = a % 5`

---

7. Ternary Operator (`?:`)

Used as a shorthand for `if-else`.

Syntax

```
condition ? value_if_true : value_if_false;
```

Example

```
public class TernaryExample {
    public static void main(String[] args) {
        int a = 10, b = 5;
        int min = (a < b) ? a : b;
        System.out.println("Minimum: " + min); // 5
    }
}
```

---

8. Unary Operators

Operator	Example	Effect
`+`	`+a`	Positive value
`-`	`-a`	Negative value
`++`	`a++` or `++a`	Increments by 1
`--`	`a--` or `--a`	Decrements by 1

Conclusion

- **Arithmetic** → Math operations
- **Relational** → Compare values
- **Logical** → Boolean logic
- **Bitwise** → Binary operations
- **Assignment** → Assign values
- **Ternary** → Short `if-else`
- **Unary** → Single operand

Type Conversion and Casting in Java

In Java, **type conversion** is the process of converting one data type into another. There are two main types:

1. **Implicit Type Conversion (Widening)**
2. **Explicit Type Conversion (Narrowing or Casting)**

---

1. Implicit Type Conversion (Widening)

- Happens **automatically** when converting a **smaller** data type into a **larger** data type.
- No data loss occurs.
- Example: `byte → short → int → long → float → double`

Example Code

```
public class WideningExample {
    public static void main(String[] args) {
        int num = 100;
        double d = num; // Automatic conversion from int to double
        System.out.println("Int value: " + num);
        System.out.println("Double value: " + d);
    }
}
```

✅ Output:

```
Int value: 100
Double value: 100.0
```

🔷 Why does it work?

- `int` (4 bytes) → `double` (8 bytes)
- Java automatically converts `int` to `double` without data loss.

---

2. Explicit Type Conversion (Narrowing or Casting)

- **Manually** converting a **larger** data type into a **smaller** one.

- **Data loss may occur** if the value exceeds the range of the target type.
- Uses **casting** syntax:
- `dataType variableName = (dataType) value;`

Example Code

```
public class NarrowingExample {
    public static void main(String[] args) {
        double d = 100.99;
        int num = (int) d; // Manual casting from double to int
        System.out.println("Double value: " + d);
        System.out.println("Integer value after casting: " + num);
    }
}
```

✅ Output:

```
Double value: 100.99
Integer value after casting: 100
```

🔷 Why is `99` removed?

- **Double (`100.99`) → Integer (`100`)**
- The decimal part is **truncated** (not rounded).

---

3. Type Conversion between Primitive and Non-Primitive Types

(a) Converting `int` to `String`

```
public class IntToString {
    public static void main(String[] args) {
        int num = 123;
        String str = Integer.toString(num); // Convert int to String
        System.out.println("String value: " + str);
    }
}
```

(b) Converting `String` to `int`

```
public class StringToInt {
    public static void main(String[] args) {
        String str = "123";
        int num = Integer.parseInt(str); // Convert String to int
        System.out.println("Integer value: " + num);
    }
}
```

---

4. Type Conversion Between `char` and `int`

Example

```
public class CharToInt {
    public static void main(String[] args) {
        char c = 'A';
        int ascii = c; // Implicit conversion (char → int)
        System.out.println("ASCII value of 'A': " + ascii);
    }
}
```

✅ **Output:**

```
ASCII value of 'A': 65
```

---

5. Automatic Type Promotion in Expressions

When performing calculations, smaller types (`byte`, `short`, `char`) are automatically converted to **int**.

Example

```
public class TypePromotion {
    public static void main(String[] args) {
        byte a = 10;
        byte b = 20;
        int result = a * b; // byte is converted to int automatically
        System.out.println("Result: " + result);
    }
}
```

✅ **Output:**

```
Result: 200
```

---

6. Summary Table

Conversion Type	Example	Automatic?	Data Loss?
Widening (Implicit)	`int → double`	✅ Yes	❌ No
Narrowing (Explicit)	`double → int`	❌ No	✅ Yes (truncation)
String to int	`Integer.parseInt("123")`	❌ No	❌ No
int to String	`Integer.toString(123)`	❌ No	❌ No
char to int	`'A' → 65`	✅ Yes	❌ No

---

7. Key Takeaways

✅ **Widening** happens automatically (no data loss).
✅ **Narrowing** requires explicit casting (possible data loss).
✅ **String to int** conversion uses `Integer.parseInt()`.
✅ **int to String** conversion uses `Integer.toString()`.

Accepting Input from the Keyboard in Java

In Java, you can take user input from the keyboard using different methods. The most common ways are:

1. **Using `Scanner` (Recommended)**
2. **Using `BufferedReader` (More Efficient)**
3. **Using `Console` (For Secure Input like Passwords)**

---

1. Using `Scanner` (Most Common & Easy)

The `Scanner` class (from `java.util` package) is commonly used to take input of different data types like `int`, `double`, `String`, etc.

Example: Accepting Different Inputs

```
import java.util.Scanner;

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

        // Taking integer input
        System.out.print("Enter an integer: ");
        int num = scanner.nextInt();

        // Taking double input
        System.out.print("Enter a decimal number: ");
        double decimal = scanner.nextDouble();

        // To consume the leftover newline character
        scanner.nextLine();

        // Taking string input
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.println("\nYou entered:");
        System.out.println("Integer: " + num);
        System.out.println("Decimal: " + decimal);
        System.out.println("Name: " + name);

        scanner.close(); // Close scanner to prevent resource leak
    }
}
```

Explanation:

- `nextInt()` → Reads an **integer**.
- `nextDouble()` → Reads a **double**.
- `nextLine()` → Reads a **full line (String)**.
- `scanner.nextLine();` is used after `nextInt()` or `nextDouble()` to consume the leftover **newline (`\n`)**.

✅ **Output Example:**

```
Enter an integer: 10
Enter a decimal number: 20.5
Enter your name: John Doe

You entered:
Integer: 10
Decimal: 20.5
Name: John Doe
```

---

2. Using `BufferedReader` (More Efficient)

The `BufferedReader` class reads input more efficiently than `Scanner`, but it requires manual **parsing** for numbers.

Example:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        System.out.print("Enter an integer: ");
        int num = Integer.parseInt(reader.readLine());

        System.out.print("Enter a decimal number: ");
        double decimal = Double.parseDouble(reader.readLine());

        System.out.print("Enter your name: ");
        String name = reader.readLine();

        System.out.println("\nYou entered:");
        System.out.println("Integer: " + num);
        System.out.println("Decimal: " + decimal);
        System.out.println("Name: " + name);
    }
}
```

Explanation:

- `reader.readLine()` reads the input as a **string**.
- `Integer.parseInt(reader.readLine())` converts it to an **integer**.
- `Double.parseDouble(reader.readLine())` converts it to a **double**.

✅ **Pros:** Faster than `Scanner` for large inputs.
❌ **Cons:** More complex due to manual parsing.

---

3. Using `Console` (For Password Input)

If you're taking **password input**, `Console` is safer because it **hides user input**.

Example:

```
import java.io.Console;

public class ConsoleExample {
    public static void main(String[] args) {
        Console console = System.console();

        if (console == null) {
            System.out.println("No console available");
            return;
        }

        String name = console.readLine("Enter your name: ");
        char[] password = console.readPassword("Enter your password: ");

        System.out.println("Name: " + name);
        System.out.println("Password: " + new String(password));
    }
}
```

✅ **Pros:** Secure input for passwords.
❌ **Cons:** Doesn't work in some IDEs (e.g., Eclipse, IntelliJ). Works in **command line (CMD, Terminal).**

Comparison Table

Method	Best For	Works in IDEs?	Handles Multiple Data Types?	Performance
`Scanner`	Simple user input	✅ Yes	✅ Yes	Medium
`BufferedReader`	Large input (performance)	✅ Yes	❌ No (manual conversion)	High
`Console`	Secure password input	❌ No (only CMD)	❌ No (String only)	Medium

---

Conclusion

✅ **Use `Scanner`** for most cases.
✅ **Use `BufferedReader`** for performance-heavy input.
✅ **Use `Console`** for password input in command line.

Reading Input with `java.util.Scanner` Class

The `Scanner` class in Java is the most commonly used method for reading user input. It allows reading input of various data types like `int`, `double`, `String`, and more.

---

1. Importing the Scanner Class

To use the `Scanner` class, you need to import it from the `java.util` package:

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

---

2. Creating a Scanner Object

A `Scanner` object is created using:

```
Scanner scanner = new Scanner(System.in);
```

- `System.in` refers to **standard input** (keyboard).
- The `Scanner` object can now be used to read different types of input.

---

3. Reading Different Data Types

The `Scanner` class provides various methods to read different types of input.

Method	Data Type Read
`nextInt()`	Integer (`int`)
`nextDouble()`	Decimal (`double`)
`nextFloat()`	Decimal (`float`)
`nextLong()`	Long Integer (`long`)
`nextBoolean()`	Boolean (`true/false`)
`next()`	Single word (`String`)
`nextLine()`	Full sentence (`String`)

---

4. Example: Reading Different Inputs

```
import java.util.Scanner;

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

        // Integer input
        System.out.print("Enter an integer: ");
        int num = scanner.nextInt();

        // Double input
        System.out.print("Enter a decimal number: ");
        double decimal = scanner.nextDouble();

        // To consume the newline left by nextInt() or nextDouble()
        scanner.nextLine();

        // String input (single word)
        System.out.print("Enter a single word: ");
        String word = scanner.next();

        // String input (full line)
        scanner.nextLine(); // Consume the leftover newline
        System.out.print("Enter a full sentence: ");
        String sentence = scanner.nextLine();

        // Boolean input
        System.out.print("Enter true or false: ");
        boolean boolValue = scanner.nextBoolean();

        System.out.println("\nYou entered:");
        System.out.println("Integer: " + num);
        System.out.println("Decimal: " + decimal);
        System.out.println("Single word: " + word);
        System.out.println("Full sentence: " + sentence);
        System.out.println("Boolean: " + boolValue);

        scanner.close(); // Close scanner to prevent resource leak
    }
}
```

✅ Sample Output:

```
Enter an integer: 10
Enter a decimal number: 3.14
Enter a single word: Hello
Enter a full sentence: Java is fun!
Enter true or false: true

You entered:
Integer: 10
Decimal: 3.14
Single word: Hello
Full sentence: Java is fun!
Boolean: true
```

---

5. Handling `nextLine()` Issue

🔷 If `nextLine()` is used after `nextInt()` or `nextDouble()`, a **newline issue** occurs because `nextInt()` and `nextDouble()` don't consume the `Enter` key (`\n`).

💡 **Solution:** Add an extra `scanner.nextLine();` to consume the newline before using `nextLine()`.

```
scanner.nextLine(); // Consume the newline left by nextInt() or nextDouble()
```

---

6. Checking If Input is Available (`hasNext()`)

To prevent errors, you can check if the input is available before reading it:

```
if (scanner.hasNextInt()) {
    int num = scanner.nextInt();
    System.out.println("You entered: " + num);
} else {
    System.out.println("Invalid input! Please enter an integer.");
}
```

---

7. Summary

✅ Use `Scanner` to read different data types.
✅ Always close `Scanner` after use (`scanner.close();`).
✅ Use `scanner.nextLine();` after `nextInt()` or `nextDouble()` to avoid newline issues.

Displaying Output with `System.out.printf()` in Java

In Java, `System.out.printf()` is used for **formatted output**. It allows precise control over how data is displayed, making it useful for aligning text, formatting numbers, and controlling decimal places.

---

1. Basic Syntax of `printf()`

```
System.out.printf("format-string", arguments);
```

- `"format-string"` → Defines how the output is formatted.
- `arguments` → Values to be inserted into the format string.

---

2. Common Format Specifiers

Specifier	Data Type	Example
`%d`	Integer (`int`)	`printf("%d", 10);` → `10`
`%f`	Floating point (`float`/`double`)	`printf("%.2f", 3.14159);` → `3.14`
`%s`	String (`String`)	`printf("%s", "Java");` → `Java`
`%c`	Character (`char`)	`printf("%c", 'A');` → `A`
`%b`	Boolean (`boolean`)	`printf("%b", true);` → `true`

---

3. Example: Using `printf()`

```
public class PrintfExample {
    public static void main(String[] args) {
        String name = "Alice";
        int age = 25;
        double height = 5.6789;

        System.out.printf("Name: %s, Age: %d, Height: %.2f meters%n", name,
age, height);
    }
}
```

✅ Output:

```
Name: Alice, Age: 25, Height: 5.68 meters
```

- `%s` → Inserts `name` (`Alice`).
- `%d` → Inserts `age` (`25`).
- `%.2f` → Formats `height` (`5.6789`) to **2 decimal places** (`5.68`).
- `%n` → Inserts a **new line** (same as `\n` but platform-independent).

---

4. Controlling Width and Alignment

You can control the **width** of printed values:

Format	Meaning
`%5d`	Prints integer with **at least** 5 spaces (right-aligned).
`%-5d`	Left-aligns integer within 5 spaces.

Format	Meaning
`%10s`	Prints a string with **at least** 10 spaces (right-aligned).
`%-10s`	Left-aligns a string within 10 spaces.

Example: Alignment

```
public class AlignmentExample {
    public static void main(String[] args) {
        System.out.printf("|%10s|%5d|%7.2f|%n", "John", 25, 3.14159);
        System.out.printf("|%-10s|%-5d|%-7.2f|%n", "Alice", 30, 2.71828);
    }
}
```

✅ **Output:**

```
|      John|   25|   3.14|
|Alice     |30   |2.72   |
```

- `%10s` → **Right-aligns** `"John"` in a **10-character space**.
- `%-10s` → **Left-aligns** `"Alice"` in a **10-character space**.
- `%7.2f` → **Right-aligns** `3.14159`, showing **2 decimal places**.

---

5. Formatting Numbers with Thousands Separator

Use **,** **(comma)** to add a thousands separator.

```
System.out.printf("Formatted number: %,d%n", 1000000);
```

✅ **Output:**

```
Formatted number: 1,000,000
```

---

6. Displaying Percentage

Use `%%` to print a literal `%` symbol.

```
System.out.printf("Pass rate: %.2f%%%n", 95.5);
```

✅ **Output:**

```
Pass rate: 95.50%
```

---

7. Summary

Feature	Example	Output
Integer	`printf("%d", 100);`	`100`
Float (2 decimal places)	`printf("%.2f", 3.14159);`	`3.14`
Right-aligned text	`printf("	%10s
Left-aligned text	`printf("	%-10s
Thousands separator	`printf("%,d", 1000000);`	`1,000,000`
Percentage	`printf("%.2f%%", 95.5);`	`95.50%`

🚀 Conclusion

✅ **Use `printf()` for formatted output.**
✅ **Control decimal places with `%.xf` (e.g., `%.2f`).**
✅ **Align text using width specifiers (e.g., `%10s`, `%-10s`).**
✅ **Use `%,d` for large numbers and `%%` for percentages.**

Displaying Formatted Output with `String.format()` in Java

The `String.format()` method in Java allows us to format and store strings without printing them immediately. It works similarly to `System.out.printf()` but returns a **formatted string** instead of directly displaying it.

---

1. Basic Syntax of `String.format()`

```
String formattedString = String.format("format-string", arguments);
```

- `"format-string"` → Defines the formatting pattern.
- `arguments` → Values inserted into the formatted string.

📌 **Unlike `printf()`, `String.format()` does NOT print output directly. Instead, it returns a formatted string that you can store in a variable.**

---

2. Common Format Specifiers

Specifier	Data Type	Example
`%d`	Integer (`int`)	`String.format("%d", 100)` → `"100"`
`%f`	Floating point (`float`/`double`)	`String.format("%.2f", 3.14159)` → `"3.14"`
`%s`	String (`String`)	`String.format("%s", "Java")` → `"Java"`
`%c`	Character (`char`)	`String.format("%c", 'A')` → `"A"`
`%b`	Boolean (`boolean`)	`String.format("%b", true)` → `"true"`

---

3. Example: Using `String.format()`

```
public class StringFormatExample {
    public static void main(String[] args) {
        String name = "Alice";
        int age = 25;
        double height = 5.6789;

        // Using String.format() to create a formatted string
        String result = String.format("Name: %s, Age: %d, Height: %.2f
meters", name, age, height);

        // Printing the formatted string
        System.out.println(result);
    }
}
```

✅ Output:

```
Name: Alice, Age: 25, Height: 5.68 meters
```

- `%s` → Inserts `name` (`Alice`).
- `%d` → Inserts `age` (`25`).
- `%.2f` → Formats `height` (`5.6789`) to **2 decimal places** (`5.68`).

---

4. Controlling Width and Alignment

Just like `printf()`, you can control **width and alignment** using `String.format()`.

Format	Meaning
`%5d`	Right-aligns integer within 5 spaces.

Format	Meaning
`%-5d`	Left-aligns integer within 5 spaces.
`%10s`	Right-aligns string within 10 spaces.
`%-10s`	Left-aligns string within 10 spaces.

Example: Right & Left Alignment

```
public class AlignmentExample {
    public static void main(String[] args) {
        String rightAligned = String.format("|%10s|%5d|%7.2f|", "John", 25,
3.14159);
        String leftAligned = String.format("|%-10s|%-5d|%-7.2f|", "Alice",
30, 2.71828);

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

✅ **Output:**

```
|      John|   25|   3.14|
|Alice     |30   |2.72   |
```

- `%10s` → **Right-aligns** `"John"` in a **10-character space**.
- `%-10s` → **Left-aligns** `"Alice"` in a **10-character space**.
- `%7.2f` → **Right-aligns** `3.14159`, showing **2 decimal places**.

---

5. Formatting Numbers with Thousands Separator

Use `,` **(comma)** to add a thousands separator.

```
String formattedNumber = String.format("Formatted number: %,d", 1000000);
System.out.println(formattedNumber);
```

✅ **Output:**

```
Formatted number: 1,000,000
```

---

6. Displaying Percentage

Use `%%` to print a `%` symbol.

```
String percentage = String.format("Pass rate: %.2f%%", 95.5);
System.out.println(percentage);
```

✅ **Output:**

```
Pass rate: 95.50%
```

---

7. Using `String.format()` in UI Development

`String.format()` is useful in UI applications when formatting text before displaying it.

Example: Creating a Message for a GUI Label

```
String message = String.format("Hello %s! Your score is %d.", "John", 95);
System.out.println(message);
```

✅ **Output:**

```
Hello John! Your score is 95.
```

---

8. Summary

Feature	Example	Output
Integer	`String.format("%d", 100);`	`"100"`
Float (2 decimal places)	`String.format("%.2f", 3.14159);`	`"3.14"`
Right-aligned text	`String.format("	%10s
Left-aligned text	`String.format("	%-10s
Thousands separator	`String.format("%,d", 1000000);`	`"1,000,000"`
Percentage	`String.format("%.2f%%", 95.5);`	`"95.50%"`

🚀 Conclusion

✅ **Use `String.format()` when you need a formatted string without printing immediately.**
✅ **Format numbers, strings, and text alignment easily.**
✅ **Great for UI messages, reports, and logs.**

Control Statements in Java

Control statements in Java **direct the flow of execution** in a program. They are categorized into three types:

1. **Selection (Decision-Making) Statements**
    - `if`, `if-else`, `if-else-if`
    - `switch`
2. **Iteration (Looping) Statements**
    - `for`, `while`, `do-while`
3. **Jump Statements**
    - `break`, `continue`, `return`

---

1. Selection (Decision-Making) Statements

(a) `if` Statement

Executes a block of code **only if** a condition is `true`.

```
if (condition) {
    // Code to execute if condition is true
}
```

✅ **Example:**

```
int num = 10;
if (num > 0) {
    System.out.println("Positive Number");
}
```

Output:

```
Positive Number
```

(b) `if-else` Statement

Executes one block if `true`, another if `false`.

```
if (condition) {
    // Executes if condition is true
} else {
    // Executes if condition is false
}
```

✅ **Example:**

```
int num = -5;
if (num > 0) {
```

```
    System.out.println("Positive");
} else {
    System.out.println("Negative");
}
```

Output:

```
Negative
```

(c) `if-else-if` Ladder

Used for multiple conditions.

```
if (condition1) {
    // Code for condition1
} else if (condition2) {
    // Code for condition2
} else {
    // Default case
}
```

✅ **Example:**

```
int marks = 85;
if (marks >= 90) {
    System.out.println("Grade: A");
} else if (marks >= 75) {
    System.out.println("Grade: B");
} else {
    System.out.println("Grade: C");
}
```

Output:

```
Grade: B
```

(d) `switch` Statement

Used when a variable has multiple possible values.

```
switch (expression) {
    case value1:
        // Code for value1
        break;
    case value2:
        // Code for value2
        break;
    default:
        // Code if no match found
}
```

✅ **Example:**

```
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
```

---

2. Iteration (Looping) Statements

(a) `for` Loop

Used when the number of iterations is known.

```
for (initialization; condition; update) {
    // Loop body
}
```

✅ Example:

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Count: " + i);
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

(b) `while` Loop

Executes **as long as** the condition is `true`.

```
while (condition) {
    // Loop body
}
```

✅ Example:

```
int i = 1;
while (i <= 5) {
    System.out.println("Iteration: " + i);
    i++;
}
```

(c) `do-while` Loop

Similar to `while`, but **executes at least once**.

```
do {
    // Loop body
} while (condition);
```

✅ **Example:**

```
int i = 1;
do {
    System.out.println("Value: " + i);
    i++;
} while (i <= 3);
```

Output:

```
Value: 1
Value: 2
Value: 3
```

---

3. Jump Statements

(a) `break` Statement

Exits a loop immediately.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        break;
    }
    System.out.println(i);
}
```

Output:

```
1
2
```

(b) `continue` Statement

Skips the current iteration and continues.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println(i);
}
```

Output:

```
1
2
4
5
```

(c) `return` Statement

Exits from a method.

```
public class ReturnExample {
    static void check(int num) {
        if (num < 0) {
            System.out.println("Negative number");
            return; // Exit method
        }
        System.out.println("Positive number");
    }

    public static void main(String[] args) {
        check(-5);
        check(10);
    }
}
```

Output:

```
Negative number
Positive number
```

Summary of Control Statements

Category	Statement	Purpose
Selection	`if`, `if-else`, `switch`	Decision-making
Iteration	`for`, `while`, `do-while`	Repeating execution
Jump	`break`, `continue`, `return`	Control flow

Would you like **practice problems** on control statements? 🚀

UNIT – II

Arrays in Java

An **array** is a data structure that stores multiple values of the **same data type** in contiguous memory locations. It allows **fast access** and **efficient storage** of elements.

---

1. Declaring and Initializing Arrays

(a) Declaration

```
dataType[] arrayName;
```

OR

```
dataType arrayName[];
```

📌 **Example:** Declaring an integer array

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

(b) Allocation (Instantiation)

```
arrayName = new dataType[size];
```

📌 **Example:** Allocating an array of size 5

```
numbers = new int[5];
```

(c) Declaration + Initialization

```
int[] numbers = new int[5];  // Creates an array of size 5 with default values (0)
```

(d) Direct Initialization (Shortcut)

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

---

2. Accessing Array Elements

Each element in an array is accessed using an **index**, starting from `0`.

✅ **Example: Accessing and Modifying Elements**

```
int[] numbers = {10, 20, 30, 40, 50};
System.out.println(numbers[0]);  // Output: 10
```

```
// Modifying an element
numbers[2] = 100;
System.out.println(numbers[2]);  // Output: 100
```

📌 **Index starts from `0` and goes up to `length - 1`**
🚨 Accessing an invalid index (`numbers[5]`) will cause an **ArrayIndexOutOfBoundsException**.

---

3. Traversing an Array

(a) Using a `for` Loop

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

for (int i = 0; i < numbers.length; i++) {
    System.out.println("Element at index " + i + ": " + numbers[i]);
}
```

✅ **Output:**

```
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
```

(b) Using an Enhanced `for` Loop (for-each)

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

for (int num : numbers) {
    System.out.println(num);
}
```

✅ **Output:**

```
10
20
30
40
50
```

📌 **Note:** In a `for-each` loop, you **cannot modify** array elements.

---

4. Finding the Length of an Array

Use the `.length` property to find the size of an array.

```
int[] numbers = {10, 20, 30, 40, 50};
System.out.println("Array Length: " + numbers.length);
```

✅ **Output:**

```
Array Length: 5
```

---

5. Multi-Dimensional Arrays

An **array of arrays**, also called a **matrix**, is declared as follows:

(a) Declaring a 2D Array

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

(b) Initializing a 2D Array

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

(c) Accessing Elements in a 2D Array

```
System.out.println(matrix[1][2]);  // Output: 6
```

(d) Traversing a 2D Array

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

✅ **Output:**

```
1 2 3
4 5 6
7 8 9
```

---

6. Copying an Array

(a) Using a Loop

```
int[] original = {1, 2, 3, 4, 5};
int[] copy = new int[original.length];

for (int i = 0; i < original.length; i++) {
    copy[i] = original[i];
}
```

(b) Using `System.arraycopy()`

```
System.arraycopy(original, 0, copy, 0, original.length);
```

(c) Using `Arrays.copyOf()`

```
int[] copy = java.util.Arrays.copyOf(original, original.length);
```

---

7. Sorting an Array

Use `Arrays.sort()` to sort an array in ascending order.

```
import java.util.Arrays;

int[] numbers = {5, 2, 8, 1, 3};
Arrays.sort(numbers);

System.out.println(Arrays.toString(numbers));  // Output: [1, 2, 3, 5, 8]
```

---

8. Searching in an Array

Use **linear search** or **binary search**:

(a) Linear Search

```
public class LinearSearch {
    public static int search(int[] arr, int key) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == key) {
                return i;  // Return index
            }
        }
        return -1;  // Not found
    }

    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        System.out.println("Index: " + search(numbers, 30));  // Output: 2
    }
}
```

(b) Binary Search (Array must be sorted)

```
import java.util.Arrays;

int[] numbers = {10, 20, 30, 40, 50};
int index = Arrays.binarySearch(numbers, 30);

System.out.println("Index: " + index);  // Output: 2
```

---

9. Summary of Java Arrays

Feature	Example
Declaration	`int[] arr;`
Allocation	`arr = new int[5];`
Initialization	`int[] arr = {1, 2, 3};`
Access Element	`arr[0]`
Length	`arr.length`
Looping (for)	`for (int i = 0; i < arr.length; i++)`
Looping (for-each)	`for (int num : arr)`
2D Array Declaration	`int[][] matrix = new int[3][3];`
2D Array Access	`matrix[1][2]`
Sorting	`Arrays.sort(arr);`
Searching	`Arrays.binarySearch(arr, key);`

🚀 Conclusion

✅ Arrays allow **efficient storage and retrieval** of data.
✅ You can use **loops** and **utility methods** to process arrays effectively.
✅ Java provides powerful **sorting and searching** operations using `Arrays` class.

Would you like to see **LinkedLists vs Arrays comparison** next? 🤔

Command Line Arguments in Java

Command-line arguments allow users to **pass information** to a Java program when it is executed from the terminal or command prompt. These arguments are **received as an array of strings (`String[] args`)** in the `main` method.

---

1. Accessing Command-Line Arguments

Example: Basic Program to Print Command-Line Arguments

```
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 + 1) + ": " + args[i]);
        }
    }
}
```

How to Run this Program?

1. Compile:
2. `javac CommandLineDemo.java`
3. Execute with arguments:
4. `java CommandLineDemo Hello World 123`

✅ Output:

```
Number of Arguments: 3
Argument 1: Hello
Argument 2: World
Argument 3: 123
```

---

2. Converting Command-Line Arguments to Other Data Types

Since all arguments are received as **strings**, you may need to **convert them** into numbers using **parsing methods**.

Example: Sum of Two Numbers Passed as Command-Line Arguments

```
public class SumCalculator {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Please provide two numbers.");
            return;
        }

        // Convert strings to integers
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);

        int sum = num1 + num2;
        System.out.println("Sum: " + sum);
```

```
    }
}
```

Run the Program:

```
java SumCalculator 10 20
```

✅ Output:

```
Sum: 30
```

---

3. Handling Errors & Exceptions

If the user provides **invalid inputs** (e.g., non-numeric values for an expected number), the program may throw an exception. To prevent this, we handle errors using `try-catch`.

Example: Handling Invalid Inputs

```
public class SafeCalculator {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Usage: java SafeCalculator <num1> <num2>");
            return;
        }

        try {
            int num1 = Integer.parseInt(args[0]);
            int num2 = Integer.parseInt(args[1]);
            System.out.println("Sum: " + (num1 + num2));
        } catch (NumberFormatException e) {
            System.out.println("Error: Please enter valid numbers.");
        }
    }
}
```

Run the Program:

```
java SafeCalculator 10 xyz
```

✅ Output:

```
Error: Please enter valid numbers.
```

---

4. Passing Multiple Arguments

You can pass **multiple arguments** and process them using loops.

Example: Finding the Largest Number from Command-Line Arguments

```
public class FindMax {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Please provide numbers.");
            return;
        }

        try {
```

```
            int max = Integer.parseInt(args[0]);
            for (int i = 1; i < args.length; i++) {
                int num = Integer.parseInt(args[i]);
                if (num > max) {
                    max = num;
                }
            }
            System.out.println("Maximum Number: " + max);
        } catch (NumberFormatException e) {
            System.out.println("Error: Please enter only numbers.");
        }
    }
}
```

Run the Program:

```
java FindMax 12 45 78 34 90 23
```

✅ Output:

```
Maximum Number: 90
```

---

5. Advantages of Command-Line Arguments

✓ **Dynamic Input**: No need to modify code for different inputs.
✓ **Automation**: Useful in **scripts, batch processing, and automation**.
✓ **Efficiency**: Avoids **hardcoded values** inside the program.

---

6. Summary of Java Command-Line Arguments

Feature	Example
Passing arguments	`java ProgramName arg1 arg2 arg3`
Accessing arguments	`args[0], args[1]...`
Length of arguments	`args.length`
Parsing integers	`Integer.parseInt(args[i])`
Handling errors	`try-catch` for `NumberFormatException`
Processing multiple inputs	Loop through `args[]`

Strings in Java – String Class & Methods

A **String** in Java is an **immutable sequence of characters**. Java provides the `String` class in the `java.lang` package with various built-in methods for string manipulation.

---

1. Creating Strings

(a) Using String Literals (Recommended)

```
String str1 = "Hello";
```

📌 **Stored in the String Pool (memory optimization).**

(b) Using `new` Keyword

```
String str2 = new String("Hello");
```

📌 **Stored in Heap Memory (separate object is created).**

---

2. String Methods in Java

(a) Getting String Length

```
String text = "Hello World";
System.out.println(text.length());  // Output: 11
```

(b) Converting Case

```
String message = "Java Programming";
System.out.println(message.toUpperCase());  // Output: JAVA PROGRAMMING
System.out.println(message.toLowerCase());  // Output: java programming
```

(c) Concatenation

```
String first = "Hello";
String second = "World";
String result = first + " " + second;  // Using + operator
System.out.println(result);  // Output: Hello World

// Using concat() method
System.out.println(first.concat(" ").concat(second));  // Output: Hello World
```

(d) Comparing Strings

Using `equals()` (Case-Sensitive)

```
String a = "Java";
String b = "java";
System.out.println(a.equals(b));  // Output: false
```

Using `equalsIgnoreCase()` (Case-Insensitive)

```
System.out.println(a.equalsIgnoreCase(b));  // Output: true
```

Using `compareTo()` (Lexicographic Comparison)

```
String x = "Apple";
String y = "Banana";
System.out.println(x.compareTo(y));  // Output: -1 (Apple comes before
Banana)
```

(e) Checking if a String Contains a Substring

```
String sentence = "Java is fun";
System.out.println(sentence.contains("Java"));  // Output: true
```

(f) Checking if a String Starts/Ends With a Substring

```
String filename = "document.pdf";
System.out.println(filename.startsWith("doc"));  // Output: true
System.out.println(filename.endsWith(".pdf"));  // Output: true
```

(g) Extracting Characters

Getting a Character at an Index

```
String word = "Hello";
System.out.println(word.charAt(1));  // Output: e
```

Extracting a Substring

```
String phrase = "Java Programming";
System.out.println(phrase.substring(5));  // Output: Programming
System.out.println(phrase.substring(5, 11));  // Output: Progra
```

(h) Replacing Characters or Words

```
String text = "I love Java";
System.out.println(text.replace("Java", "Python"));  // Output: I love Python
```

(i) Splitting a String

```
String data = "Apple,Orange,Banana";
String[] fruits = data.split(",");

for (String fruit : fruits) {
    System.out.println(fruit);
}
```

✅ **Output:**

```
Apple
Orange
Banana
```

(j) Removing Whitespace (`trim()`)

```
String str = "   Hello World   ";
System.out.println(str.trim());  // Output: "Hello World"
```

3. String Formatting

```
String name = "Alice";
int age = 25;
String formatted = String.format("Name: %s, Age: %d", name, age);
System.out.println(formatted);  // Output: Name: Alice, Age: 25
```

---

4. Converting Other Data Types to String

```
int num = 100;
String strNum = String.valueOf(num);
System.out.println(strNum + 50);  // Output: 10050 (Concatenation)
```

---

5. Checking if a String is Empty or Blank

```
String emptyString = "";
String blankString = "   ";

System.out.println(emptyString.isEmpty());  // true
System.out.println(blankString.isBlank());  // true
```

---

6. StringBuilder vs StringBuffer (Mutable Strings)

Since `String` is **immutable**, Java provides `StringBuilder` and `StringBuffer` for **mutable** string operations.

Using `StringBuilder` (Faster, Not Thread-Safe)

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

Using `StringBuffer` (Thread-Safe, Slower)

```
StringBuffer sbf = new StringBuffer("Java");
sbf.append(" Rocks!");
System.out.println(sbf);  // Output: Java Rocks!
```

---

7. Summary of Important String Methods

Method	Description	Example
`length()`	Returns string length	`"Hello".length()` → `5`
`toUpperCase()`	Converts to uppercase	`"hello".toUpperCase()` → `"HELLO"`
`toLowerCase()`	Converts to lowercase	`"HELLO".toLowerCase()` → `"hello"`
`concat()`	Concatenates two strings	`"Hello".concat(" World")` → `"Hello World"`

Method	Description	Example
`equals()`	Checks string equality	`"Java".equals("java")` → `false`
`equalsIgnoreCase()`	Compares ignoring case	`"Java".equalsIgnoreCase("java")` → `true`
`contains()`	Checks if string contains substring	`"Hello".contains("He")` → `true`
`startsWith()`	Checks if starts with substring	`"Java".startsWith("J")` → `true`
`endsWith()`	Checks if ends with substring	`"file.txt".endsWith(".txt")` → `true`
`charAt()`	Returns character at index	`"Hello".charAt(1)` → `'e'`
`substring()`	Extracts substring	`"Java".substring(1, 3)` → `"av"`
`replace()`	Replaces characters/words	`"Hello".replace("H", "J")` → `"Jello"`
`split()`	Splits string into array	`"A,B,C".split(",")` → `["A", "B", "C"]`
`trim()`	Removes whitespace	`" Hello ".trim()` → `"Hello"`
`format()`	Formats string	`String.format("Age: %d", 25)` → `"Age: 25"`
`valueOf()`	Converts data type to string	`String.valueOf(123)` → `"123"`

Classes & Objects in Java

In Java, **a class is a blueprint** for creating objects, and **an object is an instance of a class**. Java is an **object-oriented programming (OOP)** language, meaning everything revolves around objects.

---

1. What is a Class?

A **class** is a template that defines properties (fields/attributes) and behaviors (methods). It does not store actual data; instead, it describes how an object should behave.

Syntax of a Class

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

    // Constructor (optional)
    ClassName() {
        // Initialization code
    }

    // Methods (behavior)
    returnType methodName(parameters) {
        // Method body
    }
}
```

---

2. What is an Object?

An **object** is an instance of a class. It contains actual values and allows us to use the defined methods.

Creating an Object in Java

```
ClassName objectName = new ClassName();
```

---

3. Example: Creating a Class and Objects

Let's create a **Car** class with attributes and behaviors.

```
class Car {
    // Attributes (Instance Variables)
    String brand;
    int speed;

    // Constructor (Initializes attributes)
    Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }

    // Method to display car details
    void displayInfo() {
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
    }
}

public class CarDemo {
    public static void main(String[] args) {
        // Creating objects of the Car class
        Car car1 = new Car("Toyota", 180);
        Car car2 = new Car("BMW", 250);

        // Calling methods
        car1.displayInfo();
```

```
        car2.displayInfo();
    }
}
```

✅ **Output:**

```
Brand: Toyota
Speed: 180 km/h
Brand: BMW
Speed: 250 km/h
```

---

4. Understanding the Parts of a Class

(a) Fields (Instance Variables)

- Variables that **store object data**.
- Declared inside the class but outside methods.

```
String brand;
int speed;
```

(b) Constructors

- A **special method** that initializes objects.
- Has the **same name** as the class.
- Called **automatically** when an object is created.

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

(c) Methods (Functions in Classes)

- Define the **behavior** of an object.
- Can **return a value** or be `void` (no return value).

```
void displayInfo() {
    System.out.println("Brand: " + brand);
    System.out.println("Speed: " + speed + " km/h");
}
```

---

5. Access Modifiers (Encapsulation)

Access modifiers **control visibility** of class members.

Modifier	Accessible within class	Accessible in the same package	Accessible in a subclass (different package)	Accessible everywhere
`private`	✅	❌	❌	❌
`default`	✅	✅	❌	❌
`protected`	✅	✅	✅	❌
`public`	✅	✅	✅	✅

---

6. Example: Encapsulation with Getters & Setters

Encapsulation means **hiding data** and accessing it through methods.

```
class BankAccount {
    private double balance; // Private field

    // Constructor
    BankAccount(double balance) {
        this.balance = balance;
    }

    // Getter method (Read balance)
    public double getBalance() {
        return balance;
    }

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

public class BankDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(5000);
        account.deposit(2000);
        System.out.println("Balance: " + account.getBalance());
    }
}
```

✅ **Output:**

```
Balance: 7000.0
```

---

7. Static Members (Class-Level Members)

- `static` variables/methods **belong to the class, not objects**.
- Shared among all instances.

Example: Static Variables & Methods

```
class Student {
    static int count = 0; // Shared by all objects

    Student() {
        count++; // Increment count for each object
    }

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

public class StaticDemo {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student();
        Student.showCount();  // Output: Total Students: 2
    }
}
```

---

8. Difference Between Class & Object

Feature	Class	Object
Definition	A blueprint for objects	An instance of a class
Memory	No memory is allocated until an object is created	Takes up memory when instantiated
Usage	Defines attributes and behaviors	Stores data and calls methods

---

9. Summary of Key Concepts

Concept	Description
Class	Blueprint for objects
Object	Instance of a class
Constructor	Initializes objects
Methods	Define object behaviors

Concept	Description
Encapsulation	Hiding data using `private` and accessing via getters/setters
Static Members	Shared across all instances

Creating Classes in Java

In Java, a **class** is a blueprint that defines the structure and behavior of objects. It contains **fields (attributes)** and **methods (functions)** to describe an object.

---

1. Syntax for Creating a Class

```
class ClassName {
    // Fields (Instance Variables)
    dataType variableName;

    // Constructor
    ClassName() {
        // Initialization code
    }

    // Methods (Behavior)
    returnType methodName(parameters) {
        // Method body
    }
}
```

---

2. Example: Creating a Class

Let's create a **Car** class.

```
// Defining a class
class Car {
    // Attributes (Instance Variables)
    String brand;
    int speed;

    // Constructor
    Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }

    // Method to display car details
    void displayInfo() {
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
    }
```

```
}

// Main Class
public class CarDemo {
    public static void main(String[] args) {
        // Creating Objects
        Car car1 = new Car("Toyota", 180);
        Car car2 = new Car("BMW", 250);

        // Calling Methods
        car1.displayInfo();
        car2.displayInfo();
    }
}
```

✅ **Output:**

```
Brand: Toyota
Speed: 180 km/h
Brand: BMW
Speed: 250 km/h
```

---

3. Class Components

(a) Fields (Instance Variables)

- Define object properties.
- Declared inside the class but outside methods.

```
String brand;
int speed;
```

(b) Constructor

- A **special method** that initializes an object.
- Has the **same name** as the class.
- Called **automatically** when an object is created.

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

(c) Methods (Functions in a Class)

- Define **object behavior**.
- Can **return a value** or be `void`.

```
void displayInfo() {
    System.out.println("Brand: " + brand);
    System.out.println("Speed: " + speed + " km/h");
}
```

---

4. Creating Objects

An **object** is an instance of a class.

Creating an Object

```
Car myCar = new Car("Tesla", 220);
```

Accessing Methods

```
myCar.displayInfo();
```

---

5. Example: Class with Getters & Setters (Encapsulation)

Encapsulation means **hiding data** using `private` fields and accessing them via methods.

```
class BankAccount {
    private double balance; // Private field

    // Constructor
    BankAccount(double balance) {
        this.balance = balance;
    }

    // Getter method
    public double getBalance() {
        return balance;
    }

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

public class BankDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(5000);
        account.deposit(2000);
        System.out.println("Balance: " + account.getBalance());
    }
}
```

✅ **Output:**

```
Balance: 7000.0
```

---

6. Summary

Concept	Description
Class	Blueprint for objects
Object	Instance of a class
Constructor	Initializes objects
Methods	Define object behaviors
Encapsulation	Hiding data using `private` and accessing via methods

---

Declaring Objects in Java

In Java, an **object** is an instance of a class. Objects store data and allow us to call methods defined in the class.

---

1. How to Declare an Object

Syntax:

```
ClassName objectName;  // Object declaration
objectName = new ClassName();  // Object creation
```

Or, we can **combine both steps** in one line:

```
ClassName objectName = new ClassName();
```

---

2. Example: Declaring and Creating Objects

Let's create a **Car** class and declare objects.

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

    // Constructor
    Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }

    // Method to display car details
    void displayInfo() {
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
```

```
    }
}

public class CarDemo {
    public static void main(String[] args) {
        // Declaring and creating objects
        Car car1 = new Car("Toyota", 180);
        Car car2 = new Car("BMW", 250);

        // Calling methods
        car1.displayInfo();
        car2.displayInfo();
    }
}
```

✅ **Output:**

```
Brand: Toyota
Speed: 180 km/h
Brand: BMW
Speed: 250 km/h
```

---

3. Ways to Declare and Initialize Objects

(a) Using a Constructor

```
Car myCar = new Car("Honda", 200);
```

(b) Using Default Constructor

If a class **does not have a constructor**, Java provides a **default constructor**.

```
class Person {
    String name;
    int age;
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person(); // Default constructor
        p1.name = "John";
        p1.age = 25;
        System.out.println(p1.name + " is " + p1.age + " years old.");
    }
}
```

✅ **Output:**

```
John is 25 years old.
```

(c) Declaring Multiple Objects

```
Car car1, car2, car3;  // Declaring multiple objects
car1 = new Car("Ford", 200);
car2 = new Car("Nissan", 220);
```

```
car3 = new Car("Tesla", 240);
```

(d) Anonymous Objects (Without a Name)

Used when we need an object only **once**.

```
new Car("Ferrari", 300).displayInfo();
```

✅ **Output:**

```
Brand: Ferrari
Speed: 300 km/h
```

---

4. Summary

Concept	Example
Declaring an Object	`Car myCar;`
Creating an Object	`myCar = new Car();`
Declaration & Creation (Single Line)	`Car myCar = new Car();`
Multiple Objects	`Car c1, c2, c3;`
Anonymous Object	`new Car("Ferrari", 300).displayInfo();`

🚀 Conclusion

✅ **Objects** are instances of a class.
✅ Use **constructors** to initialize objects.
✅ Objects allow us to **access methods and fields** of a class.
✅ **Anonymous objects** are used when no reference is needed.

Would you like to explore **constructors, object passing, or object reference** next? 😊

Methods in Java

A **method** in Java is a block of code that performs a specific task. Methods allow **code reuse, modularity, and better organization**.

---

1. Declaring a Method

Syntax:

```
returnType methodName(parameters) {
    // Method body
    return value;  // (if returnType is not void)
}
```

Example: A Simple Method

```
class MathOperations {
    // Method to add two numbers
    int add(int a, int b) {
        return a + b;
    }
}
```

---

2. Calling a Method

To use a method, we need to call it using an **object**.

```
public class MethodDemo {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();  // Creating an object
        int sum = math.add(5, 10);  // Calling the method
        System.out.println("Sum: " + sum);
    }
}
```

✅ **Output:**

```
Sum: 15
```

---

3. Types of Methods in Java

(a) Instance Methods

- Belong to an **object**.
- Require an **object** to be called.
- Can **access instance variables**.

```
class Person {
    String name;

    void sayHello() {  // Instance method
        System.out.println("Hello, my name is " + name);
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.name = "Alice";
        p1.sayHello();  // Calling instance method
    }
}
```

✅ **Output:**

```
Hello, my name is Alice
```

(b) Static Methods

- Belong to the **class**, not objects.
- Can be called **without creating an object**.
- Cannot access **instance variables** directly.

```
class MathUtils {
    static int square(int num) {  // Static method
        return num * num;
    }
}

public class Main {
    public static void main(String[] args) {
        int result = MathUtils.square(5);  // Calling static method
        System.out.println("Square: " + result);
    }
}
```

✅ **Output:**

```
Square: 25
```

(c) Methods with Parameters

Methods can **take input parameters**.

```
class Greetings {
    void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }
}

public class Main {
    public static void main(String[] args) {
        Greetings g = new Greetings();
        g.greet("Alice");  // Passing "Alice" as an argument
    }
```

```
}
```

✅ **Output:**

```
Hello, Alice!
```

(d) Method Returning a Value

Methods can **return values** using the `return` keyword.

```
class MathOperations {
    int multiply(int a, int b) {
        return a * b;  // Returning the product
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        int result = math.multiply(4, 5);
        System.out.println("Product: " + result);
    }
}
```

✅ **Output:**

```
Product: 20
```

(e) Method Overloading (Multiple Methods with the Same Name)

- Methods can have **the same name but different parameters**.
- Java automatically chooses the correct method based on arguments.

```
class Calculator {
    int add(int a, int b) {  // Method 1
        return a + b;
    }

    double add(double a, double b) {  // Method 2 (different parameters)
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum (int): " + calc.add(5, 10));
        System.out.println("Sum (double): " + calc.add(5.5, 10.5));
    }
}
```

✅ **Output:**

```
Sum (int): 15
Sum (double): 16.0
```

---

4. Pass by Value (How Java Passes Arguments)

In Java, **primitive data types** are passed **by value** (a copy is passed, original data is not modified).

```
class NumberChanger {
    void modify(int num) {
        num = num + 10;
    }
}

public class Main {
    public static void main(String[] args) {
        int number = 20;
        NumberChanger changer = new NumberChanger();
        changer.modify(number);
        System.out.println("Original Number: " + number);  // Number remains
20
    }
}
```

✅ **Output:**

```
Original Number: 20
```

---

5. Summary

Concept	Description
Instance Method	Belongs to an object, requires an instance
Static Method	Belongs to a class, can be called without an object
Method with Parameters	Accepts input arguments
Method Returning a Value	Uses `return` to send a result
Method Overloading	Multiple methods with the same name but different parameters
Pass by Value	Java passes primitive data types by value (copy)

Parameter Passing in Java

In Java, when calling a method, we can pass **arguments (actual values)** to the method's **parameters (formal variables)**. Java **always** uses **pass-by-value**, meaning it passes a **copy** of the actual value, not the original variable itself.

---

1. Types of Parameter Passing in Java

1. **Passing Primitive Data Types** (Pass by Value)
2. **Passing Objects (Reference Types)**
3. **Passing Arrays**

---

2. Passing Primitive Data Types (Pass by Value)

Java passes primitive data types like `int`, `double`, `char`, and `boolean` **by value**.
➞ **A copy** of the value is sent to the method, so the original variable remains unchanged.

Example: Pass by Value

```
class Example {
    void modify(int num) {
        num = num + 10;  // This changes only the local copy
        System.out.println("Inside method: " + num);
    }
}

public class Main {
    public static void main(String[] args) {
        int number = 50;
        Example obj = new Example();
        obj.modify(number);  // Passing a copy of 'number'
        System.out.println("After method call: " + number);  // Original
'number' remains unchanged
    }
}
```

✅ Output:

```
Inside method: 60
After method call: 50
```

🔷 Explanation:

- `num` inside `modify()` is a **copy** of `number`.
- Changing `num` **does not affect** the original `number`.

---

3. Passing Objects (Reference Types)

Java **still** uses pass-by-value for objects, but **the reference (memory address) is copied**, not the actual object.

Example: Modifying an Object

```
class Person {
    String name;

    // Method to modify object property
    void changeName(Person p) {
        p.name = "Alice";  // Modifying the object's attribute
    }
}

public class Main {
    public static void main(String[] args) {
        Person person1 = new Person();
        person1.name = "Bob";

        System.out.println("Before: " + person1.name);

        person1.changeName(person1);  // Passing object reference

        System.out.println("After: " + person1.name);
    }
}
```

✅ Output:

```
Before: Bob
After: Alice
```

🔹 Explanation:

- The reference (memory address) of `person1` is copied and passed to `changeName()`.
- Since `p` and `person1` refer to the **same memory location**, modifying `p.name` also changes `person1.name`.

---

4. Passing Arrays (Reference Types)

Arrays behave similarly to objects. Since arrays are **reference types**, changes inside the method affect the original array.

Example: Passing an Array

```
class ArrayExample {
    void modifyArray(int[] arr) {
        arr[0] = 99;  // Changing the first element
    }
}

public class Main {
```

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

        System.out.println("Before: " + numbers[0]);

        ArrayExample obj = new ArrayExample();
        obj.modifyArray(numbers);  // Passing array reference

        System.out.println("After: " + numbers[0]);  // Original array is
modified
    }
}
```

✅ Output:

```
Before: 1
After: 99
```

🔷 Explanation:

- The **array reference** is passed, so modifications inside `modifyArray()` affect the **original array**.

---

5. Preventing Modification of Objects

To prevent modification, pass an **immutable object** like `String`, or use the `final` keyword.

Example: Using Immutable Strings

```
class Example {
    void modifyString(String str) {
        str = str + " World";  // This creates a new String object
    }
}

public class Main {
    public static void main(String[] args) {
        String text = "Hello";
        Example obj = new Example();
        obj.modifyString(text);
        System.out.println(text);  // Original text remains unchanged
    }
}
```

✅ Output:

```
Hello
```

🔷 Explanation:

- Strings are **immutable** in Java, so modifying `str` creates a **new** string object, leaving `text` unchanged.

---

6. Summary

Concept	Behavior
Primitive Types (`int`, `double`, `char`, etc.)	Passed by value (copy of data)
Objects (`Person`, `Car`, etc.)	Passed by value (copy of reference), so changes affect the original object
Arrays (`int[]`, `String[]`)	Passed by value (copy of reference), so changes affect the original array
Immutable Objects (`String`)	Cannot be modified directly, new objects are created instead

---

🚀 Conclusion

✅ **Primitive types** are always **pass-by-value** (copy of data).
✅ **Objects and arrays** are **passed by reference copy**, meaning changes affect the original data.
✅ **Immutable objects** like `String` cannot be modified directly.
✅ To prevent modification, use **immutable classes** or **final keyword**.

Would you like to explore **returning objects, method overloading, or functional programming next?** 😊

Static Fields and Methods in Java

In Java, the `static` keyword is used to define **fields (variables) and methods** that belong to the **class itself**, rather than to instances (objects) of the class.

---

1. Static Fields (Class Variables)

A **static field** (also called a **class variable**) belongs to the **class, not to any object**.

- It is shared among **all instances** of the class.
- It is **initialized only once**, at the start of program execution.

Example: Static Field

```
class Car {
    static int totalCars = 0;  // Static field (shared by all objects)
    String brand;

    // Constructor
```

```
    Car(String brand) {
        this.brand = brand;
        totalCars++;  // Increments totalCars for every new object
    }

    // Method to display details
    void displayCar() {
        System.out.println("Brand: " + brand);
        System.out.println("Total Cars: " + totalCars);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Toyota");
        Car car2 = new Car("BMW");

        car1.displayCar();
        car2.displayCar();

        // Accessing static field directly using class name
        System.out.println("Total cars (from class): " + Car.totalCars);
    }
}
```

✅ Output:

```
Brand: Toyota
Total Cars: 2
Brand: BMW
Total Cars: 2
Total cars (from class): 2
```

🔷 Explanation:

- `totalCars` is a **static field**, shared among all objects.
- It increments whenever a new `Car` object is created.
- **Access static fields using the class name**: `Car.totalCars`.

---

2. Static Methods (Class Methods)

A **static method** belongs to the **class**, not to any specific object.

- It **can be called without creating an object**.
- It **cannot** access **non-static (instance) variables or methods**.

Example: Static Method

```
class MathUtils {
    // Static method
    static int square(int num) {
        return num * num;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Calling static method without creating an object
        int result = MathUtils.square(5);
        System.out.println("Square: " + result);
    }
}
```

✅ Output:

```
Square: 25
```

🔷 Explanation:

- `square()` is **static**, so it is called using the class name: `MathUtils.square(5)`.
- It **does not need an object** to be invoked.

---

3. Accessing Static Members

Static Member	Accessed Using
Static Field	`ClassName.fieldName`
Static Method	`ClassName.methodName()`
Inside the same class	Can be accessed directly

Example: Static vs. Non-Static Access

```
class Example {
    static int staticVar = 10;  // Static field
    int instanceVar = 20;       // Instance field

    // Static method
    static void staticMethod() {
        System.out.println("Static method called.");
        System.out.println("StaticVar: " + staticVar);
        // System.out.println(instanceVar);  ❌ ERROR: Cannot access instance
variable
    }

    // Instance method
    void instanceMethod() {
        System.out.println("Instance method called.");
        System.out.println("InstanceVar: " + instanceVar);
        System.out.println("StaticVar: " + staticVar);  // ✅ Allowed
    }
}

public class Main {
    public static void main(String[] args) {
        // Calling static method using class name
        Example.staticMethod();
```

```
        // Creating an object to call instance method
        Example obj = new Example();
        obj.instanceMethod();
    }
}
```

✅ Output:

```
Static method called.
StaticVar: 10
Instance method called.
InstanceVar: 20
StaticVar: 10
```

🔹 Key Rules:

1. ✅ Static methods **can access only static fields/methods**.
2. ❌ Static methods **cannot access instance variables/methods** directly.
3. ✅ Instance methods **can access both static and non-static members**.

---

4. When to Use Static Fields & Methods?

Feature	Use Static When...
Fields (Variables)	The value is **shared** among all objects (e.g., `totalCars`).
Methods	The method does **not** depend on instance variables (e.g., utility functions like `Math.sqrt()`).
Memory Efficiency	Static fields are stored **once**, reducing memory usage.

---

5. Static Block (for Initialization)

A **static block** is used to **initialize static fields** before anything else runs.

Example: Using a Static Block

```
class Example {
    static int staticVar;

    // Static block (runs once when class is loaded)
    static {
        staticVar = 100;
        System.out.println("Static block executed.");
    }
}

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

```
        System.out.println("StaticVar: " + Example.staticVar);
    }
}
```

✅ Output:

```
Static block executed.
StaticVar: 100
```

🔷 Why use a static block?

- Runs **only once** when the class is loaded.
- Useful for **initializing static variables**.

---

6. Static vs. Instance Comparison

Feature	Static (Class-Level)	Instance (Object-Level)
Belongs to	Class	Individual Object
Stored in	Method Area	Heap Memory
Accessed via	`ClassName.field/method`	`object.field/method`
Requires Object?	❌ No	✅ Yes
Can Access Static Members?	✅ Yes	✅ Yes
Can Access Instance Members?	❌ No	✅ Yes

---

7. Real-World Examples

(a) Utility Methods (Math Class)

```
System.out.println(Math.pow(2, 3));  // Calls static method pow()
```

(b) Constants (`final static`)

```
class Config {
    static final double PI = 3.14159;
}
System.out.println(Config.PI);
```

(c) Keeping Track of Objects

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

    Counter() {
        count++;
    }
}
Counter c1 = new Counter();
```

```
Counter c2 = new Counter();
System.out.println(Counter.count);  // Output: 2
```

---

8. Summary

Concept	Static Field	Static Method
Belongs to	Class	Class
Shared among objects?	✅ Yes	✅ Yes
Access using ClassName?	✅ Yes	✅ Yes
Can access instance variables?	❌ No	❌ No
Called without object?	✅ Yes	✅ Yes

🚀 Conclusion

✅ **Static fields** are shared among all instances of a class.
✅ **Static methods** can be called without creating an object.
✅ **Static methods cannot access instance variables/methods directly.**
✅ **Use `static` for utility functions, constants, and shared data.**

Would you like to explore **static inner classes, final static variables, or singleton patterns next?** 😊

Constructors in Java

A **constructor** in Java is a special method used to **initialize objects**. It is automatically called when an object of a class is created.

---

1. Characteristics of a Constructor

✅ It has the **same name** as the class.
✅ It **does not** have a return type (not even `void`).
✅ It is **invoked automatically** when an object is created.
✅ It can be **overloaded** (multiple constructors with different parameters).

---

2. Types of Constructors in Java

1. **Default Constructor (No-Argument Constructor)**
2. **Parameterized Constructor**
3. **Copy Constructor (Manually Implemented)**
4. **Constructor Overloading**
5. **Constructor Chaining**

---

3. Default Constructor

A **default constructor** is a constructor that takes **no arguments** and initializes default values.

Example: Default Constructor

```
class Car {
    String brand;

    // Default constructor
    Car() {
        brand = "Unknown";
        System.out.println("Default Constructor Called!");
    }

    void display() {
        System.out.println("Brand: " + brand);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();  // Default constructor is called automatically
        car1.display();
    }
}
```

✅ Output:

```
Default Constructor Called!
Brand: Unknown
```

🔹 Explanation:

- The default constructor is called when `new Car()` is executed.
- It initializes `brand` to `"Unknown"`.

---

4. Parameterized Constructor

A **parameterized constructor** allows passing values to **initialize** object attributes.

Example: Parameterized Constructor

```
class Car {
    String brand;

    // Parameterized constructor
    Car(String b) {
        brand = b;
    }

    void display() {
        System.out.println("Brand: " + brand);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Toyota");
        Car car2 = new Car("BMW");

        car1.display();
        car2.display();
    }
}
```

✅ Output:

```
Brand: Toyota
Brand: BMW
```

🔷 Explanation:

- The **constructor takes a parameter** (`String b`) and initializes `brand`.
- The constructor is automatically called when `new Car("Toyota")` is executed.

---

5. Copy Constructor

A **copy constructor** creates a new object by copying an existing object.

Example: Copy Constructor

```
class Car {
    String brand;

    // Parameterized constructor
    Car(String b) {
        brand = b;
    }

    // Copy constructor
    Car(Car oldCar) {
        this.brand = oldCar.brand;
    }

    void display() {
        System.out.println("Brand: " + brand);
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Honda");
        Car car2 = new Car(car1);  // Using copy constructor

        car1.display();
        car2.display();
    }
}
```

✅ Output:

```
Brand: Honda
Brand: Honda
```

🔷 Explanation:

- The copy constructor takes an object of the same class as a parameter.
- It copies values from `car1` to `car2`.

---

6. Constructor Overloading

Java allows **multiple constructors** in the same class, as long as they have **different parameters**.

Example: Constructor Overloading

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

    // Default constructor
    Car() {
        brand = "Unknown";
        year = 2000;
    }

    // Parameterized constructor with one argument
    Car(String b) {
        brand = b;
        year = 2000;
    }

    // Parameterized constructor with two arguments
    Car(String b, int y) {
        brand = b;
        year = y;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();            // Calls default constructor
        Car car2 = new Car("Ford");      // Calls constructor with one
argument
        Car car3 = new Car("Tesla", 2023); // Calls constructor with two
arguments

        car1.display();
        car2.display();
        car3.display();
    }
}
```

✅ Output:

```
Brand: Unknown, Year: 2000
Brand: Ford, Year: 2000
Brand: Tesla, Year: 2023
```

🔷 Explanation:

- The **constructor called depends on the arguments** passed when creating an object.

---

7. Constructor Chaining

Constructor chaining is when one constructor calls another constructor within the same class using `this()`.

Example: Constructor Chaining

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

    // Default constructor
    Car() {
        this("Unknown", 2000);  // Calls the two-argument constructor
    }

    // Single parameter constructor
    Car(String brand) {
        this(brand, 2000);  // Calls the two-argument constructor
    }

    // Two parameter constructor
    Car(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();       // Calls default constructor
        Car car2 = new Car("Audi"); // Calls one-parameter constructor
        Car car3 = new Car("BMW", 2022); // Calls two-parameter constructor

        car1.display();
        car2.display();
        car3.display();
    }
}
```

✅ Output:

```
Brand: Unknown, Year: 2000
Brand: Audi, Year: 2000
Brand: BMW, Year: 2022
```

🔷 Explanation:

- The `this()` keyword is used to **call another constructor** within the same class.
- This avoids **code duplication**.

---

8. Summary

Feature	Description
Default Constructor	No parameters, assigns default values.
Parameterized Constructor	Accepts parameters to initialize fields.
Copy Constructor	Creates a new object by copying another object.
Constructor Overloading	Multiple constructors with different parameter lists.
Constructor Chaining	One constructor calls another using `this()`.

🚀 Conclusion

✅ **Constructors initialize objects automatically.**
✅ **They do not have a return type.**
✅ **They can be overloaded and chained for flexibility.**
✅ **Using `this()` inside a constructor allows constructor chaining.**

this Keyword in Java

The **`this`** keyword in Java is a reference variable that refers to the **current object** of the class. It is used to **eliminate ambiguity**, **access instance variables**, **call other constructors**, and **return the current instance**.

---

1. Uses of `this` Keyword

1. **Referring to Instance Variables** (when local and instance variables have the same name)
2. **Calling Another Constructor (Constructor Chaining)**
3. **Returning the Current Object**
4. **Passing the Current Object as an Argument**
5. **Accessing an Inner Class Instance**

---

2. Referring to Instance Variables

If a constructor or method has **local variables** with the same name as instance variables, `this` helps differentiate between them.

Example: Using `this` to Refer to Instance Variables

```
class Car {
    String brand; // Instance variable

    Car(String brand) { // Local variable (same name)
        this.brand = brand; // 'this.brand' refers to the instance variable
    }

    void display() {
        System.out.println("Brand: " + brand);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Toyota");
        car1.display();
    }
}
```

✅ Output:

```
Brand: Toyota
```

🔹 Explanation:

- The constructor parameter **`brand`** has the same name as the instance variable.
- Using `this.brand = brand;` assigns the parameter value to the instance variable.

3. Calling Another Constructor (Constructor Chaining)

We can use `this()` inside a constructor to call another constructor **in the same class**.

Example: Constructor Chaining using `this()`

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

    // Default constructor
    Car() {
        this("Unknown", 2000); // Calls the constructor with two arguments
    }

    // Constructor with one parameter
    Car(String brand) {
        this(brand, 2000); // Calls the constructor with two parameters
    }

    // Constructor with two parameters
    Car(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Ford");
        Car car3 = new Car("Tesla", 2023);

        car1.display();
        car2.display();
        car3.display();
    }
}
```

✅ Output:

```
Brand: Unknown, Year: 2000
Brand: Ford, Year: 2000
Brand: Tesla, Year: 2023
```

🔷 Explanation:

- The `this()` statement is used to call another constructor in the same class.
- This **avoids duplicate code** and improves maintainability.

4. Returning the Current Object

The `this` keyword can be used to return the **current class instance**.

Example: Returning the Current Object

```
class Car {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }

    Car getCar() {
        return this; // Returning current object
    }

    void display() {
        System.out.println("Brand: " + brand);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Mercedes");
        car1.getCar().display(); // Using returned object
    }
}
```

✅ Output:

```
Brand: Mercedes
```

🔷 Explanation:

- The `getCar()` method returns the **current instance** using `this`.
- `car1.getCar().display();` calls `display()` on the returned object.

---

5. Passing the Current Object as an Argument

Sometimes, we need to pass the **current instance** to another method or constructor.

Example: Passing `this` to Another Method

```
class Car {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }

    void display(Car obj) {
        System.out.println("Brand: " + obj.brand);
    }
```

```
    void show() {
        display(this); // Passing current object
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("BMW");
        car1.show();
    }
}
```

✅ **Output:**

```
Brand: BMW
```

🔷 **Explanation:**

- The `show()` method calls `display(this);` and passes the **current object** to the `display()` method.

---

6. Accessing an Inner Class Instance

If an **inner class** has the same variable name as the outer class, `this` helps access the outer class variable.

Example: Using `this` with Inner Class

```
class Outer {
    int x = 10;

    class Inner {
        int x = 20;

        void display() {
            System.out.println("Inner x: " + x);
            System.out.println("Outer x: " + Outer.this.x); // Accessing
outer class variable
        }
    }
}

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

✅ **Output:**

```
Inner x: 20
Outer x: 10
```

🔷 Explanation:

- The inner class has a variable `x`, and so does the outer class.
- `Outer.this.x` refers to the outer class variable.

---

7. Summary

Use Case	Description
Referring to Instance Variables	Resolves conflicts between instance and local variables.
Calling Another Constructor	`this()` is used for constructor chaining.
Returning the Current Object	`this` can return the current class instance.
Passing the Current Object	`this` is passed as an argument to another method or constructor.
Accessing Outer Class Instance	`Outer.this.variable` allows access to outer class variables.

🚀 Conclusion

✅ **The `this` keyword helps avoid ambiguity between local and instance variables.**
✅ **It is useful for constructor chaining, method chaining, and returning objects.**
✅ **It allows passing the current instance as an argument and accessing outer class members.**

Method Overloading and Access Modifiers in Java

1. Method Overloading in Java

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

🔷 Key Points:

- The **method name remains the same**.
- The **parameter list must be different**.
- Return type **does not** differentiate overloaded methods.
- Overloading improves **code readability** and **reusability**.

Example 1: Method Overloading with Different Number of Parameters

```
class MathOperations {
    // Method with one parameter
    int multiply(int a) {
```

```
        return a * a;
    }

    // Method with two parameters
    int multiply(int a, int b) {
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperations obj = new MathOperations();
        System.out.println("Multiply 5: " + obj.multiply(5));
        System.out.println("Multiply 5 and 4: " + obj.multiply(5, 4));
    }
}
```

✅ **Output:**

```
Multiply 5: 25
Multiply 5 and 4: 20
```

Example 2: Method Overloading with Different Data Types

```
class Display {
    // Overloaded method for int
    void show(int a) {
        System.out.println("Integer: " + a);
    }

    // Overloaded method for double
    void show(double a) {
        System.out.println("Double: " + a);
    }

    // Overloaded method for String
    void show(String str) {
        System.out.println("String: " + str);
    }
}

public class Main {
    public static void main(String[] args) {
        Display obj = new Display();
        obj.show(10);
        obj.show(5.5);
        obj.show("Hello");
    }
}
```

✅ **Output:**

```
Integer: 10
Double: 5.5
String: Hello
```

Example 3: Method Overloading with Different Order of Parameters

```
class Example {
    void display(int a, String b) {
        System.out.println("Integer: " + a + ", String: " + b);
    }

    void display(String b, int a) {
        System.out.println("String: " + b + ", Integer: " + a);
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example();
        obj.display(10, "Java");
        obj.display("Overloading", 20);
    }
}
```

✅ **Output:**

```
Integer: 10, String: Java
String: Overloading, Integer: 20
```

Key Benefits of Method Overloading

✅ **Increases readability** – same method name, different purposes.
✅ **Code reusability** – avoids redundant method names.
✅ **Helps in polymorphism** – method behavior depends on arguments.

---

2. Access Modifiers in Java

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

Modifier	Scope	Access Level
`public`	**Everywhere**	Accessible from anywhere.
`private`	**Within the same class**	Not accessible outside the class.
`protected`	**Same package + Subclasses**	Accessible within the package and subclasses.
(default) (No modifier)	**Same package only**	Accessible within the same package.

Example: Using Access Modifiers

```
class Example {
```

```
    public int publicVar = 10;
    private int privateVar = 20;
    protected int protectedVar = 30;
    int defaultVar = 40; // Default (no modifier)

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

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

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

    void showDefault() { // Default method
        System.out.println("Default method.");
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example();

        // ✅ Accessible
        System.out.println("Public: " + obj.publicVar);
        System.out.println("Protected: " + obj.protectedVar);
        System.out.println("Default: " + obj.defaultVar);
        obj.showPublic();
        obj.showProtected();
        obj.showDefault();

        // ❌ Not accessible (private variable and method)
        // System.out.println(obj.privateVar);  // Error
        // obj.showPrivate();                   // Error
    }
}
```

✅ Output:

```
Public: 10
Protected: 30
Default: 40
Public method.
Protected method.
Default method.
```

❌ Private members cannot be accessed outside the class.

Access Modifiers and Inheritance

When a subclass inherits a superclass, access modifiers affect what gets inherited.

Modifier	Same Class	Same Package	Subclasses (Different Package)	Outside Package
`public`	✅ Yes	✅ Yes	✅ Yes	✅ Yes
`private`	✅ Yes	❌ No	❌ No	❌ No
`protected`	✅ Yes	✅ Yes	✅ Yes	❌ No
(default) (No modifier)	✅ Yes	✅ Yes	❌ No	❌ No

Example: Access Modifiers in Inheritance

```
class Parent {
    public String publicVar = "Public";
    private String privateVar = "Private";
    protected String protectedVar = "Protected";
    String defaultVar = "Default"; // Default access

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

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

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

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

class Child extends Parent {
    void display() {
        // ✅ Accessible
        System.out.println(publicVar);
        System.out.println(protectedVar);
        System.out.println(defaultVar);

        showPublic();
        showProtected();
        showDefault();

        // ❌ Not accessible (private)
        // System.out.println(privateVar);  // Error
        // showPrivate();                   // Error
    }
}

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

```
        Child obj = new Child();
        obj.display();
    }
}
```

✅ **Output:**

```
Public
Protected
Default
Public method
Protected method
Default method
```

❌ **Private members (`privateVar` and `showPrivate()`) are not inherited.**

---

3. Summary

Feature	Method Overloading	Access Modifiers
Definition	Methods with the same name but different parameters.	Controls access to class members.
Key Benefit	Improves code readability and reusability.	Ensures encapsulation and security.
Example Use Case	`print(int)` vs. `print(String)`	`public show()` vs. `private calculate()`

🚀 Conclusion

✅ **Method Overloading** allows defining multiple methods with the same name but different arguments.
✅ **Access Modifiers** help control the visibility of class members.
✅ **Private members are not inherited, but protected members are accessible in subclasses.**

Inheritance:

In Java, **inheritance** allows a child class (**subclass**) to acquire properties and behaviors from a parent class (**superclass**). It is one of the core principles of **Object-Oriented Programming (OOP)** and helps in code reusability and maintainability.

◈ How Inheritance Works in Java

- The `extends` keyword is used to define inheritance.
- The child class can access **non-private** fields and methods of the parent class.
- The child class can **override** methods from the parent class to provide its own implementation.

◈ Types of Inheritance in Java

1. **Single Inheritance** – A class inherits from a single parent class.
2. **Multilevel Inheritance** – A class inherits from a derived class.
3. **Hierarchical Inheritance** – Multiple classes inherit from a single parent class.
4. **Hybrid Inheritance** – Combination of multiple inheritance types (not directly supported in Java).
5. **Multiple Inheritance (via Interfaces)** – Java does not support multiple inheritance with classes but allows it using **interfaces**.

◈ Example of Inheritance in Java

☑ Single Inheritance

```
// Parent Class
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

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

// Main Class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound(); // Inherited from Animal class
        myDog.bark();      // Defined in Dog class
    }
}
```

🔷 Output:

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

✅ Method Overriding in Inheritance

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

// Child class overriding the method
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Polymorphism
        myAnimal.makeSound(); // Output: Dog barks
    }
}
```

✅ Multilevel Inheritance

```
// Grandparent class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Parent class
class Mammal extends Animal {
    void walk() {
        System.out.println("Mammal walks.");
    }
}

// Child class
class Dog extends Mammal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited from Animal
        myDog.walk(); // Inherited from Mammal
```

```
        myDog.bark(); // Defined in Dog
    }
}
```

🔷 Output:

```
This animal eats food.
Mammal walks.
Dog barks.
```

☑️ Multiple Inheritance using Interfaces

Java does **not** support multiple inheritance with classes but supports it using **interfaces**.

```
// First Interface
interface Animal {
    void eat();
}

// Second Interface
interface Pet {
    void play();
}

// Class implementing multiple interfaces
class Dog implements Animal, Pet {
    public void eat() {
        System.out.println("Dog eats.");
    }

    public void play() {
        System.out.println("Dog plays.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.play();
    }
}
```

🔷 Key Points About Java Inheritance

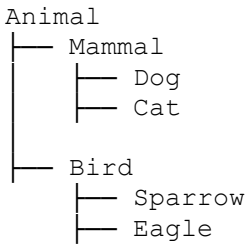
- ☑️ **Java supports single, multilevel, and hierarchical inheritance.**
- ☑️ **Java does NOT support multiple inheritance with classes to avoid ambiguity.**
- ☑️ **Method overriding is used to redefine methods in the child class.**
- ☑️ **The `super` keyword is used to call parent class methods or constructors.**
- ☑️ **Interfaces allow multiple inheritance in Java.**

Inheritance Hierarchies in Java

An **inheritance hierarchy** is the structure formed when multiple classes are connected through **inheritance** in a hierarchical manner. It helps in **code reuse, scalability, and organization** in object-oriented programming.

Example of an Inheritance Hierarchy

Consider a scenario where different types of animals inherit common characteristics from a base class.



Java Implementation

```
// Parent class (Base class)
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Intermediate class (Mammal)
class Mammal extends Animal {
    void walk() {
        System.out.println("This mammal walks.");
    }
}

// Intermediate class (Bird)
class Bird extends Animal {
    void fly() {
        System.out.println("This bird can fly.");
    }
}

// Derived class (Dog)
class Dog extends Mammal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

// Derived class (Cat)
class Cat extends Mammal {
    void meow() {
        System.out.println("Cat meows.");
    }
```

```
}

// Derived class (Sparrow)
class Sparrow extends Bird {
    void chirp() {
        System.out.println("Sparrow chirps.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited from Animal
        myDog.walk(); // Inherited from Mammal
        myDog.bark(); // Defined in Dog

        System.out.println();

        Sparrow mySparrow = new Sparrow();
        mySparrow.eat();  // Inherited from Animal
        mySparrow.fly();  // Inherited from Bird
        mySparrow.chirp(); // Defined in Sparrow
    }
}
```

◈ Output:

```
This animal eats food.
This mammal walks.
Dog barks.

This animal eats food.
This bird can fly.
Sparrow chirps.
```

◈ Advantages of Inheritance Hierarchies

✅ **Code Reusability** – Common methods are written once in the base class and reused by derived classes.
✅ **Scalability** – New classes can be added without modifying existing ones.
✅ **Better Organization** – Creates a clear relationship between classes.
✅ **Polymorphism Support** – Objects of different classes can be treated as objects of a common superclass.

◈ Things to Consider

- **Deep hierarchies can become complex and difficult to manage.**
- **Use the `super` keyword to call parent class methods when overridden.**
- **Java does not support multiple inheritance with classes but allows it using interfaces.**

◈ super Keyword and Subclasses in Java

In Java, **subclasses** are classes that inherit from another class (**superclass**). The `super` keyword is used within a subclass to refer to members (methods and constructors) of its **immediate parent class**.

◈ What is `super` Used For?

The `super` keyword can be used in three ways:

1. **Access Superclass Methods** – Call an overridden method from the parent class.
2. **Access Superclass Constructor** – Call the parent class constructor.
3. **Access Superclass Fields** – Access variables from the parent class if they are not overridden.

1️⃣ Using `super` to Call a Superclass Method

If a subclass **overrides** a method from its superclass but still wants to call the parent method, `super.methodName()` is used.

✅ Example: Overriding with `super`

```
// Superclass (Parent)
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass (Child)
class Dog extends Animal {
    void makeSound() {
        super.makeSound(); // Calls the parent class method
        System.out.println("Dog barks");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

◈ Output:

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

2️⃣ Using `super` to Call a Superclass Constructor

When a subclass object is created, the parent class constructor is **automatically called first**. If the parent class has a **parameterized constructor**, the subclass must explicitly call it using `super(parameters);`.

✅ Example: Calling Parent Constructor with `super`

```
// Superclass (Parent)
class Animal {
    String name;

    // Parameterized constructor
    Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor: " + name);
    }
}

// Subclass (Child)
class Dog extends Animal {
    Dog(String name) {
        super(name); // Calls the Animal class constructor
        System.out.println("Dog constructor: " + name);
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy");
    }
}
```

🔹 Output:

```
Animal constructor: Buddy
Dog constructor: Buddy
```

🔷 Why use `super(name);`?

- If we don't explicitly call `super(name)`, Java automatically calls the **default constructor** of the superclass.
- If the parent class does not have a **default constructor**, we must call a parameterized one explicitly.

3️⃣ Using `super` to Access Superclass Fields

If a subclass defines a field with the **same name** as its superclass, the `super` keyword can differentiate them.

✅ Example: Accessing Parent Class Variables with `super`

```
// Superclass (Parent)
class Animal {
```

```
    String type = "Animal";
}

// Subclass (Child)
class Dog extends Animal {
    String type = "Dog";

    void printType() {
        System.out.println("Child class type: " + type);
        System.out.println("Super class type: " + super.type); // Access
parent field
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.printType();
    }
}
```

◈ Output:

```
Child class type: Dog
Super class type: Animal
```

◈ Key Takeaways

- ✅ `super.methodName()` → Calls the parent class's method if overridden.
- ✅ `super(parameters)` → Calls the parent class's constructor explicitly.
- ✅ `super.variableName` → Accesses a field from the parent class.

◈ Important Notes

1. `super()` **must be the first statement** inside a subclass constructor.
2. If a parent class has **only a parameterized constructor**, the subclass **must explicitly call** `super(args)`, or a compilation error will occur.
3. The `super` keyword **cannot be used in static methods**, because `super` refers to an instance of a class.

◈ Member Access Rules in Java (Fields, Methods, Constructors)

Java has **four access modifiers** that control the visibility of class members (fields, methods, and constructors) within different parts of a program. These modifiers define **where** a member can be accessed.

🔰 Access Modifiers in Java

Modifier	Within Class	Within Package	Subclass (Other Package)	Other Classes
`private`	✅ Yes	❌ No	❌ No	❌ No
(default) (No modifier)	✅ Yes	✅ Yes	❌ No	❌ No
`protected`	✅ Yes	✅ Yes	✅ Yes	❌ No
`public`	✅ Yes	✅ Yes	✅ Yes	✅ Yes

1️⃣ `private` Access Modifier

- The **most restrictive** access level.
- Members are **accessible only within the same class**.
- **Not inherited by subclasses**.

✅ Example: `private` members

```
class Animal {
    private String name = "Lion"; // Private variable

    private void display() { // Private method
        System.out.println("Animal name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
        // System.out.println(a.name); // ❌ ERROR: name is private
        // a.display(); // ❌ ERROR: display() is private
    }
}
```

🔷 **Key Takeaway:** Private members are **not accessible outside the class**, even in subclasses.

2️⃣ Default (No Modifier) – Package-Private

- If no modifier is specified, the member is **only accessible within the same package**.
- It is **visible to all classes in the same package** but **not to subclasses outside the package**.

✅ Example: Default Access (Same Package)

```
// File: Animal.java (in package `animals`)
```

```
package animals;

class Animal {
    void display() { // Default access modifier
        System.out.println("Animal method");
    }
}
// File: Main.java (in package `animals`)
package animals;

public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.display(); // ✅ Allowed (same package)
    }
}
// File: Test.java (in a different package)
package test;

import animals.Animal;

public class Test {
    public static void main(String[] args) {
        Animal a = new Animal();
        // a.display(); // ❌ ERROR: Cannot access (default is package-
private)
    }
}
```

🔷 **Key Takeaway: No modifier = package-private**, meaning it's only accessible within the same package.

---

3 `protected` Access Modifier

- Members are **accessible within the same package** and **in subclasses (even in different packages)**.
- **Unlike default access, `protected` allows subclass access outside the package**.

✅ Example: `protected` Access in a Subclass

```
// File: Animal.java (in package `animals`)
package animals;

public class Animal {
    protected String type = "Mammal";

    protected void display() {
        System.out.println("This is a protected method");
    }
}
// File: Dog.java (in package `test`)
package test;

import animals.Animal; // Importing Animal class
```

```
class Dog extends Animal {
    void show() {
        System.out.println("Animal Type: " + type); // ✅ Allowed (subclass)
        display(); // ✅ Allowed (subclass)
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.show();

        // Animal a = new Animal();
        // a.display(); // ❌ ERROR: Cannot access protected method outside
the package
    }
}
```

🔷 Key Takeaway:

- **`protected` allows access in the same package + subclasses in different packages.**
- **It does NOT allow direct access from a non-subclass in another package.**

---

4 `public` Access Modifier

- The **least restrictive** access level.
- Members are **accessible from anywhere** (same class, same package, different package, subclass, non-subclass).

✅ Example: `public` Access

```
// File: Animal.java
package animals;

public class Animal {
    public String name = "Elephant";

    public void display() {
        System.out.println("Public method: " + name);
    }
}
// File: Main.java (different package)
package test;

import animals.Animal; // Importing Animal class

public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
        System.out.println(a.name); // ✅ Allowed
        a.display(); // ✅ Allowed
    }
}
```

🔷 Key Takeaway:

- **Public members are accessible from anywhere.**
- **Best for global access, such as API methods.**

🔰 Access Rules for Inheritance

Access Modifier	Inherited in Subclass (Same Package)?	Inherited in Subclass (Different Package)?
`private`	❌ No	❌ No
(default)	✅ Yes	❌ No
`protected`	✅ Yes	✅ Yes (Only via Inheritance)
`public`	✅ Yes	✅ Yes

✅ Example: Understanding Member Access in Inheritance

```
class Parent {
    private int privateVar = 10; // ❌ Not accessible in subclass
    int defaultVar = 20; // ✅ Accessible within the same package
    protected int protectedVar = 30; // ✅ Accessible within package & subclasses
    public int publicVar = 40; // ✅ Accessible everywhere
}

class Child extends Parent {
    void show() {
        // System.out.println(privateVar); // ❌ ERROR: Private members are NOT inherited
        System.out.println(defaultVar); // ✅ Allowed (same package)
        System.out.println(protectedVar); // ✅ Allowed (protected)
        System.out.println(publicVar); // ✅ Allowed (public)
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.show();
    }
}
```

🔷 Output:

```
20
30
40
```

◈ Summary of Java Member Access Rules

Modifier	Accessible in Same Class?	Accessible in Same Package?	Accessible in Subclass (Other Package)?	Accessible Anywhere?
`private`	✅ Yes	❌ No	❌ No	❌ No
(default) (Package-Private)	✅ Yes	✅ Yes	❌ No	❌ No
`protected`	✅ Yes	✅ Yes	✅ Yes (Only in Subclasses)	❌ No
`public`	✅ Yes	✅ Yes	✅ Yes	✅ Yes

◈ Key Takeaways

✅ **Use `private` for encapsulation** – Hide implementation details.
✅ **Use `default` (package-private) for package-level access** – Useful for classes within the same package.
✅ **Use `protected` for inheritance** – Allows access to subclasses even outside the package.
✅ **Use `public` for global access** – Suitable for APIs and widely used methods.

◈ super Keyword in Java

The `super` keyword in Java is used within a **subclass** to refer to members (methods, constructors, and variables) of its **immediate superclass**. It helps resolve conflicts when subclass and superclass have members with the same name.

◈ Uses of `super` in Java

1. **Access Superclass Methods** – Calls an overridden method from the parent class.
2. **Access Superclass Variables** – Refers to a field from the parent class when overridden in the child class.
3. **Call Superclass Constructor** – Calls a parent class constructor from a subclass.

1️⃣ Using `super` to Call a Superclass Method

If a subclass **overrides** a method from the parent class but still wants to call the original method, `super.methodName()` is used.

✅ Example: Calling an Overridden Method

```
// Superclass (Parent)
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass (Child)
class Dog extends Animal {
    void makeSound() {
        super.makeSound(); // Calls parent class method
        System.out.println("Dog barks");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

🔹 Output:

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

🔷 **Explanation:** `super.makeSound();` first calls the parent class method before executing the subclass method.

2️⃣ Using `super` to Access Superclass Variables

If a subclass defines a field with the **same name** as its superclass, the `super` keyword can differentiate them.

✅ Example: Accessing Superclass Variable

```
// Superclass (Parent)
class Animal {
    String type = "Animal";
}

// Subclass (Child)
class Dog extends Animal {
    String type = "Dog";

    void displayType() {
```

```
        System.out.println("Child class type: " + type);
        System.out.println("Super class type: " + super.type); // Access
parent field
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.displayType();
    }
}
```

🔷 Output:

```
Child class type: Dog
Super class type: Animal
```

🔹 **Explanation:** `super.type` accesses the `type` field from the superclass.

3️⃣ Using `super` to Call a Superclass Constructor

A subclass constructor **automatically** calls the parent class constructor **before executing**.
If the parent class has a **parameterized constructor**, the subclass **must explicitly call it** using `super(parameters);`.

✅ Example: Calling Parent Constructor

```
// Superclass (Parent)
class Animal {
    Animal(String name) {
        System.out.println("Animal constructor: " + name);
    }
}

// Subclass (Child)
class Dog extends Animal {
    Dog(String name) {
        super(name); // Calls the Animal constructor
        System.out.println("Dog constructor: " + name);
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy");
    }
}
```

🔷 Output:

```
Animal constructor: Buddy
Dog constructor: Buddy
```

🔹 **Explanation:**

- `super(name);` calls the constructor of `Animal` before executing the `Dog` constructor.
- Without `super(name)`, Java would try to call the default constructor of `Animal`, which **does not exist**, causing an error.

🔷 Key Points to Remember

✅ `super.methodName()` → Calls the parent class's method if overridden.
✅ `super.variableName` → Accesses a field from the parent class if overridden in the child class.
✅ `super(parameters)` → Calls the parent class's constructor explicitly.
✅ `super()` **must be the first statement** in a subclass constructor.
✅ If a superclass has **only parameterized constructors**, the subclass **must explicitly call** `super(arguments)`.
✅ **Cannot use `super` in static methods** because `super` refers to an instance of a class.

🔷 Preventing Inheritance in Java

Java provides several ways to **prevent inheritance**, ensuring that a class or method cannot be overridden or extended.

1️⃣ Using the `final` Keyword with a Class

If a class is marked as `final`, it **cannot be extended** by any other class.

✅ Example: Preventing Inheritance with `final` Class

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

// ❌ ERROR: Cannot extend a final class
class Dog extends Animal { // Compilation Error
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.sound();
    }
}
```

🔷 Explanation:

- The `final class Animal` **prevents subclasses from inheriting it**.
- The compiler will throw an error if any class tries to extend `Animal`.

2️⃣ Using `final` with Methods

A method marked as `final` **cannot be overridden** in a subclass.

✅ Example: Preventing Method Overriding with `final`

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

class Dog extends Animal {
    // ❌ ERROR: Cannot override final method
    /* void makeSound() {
        System.out.println("Dog barks");
    } */
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.makeSound(); // Calls Animal's makeSound()
    }
}
```

🔷 Explanation:

- The `final` method **prevents overriding** in any subclass.
- If a subclass tries to override `makeSound()`, the compiler throws an error.

3️⃣ Using a `private` Constructor (Singleton Pattern)

A class with a **private constructor cannot be extended**, because no subclass can call its constructor.

✅ Example: Preventing Inheritance with a `private` Constructor

```
class Animal {
    private Animal() { // Private constructor
        System.out.println("Animal created");
    }
}

// ❌ ERROR: Cannot extend a class with a private constructor
class Dog extends Animal { // Compilation Error
}
```

```
public class Main {
    public static void main(String[] args) {
        // Animal a = new Animal(); // ✕ ERROR: Cannot instantiate (private
constructor)
    }
}
```

🔷 Explanation:

- The `private` constructor **prevents class instantiation** and **inheritance**.
- Often used in **Singleton design patterns**.

4️⃣ Using `sealed` Classes (Java 17+)

Java 17 introduced **sealed classes**, allowing you to **restrict** which classes can inherit.

✅ Example: Using `sealed` to Restrict Inheritance

```
sealed class Animal permits Dog, Cat { // Only Dog and Cat can extend Animal
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

final class Dog extends Animal { } // ✅ Allowed
final class Cat extends Animal { } // ✅ Allowed

// ✕ ERROR: Fox cannot extend Animal (not in permits list)
// class Fox extends Animal { }

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.makeSound();
    }
}
```

🔷 Explanation:

- `sealed class Animal` **restricts** which classes can extend it.
- Only `Dog` and `Cat` are **permitted subclasses**.
- Any other class (e.g., `Fox`) will **cause a compilation error**.

🔹 Summary: Ways to Prevent Inheritance

Method	Effect
`final class`	**Prevents** a class from being extended.

Method	Effect
`final method`	**Prevents** a method from being overridden.
`private constructor`	**Prevents** both instantiation and inheritance.
`sealed class` (Java 17+)	**Restricts** which classes can inherit.

◈ When to Prevent Inheritance?

✅ When designing **immutable** classes (e.g., `String` is `final`).
✅ When implementing **Singletons** (private constructor).
✅ When creating **security-sensitive** classes.
✅ When **controlling API usage** with `sealed classes` (Java 17+).

◈ `final` Classes and Methods in Java

In Java, the `final` keyword is used to **restrict modifications**. It can be applied to **variables, methods, and classes**.

- `final class` → **Prevents inheritance** (no subclass can extend it).
- `final method` → **Prevents overriding** (subclass cannot change the method behavior).

---

1 `final` Classes – Preventing Inheritance

If a class is declared as `final`, it **cannot be extended** by any other class. This is useful when you want to **prevent modification** or ensure security in critical classes.

✅ Example: `final` Class

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

// ❌ ERROR: Cannot extend a final class
class Dog extends Animal { // Compilation error
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.makeSound();
```

```
    }
}
```

🔷 Output (Compilation Error)

```
Error: Cannot inherit from final class 'Animal'
```

🔷 Why?

- `final class Animal` **prevents subclasses** from extending it.
- If `Dog` tries to extend `Animal`, **compilation fails**.

✅ Real-world Example: `final` Class in Java API

Many core Java classes are `final` for **security and immutability**.
For example, `String` is a `final` class:

```
final class String {
    // Cannot be extended by subclasses
}
```

🔷 Why?

- Prevents someone from overriding String methods (security).
- Ensures string immutability (performance optimization).

2️⃣ `final` Methods – Preventing Overriding

A method declared as `final` **cannot be overridden** by a subclass. This is useful when you **want to keep a method's behavior unchanged**.

✅ Example: `final` Method

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

class Dog extends Animal {
    // ❌ ERROR: Cannot override final method
    /* void makeSound() {
        System.out.println("Dog barks");
    } */
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.makeSound(); // Calls Animal's method
    }
}
```

🔷 Output

```
Animal makes a sound
```

🔹 Why?

- `final void makeSound()` prevents overriding.
- If `Dog` tries to override it, **compilation fails**.

✅ Real-world Example: `final` Methods in Java API

- The `Thread` class has a `final` method `start()`, meaning you **cannot override** it:

```
class Thread {
    public final void start() {
        // Start the thread
    }
}
```

- This **ensures thread execution logic remains consistent**.

🔷 When to Use `final` in Java?

✅ Use `final class` when you want to **prevent inheritance** (e.g., `String`, `Wrapper classes`).
✅ Use `final method` when you want to **prevent method modification** (e.g., security-critical functions).
✅ Use `final variables` to **create constants** (`final int MAX_VALUE = 100;`).

🔷 Summary Table

`final` Keyword Usage	**Effect**
`final class`	**Prevents inheritance** (no subclasses).
`final method`	**Prevents overriding** (subclasses cannot modify it).
`final variable`	**Creates a constant** (value cannot be changed).

◈ The Object Class and Its Methods in Java

In Java, the `Object` class is **the root class of all Java classes**. Every class in Java **implicitly** inherits from `java.lang.Object`, either directly or indirectly.

◈ Key Points About `Object` Class

✅ `Object` is the **parent** of all Java classes.
✅ It provides **common methods** that all Java objects inherit.
✅ You can **override** some of these methods to define custom behavior.

---

1 Methods of `Object` Class

Method	Description
`equals(Object obj)`	Compares two objects for equality.
`hashCode()`	Returns the **hash code** of an object.
`toString()`	Returns a **string representation** of the object.
`getClass()`	Returns the **runtime class** of the object.
`clone()`	Creates a **copy** of the object (must implement `Cloneable`).
`finalize()`	Called before garbage collection.
`wait()`, `notify()`, `notifyAll()`	Used in **multithreading** for synchronization.

---

2 `equals(Object obj)` – Comparing Objects

By default, `equals()` **compares memory addresses** (same as ==).
To compare object **values**, you must **override** it.

✅ Example: Default `equals()` (Compares Memory)

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }
}

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

```
        Car car1 = new Car("Tesla");
        Car car2 = new Car("Tesla");

        System.out.println(car1.equals(car2)); // ❌ false (different
objects)
    }
}
```

🔷 Output

```
false
```

🔷 Why?

- `equals()` checks if `car1` and `car2` **are the same object in memory**, which they are not.

✅ Example: Overriding `equals()` to Compare Values

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Car car = (Car) obj;
        return model.equals(car.model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla");
        Car car2 = new Car("Tesla");

        System.out.println(car1.equals(car2)); // ✅ true (same model)
    }
}
```

🔷 Output

```
true
```

🔷 Why?

- Now `equals()` compares **values instead of memory addresses**.

3️⃣ `hashCode()` – Generating a Hash Code

A hash code is a **unique integer identifier** for an object, useful in **hash-based collections** like `HashMap`.

✅ Example: Default `hashCode()`

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla");
        Car car2 = new Car("Tesla");

        System.out.println(car1.hashCode());
        System.out.println(car2.hashCode());
    }
}
```

🔹 Output

```
12345678  // Example hash
87654321  // Different hash
```

🔷 Why?

- Since `hashCode()` is not overridden, each object has a **different hash**.

✅ Example: Overriding `hashCode()`

If you override `equals()`, you **must** override `hashCode()`, ensuring equal objects have the same hash.

```
import java.util.Objects;

class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    @Override
    public int hashCode() {
        return Objects.hash(model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla");
        Car car2 = new Car("Tesla");

        System.out.println(car1.hashCode()); // Same hash for both
        System.out.println(car2.hashCode());
    }
}
```

📤 Output

```
11223344
11223344
```

🔹 Why?

- Now, equal objects return the **same hash code**.

4️⃣ `toString()` – String Representation

By default, `toString()` returns:

```
ClassName@HexHashCode
```

✅ Example: Default `toString()`

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car("Tesla");
        System.out.println(car.toString());
    }
}
```

📤 Output

```
Car@1b6d3586  // Default output (memory address)
```

✅ Example: Overriding `toString()`

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    @Override
    public String toString() {
        return "Car Model: " + model;
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car("Tesla");
        System.out.println(car); // Calls overridden toString()
    }
}
```

📤 Output

```
Car Model: Tesla
```

🔷 Why?

- Now `toString()` prints a **custom message** instead of a memory reference.

5️⃣ `getClass()` – Getting Class Information

Returns the **runtime class** of an object.

✅ Example: Using `getClass()`

```
class Car { }

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        System.out.println(car.getClass());
    }
}
```

🔹 Output

```
class Car
```

🔷 Why?

- It returns the **class type** of the object.

6️⃣ `clone()` – Cloning an Object

Creates a **duplicate** object. To use it: ✅ The class must **implement** **`Cloneable`**.
✅ Override `clone()` inside the class.

✅ Example: Using `clone()`

```
class Car implements Cloneable {
    String model;

    Car(String model) {
        this.model = model;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Main {
    public static void main(String[] args) throws CloneNotSupportedException
{
        Car car1 = new Car("Tesla");
        Car car2 = (Car) car1.clone();
```

```
        System.out.println(car1.model);
        System.out.println(car2.model);
    }
}
```

Output

```
Tesla
Tesla
```

Why?

- `car2` is a **separate copy** of `car1`.

Summary Table of Object Methods

Method	Purpose
`equals(Object obj)`	Compares objects.
`hashCode()`	Generates a unique integer for an object.
`toString()`	Returns a string representation.
`getClass()`	Gets the class of an object.
`clone()`	Creates a copy of the object.
`finalize()`	Called before garbage collection.

Polymorphism in Java

Polymorphism is one of the core **OOP (Object-Oriented Programming)** concepts in Java. It allows objects to be treated as instances of their parent class, enabling **code reusability** and **flexibility**.

What is Polymorphism?

✅ **Poly (many) + Morphism (forms) → "Many Forms"**
✅ It allows **one interface** to be used for **multiple implementations**.
✅ Achieved through:

- **Method Overloading** (Compile-time polymorphism)
- **Method Overriding** (Runtime polymorphism)

1️⃣ Compile-Time Polymorphism (Method Overloading)

- **Same method name, different parameters** (number/type/order).
- Decided **at compile time** (static binding).
- Also called **static polymorphism**.

✅ Example: Method Overloading

```
class MathUtils {
    // Method with two parameters
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method with different data type
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathUtils math = new MathUtils();

        System.out.println(math.add(5, 10));       // 15
        System.out.println(math.add(5, 10, 15));   // 30
        System.out.println(math.add(5.5, 10.5));   // 16.0
    }
}
```

🔷 Output

```
15
30
16.0
```

🔷 Why?

- The compiler selects the correct method **based on arguments**.

2️⃣ Runtime Polymorphism (Method Overriding)

- **Same method signature, different implementation in subclass.**
- Decided **at runtime** (dynamic binding).
- Also called **dynamic polymorphism**.

✅ Example: Method Overriding

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

```
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Calls Dog's version
    }
}
```

🔷 Output

```
Dog barks
```

🔷 Why?

- Even though `myAnimal` is declared as `Animal`, it **calls `Dog`'s overridden method** at runtime.

3️⃣ Upcasting & Downcasting in Polymorphism

✅ Upcasting (Parent reference → Child object)

- Parent class reference **points to** a child class object.
- Only **parent class methods** can be accessed (unless overridden).

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // ✅ Calls overridden method
        // myAnimal.bark(); // ❌ ERROR: Cannot access child-specific methods
    }
}
```

✅ Downcasting (Converting Parent → Child)

- Converts a **parent reference** back into a **child reference**.
- Requires **explicit casting**.

```
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        Dog myDog = (Dog) myAnimal; // Downcasting
        myDog.bark(); // ✅ Now we can call Dog-specific methods
    }
}
```

🔷 Output

```
Dog barks
```

4️⃣ Polymorphism with Interfaces

Since Java **does not support multiple inheritance**, **interfaces** allow polymorphism by defining a common method signature.

✅ Example: Polymorphism with Interfaces

```
interface Animal {
    void makeSound(); // Abstract method
}

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

class Cat implements Animal {
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal1 = new Dog();
        Animal myAnimal2 = new Cat();

        myAnimal1.makeSound(); // Dog barks
        myAnimal2.makeSound(); // Cat meows
    }
}
```

🔷 Output

```
Dog barks
Cat meows
```

🔹 Why?

- Both `Dog` and `Cat` implement `Animal`, but provide **different behaviors**.

◈ Method Overloading vs Method Overriding

Feature	Method Overloading	Method Overriding
Definition	Multiple methods with **same name**, but different **parameters**	Redefining a method in **subclass**
Type	**Compile-time polymorphism**	**Runtime polymorphism**
Parameters	**Must be different**	**Must be the same**
Return Type	Can be **different**	Must be **same or covariant**
Access Modifier	No restrictions	Cannot reduce visibility (`protected` → `private` ✗)
Static Methods	Can be overloaded	Cannot be overridden

◈ When to Use Polymorphism?

✅ **Code Reusability** → Avoid duplicating similar methods.
✅ **Maintainability** → Makes adding new behaviors easier.
✅ **Flexibility** → Use parent class reference for multiple objects.
✅ **Extensibility** → Future-proofing your code by supporting multiple types.

◈ Summary

Type	Definition	Example
Method Overloading	Same method, different parameters	`add(int, int), add(double, double)`
Method Overriding	Same method, redefined in subclass	`Animal → Dog.makeSound()`
Upcasting	Parent reference → Child object	`Animal a = new Dog();`
Downcasting	Child reference from parent	`Dog d = (Dog) a;`
Interface Polymorphism	Multiple classes implementing the same interface	`Animal a = new Cat();`

🔷 Dynamic Binding in Java (Runtime Polymorphism)

🔷 What is Binding in Java?

Binding refers to the process of **linking a method call to its actual implementation**.
Java has two types of binding:

1️⃣ **Static Binding (Early Binding)** → Happens at **compile-time**.
2️⃣ **Dynamic Binding (Late Binding)** → Happens at **runtime**.

1️⃣ What is Dynamic Binding?

✅ **Dynamic Binding (Late Binding)** means that the method to be executed is determined at **runtime** based on the actual object type, not the reference type.
✅ It is associated with **Method Overriding** and **Polymorphism**.
✅ Achieved using **upcasting** (parent class reference pointing to a child class object).
✅ **Final, static, and private methods do not support dynamic binding**.

2️⃣ Example of Dynamic Binding

```
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 myAnimal = new Dog(); // Upcasting
        myAnimal.sound(); // ✅ Calls Dog's overridden method
    }
}
```

🔷 Output

```
Dog barks
```

🔹 Why?

- Even though `myAnimal` is declared as `Animal`, **Java binds the method at runtime** based on the actual object type (`Dog`).

3️⃣ Static Binding vs Dynamic Binding

Feature	Static Binding	Dynamic Binding
When it happens?	**Compile-time**	**Runtime**
Methods Involved	**Static, final, private** methods	**Overridden (instance) methods**
Performance	Faster	Slower (due to runtime decision)
Example	**Method Overloading**	**Method Overriding**

✅ Example of Static Binding (Compile-Time)

```
class MathUtils {
    static void add(int a, int b) {
        System.out.println(a + b);
    }
}

public class Main {
    public static void main(String[] args) {
        MathUtils.add(5, 10); // ✅ Compile-time binding
    }
}
```

🔹 Output

```
15
```

🔷 Why?

- Since `add()` is **static**, the method is resolved at **compile-time**.

4️⃣ Dynamic Binding with Upcasting

```
class Parent {
    void show() {
        System.out.println("Parent's show()");
    }
}

class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child's show()");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child(); // ✅ Upcasting
        obj.show(); // ✅ Calls Child's overridden method at runtime
    }
```

```
}
```

🔷 Output

```
Child's show()
```

5️⃣ Dynamic Binding and Final/Static/Private Methods

Static, final, and private methods are not overridden, so they use static binding instead of dynamic binding.

✅ Example: Static Method (Static Binding)

```
class Parent {
    static void display() {
        System.out.println("Parent's display()");
    }
}

class Child extends Parent {
    static void display() {
        System.out.println("Child's display()");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display(); // Calls Parent's display() (NOT overridden)
    }
}
```

🔷 Output

```
Parent's display()
```

🔹 **Why?**

- **Static methods do not support dynamic binding**.

6️⃣ Real-World Use of Dynamic Binding

Dynamic binding is used in **frameworks, design patterns, and libraries** where method calls are resolved dynamically.

✅ Example: Using a Parent Class for Multiple Subclasses

```
abstract class Employee {
    abstract void work();
}

class Developer extends Employee {
    void work() {
        System.out.println("Developer writes code");
    }
}
```

```
class Tester extends Employee {
    void work() {
        System.out.println("Tester tests applications");
    }
}

public class Main {
    public static void main(String[] args) {
        Employee emp1 = new Developer();
        Employee emp2 = new Tester();

        emp1.work(); // Developer writes code
        emp2.work(); // Tester tests applications
    }
}
```

🔹 Output

```
Developer writes code
Tester tests applications
```

🔷 Why?

- **The method to execute is decided at runtime**.

🔹 Summary

Type	When it Happens?	Methods Affected	Example
Static Binding	Compile-time	Static, final, private, overloaded methods	`static void display()`
Dynamic Binding	Runtime	Overridden methods (instance methods)	`void sound()` (overridden)

🔥 Key Takeaways

✅ **Dynamic Binding enables runtime method selection**.
✅ **Supports flexibility in OOP through polymorphism**.
✅ **Final, static, and private methods use static binding**.

🔷 Method Overriding in Java

🔷 What is Method Overriding?

✅ **Method Overriding** is a feature in Java that allows a **subclass** to provide a specific implementation of a method already defined in its **superclass**.
✅ The overridden method in the child class must have the **same name, return type, and parameters** as the method in the parent class.
✅ It supports **Runtime Polymorphism** (Dynamic Binding).
✅ Helps in **code reusability** and **method customization**.

1️⃣ Rules for Method Overriding

🔹 The method must have **the same name and parameters**.
🔹 The **return type must be the same or a subclass (covariant return type)**.
🔹 The **access modifier** cannot be more restrictive than the superclass method.
🔹 The method **cannot be static, final, or private**.
🔹 The method must be in a **subclass** (inheritance required).

2️⃣ Basic Example of Method Overriding

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Calls Dog's version
    }
}
```

🔷 Output

```
Dog barks
```

🔹 Why?

- Even though `myAnimal` is declared as `Animal`, at **runtime** Java calls the overridden method in `Dog`.

3️⃣ Using `super` to Call Parent Method

🔹 The `super` keyword allows calling the **overridden method** from the superclass.

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

class Dog extends Animal {
    @Override
    void makeSound() {
        super.makeSound(); // Calls the superclass method
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

🔷 Output

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

🔹 Why?

- `super.makeSound();` calls the **parent class** method before executing the child's method.

4️⃣ Overriding with Different Access Modifiers

🔹 The overridden method **cannot have a more restrictive access modifier** than the parent method.

Parent Method	Allowed in Child?
`public`	✅ `public`
`protected`	✅ `protected` or `public`
`default` (package-private)	✅ `default`, `protected`, or `public` (same package)
`private`	❌ **Cannot be overridden**

```
class Parent {
```

```
    protected void display() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    @Override
    public void display() { // ✅ Allowed (wider access)
        System.out.println("Child method");
    }
}
```

---

5 Covariant Return Type in Overriding

🔷 Java allows **overriding methods** to return a **subclass of the return type** from the parent class.

```
class Animal {
    Animal getInstance() {
        return new Animal();
    }
}

class Dog extends Animal {
    @Override
    Dog getInstance() { // ✅ Covariant return type
        return new Dog();
    }
}
```

✅ Why?

- `Dog` is a **subtype of** `Animal`, so it is allowed.

---

6 Overriding vs Overloading

Feature	Method Overriding	Method Overloading
Definition	Subclass provides a new implementation of a method	Multiple methods with the same name but different parameters
Return Type	Must be the same or a subclass (covariant return type)	Can be different
Access Modifier	Cannot reduce visibility	No restrictions
Binding Type	**Runtime (Dynamic Binding)**	**Compile-time (Static Binding)**

Feature	Method Overriding	Method Overloading
Static Methods	Cannot be overridden	Can be overloaded

7️⃣ Methods That Cannot Be Overridden

Method Type	Can It Be Overridden?	Why?
`static`	❌ No	Static methods belong to the **class, not instance**
`final`	❌ No	Final methods **cannot be modified**
`private`	❌ No	Private methods are **not inherited**
`constructors`	❌ No	Constructors are **not inherited**

✅ Example: `final` Method

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

class Child extends Parent {
    // ❌ ERROR: Cannot override final method
    // void show() { System.out.println("New method"); }
}
```

8️⃣ Real-World Example

🔹 Imagine a **bank system** where different account types have different interest calculations.

```
class Bank {
    double getInterestRate() {
        return 5.0; // Default interest rate
    }
}

class SavingsAccount extends Bank {
    @Override
    double getInterestRate() {
        return 6.5;
    }
}

class FixedDeposit extends Bank {
    @Override
    double getInterestRate() {
        return 7.5;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Bank acc1 = new SavingsAccount();
        Bank acc2 = new FixedDeposit();

        System.out.println("Savings Account Interest: " +
acc1.getInterestRate());
        System.out.println("Fixed Deposit Interest: " +
acc2.getInterestRate());
    }
}
```

Output

```
Savings Account Interest: 6.5
Fixed Deposit Interest: 7.5
```

Why?

- The method called is determined at **runtime**.

Summary

Feature	Overriding
Purpose	Modify inherited method behavior
Where?	Subclass
Binding	**Dynamic Binding (Runtime)**
Access Modifier	Cannot be more restrictive than the parent
Static Methods	Cannot be overridden
Final Methods	Cannot be overridden

Key Takeaways

- ✅ **Supports Runtime Polymorphism (Dynamic Binding).**
- ✅ **Allows behavior modification in subclasses.**
- ✅ **Super keyword can be used to call the parent method.**
- ✅ **Final, static, and private methods cannot be overridden.**

🔷 Abstract Classes and Methods in Java

1️⃣ What is an Abstract Class?

✅ An **abstract class** in Java is a class that **cannot be instantiated** (i.e., you cannot create objects from it).
✅ It is used as a **base class** for other classes to extend.
✅ Abstract classes can have **both abstract (unimplemented) and concrete (implemented) methods**.
✅ It helps in **code reuse and enforcing a contract for subclasses**.

```
abstract class Animal {  // Abstract class
    abstract void makeSound();  // Abstract method (no body)

    void sleep() {  // Concrete method (has body)
        System.out.println("Animal is sleeping");
    }
}
```

2️⃣ What is an Abstract Method?

✅ An **abstract method** is a method that **does not have a body** and must be implemented by subclasses.
✅ It is declared using the `abstract` keyword inside an **abstract class**.

```
abstract class Animal {
    abstract void makeSound(); // Abstract method (no implementation)
}
```

🔹 Rules for Abstract Methods:

- **Must be inside an abstract class**.
- **No method body (implementation)**.
- **Subclasses must override it** unless they are abstract.

3️⃣ Example: Abstract Class & Method

```
abstract class Animal {
    abstract void makeSound(); // Abstract method

    void sleep() { // Concrete method
        System.out.println("Animal is sleeping...");
    }
}

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

```
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Upcasting
        myDog.makeSound(); // Calls Dog's overridden method
        myDog.sleep(); // Calls concrete method from Animal class
    }
}
```

🔷 Output

```
Dog barks
Animal is sleeping...
```

✅ Why?

- `Dog` extends `Animal`, so it **must override the abstract method**.
- `sleep()` is a **concrete method**, so it is inherited.

---

4 Key Features of Abstract Classes

✅ **Cannot create objects** of an abstract class.
✅ Can have **abstract and non-abstract methods**.
✅ Can have **constructors** (used by subclasses).
✅ Can have **instance variables and static variables**.
✅ Can have **final methods**, which cannot be overridden.

---

5 Can an Abstract Class Have a Constructor?

✅ **Yes!** Abstract classes can have constructors, but they **cannot be instantiated directly**.

```
abstract class Vehicle {
    Vehicle() {
        System.out.println("Vehicle constructor called");
    }
}

class Car extends Vehicle {
    Car() {
        System.out.println("Car constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
    }
}
```

🔷 Output

```
Vehicle constructor called
Car constructor called
```

✅ Why?

- When `Car` is created, **the abstract class constructor is called first**.

6️⃣ Abstract Class vs Interface

Feature	Abstract Class	Interface
Methods	Can have **both abstract and concrete** methods	**All methods are abstract** (before Java 8)
Access Modifiers	Methods can be **public, protected, private**	Methods are **public by default**
Variables	Can have **instance and static variables**	Only **public, static, final (constants)**
Multiple Inheritance	**Not supported**	**Supported**
Constructors	**Can have a constructor**	**Cannot have a constructor**
Usage	Used when there is **common behavior** across classes	Used for **100% abstraction and multiple inheritance**

7️⃣ Example: Abstract Class vs Interface

```
// Abstract Class
abstract class Animal {
    abstract void makeSound();
    void sleep() {
        System.out.println("Animal sleeps");
    }
}

// Interface
interface Pet {
    void play();
}

// Concrete Class Implementing Both
class Dog extends Animal implements Pet {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
```

```
    @Override
    public void play() {
        System.out.println("Dog is playing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
        myDog.sleep();
        myDog.play();
    }
}
```

🔷 Output

```
Dog barks
Animal sleeps
Dog is playing
```

✅ Why?

- **Abstract class** provides a mix of **implemented and abstract methods**.
- **Interface** allows for **multiple inheritance**.

---

8 When to Use an Abstract Class vs an Interface?

✅ Use an **abstract class** when:
✓ You want to provide **some common implementation**.
✓ You expect **future modifications** to method implementations.

✅ Use an **interface** when:
✓ You need **100% abstraction**.
✓ You want **multiple inheritance**.
✓ You are defining **behavior (e.g., Comparable, Cloneable)**.

🔷 Summary

Feature	Abstract Class	Abstract Method
Can it be instantiated?	❌ No	❌ No
Can have concrete methods?	✅ Yes	❌ No
Must be overridden?	❌ No (only abstract methods)	✅ Yes
Can it have constructors?	✅ Yes	❌ No

🔥 Key Takeaways

- ✅ **Abstract classes enforce a common structure for subclasses**.
- ✅ **Cannot be instantiated directly**.
- ✅ **Can contain both abstract and concrete methods**.
- ✅ **Subclasses must override all abstract methods**.
- ✅ **Use interfaces for multiple inheritance, use abstract classes for common behavior**.

UNIT – III

🔷 Interfaces in Java

1️⃣ What is an Interface?

✅ An **interface** in Java is a **blueprint** for a class that defines **what** methods a class must implement, but not **how** they are implemented.
✅ **100% abstraction** (before Java 8): It **only contains method signatures**, not method bodies.
✅ **Supports multiple inheritance** (unlike abstract classes).
✅ **Encapsulation of behavior**: Defines what a class should do, but not how.

2️⃣ Declaring an Interface

🔹 An interface is declared using the `interface` keyword.

```
interface Animal {
    void makeSound(); // Abstract method (no body)
}
```

🔹 **Key Points:**

- All methods are **implicitly `public` and `abstract`** (before Java 8).
- Interfaces **cannot have constructors** (they cannot be instantiated).

3️⃣ Implementing an Interface

🔹 A class **implements** an interface using the `implements` keyword and must **override all its methods**.

```
interface Animal {
    void makeSound(); // Abstract method
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

🔷 Output

```
Dog barks
```

✅ Why?

- `Dog` **implements** `Animal`, so it **must override** the `makeSound()` method.

4️⃣ Multiple Inheritance Using Interfaces

✅ **A class can implement multiple interfaces**, unlike abstract classes.

```
interface Animal {
    void eat();
}

interface Pet {
    void play();
}

class Dog implements Animal, Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }

    @Override
    public void play() {
        System.out.println("Dog is playing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.play();
    }
}
```

🔷 Output

```
Dog is eating
Dog is playing
```

✅ Why?

- `Dog` implements both `Animal` and `Pet`, so it **inherits behavior from multiple interfaces**.

5️⃣ Interface with Variables

🔹 **Interface variables are implicitly**:

- `public` (accessible everywhere)
- `static` (belongs to the interface, not objects)
- `final` (cannot be changed)

```
interface Game {
    int MAX_PLAYERS = 4; // Implicitly public, static, final
}
```

---

6 Default & Static Methods in Interfaces (Java 8+)

🔷 **Java 8 introduced** `default` and `static` methods in interfaces.

✅ Default Methods (Can have implementation)

- Allows **adding new methods** without breaking existing implementations.

```
interface Animal {
    default void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog implements Animal {}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sleep(); // Uses the default method
    }
}
```

🔹 Output

```
Animal is sleeping
```

✅ Why?

- `Dog` does not override `sleep()`, so it uses the **default implementation**.

✅ Static Methods (Belong to Interface)

- Can be called without creating an object.

```
interface Animal {
    static void info() {
        System.out.println("Animals have different sounds.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal.info(); // Calling static method from the interface
```

```
    }
}
```

🔷 Output

```
Animals have different sounds.
```

7️⃣ Functional Interfaces (Java 8)

✅ **A functional interface has exactly one abstract method** and can be used with **lambda expressions**.

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Calculator sum = (a, b) -> a + b; // Lambda expression
        System.out.println(sum.add(5, 3));
    }
}
```

🔷 Output

```
8
```

✅ Why?

- **Lambda expressions** work because `Calculator` is a **functional interface**.

8️⃣ Interface vs Abstract Class

Feature	Interface	Abstract Class
Methods	**Only abstract (before Java 8)**	Can have **abstract & concrete** methods
Access Modifiers	Methods are **public by default**	Methods can be `public`, `protected`, `private`
Multiple Inheritance	✅ **Allowed**	❌ **Not allowed**
Variables	`public static final` (constants)	Can have instance variables
Constructors	❌ No	✅ Yes
Performance	Slower (indirect access)	Faster (direct method calls)

---

9 When to Use an Interface?

✅ Use an **interface** when:
✓ You need **multiple inheritance**.
✓ You want to define **common behavior across unrelated classes**.
✓ You want to provide a **contract without implementation**.

✅ Use an **abstract class** when:
✓ You need **shared code** for subclasses.
✓ You want to define **some methods with implementations**.
✓ You are creating a **base class**.

🔷 Summary

Feature	Interface
Purpose	Defines a contract (behavior) for classes
Abstraction	**100% abstraction** (before Java 8)
Methods	Only abstract (before Java 8), **default & static allowed in Java 8+**
Variables	`public static final` (constants)
Multiple Inheritance	✅ Allowed
Performance	❌ Slower than abstract classes
Usage	Used for defining common behavior across multiple unrelated classes

---

🔥 Key Takeaways

✅ **Interfaces define behavior but do not implement it (before Java 8).**
✅ **Support multiple inheritance**, unlike abstract classes.
✅ **All methods are public and abstract by default (before Java 8).**
✅ **Default and static methods** allow adding new functionality without breaking existing code.
✅ **Functional interfaces** enable **lambda expressions** in Java.

🔷 Interfaces vs Abstract Classes in Java

Both **interfaces** and **abstract classes** are used to achieve **abstraction**, but they have key differences. Below is a detailed comparison.

1️⃣ Basic Differences

Feature	Interface	Abstract Class
Definition	A contract that defines methods but does not implement them (before Java 8).	A class that can have both **abstract (unimplemented)** and **concrete (implemented)** methods.
Abstraction Level	**100% abstraction** (before Java 8).	**0% to 100% abstraction** (can have both abstract and concrete methods).
Implementation	Must be **implemented** by a class using `implements`.	Must be **extended** by a class using `extends`.
Method Types	🔹 Abstract methods (before Java 8) 🔹 Default & Static methods (Java 8+)	🔹 Abstract methods 🔹 Concrete (implemented) methods
Variables	🔹 `public static final` (constants only)	🔹 Can have **instance variables, static variables, and constants**
Multiple Inheritance	✅ **Allowed** (A class can implement multiple interfaces).	❌ **Not allowed** (A class can extend only one abstract class).
Constructors	❌ **Cannot have a constructor**	✅ **Can have constructors**
Performance	❌ **Slightly slower** due to interface method lookup.	✅ **Faster** due to direct method calls.
Access Modifiers	Methods are **public** by default.	Methods can be **private, protected, or public**.
Usage	Used when multiple classes need **common behavior** but not common implementation.	Used when multiple classes share **common behavior and implementation**.

---

2 Example: Interface vs Abstract Class

✅ Interface Example

```
interface Animal {
    void makeSound();  // Abstract method (no implementation)
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

🔷 Output

```
Dog barks
```

✅ Why?

- `Dog` implements `Animal` and provides its own version of `makeSound()`.

✅ Abstract Class Example

```
abstract class Animal {
    abstract void makeSound();  // Abstract method
    void sleep() { // Concrete method
        System.out.println("Animal is sleeping...");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();  // Calls overridden method
        myDog.sleep();  // Calls method from abstract class
    }
}
```

🔷 Output

```
Dog barks
Animal is sleeping...
```

✅ Why?

- `Dog` **inherits** `sleep()` from `Animal` (abstract class).
- `Dog` must override `makeSound()` because it's abstract.

---

3 Multiple Inheritance Example (Only with Interfaces)

✅ **A class can implement multiple interfaces** but **cannot extend multiple classes**.

```
interface Animal {
    void eat();
}

interface Pet {
    void play();
}

class Dog implements Animal, Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }

    @Override
    public void play() {
        System.out.println("Dog is playing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.play();
    }
}
```

🔷 Output

```
Dog is eating
Dog is playing
```

✅ Why?

- `Dog` implements both `Animal` and `Pet`, allowing multiple inheritance.

---

4 Key Differences: Interface vs Abstract Class

Feature	Interface	Abstract Class
Extending & Implementing	A class **implements** an interface.	A class **extends** an abstract class.
Multiple Inheritance	✅ **Allowed**	❌ **Not Allowed**

Feature	Interface	Abstract Class
Default Methods (Java 8+)	✅ Allowed	✅ Allowed
Static Methods (Java 8+)	✅ Allowed	✅ Allowed
Instance Variables	❌ **Not allowed** (Only `public static final` constants)	✅ **Allowed**
Method Implementation	❌ Before Java 8, **all methods must be abstract**	✅ Can have **both abstract & concrete methods**
Constructors	❌ No constructors	✅ Can have constructors

---

5 When to Use an Interface vs Abstract Class?

✅ Use an Interface when:

✔ You need **multiple inheritance**.
✔ You want to define **common behavior across unrelated classes**.
✔ You want to provide a **contract without implementation**.
✔ You need **functional interfaces** (Java 8, used with lambda expressions).

✅ Use an Abstract Class when:

✔ You need **shared code** among subclasses.
✔ You want to define **some methods with implementations**.
✔ You are creating a **base class** that provides **common fields and methods**.
✔ You need a class that **partially implements methods**.

---

6 Real-World Example

Interface Example (Common Behavior Across Unrelated Classes)

```
interface Flyable {
    void fly();
}

class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird flies in the sky");
    }
}
```

```
class Airplane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Airplane flies with engines");
    }
}

public class Main {
    public static void main(String[] args) {
        Flyable bird = new Bird();
        Flyable plane = new Airplane();

        bird.fly();
        plane.fly();
    }
}
```

🔷 Output

```
Bird flies in the sky
Airplane flies with engines
```

✅ Why?

- `Bird` and `Airplane` are **unrelated**, but both can **fly**.

Abstract Class Example (Shared Code for Related Classes)

```
abstract class Vehicle {
    void start() {
        System.out.println("Vehicle is starting...");
    }

    abstract void fuelType(); // Abstract method
}

class Car extends Vehicle {
    @Override
    void fuelType() {
        System.out.println("Car runs on petrol");
    }
}

class Truck extends Vehicle {
    @Override
    void fuelType() {
        System.out.println("Truck runs on diesel");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        Vehicle myTruck = new Truck();

        myCar.start();
        myCar.fuelType();
```

```
        myTruck.start();
        myTruck.fuelType();
    }
}
```

🔷 Output

```
Vehicle is starting...
Car runs on petrol
Vehicle is starting...
Truck runs on diesel
```

✅ Why?

- `Car` and `Truck` share **common behavior (`start()`)** from `Vehicle`.

7️⃣ Conclusion: When to Choose What?

Scenario	Choose
Need **100% abstraction**	Interface
Need **partial abstraction**	Abstract Class
Need **multiple inheritance**	Interface
Need **default behavior for some methods**	Abstract Class
Need to define **common behavior for unrelated classes**	Interface
Need to define **common behavior for related classes**	Abstract Class

---

🔥 Final Thoughts

✅ **Use interfaces** for **multiple inheritance & behavior definition**.
✅ **Use abstract classes** when **you have shared code** and **need partial implementation**.
✅ Java 8+ allows **default & static methods in interfaces**, making them more flexible.

🔷 Defining an Interface in Java

1️⃣ What is an Interface?

An **interface** in Java is a **blueprint** for a class that defines **what** methods a class must implement but not **how** they are implemented.

✅ **100% abstraction** (before Java 8).
✅ **No instance variables** (only constants: `public static final`).
✅ **Supports multiple inheritance** (a class can implement multiple interfaces).
✅ **Introduced `default` and `static` methods in Java 8**.

2️⃣ How to Define an Interface

🔹 Basic Syntax

```
interface InterfaceName {
    // Abstract methods (implicitly public and abstract)
    returnType methodName(parameters);
}
```

✅ Example

```
interface Animal {
    void makeSound();  // Abstract method (no body)
}
```

🔷 **Key Points:**

- **Methods inside an interface are abstract by default** (before Java 8).
- **All methods are implicitly `public` and `abstract`.**
- **Interfaces cannot have constructors** because they cannot be instantiated.

3️⃣ Implementing an Interface

A class **implements** an interface using the `implements` keyword.

```
interface Animal {
    void makeSound();  // Abstract method
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

🔹 Output

```
Dog barks
```

✅ **Why?**

- `Dog` implements `Animal` and must provide an implementation for `makeSound()`.

---

4 Interface with Multiple Methods

An interface can define multiple abstract methods.

```
interface Vehicle {
    void start();
    void stop();
}

class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car is starting...");
    }

    @Override
    public void stop() {
        System.out.println("Car is stopping...");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();
        myCar.stop();
    }
}
```

◈ Output

```
Car is starting...
Car is stopping...
```

☑ Why?

- `Car` implements `Vehicle` and provides implementations for all methods.

---

5 Variables in an Interface

🔷 All variables in an interface are implicitly:

- `public`
- `static`
- `final` (constants, cannot be changed)

```
interface Game {
    int MAX_PLAYERS = 4; // public static final by default
}

public class Main {
```

```
    public static void main(String[] args) {
        System.out.println("Max players allowed: " + Game.MAX_PLAYERS);
    }
}
```

🔷 Output

```
Max players allowed: 4
```

6️⃣ Default and Static Methods (Java 8+)

✅ Default Method (Method with Implementation)

- Used to add new methods **without breaking existing implementations**.

```
interface Animal {
    default void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog implements Animal {}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sleep(); // Uses the default method
    }
}
```

🔷 Output

```
Animal is sleeping
```

✅ Why?

- `Dog` does not override `sleep()`, so it uses the **default implementation**.

✅ Static Method (Belongs to Interface)

- Can be called **without creating an object**.

```
interface Animal {
    static void info() {
        System.out.println("Animals have different sounds.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal.info(); // Calling static method from the interface
    }
}
```

🔷 Output

```
Animals have different sounds.
```

7️⃣ Multiple Inheritance Using Interfaces

✅ **A class can implement multiple interfaces**, unlike abstract classes.

```
interface Animal {
    void eat();
}

interface Pet {
    void play();
}

class Dog implements Animal, Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }

    @Override
    public void play() {
        System.out.println("Dog is playing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.play();
    }
}
```

🔷 Output

```
Dog is eating
Dog is playing
```

✅ **Why?**

- `Dog` implements both `Animal` and `Pet`, allowing **multiple inheritance**.

8️⃣ Functional Interfaces (Java 8)

✅ **A functional interface has exactly one abstract method** and can be used with **lambda expressions**.

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);
}

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

```
        Calculator sum = (a, b) -> a + b; // Lambda expression
        System.out.println(sum.add(5, 3));
    }
}
```

🔷 Output

```
8
```

✅ Why?

- **Lambda expressions** work because `Calculator` is a **functional interface**.

9️⃣ Interface vs Abstract Class

Feature	Interface	Abstract Class
Methods	**Only abstract (before Java 8)**	Can have **abstract & concrete** methods
Access Modifiers	Methods are **public by default**	Methods can be `public`, `protected`, `private`
Multiple Inheritance	✅ **Allowed**	❌ **Not allowed**
Variables	`public static final` (constants)	Can have instance variables
Constructors	❌ No	✅ Yes
Performance	Slower (indirect access)	Faster (direct method calls)

🔟 When to Use an Interface?

✅ Use an **interface** when:
✔ You need **multiple inheritance**.
✔ You want to define **common behavior across unrelated classes**.
✔ You want to provide a **contract without implementation**.

✅ Use an **abstract class** when:
✔ You need **shared code** for subclasses.
✔ You want to define **some methods with implementations**.
✔ You are creating a **base class**.

Summary

Feature	Interface
Purpose	Defines a contract (behavior) for classes
Abstraction	**100% abstraction** (before Java 8)
Methods	Only abstract (before Java 8), **default & static allowed in Java 8+**
Variables	`public static final` (constants)
Multiple Inheritance	✅ Allowed
Performance	❌ Slower than abstract classes
Usage	Used for defining common behavior across multiple unrelated classes

---

Key Takeaways

- ✅ **Interfaces define behavior but do not implement it (before Java 8).**
- ✅ **Support multiple inheritance**, unlike abstract classes.
- ✅ **All methods are public and abstract by default (before Java 8).**
- ✅ **Default and static methods** allow adding new functionality without breaking existing code.
- ✅ **Functional interfaces** enable **lambda expressions** in Java.

Implementing Interfaces in Java

When a **class implements an interface**, it **inherits** the abstract methods of the interface and **must provide concrete implementations** for them.

- ✅ **A class uses `implements` to implement an interface**.
- ✅ **Multiple interfaces can be implemented in a single class**.
- ✅ **A class must override all abstract methods of the interface** unless it's declared `abstract`.

---

1 Basic Syntax

```
interface InterfaceName {
    returnType methodName(parameters);  // Abstract method
}

class ClassName implements InterfaceName {
```

```
    @Override
    public returnType methodName(parameters) {
        // Method implementation
    }
}
```

2️⃣ Example: Implementing a Single Interface

```
// Define an interface
interface Animal {
    void makeSound();  // Abstract method
}

// Implementing the interface in a class
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

🔷 Output

```
Dog barks
```

✅ Why?

- `Dog` **implements** `Animal`, so it must provide a concrete version of `makeSound()`.

3️⃣ Implementing Multiple Interfaces

A class can implement **multiple interfaces** to support **multiple inheritance**.

```
interface Animal {
    void eat();
}

interface Pet {
    void play();
}

class Dog implements Animal, Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }

    @Override
```

```
    public void play() {
        System.out.println("Dog is playing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.play();
    }
}
```

🔷 Output

```
Dog is eating
Dog is playing
```

✅ Why?

- `Dog` implements **both** `Animal` and `Pet`, so it provides implementations for **both methods**.

4️⃣ Implementing an Interface in an Abstract Class

An **abstract class** can **implement** an interface but **does not have to implement all methods**. The subclass must complete the implementation.

```
interface Vehicle {
    void start();
}

// Abstract class implementing an interface
abstract class Car implements Vehicle {
    void fuel() {
        System.out.println("Filling fuel...");
    }
}

// Concrete class implementing remaining methods
class Tesla extends Car {
    @Override
    public void start() {
        System.out.println("Tesla is starting silently...");
    }
}

public class Main {
    public static void main(String[] args) {
        Tesla myTesla = new Tesla();
        myTesla.start();
        myTesla.fuel();
    }
}
```

🔷 Output

```
Tesla is starting silently...
Filling fuel...
```

✅ Why?

- `Car` **partially implements** `Vehicle`.
- `Tesla` **fully implements** all methods.

5️⃣ Using Default Methods (Java 8+)

Java 8 introduced **default methods** to provide method implementations **inside interfaces**.

```
interface Animal {
    void makeSound();

    default void sleep() {
        System.out.println("Animal is sleeping...");
    }
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
        myDog.sleep();  // Uses default method
    }
}
```

🔷 Output

```
Dog barks
Animal is sleeping...
```

✅ Why?

- `Dog` **inherits** `sleep()` from `Animal` **without overriding it**.

6️⃣ Using Static Methods in Interfaces (Java 8+)

Static methods can be **called directly on an interface**.

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

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

```
        System.out.println(MathOperations.add(5, 3));  // Static method call
    }
}
```

🔷 Output

```
8
```

✅ Why?

- **Static methods belong to the interface** and are **called without an object**.

7️⃣ Real-World Example: Interface in Banking System

```
interface Bank {
    void deposit(double amount);
    void withdraw(double amount);
}

class SavingsAccount implements Bank {
    private double balance = 0;

    @Override
    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: $" + amount);
    }

    @Override
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: $" + amount);
        } else {
            System.out.println("Insufficient funds!");
        }
    }

    void checkBalance() {
        System.out.println("Current Balance: $" + balance);
    }
}

public class Main {
    public static void main(String[] args) {
        SavingsAccount myAccount = new SavingsAccount();
        myAccount.deposit(500);
        myAccount.withdraw(200);
        myAccount.checkBalance();
    }
}
```

🔷 Output

```
Deposited: $500
Withdrawn: $200
Current Balance: $300
```

✅ Why?

- `SavingsAccount` **implements** `Bank` and provides **banking operations**.

---

8 Summary

Feature	Interface
Purpose	Defines a contract (behavior) for classes
Abstraction	**100% abstraction** (before Java 8)
Methods	Only abstract (before Java 8), **default & static allowed in Java 8+**
Variables	`public static final` (constants)
Multiple Inheritance	✅ Allowed
Constructors	❌ Not allowed
Usage	Used for defining common behavior across multiple unrelated classes

---

9 Key Takeaways

✅ **Interfaces define behavior but do not implement it (before Java 8).**
✅ **Support multiple inheritance**, unlike abstract classes.
✅ **All methods are public and abstract by default (before Java 8).**
✅ **Default and static methods** allow adding new functionality without breaking existing code.
✅ **Functional interfaces** enable **lambda expressions** in Java.

🔷 Accessing Implementations Through Interface References in Java

In Java, you can **access an implementation class through an interface reference**. This enables **loose coupling**, **polymorphism**, and **better code maintainability**.

✅ Why use Interface References?

- Enables **polymorphism** (one interface, multiple implementations).
- Makes the code **flexible** and **maintainable**.
- Hides implementation details from the user.

1️⃣ Syntax

```
InterfaceName ref = new ImplementationClass();
```

- **Left side** → Interface reference (`InterfaceName`).
- **Right side** → Implementation object (`new ImplementationClass()`).

✅ The reference **can only call methods defined in the interface** (not specific to the class).

2️⃣ Example: Interface Reference with a Single Implementation

```
interface Animal {
    void makeSound();
}

// Implementation class
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myPet = new Dog();  // Interface reference
        myPet.makeSound();  // Calls Dog's implementation
    }
}
```

🔷 Output

```
Dog barks
```

✅ Why?

- `myPet` is an `Animal` reference pointing to a `Dog` object.
- It can **only access `Animal`'s methods**, even though `Dog` has implemented them.

3️⃣ Example: Multiple Implementations Using Interface Reference

An interface reference can refer to **any object of a class that implements the interface**.

```
interface Animal {
    void makeSound();
}

// Two different implementations
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}
```

```
class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal;  // Interface reference

        myAnimal = new Dog();
        myAnimal.makeSound();  // Calls Dog's method

        myAnimal = new Cat();
        myAnimal.makeSound();  // Calls Cat's method
    }
}
```

🔷 Output

```
Dog barks
Cat meows
```

✅ Why?

- The **same interface reference (`myAnimal`) is used for different implementations**.
- **Polymorphism**: At runtime, the correct method is called based on the actual object.

4️⃣ Example: Interface References in Method Parameters

You can pass an interface reference to a method, allowing it to work with **multiple implementations**.

```
interface Vehicle {
    void start();
}

class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car is starting...");
    }
}

class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike is starting...");
    }
}

// Method accepting interface reference
public class Main {
    static void testVehicle(Vehicle v) {
        v.start();
```

```
    }

    public static void main(String[] args) {
        Vehicle myCar = new Car();
        Vehicle myBike = new Bike();

        testVehicle(myCar);  // Calls Car's start()
        testVehicle(myBike); // Calls Bike's start()
    }
}
```

🔷 Output

```
Car is starting...
Bike is starting...
```

✅ Why?

- `testVehicle(Vehicle v)` works with **any class that implements `Vehicle`**.
- **Reusability**: No need for separate methods for `Car` and `Bike`.

5️⃣ Example: Interface References in Arrays

An array of interface references can store **multiple objects of different implementations**.

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[2];  // Array of interface references
        shapes[0] = new Circle();
        shapes[1] = new Rectangle();

        for (Shape s : shapes) {
            s.draw();  // Calls the correct draw() method
        }
    }
}
```

🔷 Output

```
Drawing Circle
```

```
Drawing Rectangle
```

✅ Why?

- `shapes` stores objects of **different classes (`Circle`, `Rectangle`)** using the `Shape` interface.
- **Polymorphism allows calling the correct `draw()` method**.

---

6 Interface Reference with Default Methods (Java 8+)

Since Java 8, **interfaces can have default methods**, which can be accessed via interface references.

```
interface Animal {
    void makeSound();

    default void sleep() {
        System.out.println("Animal is sleeping...");
    }
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myPet = new Dog();
        myPet.makeSound();  // Calls overridden method
        myPet.sleep();      // Calls default method in the interface
    }
}
```

🔷 Output

```
Dog barks
Animal is sleeping...
```

✅ Why?

- The `sleep()` method is **inherited from the interface**.
- **No need for the `Dog` class to override it**.

---

7 Interface Reference and Downcasting

If a method **only exists in the implementation class**, the interface reference **cannot access it directly**.

```
interface Animal {
    void makeSound();
```

```
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }

    void wagTail() {
        System.out.println("Dog is wagging tail");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myPet = new Dog();
        myPet.makeSound();
        // myPet.wagTail(); // ✕ Compile-time error

        // Downcasting to access Dog-specific methods
        if (myPet instanceof Dog) {
            Dog myDog = (Dog) myPet;
            myDog.wagTail();
        }
    }
}
```

◈ Output

```
Dog barks
Dog is wagging tail
```

☑ Why?

- **`wagTail()` is not in `Animal`, so `myPet` cannot call it.**
- **Downcasting (`(Dog) myPet`) allows access to `Dog`-specific methods.**

---

8 Summary

Feature	Description
Interface Reference	Holds objects of any class that implements the interface
Polymorphism	Calls correct method based on the actual object type
Method Parameters	Interface references allow passing multiple implementations to a method
Arrays	Can store multiple objects of different implementations
Downcasting	Required to access class-specific methods

9️⃣ Key Takeaways

✅ **Interface references allow polymorphism** – multiple classes can be accessed using a single reference type.
✅ **Only methods defined in the interface can be called** through the reference.
✅ **Useful in method parameters, arrays, and collections** for flexible and reusable code.
✅ **Downcasting allows access to class-specific methods** not declared in the interface.

🔷 Extending Interfaces in Java

In Java, an interface can **extend another interface** using the `extends` keyword. This allows interfaces to **inherit methods** from other interfaces, promoting **code reusability and modularity**.

✅ Why Extend an Interface?

- Supports **interface inheritance** (similar to class inheritance).
- Allows **adding more methods** without modifying existing interfaces.
- Enables **hierarchical structure** for better code organization.
- Supports **multiple inheritance** of interfaces.

1️⃣ Basic Syntax

```
interface ParentInterface {
    void method1();
}

// Child interface extending ParentInterface
interface ChildInterface extends ParentInterface {
    void method2();
}

// Class implementing the child interface
class ImplementationClass implements ChildInterface {
    @Override
    public void method1() {
        System.out.println("Method1 from ParentInterface");
    }

    @Override
    public void method2() {
        System.out.println("Method2 from ChildInterface");
    }
}

public class Main {
    public static void main(String[] args) {
        ChildInterface obj = new ImplementationClass();
        obj.method1();  // Inherited from ParentInterface
```

```
        obj.method2();  // Defined in ChildInterface
    }
}
```

🔷 Output

```
Method1 from ParentInterface
Method2 from ChildInterface
```

✅ Why?

- `ChildInterface` **inherits** `method1()` from `ParentInterface`.
- `ImplementationClass` **must implement both methods**.

---

2 Extending Multiple Interfaces

A **Java interface can extend multiple interfaces**, unlike classes (which support single inheritance).

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

interface B {
    void methodB();
}

// Extending multiple interfaces
interface C extends A, B {
    void methodC();
}

// Class implementing interface C
class MultiImplementation implements C {
    @Override
    public void methodA() {
        System.out.println("Method A");
    }

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

    @Override
    public void methodC() {
        System.out.println("Method C");
    }
}

public class Main {
    public static void main(String[] args) {
        C obj = new MultiImplementation();
        obj.methodA();
        obj.methodB();
        obj.methodC();
    }
```

```
}
```

🔷 Output

```
Method A
Method B
Method C
```

✅ Why?

- `C` **inherits all methods** from `A` and `B`.
- `MultiImplementation` **implements all inherited methods**.

3️⃣ Extending an Interface with Default Methods (Java 8+)

Since Java 8, interfaces can have **default methods**, which can be inherited and overridden.

```
interface ParentInterface {
    default void greet() {
        System.out.println("Hello from ParentInterface!");
    }
}

interface ChildInterface extends ParentInterface {
    default void greet() {
        System.out.println("Hello from ChildInterface!");
    }
}

class ImplementationClass implements ChildInterface {}

public class Main {
    public static void main(String[] args) {
        ChildInterface obj = new ImplementationClass();
        obj.greet();  // Calls ChildInterface's greet()
    }
}
```

🔷 Output

```
Hello from ChildInterface!
```

✅ Why?

- `ChildInterface` **overrides** the `greet()` method from `ParentInterface`.
- `ImplementationClass` **inherits the overridden version**.

4️⃣ Extending Interfaces Without Overriding Methods

If a class implements a child interface that extends another interface, it **must implement all inherited methods**.

```
interface Animal {
```

```
    void eat();
}

interface Pet extends Animal {
    void play();
}

class Dog implements Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating...");
    }

    @Override
    public void play() {
        System.out.println("Dog is playing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Pet myPet = new Dog();
        myPet.eat();
        myPet.play();
    }
}
```

🔷 Output

```
Dog is eating...
Dog is playing...
```

✅ Why?

- `Pet` **inherits** `eat()` from `Animal`.
- `Dog` **implements both `eat()` and `play()`.**

5️⃣ Extending Interfaces in a Real-World Scenario

Imagine a payment system where different payment methods (Credit Card, PayPal) follow a common interface but also have additional functionalities.

```
interface Payment {
    void processPayment(double amount);
}

// Extending Payment for online transactions
interface OnlinePayment extends Payment {
    void authenticateUser();
}

// Implementing the OnlinePayment interface
class PayPal implements OnlinePayment {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
```

```
    @Override
    public void authenticateUser() {
        System.out.println("Authenticating PayPal user...");
    }
}

public class Main {
    public static void main(String[] args) {
        OnlinePayment payment = new PayPal();
        payment.authenticateUser();
        payment.processPayment(100.0);
    }
}
```

🔷 Output

```
Authenticating PayPal user...
Processing PayPal payment of $100.0
```

✅ Why?

- `OnlinePayment` extends `Payment`, so it inherits `processPayment()`.
- `PayPal` **implements both `processPayment()` and `authenticateUser()`.**

---

6 Summary

Feature	Description
Extending an Interface	One interface can extend another using `extends`
Multiple Inheritance	An interface can extend multiple interfaces
Method Inheritance	Child interfaces inherit methods from parent interfaces
Default Methods	Can be inherited and overridden
Implementation	Classes implementing the child interface must implement all methods

---

7 Key Takeaways

- ✅ **An interface can extend another interface** to inherit its methods.
- ✅ **Multiple interface inheritance** is allowed (unlike class inheritance).
- ✅ **A class implementing an extended interface must implement all inherited methods**.
- ✅ **Default methods can be inherited and overridden** in extended interfaces.

🔷 Packages in Java

A **package** in Java is a way to **organize classes, interfaces, and sub-packages** into a structured manner. It helps avoid name conflicts and improves code maintainability.

✅ Why Use Packages?

- **Avoids name conflicts** between classes.
- **Provides access control** (public, private, protected).
- **Easier code organization** (group related classes).
- **Enhances modularity and reusability**.

1️⃣ Types of Packages in Java

Java provides two types of packages:

Type	Description
Built-in packages	Predefined Java packages (e.g., `java.util`, `java.io`)
User-defined packages	Custom packages created by developers

2️⃣ Creating a Package

To create a package, use the **`package`** keyword at the beginning of a Java file.

Example: Creating a User-Defined Package

📌 Steps to Create and Use a Package

1. Declare the package at the top of the Java file.
2. Save the file inside a folder matching the package name.
3. Compile using `javac -d . FileName.java`.
4. Use the package in another class with `import packageName.*;`.

📝 Example 1: Creating a Package

Step 1: Create a package named `mypackage`

```
// File: MyClass.java (inside 'mypackage' folder)
package mypackage;  // Package declaration

public class MyClass {
    public void displayMessage() {
        System.out.println("Hello from MyClass in mypackage!");
```

```
    }
}
```

✅ **Save this file inside a folder named `mypackage`.**

📝 Example 2: Using the Package in Another Class

Step 2: Import and Use the `mypackage`

```
// File: Main.java (outside 'mypackage' folder)
import mypackage.MyClass;  // Import the class

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Using the package class
        obj.displayMessage();
    }
}
```

🔷 Compilation & Execution

```
javac -d . MyClass.java    # Compiles and stores class in 'mypackage' folder
javac Main.java            # Compile Main.java
java Main                  # Run the program
```

🔷 Output

```
Hello from MyClass in mypackage!
```

✅ **Why?**

- `MyClass` is inside `mypackage`, so we import it in `Main.java`.
- The **`-d .`** flag ensures the compiled `.class` file is placed in the correct package folder.

3️⃣ Importing a Package

You can import a package in three ways:

Import Statement	Description
`import packageName.ClassName;`	Imports a specific class
`import packageName.*;`	Imports all classes from the package
`import static packageName.ClassName.methodName;`	Imports a static method

📝 Example 3: Importing All Classes

```
import mypackage.*;  // Import all classes from mypackage

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

```
        MyClass obj = new MyClass();  // No need to specify package
        obj.displayMessage();
    }
}
```

✅ Why?

- `import mypackage.*;` allows access to **all classes** in `mypackage`.
- No need to import each class separately.

---

4 Access Modifiers & Packages

Modifier	Same Class	Same Package	Different Package (Subclass)	Different Package (Non-Subclass)
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
No modifier (default)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

🔹 **Example:** A `protected` method can be accessed **within the same package** or in **subclasses of different packages**.

---

5 Sub-Packages

A sub-package is a package inside another package.

📝 Example 4: Creating a Sub-Package

```
// File: mypackage/subpackage/SubClass.java
package mypackage.subpackage;  // Sub-package declaration

public class SubClass {
    public void show() {
        System.out.println("Inside SubClass of subpackage");
    }
}
```

Using the Sub-Package

```
import mypackage.subpackage.SubClass;  // Import sub-package class

public class Main {
    public static void main(String[] args) {
        SubClass obj = new SubClass();
        obj.show();
```

```
    }
}
```

🔷 Compilation & Execution

```
javac -d . mypackage/subpackage/SubClass.java
javac -d . Main.java
java Main
```

🔷 Output

```
Inside SubClass of subpackage
```

✅ Why?

- The `.` **(dot)** represents a sub-package (`mypackage.subpackage`).
- Import **must match** the full package structure.

---

6 Java Built-in Packages

Java provides several built-in packages:

Package	Description
`java.lang`	Default package (String, Math, System, etc.)
`java.util`	Collection classes (`ArrayList`, `HashMap`, etc.)
`java.io`	Input/output classes (`File`, `BufferedReader`, etc.)
`java.net`	Networking classes (`Socket`, `URL`, etc.)
`java.sql`	Database handling (`Connection`, `Statement`, etc.)

📝 Example 5: Using `java.util` Package

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        System.out.println(list);
    }
}
```

🔷 Output

```
[Java, Python]
```

✅ Why?

- `ArrayList` is in `java.util`, so we import it.
- Some packages (`java.lang`) are **imported by default**.

7️⃣ Package Naming Conventions

- **Use lowercase**: `mypackage`, `com.mycompany.project`.
- **Follow domain-style format**: `com.example.utils`, `org.company.app`.
- **Avoid Java reserved words**.

✅ Good Example

```
package com.companyname.projectname;
```

❌ Bad Example

```
package MyPackage;  // Uppercase not recommended
```

8️⃣ Summary

Feature	Description
Package	A way to organize Java classes and interfaces
Built-in Packages	Java's predefined packages (`java.util`, `java.io`, etc.)
User-Defined Packages	Custom packages created using the `package` keyword
Importing Packages	Use `import packageName.*;` to access classes
Access Modifiers	Control visibility of classes and members
Sub-Packages	Nested packages within a package (`package.subpackage`)

9️⃣ Key Takeaways

✅ **Packages help in organizing Java code effectively**.
✅ **Built-in Java packages** provide reusable functionality.
✅ **Classes in different packages require explicit import statements**.
✅ **Access modifiers control package visibility** (e.g., `public`, `protected`).
✅ **Follow naming conventions** (`com.company.project`).

🔥 Do you want more real-world examples or hands-on exercises on Java packages? 🚀😊

🔷 Packages in Java

A **package** in Java is a way to **organize classes, interfaces, and sub-packages** into a structured manner. It helps avoid name conflicts and improves code maintainability.

✅ Why Use Packages?

- **Avoids name conflicts** between classes.
- **Provides access control** (public, private, protected).
- **Easier code organization** (group related classes).
- **Enhances modularity and reusability**.

1️⃣ Types of Packages in Java

Java provides two types of packages:

Type	Description
Built-in packages	Predefined Java packages (e.g., `java.util`, `java.io`)
User-defined packages	Custom packages created by developers

2️⃣ Creating a Package

To create a package, use the **`package`** keyword at the beginning of a Java file.

Example: Creating a User-Defined Package

📌 Steps to Create and Use a Package

1. Declare the package at the top of the Java file.
2. Save the file inside a folder matching the package name.
3. Compile using `javac -d . FileName.java`.
4. Use the package in another class with `import packageName.*;`.

📝 Example 1: Creating a Package

Step 1: Create a package named `mypackage`

```
// File: MyClass.java (inside 'mypackage' folder)
package mypackage;  // Package declaration

public class MyClass {
    public void displayMessage() {
        System.out.println("Hello from MyClass in mypackage!");
```

```
    }
}
```

✅ **Save this file inside a folder named `mypackage`.**

📝 Example 2: Using the Package in Another Class

Step 2: Import and Use the `mypackage`

```
// File: Main.java (outside 'mypackage' folder)
import mypackage.MyClass;  // Import the class

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Using the package class
        obj.displayMessage();
    }
}
```

🔷 Compilation & Execution

```
javac -d . MyClass.java    # Compiles and stores class in 'mypackage' folder
javac Main.java            # Compile Main.java
java Main                  # Run the program
```

🔷 Output

```
Hello from MyClass in mypackage!
```

✅ **Why?**

- `MyClass` is inside `mypackage`, so we import it in `Main.java`.
- The **`-d .`** flag ensures the compiled `.class` file is placed in the correct package folder.

3️⃣ Importing a Package

You can import a package in three ways:

Import Statement	Description
`import packageName.ClassName;`	Imports a specific class
`import packageName.*;`	Imports all classes from the package
`import static packageName.ClassName.methodName;`	Imports a static method

📝 Example 3: Importing All Classes

```
import mypackage.*;  // Import all classes from mypackage

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

```
        MyClass obj = new MyClass();  // No need to specify package
        obj.displayMessage();
    }
}
```

✅ Why?

- `import mypackage.*;` allows access to **all classes** in `mypackage`.
- No need to import each class separately.

---

4 Access Modifiers & Packages

Modifier	Same Class	Same Package	Different Package (Subclass)	Different Package (Non-Subclass)
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
No modifier (default)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

🔹 **Example:** A `protected` method can be accessed **within the same package** or in **subclasses of different packages**.

---

5 Sub-Packages

A sub-package is a package inside another package.

📝 Example 4: Creating a Sub-Package

```
// File: mypackage/subpackage/SubClass.java
package mypackage.subpackage;  // Sub-package declaration

public class SubClass {
    public void show() {
        System.out.println("Inside SubClass of subpackage");
    }
}
```

Using the Sub-Package

```
import mypackage.subpackage.SubClass;  // Import sub-package class

public class Main {
    public static void main(String[] args) {
        SubClass obj = new SubClass();
        obj.show();
```

```
    }
}
```

🔷 Compilation & Execution

```
javac -d . mypackage/subpackage/SubClass.java
javac -d . Main.java
java Main
```

🔷 Output

```
Inside SubClass of subpackage
```

✅ Why?

- The `.` **(dot)** represents a sub-package (`mypackage.subpackage`).
- Import **must match** the full package structure.

---

6 Java Built-in Packages

Java provides several built-in packages:

Package	Description
`java.lang`	Default package (String, Math, System, etc.)
`java.util`	Collection classes (`ArrayList`, `HashMap`, etc.)
`java.io`	Input/output classes (`File`, `BufferedReader`, etc.)
`java.net`	Networking classes (`Socket`, `URL`, etc.)
`java.sql`	Database handling (`Connection`, `Statement`, etc.)

📝 Example 5: Using `java.util` Package

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        System.out.println(list);
    }
}
```

🔷 Output

```
[Java, Python]
```

✅ Why?

- `ArrayList` is in `java.util`, so we import it.
- Some packages (`java.lang`) are **imported by default**.

7️⃣ Package Naming Conventions

- **Use lowercase**: `mypackage`, `com.mycompany.project`.
- **Follow domain-style format**: `com.example.utils`, `org.company.app`.
- **Avoid Java reserved words**.

✅ Good Example

```
package com.companyname.projectname;
```

❌ Bad Example

```
package MyPackage;  // Uppercase not recommended
```

8️⃣ Summary

Feature	Description
Package	A way to organize Java classes and interfaces
Built-in Packages	Java's predefined packages (`java.util`, `java.io`, etc.)
User-Defined Packages	Custom packages created using the `package` keyword
Importing Packages	Use `import packageName.*;` to access classes
Access Modifiers	Control visibility of classes and members
Sub-Packages	Nested packages within a package (`package.subpackage`)

9️⃣ Key Takeaways

✅ **Packages help in organizing Java code effectively**.
✅ **Built-in Java packages** provide reusable functionality.
✅ **Classes in different packages require explicit import statements**.
✅ **Access modifiers control package visibility** (e.g., `public`, `protected`).
✅ **Follow naming conventions** (`com.company.project`).

🔥 Do you want more real-world examples or hands-on exercises on Java packages? 🚀😊

🔷 Defining Packages in Java

In Java, a **package** is a way to group related classes and interfaces together, similar to a folder structure in a file system. Packages help in organizing the code, avoiding name conflicts, and improving reusability.

1️⃣ How to Define a Package in Java?

To define a package, use the **`package`** keyword at the **very beginning** of a Java file.

📝 Syntax

```
package packageName;
```

📌 Steps to Define a Package:

1. Declare the package at the top of the Java file.
2. Save the file inside a folder matching the package name.
3. Compile using `javac -d . FileName.java`.
4. Import and use the package in another class.

2️⃣ Example: Creating a User-Defined Package

Step 1: Define a Package

```
// File: MyClass.java (inside 'mypackage' folder)
package mypackage;  // Package declaration

public class MyClass {
    public void displayMessage() {
        System.out.println("Hello from MyClass in mypackage!");
    }
}
```

✅ **Save this file inside a folder named `mypackage`.**

Step 2: Compile the Package

Use the **`-d .`** flag to compile the Java file into the correct package folder.

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

🔹 This command places the `.class` file inside the `mypackage` directory.

Step 3: Using the Package in Another Class

Now, create another file (`Main.java`) to **use the package**.

```
// File: Main.java (outside 'mypackage' folder)
import mypackage.MyClass;  // Import the class from the package

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Creating an object
        obj.displayMessage();  // Calling the method
    }
}
```

Step 4: Compile and Run

```
javac Main.java
java Main
```

🔷 Output

```
Hello from MyClass in mypackage!
```

✅ Why?

- `MyClass` is inside `mypackage`, so we **import** it in `Main.java`.
- The `-d .` flag ensures the compiled `.class` file is stored in the correct package structure.

3️⃣ Sub-Packages in Java

A **sub-package** is a package inside another package.

📝 Example: Creating a Sub-Package

```
// File: mypackage/subpackage/SubClass.java
package mypackage.subpackage;  // Sub-package declaration

public class SubClass {
    public void show() {
        System.out.println("Inside SubClass of subpackage");
    }
}
```

Using the Sub-Package

```
import mypackage.subpackage.SubClass;  // Import sub-package class

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

🔷 Compilation & Execution

```
javac -d . mypackage/subpackage/SubClass.java
javac -d . Main.java
```

```
java Main
```

🔷 Output

```
Inside SubClass of subpackage
```

✅ Why?

- The `.` **(dot)** represents a **sub-package** (`mypackage.subpackage`).
- Import **must match** the full package structure.

---

4 Access Modifiers & Packages

Modifier	Same Class	Same Package	Different Package (Subclass)	Different Package (Non-Subclass)
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
Default (no modifier)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

🔹 **Example:** A `protected` method can be accessed **within the same package** or in **subclasses of different packages**.

---

5 Importing Packages

You can import a package in three ways:

Import Statement	Description
`import packageName.ClassName;`	Imports a specific class
`import packageName.*;`	Imports all classes from the package
`import static packageName.ClassName.methodName;`	Imports a static method

📝 Example: Importing All Classes

```
import mypackage.*;  // Import all classes from mypackage

public class Main {
```

```
    public static void main(String[] args) {
        MyClass obj = new MyClass();  // No need to specify package
        obj.displayMessage();
    }
}
```

✅ Why?

- `import mypackage.*;` allows access to **all classes** in `mypackage`.
- No need to import each class separately.

---

6 Java Built-in Packages

Java provides several built-in packages:

Package	Description
`java.lang`	Default package (String, Math, System, etc.)
`java.util`	Collection classes (`ArrayList`, `HashMap`, etc.)
`java.io`	Input/output classes (`File`, `BufferedReader`, etc.)
`java.net`	Networking classes (`Socket`, `URL`, etc.)
`java.sql`	Database handling (`Connection`, `Statement`, etc.)

📝 Example: Using `java.util` Package

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        System.out.println(list);
    }
}
```

🔷 Output

```
[Java, Python]
```

✅ Why?

- `ArrayList` is in `java.util`, so we import it.
- Some packages (`java.lang`) are **imported by default**.

7️⃣ Package Naming Conventions

- **Use lowercase**: `mypackage`, `com.mycompany.project`.
- **Follow domain-style format**: `com.example.utils`, `org.company.app`.
- **Avoid Java reserved words**.

✅ Good Example

```
package com.companyname.projectname;
```

❌ Bad Example

```
package MyPackage;  // Uppercase not recommended
```

8️⃣ Summary

Feature	Description
Package	A way to organize Java classes and interfaces
Built-in Packages	Java's predefined packages (`java.util`, `java.io`, etc.)
User-Defined Packages	Custom packages created using the `package` keyword
Importing Packages	Use `import packageName.*;` to access classes
Access Modifiers	Control visibility of classes and members
Sub-Packages	Nested packages within a package (`package.subpackage`)

9️⃣ Key Takeaways

✅ **Packages help in organizing Java code effectively**.
✅ **Built-in Java packages** provide reusable functionality.
✅ **Classes in different packages require explicit import statements**.
✅ **Access modifiers control package visibility** (e.g., `public`, `protected`).
✅ **Follow naming conventions** (`com.company.project`).

🔥 Do you need more hands-on exercises or real-world examples on Java packages? 🚀😊

📦 Creating and Accessing a Package in Java

A **package** in Java is a way to group related classes and interfaces together. It helps in organizing the code, avoiding class name conflicts, and improving reusability.

1️⃣ Creating a Package in Java

To create a package, use the **`package`** keyword at the **top of the Java file**.

📝 Steps to Create a Package

1. Define the package name using the `package` keyword.
2. Save the file inside a directory with the same name as the package.
3. Compile the Java file using `javac -d . FileName.java`.
4. The compiled `.class` file will be placed in the specified package folder.

📝 Example: Creating a Package

Let's create a package **`mypackage`** containing a class **`MyClass`**.

```
// File: MyClass.java (inside 'mypackage' folder)
package mypackage;  // Defining a package

public class MyClass {
    public void showMessage() {
        System.out.println("Hello from MyClass in mypackage!");
    }
}
```

✅ **Save the file inside a folder named `mypackage`.**

🛠️ Compiling the Package

To compile `MyClass.java` and store the `.class` file in the `mypackage` directory, use:

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

✅ The `-d .` flag tells Java to place the compiled `.class` file inside the correct package folder.

2️⃣ Accessing a Package in Java

After creating a package, we can **access it from another class** by using the `import` statement.

📝 Example: Using the Package in Another Class

Now, create a new file **outside** the `mypackage` folder (e.g., `Main.java`) and import `MyClass`.

```
// File: Main.java (outside 'mypackage' folder)
import mypackage.MyClass;  // Import the class from the package

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();  // Creating an object of MyClass
        obj.showMessage();  // Calling the method
    }
}
```

🛠️ Compiling and Running

```
javac Main.java  # Compile the Main class
java Main        # Run the program
```

🔷 Output

```
Hello from MyClass in mypackage!
```

✅ Why?

- `MyClass` is inside `mypackage`, so we **import** it in `Main.java`.
- The `import` statement allows `Main.java` to use `MyClass` without needing the full package name every time.

3️⃣ Accessing a Package Without Import

Instead of using `import`, we can access the package using the **fully qualified class name**:

```
// File: Main.java
public class Main {
    public static void main(String[] args) {
        mypackage.MyClass obj = new mypackage.MyClass();  // Fully qualified
name
        obj.showMessage();
    }
}
```

✅ No need to import the package, but it makes the code longer.

4️⃣ Creating a Sub-Package

A **sub-package** is a package inside another package.

📝 Example: Creating a Sub-Package

```
// File: mypackage/subpackage/SubClass.java
package mypackage.subpackage;  // Defining a sub-package

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

📝 Example: Using the Sub-Package

```
import mypackage.subpackage.SubClass;  // Import the sub-package class

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

✅ Compile and run as before.

```
javac -d . mypackage/subpackage/SubClass.java
javac -d . Main.java
java Main
```

🔷 Output

```
Inside SubClass of subpackage
```

5️⃣ Summary

Feature	Description
Creating a Package	Use `package packageName;` at the top of the file
Compiling a Package	Use `javac -d . FileName.java`
Accessing a Package	Use `import packageName.ClassName;`
Sub-Packages	Defined as `package mainpackage.subpackage;`

🔷 Understanding `CLASSPATH` in Java

What is `CLASSPATH`?

In Java, the **CLASSPATH** is an environment variable that tells the Java compiler (`javac`) and Java runtime (`java`) where to look for **.class** files and **JAR (Java Archive) files** when running a program.

If a required class is not found in the CLASSPATH, you may get errors like:

```
Error: Could not find or load main class Main
```

1️⃣ Default `CLASSPATH`

By default, Java searches for classes in: 1️⃣ The **current directory (`.`)**.
2️⃣ The **JDK's standard library (`rt.jar`)**.
3️⃣ Any paths explicitly specified in the `CLASSPATH` variable.

✔️ **If `CLASSPATH` is not set, Java assumes `.` (current directory) as the default path.**

2️⃣ Setting `CLASSPATH` Temporarily

You can specify the CLASSPATH while running a Java program using the `-cp` or `-classpath` option.

```
java -cp path/to/classes Main
```

or

```
java -classpath path/to/classes Main
```

🔹 Example: If `MyClass.class` is inside the `mypackage` folder, run:

```
java -cp mypackage mypackage.MyClass
```

3️⃣ Setting `CLASSPATH` Permanently

🖥️ On Windows

Set the CLASSPATH permanently using **Environment Variables**:

1. Open **Command Prompt** (`cmd`).
2. Check the current CLASSPATH:
3. `echo %CLASSPATH%`
4. Set the CLASSPATH (example for `C:\java\classes`):

5. `set CLASSPATH=C:\java\classes;.`
6. To make it permanent:
    - Go to **System Properties → Advanced → Environment Variables**.
    - Add a new variable `CLASSPATH` and set its value.

💻 On macOS/Linux

1. Open **Terminal**.
2. Check the current CLASSPATH:
3. `echo $CLASSPATH`
4. Set the CLASSPATH temporarily:
5. `export CLASSPATH=/home/user/java/classes:.`
6. To make it permanent, add this line to **`~/.bashrc`** or **`~/.bash_profile`**:
7. `export CLASSPATH=/home/user/java/classes:.`

✅ **Use `:` as a separator in macOS/Linux and `;` in Windows.**

4️⃣ Using `CLASSPATH` with JAR Files

If you have a `.jar` file (e.g., `mylibrary.jar`), add it to the CLASSPATH:

```
java -cp mylibrary.jar MainClass
```

For multiple JARs:

```
java -cp "lib1.jar;lib2.jar;." MainClass  # Windows
java -cp "lib1.jar:lib2.jar:." MainClass  # macOS/Linux
```

5️⃣ Summary

Concept	Description
Default CLASSPATH	Current directory (`.`)
Setting CLASSPATH Temporarily	Use `java -cp path/to/classes Main`
Setting CLASSPATH Permanently	Use system environment variables (`CLASSPATH`)
Multiple Paths	Separate paths with `;` (Windows) or `:` (Linux/macOS)
Using JARs in CLASSPATH	`java -cp "mylib.jar;." MainClass`

📦 Importing Packages in Java

In Java, **importing packages** allows you to use classes and interfaces from another package in your program. This improves code organization and reusability.

1️⃣ Ways to Import a Package

There are three ways to import a package in Java:

Method	Syntax	Usage
Import a Specific Class	`import packageName.ClassName;`	Imports only the specified class
Import All Classes in a Package	`import packageName.*;`	Imports all classes in the package
No Import (Fully Qualified Name)	`packageName.ClassName obj = new packageName.ClassName();`	Directly use the full class name

---

2️⃣ Example: Importing a User-Defined Package

Let's create and import a package **`mypackage`**.

Step 1: Create the Package

```
// File: MyClass.java (inside 'mypackage' folder)
package mypackage;  // Define the package

public class MyClass {
    public void showMessage() {
        System.out.println("Hello from MyClass in mypackage!");
    }
}
```

✅ **Save this file inside a folder named `mypackage`.**

Step 2: Compile the Package

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

🔹 This places `MyClass.class` inside the `mypackage` directory.

Step 3: Import and Use the Package

Now, create another file (`Main.java`) to **import the package**.

Method 1: Import a Specific Class

```
// File: Main.java
import mypackage.MyClass;  // Import the specific class

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();  // Create an object
        obj.showMessage();  // Call the method
    }
}
```

Method 2: Import All Classes in a Package

```
// File: Main.java
import mypackage.*;  // Import all classes from mypackage

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();  // Works without specifying the class
name
        obj.showMessage();
    }
}
```

Method 3: No Import (Fully Qualified Name)

```
// File: Main.java
public class Main {
    public static void main(String[] args) {
        mypackage.MyClass obj = new mypackage.MyClass();  // Use full package
name
        obj.showMessage();
    }
}
```

✅ **No need for `import`, but the code is longer.**

Step 4: Compile and Run

```
javac Main.java  # Compile the Main class
java Main        # Run the program
```

Output

```
Hello from MyClass in mypackage!
```

---

3 Importing Built-in Java Packages

Java has many built-in packages (e.g., `java.util`, `java.io`, `java.net`).

Example: Importing `java.util.ArrayList`

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        System.out.println(list);
    }
}
```

🔷 Output

```
[Java, Python]
```

✅ Why?

- `ArrayList` is inside `java.util`, so we import it.
- Some packages like `java.lang` (String, Math, System) are **imported by default**.

4️⃣ Summary

Feature	Description
Import Specific Class	`import packageName.ClassName;`
Import All Classes	`import packageName.*;`
Use Without Import	`packageName.ClassName obj = new packageName.ClassName();`
Built-in Packages	`java.util`, `java.io`, `java.net`, etc.

🔷 Exception Handling in Java

1️⃣ What is Exception Handling?

Exception handling in Java is a mechanism to handle **runtime errors** (exceptions) to ensure smooth execution of programs without unexpected crashes.

✅ Key Benefits:

- Prevents program crashes
- Helps in debugging
- Provides meaningful error messages
- Ensures the program continues running even after an error

---

2 Types of Exceptions in Java

◈ Checked Exceptions (Compile-Time Exceptions)

These exceptions are checked at **compile-time** and must be handled using `try-catch` or declared with `throws`.
Example:

- `IOException` (file not found)
- `SQLException` (database error)
- `InterruptedException`

```
import java.io.*;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("nonexistent.txt"); // May throw
FileNotFoundException
            BufferedReader br = new BufferedReader(file);
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

◈ Unchecked Exceptions (Runtime Exceptions)

These exceptions occur at **runtime** and do not require explicit handling.
Example:

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        System.out.println(arr[5]); // Throws ArrayIndexOutOfBoundsException
    }
}
```

◈ Errors (System Failures)

Errors are **not exceptions** and indicate serious issues like memory overflow or JVM crashes.
Example:

- `StackOverflowError`
- `OutOfMemoryError`

```
public class StackOverflowExample {
    public static void infiniteRecursion() {
        infiniteRecursion(); // Causes StackOverflowError
    }

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

---

3 Exception Handling Mechanisms

1. `try-catch` Block

Used to handle exceptions gracefully.

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

2. `try-catch-finally` Block

The `finally` block always **executes** (even if an exception occurs).

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]); // Throws exception
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Index out of bounds!");
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

3. `throws` Keyword

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

```
import java.io.*;

public class ThrowsExample {
    static void readFile() throws IOException {
        FileReader file = new FileReader("file.txt"); // May throw
IOException
    }

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

```
            readFile();
        } catch (IOException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

4. `throw` Keyword

Used to manually throw an exception.

```
public class ThrowExample {
    static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Not eligible to vote.");
        }
    }

    public static void main(String[] args) {
        validateAge(16); // Throws exception
    }
}
```

---

4 Creating Custom Exceptions

You can create your own exceptions by extending the `Exception` class.

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

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

    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (AgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

---

5 Summary Table

Concept	Usage
`try-catch`	Handles exceptions

Concept	Usage
`finally`	Always executes (cleanup code)
`throws`	Declares an exception in a method
`throw`	Manually throws an exception
`Custom Exception`	Extends `Exception` class

🔷 Benefits of Exception Handling in Java

Exception handling in Java helps make programs more **robust, maintainable, and user-friendly** by managing runtime errors effectively. Here are the key benefits:

1️⃣ Prevents Program Crashes 🚀

- Without exception handling, a program **stops executing** when an error occurs.
- `try-catch` ensures that the program continues running even after encountering an exception.

✅ Example:

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Causes ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
        System.out.println("Program continues...");
    }
}
```

🔹 Output:

```
Cannot divide by zero!
Program continues...
```

Even though an error occurred, the program **did not crash**. 🎉

2️⃣ Improves Code Maintainability 📌

- Exception handling **separates error-handling logic** from the main program logic.
- Makes the code **cleaner and easier to debug**.

✅ **Example:** Without exception handling:

```
public class NoExceptionHandling {
    public static void main(String[] args) {
        int arr[] = {1, 2, 3};
        System.out.println(arr[5]); // Program crashes
    }
}
```

✅ **With exception handling:**

```
public class WithExceptionHandling {
    public static void main(String[] args) {
        try {
            int arr[] = {1, 2, 3};
            System.out.println(arr[5]); // Out of bounds
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Index is out of range!");
        }
    }
}
```

3️⃣ Helps in Debugging 🐞

- The **stack trace** helps developers identify the exact **line number and method** where an error occurred.
- The `getMessage()` and `printStackTrace()` methods provide detailed information.

✅ **Example:**

```
public class DebugExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length()); // Causes NullPointerException
        } catch (NullPointerException e) {
            e.printStackTrace(); // Prints detailed error info
        }
    }
}
```

🔹 **Output:**

```
java.lang.NullPointerException
    at DebugExample.main(DebugExample.java:6)
```

🔹 Shows that the error **occurred at line 6** in `DebugExample.java`.

4️⃣ Ensures Proper Resource Management ✅

- The `finally` block **closes resources** (e.g., files, database connections) even if an exception occurs.
- Prevents **memory leaks**.

✅ Example: Handling File Resources

```
import java.io.*;

public class FileHandlingExample {
    public static void main(String[] args) {
        FileReader file = null;
        try {
            file = new FileReader("data.txt");
            BufferedReader br = new BufferedReader(file);
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        } finally {
            try {
                if (file != null) file.close(); // Closes file safely
            } catch (IOException e) {
                System.out.println("Error closing file!");
            }
        }
    }
}
```

5️⃣ Allows Custom Exception Handling 🎯

- You can create **custom exceptions** for specific business logic errors.
- Helps make error messages more meaningful.

✅ Example: Custom Exception for Age Validation

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

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

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

```
        }
    }
}
```

🔷 Output:

```
Exception: Age must be 18 or above.
```

---

6 Enables Exception Propagation 🔄

- Using `throws`, a method can pass the exception to its caller **instead of handling it immediately**.
- Helps **avoid redundant code**.

✅ Example:

```
import java.io.*;

public class ThrowsExample {
    static void readFile() throws IOException {
        FileReader file = new FileReader("file.txt"); // Exception thrown
    }

    public static void main(String[] args) {
        try {
            readFile(); // Exception propagated here
        } catch (IOException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

---

7 Supports Multiple Exception Handling 🎭

- Java allows handling **multiple exceptions** in a single `catch` block.
- Avoids writing redundant `catch` blocks.

✅ Example: Handling Multiple Exceptions

```
public class MultipleExceptionsExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // ArithmeticException
            int arr[] = {1, 2, 3};
            System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
        } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e);
        }
    }
}
```

8️⃣ Improves User Experience 🏆

- Instead of displaying **technical errors**, Java allows displaying **user-friendly messages**.
- This enhances the **end-user experience**.

✅ Example:

```
public class UserFriendlyExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // Exception occurs
        } catch (ArithmeticException e) {
            System.out.println("Oops! Something went wrong. Please try
again.");
        }
    }
}
```

🔷 Output:

```
Oops! Something went wrong. Please try again.
```

Instead of showing `java.lang.ArithmeticException: / by zero`, the program **displays a friendly message**. 🎉

🔷 Summary Table: Benefits of Exception Handling

Benefit	Description
Prevents Crashes	Program continues even after an error.
Improves Maintainability	Error-handling code is separate from business logic.
Helps Debugging	`printStackTrace()` helps locate errors.
Ensures Resource Management	`finally` ensures resources are closed properly.
Allows Custom Errors	Create your own exceptions for better readability.
Enables Exception Propagation	Methods can pass exceptions to their caller.
Supports Multiple Exceptions	Handle different exceptions in a single `catch`.
Enhances User Experience	Displays user-friendly error messages.

✨ Conclusion

Exception handling is a **powerful feature in Java** that makes applications **robust, user-friendly, and error-free**. 🚀 It **reduces program crashes, helps debugging, and ensures proper resource management**.

🔷 Classification of Exceptions in Java

In Java, exceptions are classified into two main categories:
1️⃣ **Checked Exceptions** (Compile-time)
2️⃣ **Unchecked Exceptions** (Runtime)

Additionally, Java has **Errors**, which are serious problems that cannot be handled by normal exception handling.

1️⃣ Checked Exceptions (Compile-Time Exceptions)

✅ Definition:

- These exceptions are checked **at compile-time** by the Java compiler.
- If not handled using `try-catch` or `throws`, the program **won't compile**.
- These usually occur due to **external resources** like files, databases, or network connections.

✅ Examples of Checked Exceptions:

- `IOException` (Input/Output failure)
- `SQLException` (Database errors)
- `FileNotFoundException` (File not found)
- `InterruptedException` (Thread interruption)
- `ClassNotFoundException` (Class not found at runtime)

✅ Example: Handling Checked Exception

```
import java.io.*;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("data.txt"); // May throw
FileNotFoundException
            BufferedReader br = new BufferedReader(file);
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

🔷 **Without handling**, this code would give a **compilation error**.

2️⃣ Unchecked Exceptions (Runtime Exceptions)

✅ Definition:

- These exceptions occur **at runtime** and are **not checked at compile-time**.
- Caused by logical programming errors (e.g., dividing by zero, accessing an invalid index).
- The program **compiles successfully**, but **fails at runtime**.

✅ Examples of Unchecked Exceptions:

- `NullPointerException` (Accessing an object that is `null`)
- `ArrayIndexOutOfBoundsException` (Accessing an invalid index)
- `ArithmeticException` (Division by zero)
- `ClassCastException` (Invalid type casting)
- `IllegalArgumentException` (Invalid arguments passed)

✅ Example: Handling Unchecked Exception

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // Division by zero causes ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

🔷 **Without handling, this code would crash at runtime.**

3️⃣ Errors (System Failures)

✅ Definition:

- Errors are **not exceptions**; they represent **serious system failures**.
- Caused by **JVM issues** such as memory overflow, infinite recursion, or hardware failures.
- Cannot be recovered using `try-catch`.

✅ Examples of Errors:

- `StackOverflowError` (Infinite recursion)
- `OutOfMemoryError` (JVM runs out of memory)
- `VirtualMachineError` (JVM internal error)

- `AssertionError` (Failed assertion in debugging)

✅ Example: StackOverflowError (Infinite Recursion)

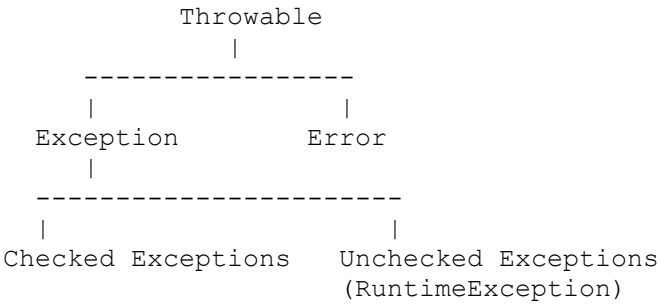
```
public class StackOverflowExample {
    public static void recursiveMethod() {
        recursiveMethod(); // Infinite recursion causes StackOverflowError
    }

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

🔷 This will cause a `StackOverflowError`, crashing the program.

🔷 Exception Hierarchy in Java

Java exceptions follow a hierarchy under the `Throwable` class.



✅ Key Points:

1. `Throwable` is the root class for both **Exceptions** and **Errors**.
2. `Exception` contains both **Checked** and **Unchecked** exceptions.
3. `Error` represents **serious system issues** that should not be caught.

🔷 Summary Table: Classification of Exceptions

Type	Description	Examples	Checked at Compile-time?
Checked Exception	External resource failures	`IOException`, `SQLException`, `FileNotFoundException`	✅ Yes

Type	Description	Examples	Checked at Compile-time?
Unchecked Exception	Logical programming errors	`NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`	❌ No
Errors	System-level failures	`StackOverflowError`, `OutOfMemoryError`	❌ No

✨ Conclusion

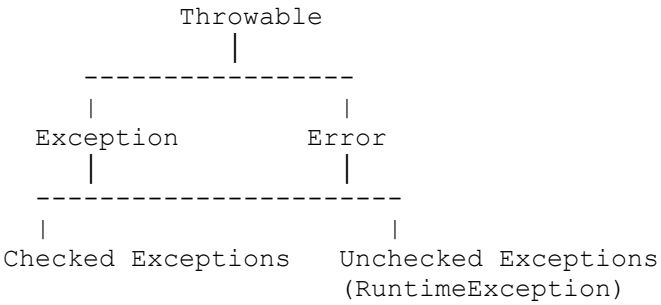
✅ **Checked Exceptions** must be handled explicitly using `try-catch` or `throws`.
✅ **Unchecked Exceptions** are programming mistakes and should be **avoided through proper logic**.
✅ **Errors** indicate **critical failures** that cannot be recovered.

Would you like more real-world examples or practice questions? 😊

🔷 Exception Hierarchy in Java

In Java, **all exceptions and errors are subclasses of the `Throwable` class**. The `Throwable` class has two main subclasses:
1️⃣ **Exception** – Represents recoverable errors (programming issues).
2️⃣ **Error** – Represents critical failures that cannot be recovered.

🔷 Exception Hierarchy Diagram



1️⃣ `Throwable` (Root Class)

- The **superclass** for all exceptions and errors.
- Contains important methods like:

- `printStackTrace()`
- `getMessage()`
- `toString()`

✅ Example: Using `printStackTrace()`

```
public class ThrowableExample {
    public static void main(String[] args) {
        try {
            int num = 5 / 0; // ArithmeticException
        } catch (Throwable t) {
            t.printStackTrace(); // Prints full error details
        }
    }
}
```

---

2 `Exception` (Recoverable Errors)

- Represents **errors that can be handled** in the program.
- Further divided into:
  - **Checked Exceptions** (Must be handled)
  - **Unchecked Exceptions** (Occur at runtime)

---

3 Checked Exceptions (Compile-Time Exceptions)

✅ Definition:

- Checked at **compile-time** by the Java compiler.
- Must be handled using `try-catch` or `throws`.

✅ Examples:

- `IOException` – Issues with file handling.
- `SQLException` – Database errors.
- `FileNotFoundException` – File does not exist.
- `InterruptedException` – Thread interruption.

✅ Example: Handling Checked Exception

```
import java.io.*;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("data.txt"); //
FileNotFoundException
        } catch (IOException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
```

```
}
```

🔹 Without handling, this will cause a **compilation error**.

4️⃣ Unchecked Exceptions (Runtime Exceptions)

✅ Definition:

- Occur at **runtime** due to programming mistakes.
- Not checked at compile-time.
- Can be avoided with better coding practices.

✅ Examples:

- `NullPointerException` – Accessing a null object.
- `ArrayIndexOutOfBoundsException` – Invalid array index.
- `ArithmeticException` – Division by zero.
- `IllegalArgumentException` – Invalid method arguments.
- `ClassCastException` – Invalid type casting.

✅ Example: Handling Unchecked Exception

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // Causes ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

🔹 Without handling, this will **crash at runtime**.

5️⃣ `Error` (Serious System Failures)

✅ Definition:

- Errors are **not exceptions**; they indicate **serious failures**.
- Cannot be handled using `try-catch`.
- Usually caused by **JVM or system-level issues**.

✅ Examples:

- `StackOverflowError` – Infinite recursion.
- `OutOfMemoryError` – JVM runs out of memory.

- `VirtualMachineError` – JVM internal error.
- `AssertionError` – Failed assertions in debugging.

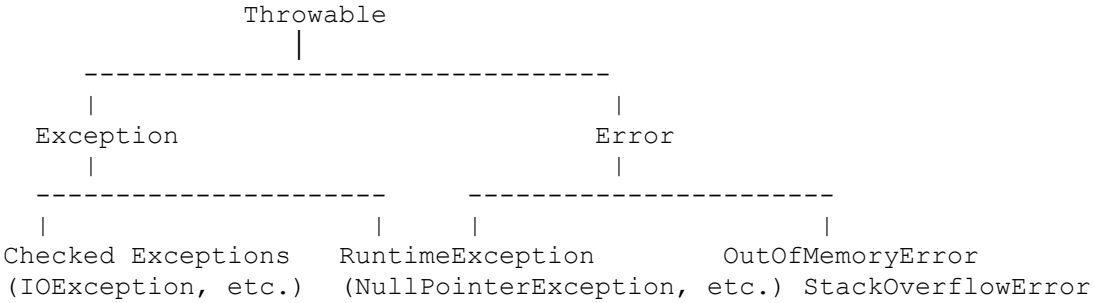
✅ Example: StackOverflowError (Infinite Recursion)

```
public class StackOverflowExample {
    public static void recursiveMethod() {
        recursiveMethod(); // Infinite recursion
    }

    public static void main(String[] args) {
        recursiveMethod(); // Causes StackOverflowError
    }
}
```

🔹 This **crashes the JVM**, as it **cannot be handled**.

🔷 Complete Exception Hierarchy Diagram



🔷 Summary Table: Exception Hierarchy

Type	Description	Examples	Checked at Compile-time?
Checked Exception	External issues, must be handled	`IOException`, `SQLException`, `FileNotFoundException`	✅ Yes
Unchecked Exception	Programming mistakes (runtime errors)	`NullPointerException`, `ArithmeticException`	❌ No
Error	Critical system failures, cannot be handled	`OutOfMemoryError`, `StackOverflowError`	❌ No

---

✨ Conclusion

1️⃣ **Checked exceptions** must be handled with `try-catch` or `throws`.
2️⃣ **Unchecked exceptions** are due to **poor logic** and should be prevented.
3️⃣ **Errors** indicate **serious problems** that **cannot be handled**.

Would you like a quiz or hands-on exercises to test your knowledge? 😊

Checked vs Unchecked Exceptions in Java

Java exceptions are divided into **Checked Exceptions** and **Unchecked Exceptions** based on when they are detected and how they are handled.

🔷 Checked Exceptions (Compile-Time Exceptions)

✅ Definition:

- Checked at **compile-time** by the Java compiler.
- If not handled using `try-catch` or `throws`, the program **won't compile**.
- Usually occur due to **external factors** (e.g., file handling, database access, network failures).

✅ Examples:

- `IOException` – Input/output operation failure.
- `SQLException` – Issues while working with databases.
- `FileNotFoundException` – File not found error.
- `InterruptedException` – Thread interruption.
- `ClassNotFoundException` – Class not found at runtime.

✅ Example: Handling a Checked Exception

```
import java.io.*;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("data.txt"); // May throw
FileNotFoundException
            BufferedReader br = new BufferedReader(file);
            System.out.println(br.readLine());
        } catch (IOException e) {  // Must handle the exception
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

🔹 Without handling, this would cause a **compilation error**.

◈ Unchecked Exceptions (Runtime Exceptions)

✅ Definition:

- **Not checked** at compile-time.
- Occur due to **programming errors** (e.g., logic mistakes, invalid operations).
- The program **compiles successfully** but **fails at runtime**.
- Usually caused by **incorrect code logic** rather than external resources.

✅ Examples:

- `NullPointerException` – Trying to access an object reference that is `null`.
- `ArrayIndexOutOfBoundsException` – Accessing an invalid array index.
- `ArithmeticException` – Division by zero.
- `IllegalArgumentException` – Passing an invalid argument to a method.
- `ClassCastException` – Invalid type conversion.

✅ Example: Handling an Unchecked Exception

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // Causes ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

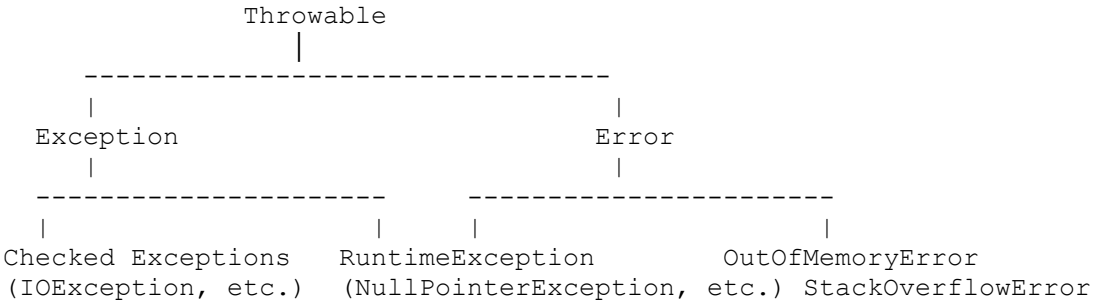
🔹 Without handling, this **crashes at runtime**.

◈ Key Differences Between Checked and Unchecked Exceptions

Feature	Checked Exceptions	Unchecked Exceptions
When detected?	At compile-time	At runtime
Compiler enforcement?	Must be handled (`try-catch` or `throws`)	No need to handle
Cause?	External issues (file, DB, network)	Programming errors

Feature	Checked Exceptions	Unchecked Exceptions
Examples	`IOException`, `SQLException`, `FileNotFoundException`	`NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`
Can be prevented?	No (depends on external factors)	Yes (better coding practices)

🔷 Exception Hierarchy in Java



🔷 Summary

1. **Checked exceptions** are **checked at compile-time** and must be handled.
2. **Unchecked exceptions** occur **at runtime** due to programming errors.
3. **Checked exceptions** handle **external resource failures**, while **unchecked exceptions** arise from **bad logic**.

Would you like a quiz or hands-on coding exercises to reinforce these concepts? 😊

Exception Handling in Java: `try`, `catch`, `throw`, `throws`, and `finally`

Java provides a robust exception-handling mechanism using the following keywords:

1. **`try`** – Defines a block of code where exceptions may occur.
2. **`catch`** – Handles exceptions thrown in the `try` block.
3. **`throw`** – Used to explicitly throw an exception.
4. **`throws`** – Declares exceptions that a method might throw.
5. **`finally`** – A block that always executes (used for cleanup).

🔷 1. `try` and `catch`

✅ **Usage:**

- The `try` block contains **code that may throw an exception**.
- The `catch` block **handles the exception** if one occurs.
- You can have **multiple `catch` blocks** to handle different exceptions.

✅ Example: Handling an Exception

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

🔹 If no exception occurs, the `catch` block is skipped.

✅ Multiple Catch Blocks

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException
        } catch (ArithmeticException e) {
            System.out.println("Math error: " + e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds!");
        }
    }
}
```

🔹 The **first matching `catch` block** is executed.

🔷 2. `throw` (Explicitly Throwing Exceptions)

✅ Usage:

- The `throw` keyword is used to **manually throw an exception** in Java.
- Used inside methods when we **want to create a custom exception condition**.

✅ Example: Throwing an Exception Manually

```
public class ThrowExample {
    static void checkAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or above.");
        }
        System.out.println("Access granted.");
    }
```

```
    public static void main(String[] args) {
        checkAge(16); // Throws IllegalArgumentException
    }
}
```

🔹 **Throws an exception if age is less than 18.**

🔹 3. `throws` (Declaring Exceptions)

✅ Usage:

- Used in **method signatures** to indicate that a method **may throw exceptions**.
- The caller of the method must handle the exception.

✅ Example: Declaring an Exception

```
import java.io.*;

public class ThrowsExample {
    static void readFile() throws IOException {
        FileReader file = new FileReader("data.txt"); // May throw
FileNotFoundException
        BufferedReader br = new BufferedReader(file);
        System.out.println(br.readLine());
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("File not found!");
        }
    }
}
```

🔹 The `readFile()` method **declares** that it may throw an `IOException`, and the caller (`main`) **handles it using `try-catch`.**

✅ Difference Between `throw` and `throws`

Keyword	Purpose	Usage
`throw`	**Manually throws an exception**	`throw new IllegalArgumentException("Error message");`
`throws`	**Declares exceptions a method may throw**	`void myMethod() throws IOException {}`

🔷 4. `finally` (Always Executes)

✅ Usage:

- The `finally` block is used for **cleanup code** that must execute **whether an exception occurs or not**.
- Commonly used to **close resources like files, databases, or network connections**.

✅ Example: Using `finally`

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

🔹 The `finally` block **executes regardless of whether an exception occurs**.

✅ Using `finally` to Close Resources

```
import java.io.*;

public class FinallyResourceExample {
    public static void main(String[] args) {
        FileReader file = null;
        try {
            file = new FileReader("data.txt");
            BufferedReader br = new BufferedReader(file);
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("File not found.");
        } finally {
            try {
                if (file != null) file.close(); // Closing the file in
finally block
            } catch (IOException e) {
                System.out.println("Error closing file.");
            }
        }
    }
}
```

🔹 The `finally` block ensures that **the file is closed** even if an exception occurs.

🔷 Summary Table

Keyword	Purpose	Usage
`try`	Defines a block where exceptions may occur	`try { int x = 10 / 0; }`
`catch`	Handles exceptions thrown in `try` block	`catch (ArithmeticException e) {}`
`throw`	Manually throws an exception	`throw new IllegalArgumentException();`
`throws`	Declares exceptions a method may throw	`void myMethod() throws IOException {}`
`finally`	Executes code after `try-catch`, even if an exception occurs	`finally { System.out.println("Cleanup done."); }`

✨ Conclusion

✅ `try-catch` **handles** exceptions.
✅ `throw` is used to **manually throw** exceptions.
✅ `throws` **declares** exceptions that a method might throw.
✅ `finally` **always executes** (used for cleanup).

Would you like a quiz or a hands-on coding exercise? 😊

🔷 Rethrowing Exceptions in Java

📌 What is Rethrowing an Exception?

Rethrowing an exception means **catching an exception and then throwing it again** to be handled by a higher-level method. This is useful when:
✔ You **want to log** an exception before passing it further.
✔ You **don't want to handle it immediately** but allow higher levels of the program to handle it.
✔ You **want to modify the exception** before rethrowing.

🔷 1. Basic Rethrowing (Same Exception)

✅ Example: Rethrowing Without Modifying

```
public class RethrowExample {
```

```
    static void riskyMethod() throws ArithmeticException {
        try {
            int num = 10 / 0; // Causes ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught in riskyMethod, rethrowing...");
            throw e; // Rethrow the same exception
        }
    }

    public static void main(String[] args) {
        try {
            riskyMethod();
        } catch (ArithmeticException e) {
            System.out.println("Caught in main: " + e);
        }
    }
}
```

🔷 Output:

```
Caught in riskyMethod, rethrowing...
Caught in main: java.lang.ArithmeticException: / by zero
```

🔷 The exception is **caught, logged, and then rethrown** to be handled by the `main()` method.

🔹 2. Rethrowing a New Exception (Wrapping Exception)

✅ Example: Wrapping and Rethrowing a Different Exception

```
public class RethrowNewException {
    static void riskyMethod() throws Exception {
        try {
            int num = 10 / 0;
        } catch (ArithmeticException e) {
            throw new Exception("Custom message: Error in riskyMethod", e);
// Wrapping original exception
        }
    }

    public static void main(String[] args) {
        try {
            riskyMethod();
        } catch (Exception e) {
            System.out.println("Caught in main: " + e.getMessage());
            System.out.println("Original cause: " + e.getCause()); //
Retrieves original exception
        }
    }
}
```

🔷 Output:

```
Caught in main: Custom message: Error in riskyMethod
```

```
Original cause: java.lang.ArithmeticException: / by zero
```

🔹 **Why Wrap an Exception?**
✔ Adds a **custom message**.
✔ Preserves **original exception details** using `getCause()`.

🔷 3. Rethrowing with `throws` (Checked Exceptions)

✅ Example: Rethrowing a Checked Exception

```
import java.io.*;

public class RethrowCheckedException {
    static void readFile() throws IOException {
        try {
            throw new IOException("File error!"); // Simulating file error
        } catch (IOException e) {
            System.out.println("Logging error: " + e.getMessage());
            throw e; // Rethrowing checked exception
        }
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Handled in main: " + e.getMessage());
        }
    }
}
```

🔹 **Output:**

```
Logging error: File error!
Handled in main: File error!
```

🔹 **Key Points:**
✔ The exception is **logged** before being passed to `main()`.
✔ The `throws` declaration ensures the exception is properly handled.

🔷 4. Using `throw e;` vs `throw new Exception(e);`

Approach	Behavior
`throw e;`	Preserves original stack trace (recommended)
`throw new Exception(e);`	Wraps the original exception (useful for adding more details)

✨ Conclusion

✔ **Rethrowing** allows an exception to be handled at a **higher level**.
✔ Use **`throw e;`** to preserve stack trace.
✔ Use **`throw new Exception(e);`** to wrap an exception with a custom message.
✔ **Checked exceptions** need to be declared using `throws` in method signatures.

Would you like to try a **coding exercise** on this? 😊

🔷 Exception Specification in Java (`throws` Keyword)

📌 What is Exception Specification?

Exception specification in Java refers to the **declaration of exceptions** that a method might throw. This is done using the **`throws`** keyword in the method signature.

It helps:
✔ Indicate potential exceptions a method may throw.
✔ Ensure **checked exceptions** are handled properly.
✔ Improve **code readability and maintainability**.

🔷 1. Using `throws` for Exception Specification

☑ Syntax:

```
returnType methodName(parameters) throws ExceptionType1, ExceptionType2 {
    // method code
}
```

☑ Example: Declaring an Exception in a Method

```
import java.io.*;

public class ExceptionSpecificationExample {
    static void readFile() throws IOException { // Declaring IOException
        FileReader file = new FileReader("data.txt"); // May throw
FileNotFoundException
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Handled IOException: " + e.getMessage());
        }
    }
```

```
}
```

🔹 **Key Points:**

✔ `readFile()` **declares** that it may throw an `IOException`.

✔ `main()` **handles** it using `try-catch`.

🔷 2. Multiple Exceptions in `throws`

A method can specify **multiple exceptions** separated by commas.

✅ Example: Declaring Multiple Exceptions

```
import java.io.*;

public class MultiExceptionExample {
    static void processFile() throws IOException, ArithmeticException {
        FileReader file = new FileReader("data.txt"); // May throw
IOException
        int result = 10 / 0; // Causes ArithmeticException
    }

    public static void main(String[] args) {
        try {
            processFile();
        } catch (IOException | ArithmeticException e) { // Handling both
exceptions
            System.out.println("Exception occurred: " + e);
        }
    }
}
```

🔹 **Java 7+ allows multiple exceptions in a single `catch` block (`|` operator).**

🔷 3. Exception Specification with Method Overriding

- In method **overriding**, the subclass method **must not throw broader exceptions** than the superclass method.
- The subclass can:
  ✓ Throw **fewer exceptions** or the **same** exception.
  ✗ Not introduce **new checked exceptions** that the superclass didn't declare.

✅ Example: Overriding with Checked Exceptions

```
import java.io.*;

class Parent {
    void show() throws IOException { // Declares IOException
        throw new IOException("File error!");
```

```
    }
}

class Child extends Parent {
    @Override
    void show() throws IOException { // Allowed (same exception)
        throw new IOException("File not found!");
    }
}

public class OverridingExceptionExample {
    public static void main(String[] args) {
        Parent obj = new Child();
        try {
            obj.show();
        } catch (IOException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

🔷 The overridden method **must not throw a broader exception** than its parent method.

✅ Invalid Case (Broader Exception)

```
class Parent {
    void show() throws IOException { }
}

class Child extends Parent {
    @Override
    void show() throws Exception { } // ❌ ERROR: Broader exception
}
```

🔷 The child method **cannot** throw a broader `Exception` than `IOException` in the parent method.

🔰 4. Checked vs. Unchecked Exceptions in `throws`

- **Checked exceptions** (e.g., `IOException`) **must** be declared in `throws` if not handled.
- **Unchecked exceptions** (e.g., `NullPointerException`, `ArithmeticException`) **don't** need to be declared in `throws`.

✅ Example: Unchecked Exception (No `throws` Needed)

```
public class UncheckedExample {
    static void divide() {
        int num = 10 / 0; // Throws ArithmeticException (Unchecked)
    }

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

```
            divide();
        } catch (ArithmeticException e) {
            System.out.println("Handled exception: " + e);
        }
    }
}
```

🔹 No `throws` needed because `ArithmeticException` is an **unchecked exception**.

🔷 Summary

Feature	Checked Exceptions	Unchecked Exceptions
Declaration in `throws`	Required	Not required
Examples	`IOException`, `SQLException`	`NullPointerException`, `ArithmeticException`
Must be handled?	Yes (either `try-catch` or `throws`)	No (optional)

✨ Conclusion

✔ `throws` **declares** exceptions that a method **might throw**.
✔ Multiple exceptions can be listed using **commas**.
✔ Overridden methods **cannot throw broader exceptions** than the parent method.
✔ **Unchecked exceptions** do not require `throws`.

Would you like a **quiz** or **coding exercise** on this? 😊

🔷 Built-in Exceptions in Java

📌 What Are Built-in Exceptions?

Java provides a set of **predefined (built-in) exceptions** that handle **common runtime errors** such as division by zero, accessing an invalid index, and file-related errors. These exceptions are part of the **Java Exception Hierarchy**.

🔷 Exception Hierarchy in Java

All exceptions in Java are subclasses of the `Throwable` class.

Exception Hierarchy

```
                Throwable
                /       \
           Exception     Error
          /         \
Checked Exceptions  Unchecked Exceptions (RuntimeException)
```

✓ **Checked Exceptions** – Must be handled (e.g., `IOException`).
✓ **Unchecked Exceptions** – Occur at runtime and can be avoided with proper coding (e.g., `NullPointerException`).
✓ **Errors** – Represent system-level issues (e.g., `OutOfMemoryError`).

🔷 1. Checked Exceptions (Compile-Time)

Checked exceptions **must** be handled using `try-catch` or declared using `throws`.

✅ Common Checked Exceptions

Exception	Cause
`IOException`	File handling errors
`SQLException`	Database errors
`InterruptedException`	Thread interruption
`FileNotFoundException`	File not found
`ClassNotFoundException`	Missing class at runtime

✅ Example: Handling `IOException`

```
import java.io.*;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("data.txt"); // May throw
FileNotFoundException
            BufferedReader br = new BufferedReader(file);
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("File error: " + e.getMessage());
        }
    }
}
```

🔷 **Explanation:**
✓ The program tries to **read a file** that may not exist.
✓ `IOException` is **handled using `try-catch`.**

◈ 2. Unchecked Exceptions (Runtime)

Unchecked exceptions **occur during execution** and do not require explicit handling.

✅ Common Unchecked Exceptions

Exception	Cause
`NullPointerException`	Accessing an object that is `null`
`ArithmeticException`	Division by zero
`ArrayIndexOutOfBoundsException`	Accessing an invalid array index
`NumberFormatException`	Invalid conversion (e.g., `"abc"` to `int`)
`IllegalArgumentException`	Invalid method argument

✅ Example: Handling `NullPointerException`

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            String text = null;
            System.out.println(text.length()); // Causes NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Null value encountered!");
        }
    }
}
```

🔷 **Explanation:**
✓ The `text` variable is **null**, so calling `.length()` **causes an exception**.
✓ The exception is **caught and handled**.

✅ Example: Handling `ArrayIndexOutOfBoundsException`

```
public class ArrayExceptionExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // Causes
ArrayIndexOutOfBoundsException
```

```
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Invalid array index!");
        }
    }
}
```

🔷 3. Errors (Serious Issues)

Errors occur **at the system level** and usually cannot be handled.

✅ Common Errors

Error	Cause
`OutOfMemoryError`	JVM runs out of memory
`StackOverflowError`	Infinite recursion
`VirtualMachineError`	JVM-related failures

✅ Example: `StackOverflowError` (Infinite Recursion)

```
public class StackOverflowExample {
    static void recursiveMethod() {
        recursiveMethod(); // Causes infinite recursion
    }

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

🔹 Explanation:

✓ This code causes **infinite recursion**, leading to a **StackOverflowError**.

🔷 Summary Table

Type	Exception Name	Example
Checked	`IOException`	File handling errors
Checked	`SQLException`	Database issues
Unchecked	`NullPointerException`	Calling method on `null` object
Unchecked	`ArithmeticException`	Division by zero

Type	Exception Name	Example
Unchecked	`ArrayIndexOutOfBoundsException`	Invalid array access
Error	`OutOfMemoryError`	Insufficient memory
Error	`StackOverflowError`	Infinite recursion

Conclusion

✓ Java has **built-in exceptions** categorized as **checked, unchecked, and errors**.
✓ Checked exceptions **must** be handled (`try-catch` or `throws`).
✓ Unchecked exceptions **occur at runtime** and can be avoided with good coding practices.
✓ Errors **cannot be recovered from** (e.g., `OutOfMemoryError`).

Would you like a **coding challenge** on this? 😊

Creating Custom Exception Subclasses in Java

Why Create Custom Exceptions?

Sometimes, built-in exceptions like `IOException` or `NullPointerException` may not be enough. Custom exceptions allow you to: ✔ **Provide meaningful error messages** relevant to your application.
✔ **Categorize errors** (e.g., `InsufficientFundsException` for banking apps).
✔ **Improve code readability and debugging**.

---

1. Creating a Custom Exception (Extending `Exception`)

To create a custom exception: 1️⃣ Extend the `Exception` class (for **checked** exceptions) or `RuntimeException` (for **unchecked** exceptions).
2️⃣ Define **constructors** to pass error messages.
3️⃣ Optionally override `toString()` or `getMessage()`.

✅ Example: Custom Checked Exception (`Exception` subclass)

```
// Step 1: Create a custom exception
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message); // Call parent constructor
    }
}
```

```
// Step 2: Use the custom exception
public class BankAccount {
    private double balance = 5000;

    void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient funds!
Balance: " + balance);
        }
        balance -= amount;
        System.out.println("Withdrawal successful! Remaining balance: " +
balance);
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        try {
            account.withdraw(6000); // Will throw custom exception
        } catch (InsufficientFundsException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

🔷 Output:

```
Exception: Insufficient funds! Balance: 5000.0
```

🔷 Explanation:

✓ `InsufficientFundsException` extends `Exception`, making it **checked**.

✓ The `withdraw()` method **throws** this exception when balance is low.

🔹 2. Creating an Unchecked Custom Exception (`RuntimeException` subclass)

☑️ Example: Custom Unchecked Exception

```
// Step 1: Create an unchecked exception
class InvalidAgeException extends RuntimeException {
    public InvalidAgeException(String message) {
        super(message);
    }
}

// Step 2: Use the custom exception
public class VotingSystem {
    static void checkEligibility(int age) {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above for
voting.");
        }
        System.out.println("You are eligible to vote!");
    }
```

```
    public static void main(String[] args) {
        checkEligibility(16); // Will throw exception
    }
}
```

🔷 Output:

```
Exception in thread "main" InvalidAgeException: Age must be 18 or above for
voting.
```

🔷 Explanation:

✓ `InvalidAgeException` extends `RuntimeException`, making it **unchecked**.
✓ No need to use `throws` in method signatures.

🔹 3. Overriding `toString()` or `getMessage()`

By overriding these methods, we can customize how the exception message appears.

✅ Example: Overriding `toString()`

```
class CustomException extends Exception {
    private int errorCode;

    public CustomException(String message, int errorCode) {
        super(message);
        this.errorCode = errorCode;
    }

    @Override
    public String toString() {
        return "Error Code " + errorCode + ": " + getMessage();
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new CustomException("File not found", 404);
        } catch (CustomException e) {
            System.out.println(e);
        }
    }
}
```

🔷 Output:

```
Error Code 404: File not found
```

🔷 Explanation:

✓ **Overriding `toString()`** helps provide a **structured error message**.

🔹 Summary

Type	Base Class	Requires `throws`?	Examples
Checked Exception	`Exception`	✅ Yes	`IOException`, `SQLException`, `CustomCheckedException`
Unchecked Exception	`RuntimeException`	❌ No	`NullPointerException`, `CustomUncheckedException`

✨ Conclusion

✓ Custom exceptions **improve error handling** by providing clear, meaningful messages.
✓ **Checked exceptions** (`extends Exception`) **must be declared** in method signatures.
✓ **Unchecked exceptions** (`extends RuntimeException`) do not require explicit handling.
✓ Overriding `toString()` and `getMessage()` **improves exception readability**.

Would you like a **coding challenge** on custom exceptions? 😊

UNIT – IV

🔷 Multithreading in Java

📌 What is Multithreading?

Multithreading is the process of **executing multiple threads simultaneously** within a program.
✓ It allows parallel execution, improving performance.
✓ Each thread runs **independently**, sharing the same resources.
✓ Java provides built-in support for multithreading via the **`Thread`** class and **`Runnable`** interface.

🔷 1. Creating Threads in Java

There are **two ways** to create threads:

🔷 Method 1: Extending the `Thread` Class

✅ Example: Creating a Thread by Extending `Thread`

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread: " + i);
            try {
                Thread.sleep(1000); // Pause for 1 second
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Create thread
        t1.start(); // Start thread execution
    }
}
```

🔹 Key Points:
✓ `MyThread` extends `Thread` and overrides the `run()` method.
✓ `start()` calls `run()` **asynchronously** in a new thread.
✓ `Thread.sleep(1000)` makes the thread **pause for 1 second**.

🔷 Method 2: Implementing the `Runnable` Interface

✅ Example: Creating a Thread by Implementing `Runnable`

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Runnable: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable obj = new MyRunnable();
        Thread t1 = new Thread(obj); // Pass object to Thread
        t1.start();
    }
}
```

🔹 Key Points:

✔ `MyRunnable` implements `Runnable` and overrides `run()`.
✔ `Thread` is created using `new Thread(obj)`.
✔ **Preferred approach** when extending another class.

🔷 2. Thread Lifecycle in Java

A thread goes through multiple **states**:

```
NEW → RUNNABLE → RUNNING → BLOCKED/WAITING → TERMINATED
```

✅ Example: Thread Lifecycle Demonstration

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

public class ThreadLifecycle {
    public static void main(String[] args) {
        LifecycleThread t1 = new LifecycleThread();
        System.out.println("Thread State: " + t1.getState()); // NEW
        t1.start();
        System.out.println("Thread State after start(): " + t1.getState());
// RUNNABLE
    }
```

```
}
```

🔷 `getState()` method shows the **current state** of the thread.

🔹 3. Thread Methods

Java provides several methods to **control thread behavior**:

Method	Description
`start()`	Starts a new thread
`run()`	Contains the task to be executed
`sleep(ms)`	Pauses thread execution for given time
`join()`	Waits for a thread to finish before proceeding
`isAlive()`	Checks if a thread is still running

✅ Example: `join()` Method (Waiting for a Thread to Finish)

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Running thread " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class JoinExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        try {
            t1.join(); // Main thread waits for t1 to finish
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        t2.start();
    }
}
```

🔹 **Key Points:**
✓ `join()` makes the **main thread wait** for `t1` to finish before running `t2`.

🔷 4. Thread Synchronization

When multiple threads access **shared resources**, synchronization ensures **data consistency**.

🔷 Without Synchronization (Race Condition)

```
class Counter {
    int count = 0;

    void increment() {
        count++; // Not synchronized
    }
}

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

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Final Count: " + counter.count);
    }
}
```

🔹 **Issue:** The output **may not always be 2000** due to **race conditions**.

🔷 With Synchronization (Thread-Safe)

✅ **Fixing the Issue Using `synchronized`**

```
class Counter {
    int count = 0;
```

```
    synchronized void increment() { // Only one thread can access at a time
        count++;
    }
}

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

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Final Count: " + counter.count); // Always 2000
    }
}
```

🔷 **Fix:** The `synchronized` keyword **prevents race conditions** by **allowing only one thread** to modify `count` at a time.

🔹 5. Inter-Thread Communication (wait() & notify())

Threads can communicate using **`wait()` and `notify()`.**

✅ Example: Producer-Consumer Problem

```
class Store {
    private int product = 0;

    synchronized void produce() throws InterruptedException {
        while (product > 0) wait(); // Wait if product is available
        product++;
        System.out.println("Produced: " + product);
        notify(); // Notify consumer
    }

    synchronized void consume() throws InterruptedException {
        while (product == 0) wait(); // Wait if no product
        System.out.println("Consumed: " + product);
```

```
            product--;
            notify(); // Notify producer
        }
    }

    public class ProducerConsumer {
        public static void main(String[] args) {
            Store store = new Store();

            Thread producer = new Thread(() -> {
                try { for (int i = 0; i < 5; i++) store.produce(); }
                catch (InterruptedException e) {}
            });

            Thread consumer = new Thread(() -> {
                try { for (int i = 0; i < 5; i++) store.consume(); }
                catch (InterruptedException e) {}
            });

            producer.start();
            consumer.start();
        }
    }
```

🔹 **Key Points:**
✓ `wait()` makes a thread **pause until notified**.
✓ `notify()` **resumes** waiting threads.

✨ Conclusion

✓ **Multithreading** improves efficiency by running multiple tasks in parallel.
✓ Threads can be created using **`Thread` class** or **`Runnable` interface**.
✓ **Synchronization (`synchronized`)** prevents race conditions.
✓ **Inter-thread communication (`wait()` & `notify()`)** allows threads to cooperate.

Would you like a **coding challenge** on this? 😊

🔷 Differences Between Multiple Processes and Multiple Threads in Java

In Java (and in general computing), **processes** and **threads** are used for executing tasks concurrently. However, they have key differences in **memory usage, execution, and communication.**

📌 Key Differences Between Processes and Threads

Feature	Multiple Processes	Multiple Threads
Definition	A process is an independent execution unit with its own memory space.	A thread is a smaller execution unit within a process that shares memory with other threads.
Memory Usage	Each process has its **own separate memory (heap, stack, etc.).**	Threads **share memory** within a process.
Communication	Inter-process communication (IPC) is **complex** and requires mechanisms like sockets, pipes, or shared memory.	Threads **communicate easily** using shared variables.
Creation Cost	Creating a process is **expensive** because it requires separate memory allocation.	Creating a thread is **lightweight** and faster.
Context Switching	Switching between processes is **slower** due to independent memory.	Switching between threads is **faster** as they share memory.
Isolation	Processes are **isolated** and do not affect each other.	Threads can **interfere with each other**, causing data inconsistency (if not synchronized).
Concurrency	True parallel execution is possible (if run on multiple CPUs).	Threads provide concurrency but may not always run in parallel (depends on JVM & CPU).
Example	Running **two Java programs** separately.	Running **multiple tasks** in the same Java program using threads.

🔷 Example of Multiple Processes in Java

Each Java program runs as a **separate process**.

✅ Example: Running Multiple Java Processes

```
import java.io.IOException;

public class MultiProcessExample {
    public static void main(String[] args) {
        try {
            // Running another Java process (another program)
            Process process = Runtime.getRuntime().exec("java
AnotherProgram");
            System.out.println("Another Java process started.");
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

🔷 **Key Point:** Each `java` command starts a **new process** with its **own memory space**.

🔹 Example of Multiple Threads in Java

Threads within a process share memory, making communication easier.

✅ **Example: Creating Multiple Threads**

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

public class MultiThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t2.start();
    }
}
```

🔷 **Key Point:** Both threads share **the same memory** but execute **independently**.

🔹 When to Use Multiple Processes vs. Multiple Threads?

Use Case	Use Multiple Processes	Use Multiple Threads
Running multiple independent applications	✅ Yes	❌ No
Heavy computation tasks	✅ Yes (distributes across CPUs)	❌ No (may block main thread)
Tasks requiring shared memory	❌ No (difficult communication)	✅ Yes (threads share memory)
Web servers (handling multiple requests)	✅ Yes (each request as a process)	✅ Yes (threads for better performance)

Use Case	Use Multiple Processes	Use Multiple Threads
Game development (handling AI, physics, rendering, etc.)	❌ No	✅ Yes (parallel game mechanics)

✨ Conclusion

✓ **Use multiple processes** when you need **independent execution** with **separate memory spaces**.
✓ **Use multiple threads** when you need **lightweight execution** with **shared memory** and **fast communication**.
✓ **Threads are preferred for concurrency**, while **processes are used for isolation and true parallelism**.

Would you like an example comparing **multi-threading vs. multi-processing performance**? 🚀

🔷 Thread States in Java

In Java, a thread can be in one of several **states** during its lifecycle. The **`Thread.State`** enum defines six possible thread states.

📌 Java Thread States

A thread goes through **six different states**:

State	Description
1. NEW	The thread is created but not yet started (`start()` not called).
2. RUNNABLE	The thread is ready to run but waiting for CPU time.
3. RUNNING	The thread is executing its task.
4. BLOCKED	The thread is waiting to enter a synchronized block or method.
5. WAITING	The thread is waiting indefinitely for another thread to notify it (`wait()`).
6. TIMED_WAITING	The thread is waiting for a specified time (`sleep()`, `join(timeout)`, `wait(timeout)`).

State	Description
7. TERMINATED (DEAD)	The thread has finished execution or was stopped.

🔷 1. NEW State

A thread is in the **NEW** state when created but not started.

✅ Example

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

public class NewStateExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        System.out.println("Thread State: " + t1.getState()); // NEW
    }
}
```

🔷 2. RUNNABLE State

The thread is **ready to run** but waiting for the CPU.

✅ Example

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

public class RunnableStateExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Thread moves from NEW → RUNNABLE
        System.out.println("Thread State: " + t1.getState()); // May be
RUNNABLE or TERMINATED
    }
}
```

🔷 3. RUNNING State

The thread is **actively executing its task**.

🔹 A thread moves from **RUNNABLE → RUNNING** when the CPU **schedules it**.
(We cannot force it; the CPU decides when to execute a thread.)

🔷 4. BLOCKED State

A thread enters **BLOCKED** state when trying to access a **synchronized resource** already held by another thread.

✅ Example

```
class SharedResource {
    synchronized void access() {
        System.out.println(Thread.currentThread().getName() + " is
accessing...");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

public class BlockedStateExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread t1 = new Thread(() -> resource.access());
        Thread t2 = new Thread(() -> resource.access());

        t1.start();
        t2.start();

        try {
            Thread.sleep(500); // Ensures t1 locks the resource first
            System.out.println("t2 State: " + t2.getState()); // BLOCKED
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

🔹 **Key Point:** `t2` enters **BLOCKED** state because `t1` is using the synchronized method.

🔷 5. WAITING State

A thread enters **WAITING** state when it calls **`wait()`** and waits indefinitely.

✅ Example

```
class WaitExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (WaitExample.class) {
                try {
                    WaitExample.class.wait(); // Moves to WAITING state
                } catch (InterruptedException e) {
                    System.out.println(e);
                }
            }
        });

        t1.start();

        try {
            Thread.sleep(500);
            System.out.println("t1 State: " + t1.getState()); // WAITING
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

🔹 **Key Point:** The thread **waits indefinitely** until another thread calls `notify()`.

🔷 6. TIMED_WAITING State

A thread enters **TIMED_WAITING** when it waits for a specific time (`sleep()`, `wait(timeout)`, `join(timeout)`).

✅ Example

```
class TimedWaitingExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            try {
                Thread.sleep(3000); // Moves to TIMED_WAITING
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        });

        t1.start();

        try {
            Thread.sleep(500);
            System.out.println("t1 State: " + t1.getState()); //
TIMED_WAITING
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

🔹 **Key Point:** The thread **resumes** after the timeout expires.

🔷 7. TERMINATED (DEAD) State

A thread enters **TERMINATED** state when it **finishes execution** or is **stopped**.

✅ Example

```
class TerminatedExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> System.out.println("Thread is
running..."));
        t1.start();

        try {
            Thread.sleep(100);
            System.out.println("t1 State: " + t1.getState()); // TERMINATED
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

🔹 **Key Point:** Once a thread **completes execution**, it cannot be restarted.

🔷 Thread State Transitions (Lifecycle Diagram)

`NEW → RUNNABLE → RUNNING → BLOCKED / WAITING / TIMED_WAITING → TERMINATED`

✨ Summary Table

State	How it Occurs?
NEW	Thread created but `start()` not called.
RUNNABLE	`start()` called, waiting for CPU.
RUNNING	Thread is actively executing.
BLOCKED	Waiting for a synchronized resource.
WAITING	Waiting indefinitely (`wait()`).
TIMED_WAITING	Waiting for a specific time (`sleep()`, `wait(time)`).
TERMINATED	Thread execution is complete.

✨ Conclusion

✓ **Threads move between states** during execution.
✓ `BLOCKED`, `WAITING`, and `TIMED_WAITING` indicate **threads are paused** for some reason.
✓ **A TERMINATED thread cannot restart** (must create a new thread).

Would you like a **real-world example** of these states in action? 🚀

🔷 Thread Life Cycle in Java

A thread in Java goes through **different states** during its execution, forming a **life cycle**. Understanding this cycle helps in writing efficient **multithreaded programs**.

📌 Thread Life Cycle Phases

A thread goes through **five main stages** in its life:

1️⃣ **NEW** → The thread is created but not yet started.
2️⃣ **RUNNABLE** → The thread is ready to run, waiting for CPU time.
3️⃣ **RUNNING** → The thread is actively executing.
4️⃣ **BLOCKED / WAITING / TIMED_WAITING** → The thread is paused, waiting for a resource or time.
5️⃣ **TERMINATED (DEAD)** → The thread has finished execution.

🖼 Life Cycle Diagram

```
NEW → RUNNABLE → RUNNING → BLOCKED / WAITING / TIMED_WAITING → TERMINATED
```

🔷 1. NEW State

A thread is in the **NEW** state when it is created but has not started.

✅ Example

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

public class NewStateExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
```

```
        System.out.println("Thread State: " + t1.getState()); // NEW
    }
}
```

🔹 **Key Point:** The thread remains **NEW** until `start()` is called.

🔷 2. RUNNABLE State

The thread moves to **RUNNABLE** after calling `start()`. It is ready to run but **waiting for CPU scheduling**.

✅ Example

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

public class RunnableStateExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Moves from NEW → RUNNABLE
        System.out.println("Thread State: " + t1.getState()); // RUNNABLE or
TERMINATED
    }
}
```

🔹 **Key Point:** The thread is now **eligible to run** but the **CPU decides when**.

🔷 3. RUNNING State

A thread moves from **RUNNABLE → RUNNING** when the **CPU schedules it**.

🔹 We **cannot force** a thread into the **RUNNING** state. The **JVM scheduler** decides.

🔷 4. BLOCKED / WAITING / TIMED_WAITING States

A running thread can pause and move into **one of these states**.

⏸ BLOCKED State

A thread is **BLOCKED** if it tries to enter a **synchronized block** already locked by another thread.

✅ Example

```
class SharedResource {
    synchronized void access() {
        System.out.println(Thread.currentThread().getName() + " is
accessing...");
        try { Thread.sleep(2000); } catch (InterruptedException e) {}
    }
}

public class BlockedStateExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        Thread t1 = new Thread(() -> resource.access());
        Thread t2 = new Thread(() -> resource.access());

        t1.start();
        t2.start();

        try {
            Thread.sleep(500); // Ensures t1 locks the resource first
            System.out.println("t2 State: " + t2.getState()); // BLOCKED
        } catch (InterruptedException e) {}
    }
}
```

🔹 **Key Point:** `t2` enters **BLOCKED** state because `t1` has **locked** the resource.

⏳ WAITING State

A thread is in **WAITING** state if it calls **`wait()`** and is waiting **indefinitely** for another thread to notify it.

✅ Example

```
class WaitExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (WaitExample.class) {
                try {
                    WaitExample.class.wait(); // Moves to WAITING state
                } catch (InterruptedException e) {}
            }
        });

        t1.start();

        try {
            Thread.sleep(500);
            System.out.println("t1 State: " + t1.getState()); // WAITING
        } catch (InterruptedException e) {}
    }
}
```

🔹 **Key Point:** The thread **waits indefinitely** until another thread calls `notify()`.

⏳ TIMED_WAITING State

A thread is in **TIMED_WAITING** state if it waits for a **specific time** using `sleep()`, `wait(time)`, or `join(time)`.

✅ Example

```
class TimedWaitingExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            try {
                Thread.sleep(3000); // Moves to TIMED_WAITING
            } catch (InterruptedException e) {}
        });

        t1.start();

        try {
            Thread.sleep(500);
            System.out.println("t1 State: " + t1.getState()); //
TIMED_WAITING
        } catch (InterruptedException e) {}
    }
}
```

🔹 **Key Point:** The thread **resumes** execution after the timeout expires.

🔚 5. TERMINATED (DEAD) State

A thread enters the **TERMINATED** state after **completing execution** or being **stopped**.

✅ Example

```
class TerminatedExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> System.out.println("Thread is
running..."));
        t1.start();

        try {
            Thread.sleep(100);
            System.out.println("t1 State: " + t1.getState()); // TERMINATED
        } catch (InterruptedException e) {}
    }
}
```

🔹 **Key Point:** Once a thread **finishes execution**, it **cannot be restarted**.

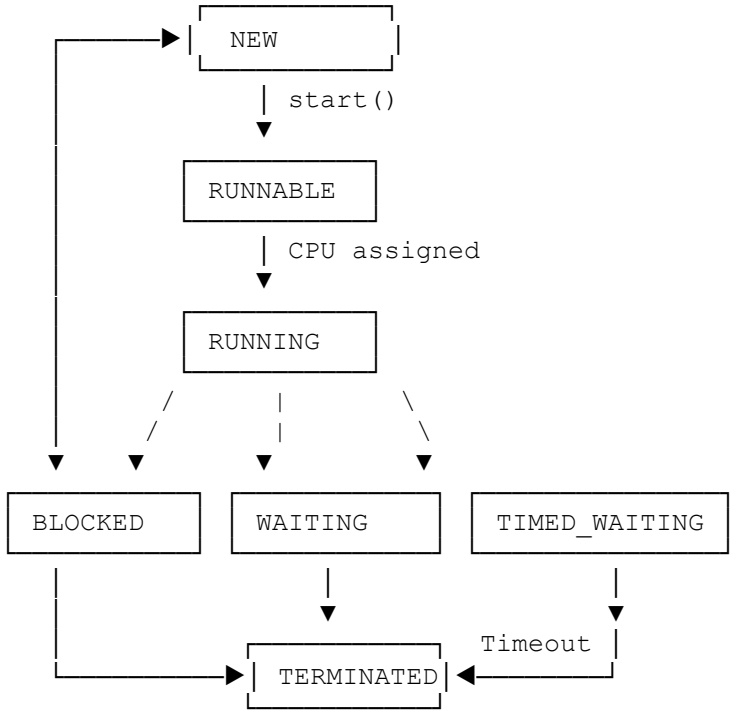
◈ Thread Life Cycle Summary

State	How it Occurs?
NEW	Thread created but `start()` not called.
RUNNABLE	`start()` called, waiting for CPU.
RUNNING	Thread is actively executing.
BLOCKED	Waiting for a synchronized resource.
WAITING	Waiting indefinitely (`wait()`).
TIMED_WAITING	Waiting for a specific time (`sleep()`, `wait(time)`).
TERMINATED	Thread execution is complete.

◈ Thread State Transition Diagram

`NEW → RUNNABLE → RUNNING → BLOCKED / WAITING / TIMED_WAITING → TERMINATED`

🖼 Visual Representation



✨ Conclusion

✓ **Threads move between states** based on execution.
✓ **WAITING and BLOCKED** threads are **paused** until a condition is met.
✓ **A TERMINATED thread cannot restart** (must create a new thread).

Would you like a **real-world example** of thread life cycle usage? 🚀

🔷 Creating Threads in Java

In Java, we can create **threads** using two main approaches:
1️⃣ **Extending `Thread` class**
2️⃣ **Implementing `Runnable` interface**

Additionally, we can create **threads using `Callable` and `Executors`**, which allow returning results.

🔷 1. Creating a Thread by Extending `Thread` Class

We can create a thread by **extending the `Thread` class** and overriding its `run()` method.

✅ Example

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

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

🔷 How It Works?

1. **Create a class** (`MyThread`) that extends `Thread`.
2. **Override `run()` method** to define the task.
3. **Create an instance** and call `start()`.

🚀 Important:

- Never call `run()` directly, as it will execute like a normal method.
- Calling `start()` executes `run()` in a **new thread**.

🔷 2. Creating a Thread by Implementing `Runnable` Interface

A more flexible way to create threads is by **implementing `Runnable`**, as Java supports only **single inheritance**.

✅ Example

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

public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable myTask = new MyRunnable(); // Create a task
        Thread t1 = new Thread(myTask); // Create a thread
        t1.start(); // Start the thread
    }
}
```

🔷 How It Works?

1. **Create a class** (`MyRunnable`) that implements `Runnable`.
2. **Override `run()` method** to define the task.
3. **Create a `Thread` object** and pass the `Runnable` instance.
4. **Call `start()`** to run the thread.

🚀 Advantages:

✅ Allows multiple inheritance (since it does not extend `Thread`).
✅ More flexible and widely used in **real-world applications**.

🔷 3. Creating Threads Using `Lambda Expressions` (Java 8+)

Since `Runnable` is a **functional interface**, we can use **lambda expressions** to create threads more concisely.

✅ Example

```
public class LambdaThreadExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> System.out.println("Thread is
running..."));
        t1.start();
    }
}
```

🔷 Benefits of Using Lambda

✓ Reduces **boilerplate code**
✓ Improves **readability**

🔷 4. Creating Threads Using `Callable` and `Future`

If we need a **thread to return a value**, we use `Callable<V>` instead of `Runnable`.

✅ Example

```
import java.util.concurrent.*;

class MyCallable implements Callable<String> {
    public String call() {
        return "Thread execution completed!";
    }
}

public class CallableExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<String> future = executor.submit(new MyCallable());

        System.out.println(future.get()); // Retrieves the result
        executor.shutdown();
    }
}
```

🚀 Key Differences Between `Runnable` and `Callable`

Feature	Runnable	Callable
Return Type	`void` (no result)	`V` (returns a value)
Exception Handling	Cannot throw checked exceptions	Can throw checked exceptions
Used With	`Thread` class	`ExecutorService`

🔷 5. Creating Threads Using `Executors` (Thread Pool)

If we need **multiple threads**, using `ExecutorService` is better than manually creating threads.

✅ Example

```
import java.util.concurrent.*;
```

```
class MyTask implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is
running...");
    }
}

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            executor.execute(new MyTask()); // Submitting tasks
        }

        executor.shutdown(); // Shutdown executor
    }
}
```

🚀 **Benefits of Using `Executors` ✓ Thread reusability** (avoids frequent creation/destruction of threads).
✓ **Better performance** for handling multiple tasks.

🔷 Choosing the Best Way to Create Threads

Method	Use Case
`Thread` class	When a class does **not need to extend another class**
`Runnable`	When a class needs **multiple inheritance**
`Lambda`	When using **Java 8+**, and code should be concise
`Callable`	When a thread **returns a result**
`Executors`	When handling **multiple threads efficiently**

✨ Conclusion

✓ Java provides **multiple ways** to create threads based on different needs.
✓ **Extending `Thread`** is simple but **not flexible**.
✓ **Implementing `Runnable`** is preferred for **better code reuse**.
✓ **Using `Callable`** allows **returning results** from threads.
✓ **Thread Pools (`Executors`)** improve **performance** in multi-threaded programs.

Would you like **real-world examples** using threads? 🚀

🔷 Interrupting Threads in Java

In Java, a **thread can be interrupted** to **request it to stop execution**. However, **interrupting a thread does not forcefully stop it**; instead, it is up to the thread to **handle the interruption properly**.

🔷 Ways to Interrupt a Thread

1️⃣ **Using `interrupt()` method** → Requests the thread to stop.
2️⃣ **Handling `InterruptedException`** → Used in methods like `sleep()`, `wait()`, etc.
3️⃣ **Checking `isInterrupted()`** → Checks if a thread was interrupted.
4️⃣ **Using `Thread.interrupted()`** → Static method that clears the interrupted status.

🔷 1. Interrupting a Thread Using `interrupt()`

We can call `interrupt()` on a **running thread**, which sets the **interrupt flag**.

✅ Example

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(i);
        }
    }
}

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

🔹 Output

```
1
2
3
4
5
```

🚀 Why does it still run?

- Because `interrupt()` **only sets a flag**.

- The thread **does not stop automatically** unless we handle it.

🔷 2. Handling `InterruptedException`

If a thread is **sleeping or waiting**, calling `interrupt()` **throws `InterruptedException`**, stopping execution.

✅ Example

```
class MyThread extends Thread {
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println(i);
                Thread.sleep(1000); // Sleep for 1 second
            }
        } catch (InterruptedException e) {
            System.out.println("Thread was interrupted!");
        }
    }
}

public class InterruptHandling {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();

        try {
            Thread.sleep(2000); // Main thread waits for 2 seconds
        } catch (InterruptedException e) {}

        t1.interrupt(); // Interrupts t1 during sleep
    }
}
```

🔹 Output

```
1
2
Thread was interrupted!
```

🚀 What happened?

- The thread was sleeping (`Thread.sleep(1000)`).
- `interrupt()` caused `InterruptedException`, exiting the loop.

🔷 3. Checking `isInterrupted()`

A thread can check if it was interrupted using `isInterrupted()`.

✅ Example

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

public class IsInterruptedExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {}

        t1.interrupt(); // Interrupting the thread
    }
}
```

🔷 Output

```
Thread is running...
Thread is running...
Thread is running...
Thread was interrupted!
```

🚀 Key Point:

- `isInterrupted()` **does not clear** the interrupt flag.
- The loop exits when `isInterrupted()` returns `true`.

🔹 4. Using `Thread.interrupted()`

`Thread.interrupted()` **checks and clears** the interrupt flag.

✅ Example

```
class MyThread extends Thread {
    public void run() {
        while (true) {
            if (Thread.interrupted()) { // Checks and clears the flag
                System.out.println("Thread was interrupted!");
                break;
            }
            System.out.println("Thread is running...");
        }
    }
}
```

```
public class InterruptedExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {}

        t1.interrupt(); // Interrupting the thread
    }
}
```

🔷 Output

```
Thread is running...
Thread is running...
Thread is running...
Thread was interrupted!
```

🚀 Key Difference:

- `Thread.interrupted()` **clears the flag** after checking.
- The next call to `Thread.interrupted()` returns `false`.

🔹 Summary of Interrupting Threads

Method	Purpose
`interrupt()`	Sets the interrupt flag for the thread.
`InterruptedException`	Stops execution when waiting or sleeping.
`isInterrupted()`	Checks the interrupt flag **without clearing** it.
`Thread.interrupted()`	Checks and **clears** the interrupt flag.

✨ Conclusion

✓ **Interrupting a thread does not stop it immediately**; it only sets a flag.
✓ We need to **handle the interruption properly** using `isInterrupted()` or `InterruptedException`.
✓ `Thread.interrupted()` clears the flag, while `isInterrupted()` does not.

Would you like a **real-world example** using thread interruption? 🚀

🔷 Thread Priorities in Java

In Java, each thread has a **priority** that helps the **Thread Scheduler** decide **which thread to run first** when multiple threads are waiting for execution.

⚡ **However, thread priority does NOT guarantee execution order.** It only gives a **hint** to the thread scheduler.

🔷 1. Understanding Thread Priority

- Java assigns **default priority = 5** to every thread.
- Thread priority **ranges from 1 to 10**:
  - 🔹 `Thread.MIN_PRIORITY` = **1**
  - 🔹 `Thread.NORM_PRIORITY` = **5** (Default)
  - 🔹 `Thread.MAX_PRIORITY` = **10**

✅ Example: Setting Thread Priorities

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Priority: " +
Thread.currentThread().getPriority());
    }
}

public class ThreadPriorityExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.setPriority(Thread.MIN_PRIORITY); // Priority = 1
        t2.setPriority(Thread.NORM_PRIORITY); // Priority = 5 (Default)
        t3.setPriority(Thread.MAX_PRIORITY); // Priority = 10

        t1.start();
        t2.start();
        t3.start();
    }
}
```

🔹 Output (Order May Vary)

```
Thread-0 Priority: 1
Thread-1 Priority: 5
Thread-2 Priority: 10
```

🚀 Key Points:

- `setPriority(int priority)` sets the thread's priority.

- `getPriority()` returns the thread's current priority.
- Execution **order is not guaranteed** because it depends on the **Thread Scheduler**.

🔷 2. Does Higher Priority Guarantee Faster Execution?

🔴 No!

- Thread scheduling depends on the **JVM implementation & OS**.
- Some OS schedulers **may ignore priority** and execute threads based on **time slicing**.
- **Example:** On Linux, thread priority may have **little or no effect**.

🔷 3. Example: Multiple Threads with Different Priorities

✅ Example: Observing Priority Impact

```
class MyThread extends Thread {
    MyThread(String name) {
        super(name);
    }
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(getName() + " is running...");
        }
    }
}

public class ThreadPriorityTest {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Low Priority");
        MyThread t2 = new MyThread("High Priority");

        t1.setPriority(Thread.MIN_PRIORITY); // Priority = 1
        t2.setPriority(Thread.MAX_PRIORITY); // Priority = 10

        t1.start();
        t2.start();
    }
}
```

🔹 Possible Output

```
High Priority is running...
High Priority is running...
High Priority is running...
Low Priority is running...
Low Priority is running...
Low Priority is running...
```

🚀 Key Observations:

- The **higher-priority thread** (`t2`) gets more CPU time.
- However, execution order **is still not guaranteed**.

🔷 4. How JVM Schedules Threads?

The **JVM uses preemptive or time-slicing scheduling**, based on:
1️⃣ **Thread Priority** → Higher-priority threads **may get more CPU time**.
2️⃣ **Operating System Scheduling** → Some OS **ignore priority** (like Linux).
3️⃣ **Thread State** → If a high-priority thread is **blocked**, a lower-priority thread can run.

🔷 5. When to Use Thread Priorities?

✅ When handling **urgent tasks** (e.g., UI updates).
✅ In **multithreaded applications** where some tasks are **more important**.
❌ **Avoid relying on priority** for crucial logic. Always assume threads will run **in any order**.

✨ Conclusion

✓ **Thread priorities range from 1 to 10** (default = 5).
✓ **Higher priority ≠ guaranteed execution first.**
✓ **Thread scheduling depends on JVM and OS.**
✓ Use priorities **only as a hint** to JVM, not as a strict rule.

Would you like an example of **real-world thread priority usage**? 🚀

🔷 Synchronizing Threads in Java

In **multithreading**, multiple threads share resources, which can lead to **race conditions** (inconsistent data). **Synchronization** helps in **controlling access** to shared resources.

🔷 1. Why Synchronization? (Race Condition Example)

If multiple threads modify a shared resource **without synchronization**, data may become inconsistent.

✅ **Example (Without Synchronization - Problem)**

```
class BankAccount {
```

```
    int balance = 100;

    void withdraw(int amount) {
        if (balance >= amount) {
            System.out.println(Thread.currentThread().getName() + " is
withdrawing " + amount);
            balance -= amount;
            System.out.println("Remaining Balance: " + balance);
        } else {
            System.out.println("Not enough balance for " +
Thread.currentThread().getName());
        }
    }
}

public class RaceConditionExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread t1 = new Thread(() -> account.withdraw(80), "Thread-1");
        Thread t2 = new Thread(() -> account.withdraw(80), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

🔷 Possible Output (Inconsistent Results)

```
Thread-1 is withdrawing 80
Remaining Balance: 20
Thread-2 is withdrawing 80
Remaining Balance: -60  ❌ (Incorrect)
```

🚀 Why?

- Both threads check `balance >= amount` at **the same time** before withdrawal.
- This **race condition** results in **over-withdrawal**.

🔷 2. Solution: Synchronization

We can prevent race conditions using **synchronized methods** or **synchronized blocks**.

✅ 2.1 Using Synchronized Methods

We declare the method as `synchronized`, ensuring only **one thread** executes it at a time.

```
class BankAccount {
    int balance = 100;

    synchronized void withdraw(int amount) {
        if (balance >= amount) {
```

```
            System.out.println(Thread.currentThread().getName() + " is
withdrawing " + amount);
            balance -= amount;
            System.out.println("Remaining Balance: " + balance);
        } else {
            System.out.println("Not enough balance for " +
Thread.currentThread().getName());
        }
    }
}

public class SynchronizedExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread t1 = new Thread(() -> account.withdraw(80), "Thread-1");
        Thread t2 = new Thread(() -> account.withdraw(80), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

🔷 Correct Output

```
Thread-1 is withdrawing 80
Remaining Balance: 20
Not enough balance for Thread-2
```

🚀 How It Works?

- The `synchronized` keyword **locks** the method.
- If one thread is executing `withdraw()`, others **must wait**.

☑️ 2.2 Using Synchronized Blocks

Instead of synchronizing the whole method, we can synchronize **only critical sections**.

```
class BankAccount {
    int balance = 100;

    void withdraw(int amount) {
        synchronized (this) { // Lock only the critical section
            if (balance >= amount) {
                System.out.println(Thread.currentThread().getName() + " is
withdrawing " + amount);
                balance -= amount;
                System.out.println("Remaining Balance: " + balance);
            } else {
                System.out.println("Not enough balance for " +
Thread.currentThread().getName());
            }
        }
    }
}
```

```
public class SynchronizedBlockExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread t1 = new Thread(() -> account.withdraw(80), "Thread-1");
        Thread t2 = new Thread(() -> account.withdraw(80), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

🚀 **When to Use Synchronized Blocks?** ✓ Use it when **only a small part** of a method needs synchronization.
✓ Improves **performance** by reducing time under the lock.

◈ 3. Static Synchronization

If multiple threads are accessing **static methods** of a class, we must synchronize at the **class level**.

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

    synchronized static void increment() { // Locks at class level
        count++;
        System.out.println(Thread.currentThread().getName() + ": " + count);
    }
}

public class StaticSyncExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(Counter::increment);
        Thread t2 = new Thread(Counter::increment);

        t1.start();
        t2.start();
    }
}
```

🚀 Key Point:

- `synchronized static` locks the **whole class**, not just an instance.

◈ 4. Locks (`ReentrantLock`) - Alternative to Synchronization

Instead of `synchronized`, Java provides **explicit locks** using `ReentrantLock`.

☑ Example Using `ReentrantLock`

```
import java.util.concurrent.locks.ReentrantLock;

class BankAccount {
    int balance = 100;
    private final ReentrantLock lock = new ReentrantLock();

    void withdraw(int amount) {
        lock.lock(); // Acquire lock
        try {
            if (balance >= amount) {
                System.out.println(Thread.currentThread().getName() + " is
withdrawing " + amount);
                balance -= amount;
                System.out.println("Remaining Balance: " + balance);
            } else {
                System.out.println("Not enough balance for " +
Thread.currentThread().getName());
            }
        } finally {
            lock.unlock(); // Release lock
        }
    }
}

public class LockExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread t1 = new Thread(() -> account.withdraw(80), "Thread-1");
        Thread t2 = new Thread(() -> account.withdraw(80), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

🚀 **Why Use `ReentrantLock`?**
✓ More flexibility (can try acquiring a lock with timeout).
✓ Supports fairness policy (**First Come First Serve scheduling**).

◈ 5. Summary of Synchronization Techniques

Method	How It Works?	Use Case
`synchronized` method	Locks entire method	Simple cases, entire method needs lock
`synchronized` block	Locks only a specific section	Performance optimization
`static synchronized`	Locks the class, not instance	Synchronizing static methods
`ReentrantLock`	Explicit lock with more features	When flexibility is needed

---

✨ Conclusion

✓ Synchronization **prevents race conditions** but **reduces performance**.
✓ **Use `synchronized` methods or blocks** for thread safety.
✓ **For advanced control, use `ReentrantLock`.**

Would you like an example of **deadlock and how to prevent it**? 🚀

🔷 Inter-Thread Communication in Java

🚀 Why Do Threads Need to Communicate?

When multiple threads share a resource, they often need to **communicate** and **coordinate** actions to avoid race conditions or unnecessary waiting.

🔹 **Example:** A **Producer thread** produces data, and a **Consumer thread** consumes it. The consumer must **wait** until the producer has produced something.

🔷 1. How Does Inter-Thread Communication Work?

Java provides **three built-in methods** inside `Object` class to enable communication between threads:

Method	Description
`wait()`	Makes the thread wait (releases the lock)
`notify()`	Wakes up **one** waiting thread
`notifyAll()`	Wakes up **all** waiting threads

✅ **These methods must be called inside a `synchronized` block** because they **work with locks**.

🔷 2. Producer-Consumer Example Using `wait()` and `notify()`

Let's implement a **Producer-Consumer problem**, where:

- The **Producer** thread produces data.
- The **Consumer** thread consumes it **only when available**.

✅ Example Code:

```
class SharedResource {
    private int data;
    private boolean hasData = false;

    public synchronized void produce(int value) {
        while (hasData) { // If data is already produced, wait
            try {
                wait(); // Release the lock and wait
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        data = value;
        System.out.println("Produced: " + data);
        hasData = true;
        notify(); // Wake up consumer
    }

    public synchronized void consume() {
        while (!hasData) { // If no data, wait
            try {
                wait(); // Release the lock and wait
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Consumed: " + data);
        hasData = false;
        notify(); // Wake up producer
    }
}

public class InterThreadCommunication {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.produce(i);
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.consume();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

🔷 Expected Output:

```
Produced: 1
```

```
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

🔷 3. Explanation of the Code

🔹 `wait()`

- Makes the **current thread wait** and **releases the lock**.
- It waits until `notify()` is called by another thread.

🔹 `notify()`

- Wakes up **one waiting thread** (if any).

🔹 `synchronized`

- Ensures that **only one thread** accesses the method at a time.

🔷 4. `notifyAll()` - Waking Up Multiple Threads

If multiple consumer threads are waiting, we use `notifyAll()` instead of `notify()`.

✅ Modified Code Using `notifyAll()`

```
public synchronized void consume() {
    while (!hasData) {
        try {
            wait(); // Consumer waits if no data
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println("Consumed: " + data);
    hasData = false;
    notifyAll(); // Wake up all waiting threads
}
```

🚀 When to Use `notifyAll()`?

- When multiple consumers are waiting for a resource.

🔷 5. Alternative: Using `BlockingQueue` (Recommended)

Instead of using `wait()` and `notify()`, Java provides `BlockingQueue` (thread-safe) for producer-consumer problems.

✅ Example Using `BlockingQueue`

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

class Producer implements Runnable {
    private BlockingQueue<Integer> queue;

    Producer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Produced: " + i);
                queue.put(i); // Automatically waits if full
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumer implements Runnable {
    private BlockingQueue<Integer> queue;

    Consumer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                int value = queue.take(); // Automatically waits if empty
                System.out.println("Consumed: " + value);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class BlockingQueueExample {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(2);

        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        producer.start();
        consumer.start();
```

```
    }
}
```

🔷 **Advantages of `BlockingQueue`:** ✓ No need for **manual synchronization** (`synchronized`, `wait()`, `notify()`).
✓ Handles **automatic waiting** when queue is **full/empty**.

🔹 6. Summary of Inter-Thread Communication

Method	Description
`wait()`	Releases lock, makes thread wait until `notify()`
`notify()`	Wakes up **one waiting thread**
`notifyAll()`	Wakes up **all waiting threads**
`BlockingQueue`	A thread-safe queue that handles waiting automatically

✨ Conclusion

✓ **Inter-thread communication** is needed when threads **share resources**.
✓ Use `wait()` & `notify()` inside **synchronized methods** for coordination.
✓ **Prefer `BlockingQueue`** for a cleaner, more efficient approach.

Would you like a **real-world example** of thread communication? 🚀

🔹 Stream-Based I/O in Java (`java.io`)

🚀 What is Stream-Based I/O?

Java uses **streams** to perform **input and output (I/O)** operations. A **stream** is a **sequence of data** that flows from a **source** to a **destination**.

🔹 1. Types of Streams in Java

Java provides **two types of streams** for handling data:

Type	Handles	Examples
Byte Streams (`InputStream`, `OutputStream`)	Binary data (images, videos, etc.)	`FileInputStream`, `FileOutputStream`
Character Streams (`Reader`, `Writer`)	Text data (UTF-8, Unicode)	`FileReader`, `FileWriter`

🔷 2. Byte Streams (For Binary Data)

Byte streams handle **raw binary data** (e.g., images, audio, video files).
They **do not** perform character encoding.

✅ Reading Bytes Using `FileInputStream`

```
import java.io.FileInputStream;
import java.io.IOException;

public class ByteStreamReadExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int i;
            while ((i = fis.read()) != -1) {
                System.out.print((char) i); // Convert byte to char
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ Writing Bytes Using `FileOutputStream`

```
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamWriteExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt")) {
            String data = "Hello, Java!";
            fos.write(data.getBytes()); // Convert string to bytes
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

🔷 3. Character Streams (For Text Data)

Character streams handle **textual data** and support **Unicode encoding**.

✅ Reading Characters Using `FileReader`

```
import java.io.FileReader;
import java.io.IOException;

public class CharacterStreamReadExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("example.txt")) {
            int i;
            while ((i = fr.read()) != -1) {
                System.out.print((char) i);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ Writing Characters Using `FileWriter`

```
import java.io.FileWriter;
import java.io.IOException;

public class CharacterStreamWriteExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("example.txt")) {
            fw.write("Hello, Java Character Streams!");
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

🔷 4. Buffered Streams (For Efficient I/O)

Buffered streams **improve performance** by reducing direct interaction with files.

✅ Using `BufferedReader` for Efficient Reading

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

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

🔷 **Why Use `BufferedReader`?** ✓ Reads **line-by-line** instead of character-by-character.
✓ Reduces **I/O operations**, improving efficiency.

🔹 5. Data Streams (`DataInputStream`, `DataOutputStream`)

Used for **reading/writing primitive data types** (e.g., `int`, `float`, `double`).

✅ Writing Data Using `DataOutputStream`

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataOutputStreamExample {
    public static void main(String[] args) {
        try (DataOutputStream dos = new DataOutputStream(new
FileOutputStream("data.bin"))) {
            dos.writeInt(100);
            dos.writeDouble(99.99);
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ Reading Data Using `DataInputStream`

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class DataInputStreamExample {
    public static void main(String[] args) {
        try (DataInputStream dis = new DataInputStream(new
FileInputStream("data.bin"))) {
            int intValue = dis.readInt();
            double doubleValue = dis.readDouble();
            System.out.println("Integer: " + intValue);
            System.out.println("Double: " + doubleValue);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

🔷 **Why Use `Data Streams`?** ✓ Stores **primitive data types** directly.
✓ Avoids manual **byte conversion**.

🔹 6. Object Streams (`ObjectInputStream`, `ObjectOutputStream`)

Used to read/write **entire objects** (**Serialization & Deserialization**).

✅ Writing an Object Using `ObjectOutputStream`

```
import java.io.*;

class Person implements Serializable { // Must implement Serializable
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class ObjectOutputStreamExample {
    public static void main(String[] args) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("person.dat"))) {
            Person person = new Person("Alice", 25);
            oos.writeObject(person);
            System.out.println("Object written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ Reading an Object Using `ObjectInputStream`

```
import java.io.*;

public class ObjectInputStreamExample {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("person.dat"))) {
            Person person = (Person) ois.readObject();
            System.out.println("Name: " + person.name + ", Age: " +
person.age);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

🔹 **Why Use Object Streams?** ✓ Stores **entire objects** in a file.
✓ Avoids **manual field writing/reading**.

🔷 7. Summary of Java I/O Streams

Stream Type	Class Used	Purpose
Byte Streams	`FileInputStream`, `FileOutputStream`	Reads/Writes binary data
Character Streams	`FileReader`, `FileWriter`	Reads/Writes text data

Stream Type	Class Used	Purpose
Buffered Streams	`BufferedReader`, `BufferedWriter`	Improves performance
Data Streams	`DataInputStream`, `DataOutputStream`	Reads/Writes primitive data types
Object Streams	`ObjectInputStream`, `ObjectOutputStream`	Reads/Writes objects (serialization)

✨ Conclusion

✓ **Streams** provide a flexible way to perform I/O operations in Java.
✓ Use **byte streams** for binary data and **character streams** for text files.
✓ **Buffered streams** improve performance, and **object streams** allow object persistence.

Would you like an example of **network streams** (`Socket` programming)? 🚀

🔷 Stream Classes in Java: Byte Streams & Character Streams

In Java, streams are used to perform **input and output (I/O)** operations on data. They provide an efficient way to read and write **bytes** or **characters**.

🔷 1. Byte Streams (For Binary Data)

Byte streams handle **raw binary data**, such as images, videos, and audio files. They read and write **data in bytes (8-bit units)**.

📌 Byte Stream Classes

Class Name	Usage	Type
`InputStream`	Base class for input byte streams	Abstract Class
`OutputStream`	Base class for output byte streams	Abstract Class
`FileInputStream`	Reads bytes from a file	Concrete Class
`FileOutputStream`	Writes bytes to a file	Concrete Class
`BufferedInputStream`	Buffers input for efficiency	Concrete Class
`BufferedOutputStream`	Buffers output for efficiency	Concrete Class

Class Name	Usage	Type
`DataInputStream`	Reads primitive data types	Concrete Class
`DataOutputStream`	Writes primitive data types	Concrete Class

✅ Example: Reading a File Using `FileInputStream`

```
import java.io.FileInputStream;
import java.io.IOException;

public class ByteStreamReadExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int i;
            while ((i = fis.read()) != -1) { // Read one byte at a time
                System.out.print((char) i);  // Convert byte to char
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ Example: Writing to a File Using `FileOutputStream`

```
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreamWriteExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt")) {
            String data = "Hello, Java!";
            fos.write(data.getBytes()); // Convert String to bytes
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Best For:** Handling **binary files** (e.g., images, audio, video).

🔷 2. Character Streams (For Text Data)

Character streams are used for **text-based data**. They work with **Unicode characters**, making them ideal for handling **text files**.

📌 Character Stream Classes

Class Name	Usage	Type
`Reader`	Base class for input character streams	Abstract Class
`Writer`	Base class for output character streams	Abstract Class
`FileReader`	Reads characters from a file	Concrete Class
`FileWriter`	Writes characters to a file	Concrete Class
`BufferedReader`	Buffers character input for efficiency	Concrete Class
`BufferedWriter`	Buffers character output for efficiency	Concrete Class

✅ Example: Reading a File Using `FileReader`

```
import java.io.FileReader;
import java.io.IOException;

public class CharacterStreamReadExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("example.txt")) {
            int i;
            while ((i = fr.read()) != -1) { // Read character by character
                System.out.print((char) i);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✅ Example: Writing to a File Using `FileWriter`

```
import java.io.FileWriter;
import java.io.IOException;

public class CharacterStreamWriteExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("example.txt")) {
            fw.write("Hello, Java Character Streams!");
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Best For:** Handling **text files** efficiently with **character encoding support**.

🔷 3. Buffered Streams (For Efficient I/O)

Buffered streams improve **performance** by **reducing the number of direct read/write operations** on a file.

✅ Example: Efficient Reading Using `BufferedReader`

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) { // Read line-by-line
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ Advantages:

- **Reads entire lines at a time** (better performance).
- **Reduces I/O operations**, making it **faster**.

🔷 4. Byte Streams vs. Character Streams

Feature	Byte Streams (`InputStream`, `OutputStream`)	Character Streams (`Reader`, `Writer`)
Handles	Binary data (images, videos, audio)	Text data (characters, strings)
Data Type	**Bytes** (8-bit)	**Characters** (16-bit, Unicode)
Performance	Faster for raw binary files	Better for text-based files
Encoding Support	No character encoding	Supports **Unicode encoding**
Examples	`FileInputStream`, `FileOutputStream`	`FileReader`, `FileWriter`

✓ Use Byte Streams when working with **binary files** (e.g., images, videos).
✓ Use Character Streams when working with **text files** (e.g., `.txt`, `.csv`).

🔷 5. Summary of Java I/O Stream Classes

Stream Type	Input Class	Output Class	Best Used For
Byte Stream	`FileInputStream`	`FileOutputStream`	Binary data (images, audio, video)
Character Stream	`FileReader`	`FileWriter`	Text data (text files, Unicode)
Buffered Stream	`BufferedReader`	`BufferedWriter`	Efficient I/O operations
Data Stream	`DataInputStream`	`DataOutputStream`	Reading/writing primitive data types

✨ Conclusion

✓ **Byte Streams (`InputStream` & `OutputStream`)** handle **binary data**.
✓ **Character Streams (`Reader` & `Writer`)** handle **text-based data**.
✓ **Buffered Streams** improve **performance** by reducing direct I/O operations.
✓ **Choose the right stream** based on your **file type (binary vs. text)**.

Would you like a **real-world example** of I/O operations (e.g., copying a file)? 🚀

🔷 Reading Console Input & Writing Console Output in Java

In Java, we can interact with the **console** using different methods to **read user input** and **display output**. The primary ways to handle console I/O are:

- **Reading Input**
  1. `Scanner` (Recommended, modern approach)
  2. `BufferedReader` (For efficient input handling)
  3. `System.in.read()` (Reads a single byte/character)
- **Writing Output**
  1. `System.out.println()` (Standard method)
  2. `System.out.print()` (Without newline)
  3. `System.out.printf()` (Formatted output)

🔷 1. Writing Console Output in Java

Java provides `System.out` to print text to the console.

✅ Using `System.out.println()`

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

```
        System.out.println("Hello, Java!"); // Prints with a newline
        System.out.print("This is a single line. "); // Prints without a
newline
        System.out.println("Next line starts here.");
    }
}
```

📌 Output:

```
Hello, Java!
This is a single line. Next line starts here.
```

✅ Using `System.out.printf()` (Formatted Output)

The `printf()` method allows formatted printing similar to C-style formatting.

```
public class FormattedOutputExample {
    public static void main(String[] args) {
        int age = 25;
        double salary = 75000.50;
        String name = "Alice";

        System.out.printf("Name: %s, Age: %d, Salary: $%.2f\n", name, age,
salary);
    }
}
```

📌 Output:

```
Name: Alice, Age: 25, Salary: $75000.50
```

✓ Format Specifiers:

- `%d` → Integer
- `%f` → Floating-point number
- `%.2f` → Floating-point with **2 decimal places**
- `%s` → String

🔷 2. Reading Console Input in Java

✅ (Method 1) Using `Scanner` (Recommended)

The `Scanner` class provides easy methods to read different types of input.

```
import java.util.Scanner;

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

        System.out.print("Enter your name: ");
```

```
        String name = scanner.nextLine(); // Reads a full line

        System.out.print("Enter your age: ");
        int age = scanner.nextInt(); // Reads an integer

        System.out.print("Enter your salary: ");
        double salary = scanner.nextDouble(); // Reads a double

        System.out.println("Hello " + name + ", you are " + age + " years old
with a salary of $" + salary);

        scanner.close(); // Always close Scanner to prevent resource leaks
    }
}
```

✓ **Best for**: Simple and flexible input reading.
✓ **Methods in `Scanner`**:

- `nextLine()` → Reads a full line (String).
- `nextInt()` → Reads an integer.
- `nextDouble()` → Reads a floating-point number.
- `next()` → Reads a **single word** (ignores spaces).

☑ (Method 2) Using `BufferedReader` (Efficient for Large Input)

The `BufferedReader` class reads text **efficiently**, but it requires exception handling.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class BufferedReaderExample {
    public static void main(String[] args) {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        try {
            System.out.print("Enter your name: ");
            String name = reader.readLine(); // Reads a full line

            System.out.print("Enter your age: ");
            int age = Integer.parseInt(reader.readLine()); // Reads a string
and converts to int

            System.out.println("Hello " + name + ", you are " + age + " years
old.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Best for**: Handling **large text input** efficiently.
✓ **Requires** explicit conversion (e.g., `Integer.parseInt()`).

✅ (Method 3) Using `System.in.read()` (Reads Single Character)

The `System.in.read()` method reads **one byte** (ASCII value).

```
import java.io.IOException;

public class SystemInExample {
    public static void main(String[] args) {
        System.out.print("Press any key: ");
        try {
            int input = System.in.read(); // Reads a single character (byte)
            System.out.println("You pressed: " + (char) input);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Best for**: **Reading single characters** from the console.
✓ **Limitation**: Only reads **one byte** at a time.

🔷 3. Summary of Console I/O Methods

Method	Class Used	Best For	Usage
`System.out.println()`	`System.out`	Printing normal output	`System.out.println("Hello")`
`System.out.printf()`	`System.out`	Formatted output	`System.out.printf("Age: %d", age)`
`Scanner`	`java.util.Scanner`	Easy input reading	`scanner.nextInt()`
`BufferedReader`	`java.io.BufferedReader`	Efficient large input	`reader.readLine()`
`System.in.read()`	`System.in`	Reading single character	`System.in.read()`

✨ Conclusion

✓ Use **`Scanner`** for **simple input reading** (best for beginners).
✓ Use **`BufferedReader`** for **efficient, large input** handling.
✓ Use **`System.out.printf()`** for **formatted output**.
✓ Use **`System.in.read()`** only when **reading a single character**.

Would you like an example of **reading multiple inputs from a single line**? 🚀

🔷 `File` Class in Java (`java.io.File`)

The `File` class in Java (part of the `java.io` package) is used to **represent file and directory pathnames** in an **abstract manner**. It allows us to **create, delete, check properties, and inspect files and directories** without directly handling file content.

🔷 1. Creating a File Object

To create a `File` object, use:

```
File file = new File("example.txt");
```

✓ This **does not create a physical file**; it just represents a file path.

🔷 2. Creating a New File

☑️ Example: Creating a File

```
import java.io.File;
import java.io.IOException;

public class FileCreateExample {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");

            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

✓ `createNewFile()` creates a new file **only if it doesn't already exist**.
✓ **Catches IOException**, which occurs if the file path is invalid.

🔷 3. Checking File Information

The `File` class provides methods to check file **properties**.

✅ Example: File Properties

```
import java.io.File;

public class FilePropertiesExample {
    public static void main(String[] args) {
        File file = new File("example.txt");

        if (file.exists()) { // Check if file exists
            System.out.println("File name: " + file.getName());
            System.out.println("Absolute path: " + file.getAbsolutePath());
            System.out.println("Writeable: " + file.canWrite());
            System.out.println("Readable: " + file.canRead());
            System.out.println("File size in bytes: " + file.length());
        } else {
            System.out.println("The file does not exist.");
        }
    }
}
```

✓ **Useful Methods:**

- `exists()` → Checks if file exists.
- `getName()` → Gets file name.
- `getAbsolutePath()` → Gets absolute path.
- `canRead()`, `canWrite()` → Checks read/write permissions.
- `length()` → Returns file size in bytes.

🔷 4. Deleting a File

Use `delete()` to remove a file.

✅ Example: Deleting a File

```
import java.io.File;

public class FileDeleteExample {
    public static void main(String[] args) {
        File file = new File("example.txt");

        if (file.delete()) {
            System.out.println("Deleted the file: " + file.getName());
        } else {
            System.out.println("Failed to delete the file.");
```

```
        }
    }
}
```

✓ **Returns `true` if successful, `false` if the file does not exist.**

🔷 5. Working with Directories

☑ Creating a Directory

```
import java.io.File;

public class DirectoryCreateExample {
    public static void main(String[] args) {
        File dir = new File("MyFolder");

        if (dir.mkdir()) {
            System.out.println("Directory created: " + dir.getName());
        } else {
            System.out.println("Failed to create directory.");
        }
    }
}
```

✓ `mkdir()` creates a single directory.
✓ `mkdirs()` creates **parent directories if missing**.

☑ Listing Files in a Directory

```
import java.io.File;

public class ListFilesExample {
    public static void main(String[] args) {
        File dir = new File("MyFolder");

        if (dir.isDirectory()) {
            String[] files = dir.list(); // List file names
            System.out.println("Files in directory:");
            for (String file : files) {
                System.out.println(file);
            }
        } else {
            System.out.println("Not a directory.");
        }
    }
}
```

✓ `list()` returns an **array of file names** in the directory.
✓ `listFiles()` returns **`File` objects** for each file.

🔷 6. Summary of File Class Methods

Method	Description
`createNewFile()`	Creates a new file if it doesn't exist.
`delete()`	Deletes the file/directory.
`exists()`	Checks if file exists.
`getName()`	Returns file name.
`getAbsolutePath()`	Returns absolute file path.
`length()`	Returns file size in bytes.
`canRead()`, `canWrite()`	Checks read/write permissions.
`mkdir()`	Creates a directory.
`list()`	Lists file names in a directory.

✨ Conclusion

✓ The `File` class **does not read/write file content** (use `FileReader`/`FileWriter` for that).
✓ It is used for **file management tasks** like creating, deleting, and checking file properties.
✓ It can **create and list directories** efficiently.

Would you like an example of **reading and writing file content**? 🚀

🔷 Reading & Writing Files in Java

In Java, we can read and write files using multiple classes from the `java.io` and `java.nio` packages. The most common methods include:

- **Reading a File**
  1. `FileReader` + `BufferedReader` (Efficient for text files)
  2. `FileInputStream` (For binary data)
  3. `Scanner` (For structured input)
  4. `Files.readAllLines()` (For small files)
- **Writing to a File**
  1. `FileWriter` + `BufferedWriter` (For text files)
  2. `FileOutputStream` (For binary data)
  3. `Files.write()` (For small files)

🔷 1. Writing to a File

✅ (Method 1) Using `FileWriter`

```
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteExample {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("output.txt"); // Open file in
write mode
            writer.write("Hello, this is a sample text.\n"); // Write content
            writer.write("This is another line.\n");
            writer.close(); // Close file
            System.out.println("File written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Creates or overwrites the file**
✓ Use `writer.append()` to add content without overwriting.

✅ (Method 2) Using `BufferedWriter` (Efficient)

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedFileWriteExample {
    public static void main(String[] args) {
        try {
            BufferedWriter writer = new BufferedWriter(new
FileWriter("output.txt"));
            writer.write("BufferedWriter is more efficient.\n");
            writer.write("It reduces the number of disk writes.\n");
            writer.close();
            System.out.println("File written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Faster than `FileWriter` alone** due to internal buffering.

✅ (Method 3) Using `Files.write()` (For Small Files)

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;
```

```
import java.util.Arrays;

public class FilesWriteExample {
    public static void main(String[] args) {
        try {
            Files.write(Paths.get("output.txt"), Arrays.asList("Line 1",
"Line 2", "Line 3"));
            System.out.println("File written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Best for small files**
✓ **Easier and cleaner** compared to `FileWriter`.

🔷 2. Reading from a File

✅ (Method 1) Using `FileReader` + `BufferedReader`

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileReadExample {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new
FileReader("output.txt"));
            String line;
            while ((line = reader.readLine()) != null) { // Read line by line
                System.out.println(line);
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Best for reading large text files efficiently**.

✅ (Method 2) Using `Scanner`

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ScannerFileReadExample {
    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("output.txt"));
            while (scanner.hasNextLine()) {
```

```
                System.out.println(scanner.nextLine());
            }
            scanner.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ Best for structured input (e.g., CSV, numbers, words).

☑ (Method 3) Using `Files.readAllLines()` (For Small Files)

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.List;

public class FilesReadExample {
    public static void main(String[] args) {
        try {
            List<String> lines = Files.readAllLines(Paths.get("output.txt"));
            for (String line : lines) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ Best for small files
✓ Reads the entire file into a `List<String>`

🔷 3. Appending to a File

```
import java.io.FileWriter;
import java.io.IOException;

public class FileAppendExample {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("output.txt", true); // true
enables append mode
            writer.write("Appending a new line.\n");
            writer.close();
            System.out.println("Content appended successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Does not overwrite existing content**

🔷 4. Reading & Writing Binary Files

☑ Writing Binary Data (`FileOutputStream`)

```
import java.io.FileOutputStream;
import java.io.IOException;

public class BinaryWriteExample {
    public static void main(String[] args) {
        try {
            FileOutputStream fos = new FileOutputStream("binary.dat");
            byte[] data = {65, 66, 67, 68}; // ASCII: A B C D
            fos.write(data);
            fos.close();
            System.out.println("Binary file written.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Best for writing images, audio, etc.**

☑ Reading Binary Data (`FileInputStream`)

```
import java.io.FileInputStream;
import java.io.IOException;

public class BinaryReadExample {
    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("binary.dat");
            int byteData;
            while ((byteData = fis.read()) != -1) {
                System.out.print((char) byteData + " ");
            }
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ **Reads binary data as bytes**.

🔷 5. Summary of File I/O Methods

Operation	Class Used	Best Use Case
Write to File	`FileWriter`	Simple text writing
Write to File	`BufferedWriter`	Large text writing
Write to File	`Files.write()`	Small file writing
Read from File	`FileReader + BufferedReader`	Efficient large file reading
Read from File	`Scanner`	Structured input (CSV, numbers)
Read from File	`Files.readAllLines()`	Simple small file reading
Binary Write	`FileOutputStream`	Writing images, PDFs, etc.
Binary Read	`FileInputStream`	Reading images, PDFs, etc.

✨ Conclusion

✓ Use **`BufferedReader` for reading** and **`BufferedWriter` for writing** for **best performance**.
✓ Use **`Files.readAllLines()`** and **`Files.write()`** for **small files**.
✓ Use **`Scanner`** for **structured input (CSV, numbers, words)**.
✓ Use **`FileOutputStream`** / **`FileInputStream`** for **binary data (images, PDFs, videos)**.

Would you like an example of **reading CSV files** or **handling file exceptions**? 🚀

🔷 `Console` Class in Java (`java.io.Console`)

The `Console` class in Java (`java.io.Console`) is used for **reading input and writing output** via the console. It is particularly useful for **reading passwords securely** without displaying them on the screen.

🚀 **Key Features of `Console` Class**:
✅ Reads user input **without echoing sensitive data** (e.g., passwords).
✅ Provides methods like `readLine()`, `readPassword()`, and `printf()`.
✅ Only works **in a real console** (not in IDEs like Eclipse or IntelliJ).

🔷 1. Getting a `Console` Instance

To use the `Console` class, call:

```
Console console = System.console();
```

🔴 **Important:** `System.console()` returns `null` if the program is run in an IDE (like Eclipse, IntelliJ) because most IDEs **don't support console input**.

🔷 2. Reading User Input (`readLine()`)

✅ Example: Getting Input from User

```
import java.io.Console;

public class ConsoleReadExample {
    public static void main(String[] args) {
        Console console = System.console();  // Get Console instance

        if (console == null) {
            System.out.println("No console available. Run the program in a
terminal.");
            return;
        }

        String name = console.readLine("Enter your name: ");
        System.out.println("Hello, " + name);
    }
}
```

✔ Uses `readLine()` to **read a string input from the user**.
✔ The prompt message `"Enter your name: "` is displayed before input.

🔷 3. Reading Password Securely (`readPassword()`)

✅ Example: Reading a Password

```
import java.io.Console;

public class ConsolePasswordExample {
    public static void main(String[] args) {
        Console console = System.console();

        if (console == null) {
            System.out.println("No console available. Run in a terminal.");
            return;
        }

        char[] passwordArray = console.readPassword("Enter your password: ");
        String password = new String(passwordArray); // Convert char[] to
String
```

```
        System.out.println("Password entered successfully.");
    }
}
```

✓ **Hides password characters** while typing.
✓ Stores the password as a `char[]` for security.

🔴 **Why `char[]` instead of `String`?**
Using `String` keeps the password in memory until garbage collected. `char[]` allows immediate **manual clearing** for security.

🔷 4. Writing Output Using `printf()`

☑ Example: Formatted Output

```
import java.io.Console;

public class ConsolePrintfExample {
    public static void main(String[] args) {
        Console console = System.console();

        if (console == null) {
            System.out.println("No console available. Run in a terminal.");
            return;
        }

        String name = console.readLine("Enter your name: ");
        int age = Integer.parseInt(console.readLine("Enter your age: "));

        console.printf("Hello %s, you are %d years old.\n", name, age);
    }
}
```

✓ `printf()` allows **formatted output** (like `System.out.printf()`).

🔷 5. Checking for a Console Before Using

Since `System.console()` may return `null`, always **check for availability** before using it:

```
Console console = System.console();
if (console == null) {
    System.out.println("No console available.");
    return;
}
```

🔴 **Why does it return `null`?**

- If the program is run inside **an IDE**.
- If the **standard input/output is redirected** (e.g., running in a script).

🔷 6. When to Use `Console` vs. `Scanner`?

Feature	Console Class (`java.io.Console`)	Scanner Class (`java.util.Scanner`)
Works in IDEs	❌ No	✅ Yes
Reads passwords securely	✅ Yes (`readPassword()`)	❌ No (password is visible)
Reads formatted input	✅ Yes (`printf()`)	✅ Yes (`nextInt()`, `nextDouble()`, etc.)
Handles exceptions	❌ No (throws `NullPointerException` if `console` is `null`)	✅ Yes (handles `InputMismatchException`)

✨ Conclusion

✔ `Console` is **useful for command-line applications** but **won't work inside IDEs**.
✔ Use **`readPassword()`** for securely handling passwords.
✔ Always **check if `System.console()` is `null`** before using it.
✔ If working inside an IDE, use **`Scanner` instead**.

Would you like an example of **reading numbers with Console**? 🚀

🔷 Serialization in Java (`java.io.Serializable`)

Serialization is the process of **converting an object into a byte stream** so that it can be **saved to a file, transmitted over a network, or stored in a database**.
The reverse process is called **Deserialization**, which reconstructs the object from the byte stream.

🔷 1. Why Use Serialization?

✅ Save and load objects from files (**persistent storage**).
✅ Send objects over a **network (RMI, sockets, etc.)**.
✅ Transfer objects between **processes or applications**.

🔷 2. Making a Class Serializable

To make a class serializable, it must **implement** the `Serializable` interface:

```
import java.io.Serializable;

class Person implements Serializable {
    private static final long serialVersionUID = 1L;  // (Recommended)

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

✓ `Serializable` is a **marker interface** (has no methods).
✓ `serialVersionUID` ensures **version compatibility**.

🔷 3. Writing (Serializing) an Object to a File

Use **`ObjectOutputStream`** and **`FileOutputStream`** to save an object:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializeExample {
    public static void main(String[] args) {
        try {
            Person person = new Person("Alice", 25);

            FileOutputStream fileOut = new FileOutputStream("person.ser");
            ObjectOutputStream objectOut = new ObjectOutputStream(fileOut);

            objectOut.writeObject(person); // Serialize object
            objectOut.close();
            fileOut.close();

            System.out.println("Object serialized successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✓ Saves the object to **`person.ser`**.
✓ `writeObject()` converts the object into bytes.

🔷 4. Reading (Deserializing) an Object from a File

Use **`ObjectInputStream`** and **`FileInputStream`** to restore an object:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializeExample {
    public static void main(String[] args) {
        try {
            FileInputStream fileIn = new FileInputStream("person.ser");
            ObjectInputStream objectIn = new ObjectInputStream(fileIn);

            Person person = (Person) objectIn.readObject(); // Deserialize
object
            objectIn.close();
            fileIn.close();

            System.out.println("Deserialized Object: " + person.name + ", " +
person.age);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

✓ Reads the object from **`person.ser`**.
✓ `readObject()` reconstructs the object from bytes.

🔷 5. Preventing Serialization (Using `transient`)

If you **don't want** a field to be serialized, use `transient`:

```
class Person implements Serializable {
    private static final long serialVersionUID = 1L;

    String name;
    transient int age; // Won't be serialized

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

✓ `age` will **not** be saved.
✓ When deserialized, `age` will be **reset to default** (`0` for `int`).

🔷 6. Custom Serialization (`writeObject()` & `readObject()`)

If `transient` fields need to be saved in a **custom way**, implement custom methods:

```
import java.io.*;

class Person implements Serializable {
    private static final long serialVersionUID = 1L;

    String name;
    transient int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();  // Serialize normal fields
        oos.writeInt(age); // Manually serialize transient field
    }

    private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {
        ois.defaultReadObject();  // Deserialize normal fields
        age = ois.readInt(); // Manually deserialize transient field
    }
}
```

✓ Preserves `age` even though it is `transient`.

🔷 7. serialVersionUID (Version Control)

```
private static final long serialVersionUID = 1L;
```

✓ Ensures **compatibility between different versions** of the class.
✓ If absent, Java generates one automatically, but changes in the class structure **break deserialization**.

🔷 8. Summary

Feature	Method
Enable Serialization	`implements Serializable`
Write Object to File	`ObjectOutputStream.writeObject()`
Read Object from File	`ObjectInputStream.readObject()`

Feature	Method
Ignore Field	`transient` keyword
Custom Serialization	`writeObject()` & `readObject()`
Version Control	`serialVersionUID`

✨ Conclusion

✓ **Serialization** is useful for saving and transferring objects.
✓ Use **`transient`** for fields you **don't want to serialize**.
✓ Implement **custom serialization** if needed.
✓ Use **`serialVersionUID`** for versioning.

Would you like an example of **serializing multiple objects** or **network-based serialization**? 🚀

UNIT - V

◈ GUI Programming with Swing in Java

Swing is Java's GUI (Graphical User Interface) toolkit that allows developers to create **rich, cross-platform desktop applications**. It is built on top of **AWT (Abstract Window Toolkit)** and provides **lightweight, flexible components**.

◈ 1. Key Features of Swing

✅ **Platform-Independent** – Works on all OS (Windows, macOS, Linux).
✅ **Lightweight** – Doesn't rely on OS-specific components.
✅ **MVC Architecture** – Separates data (Model), UI (View), and logic (Controller).
✅ **Event-Driven** – Uses **event listeners** for user interactions.
✅ **Customizable** – Supports **Look and Feel** (L&F) changes.

◈ 2. Creating a Basic Swing Window

To create a GUI in Swing, you typically **extend `JFrame`** and use components like `JButton`, `JLabel`, and `JTextField`.

✅ Example: Simple Swing Window

```
import javax.swing.*;

public class SimpleSwingApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My First Swing App");  // Create a window
        frame.setSize(400, 300);  // Set window size
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Close on
exit

        JLabel label = new JLabel("Hello, Swing!", SwingConstants.CENTER);
// Create label
        frame.add(label);  // Add label to the frame

        frame.setVisible(true);  // Make the window visible
    }
}
```

✔ **`JFrame`** – Main window.
✔ **`JLabel`** – Displays text.
✔ **`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`** – Ensures program exits when the window is closed.

🔷 3. Adding Buttons & Event Handling

Swing is **event-driven**, meaning actions (like button clicks) trigger event handlers.

✅ Example: Handling Button Click

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Click Example");
        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);  // Set layout to null (manual positioning)

        JButton button = new JButton("Click Me!");
        button.setBounds(150, 80, 100, 30); // Set button position & size

        JLabel label = new JLabel("Button not clicked.");
        label.setBounds(140, 30, 200, 30);

        // Add event listener
        button.addActionListener(e -> label.setText("Button Clicked!"));

        frame.add(button);
        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ **`JButton`** – Clickable button.
✓ **`ActionListener`** – Handles button click events.
✓ **Lambda (`e ->`)** – Shortens event handling.

🔷 4. Using Layout Managers

Layout managers automatically arrange components inside a container.

✅ Common Layout Managers

Layout Manager	Description
FlowLayout	Aligns components **left to right**
BorderLayout	Divides into **NORTH, SOUTH, EAST, WEST, CENTER**
GridLayout	Arranges in **rows and columns**

Layout Manager	Description
BoxLayout	Arranges in a **single row or column**

✅ Example: GridLayout

```
import javax.swing.*;
import java.awt.*;

public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new GridLayout(2, 2)); // 2 Rows, 2 Columns

        frame.add(new JButton("Button 1"));
        frame.add(new JButton("Button 2"));
        frame.add(new JButton("Button 3"));
        frame.add(new JButton("Button 4"));

        frame.setVisible(true);
    }
}
```

✓ **GridLayout(2,2)** → Creates a **2x2 grid**.
✓ Components automatically **resize** to fit the grid.

🔷 5. Adding Input Fields (`JTextField`)

✅ Example: Text Field & Button Interaction

```
import javax.swing.*;
import java.awt.event.*;

public class TextFieldExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("TextField Example");
        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        JTextField textField = new JTextField();
        textField.setBounds(100, 50, 200, 30);

        JButton button = new JButton("Show Text");
        button.setBounds(150, 100, 100, 30);

        JLabel label = new JLabel("Enter text above.");
        label.setBounds(140, 20, 200, 30);

        button.addActionListener(e -> label.setText("You entered: " +
textField.getText()));
```

```
        frame.add(textField);
        frame.add(button);
        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ **`JTextField`** – Allows text input.
✓ **`getText()`** – Retrieves text from input.

🔷 6. Dialog Boxes (`JOptionPane`)

Swing provides **built-in dialogs** for messages, inputs, and confirmations.

✅ Example: Message Dialog

```
JOptionPane.showMessageDialog(null, "Hello, Swing!");
```

✅ Example: Input Dialog

```
String name = JOptionPane.showInputDialog("Enter your name:");
```

✅ Example: Confirmation Dialog

```
int choice = JOptionPane.showConfirmDialog(null, "Do you want to continue?");
```

✓ Returns `YES_OPTION` (`0`), `NO_OPTION` (`1`), or `CANCEL_OPTION` (`2`).

🔷 7. Changing Look and Feel

Swing allows changing the **theme** (Look and Feel).

✅ Example: Set Windows Look and Feel

```
try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
} catch (Exception e) {
    e.printStackTrace();
}
```

✓ Uses the **native OS theme** (Windows, macOS, Linux).

🔷 8. Summary

Swing Component	Description
`JFrame`	Main window

Swing Component	Description
`JLabel`	Displays text
`JButton`	Clickable button
`JTextField`	Single-line text input
`JTextArea`	Multi-line text input
`JCheckBox`	Checkbox for selection
`JRadioButton`	Radio button (single selection)
`JComboBox`	Dropdown (Combo Box)
`JPanel`	Container for grouping components
`JMenuBar`	Menu bar with options
`JTable`	Displays tables of data
`JOptionPane`	Popup dialog boxes

✨ Conclusion

✓ Swing makes **GUI development easy** in Java.
✓ Uses **event-driven programming** for user interactions.
✓ Layout managers help organize components efficiently.
✓ `JOptionPane` simplifies dialogs.

Would you like an **example with menus (`JMenuBar`)** or a **more advanced GUI project**? 🚀

🔷 Introduction to GUI Programming with Swing in Java

🔷 What is Swing?

Swing is a part of **Java's Standard Library (`javax.swing` package)** that provides **Graphical User Interface (GUI)** components for building desktop applications. It is built on top of **AWT (Abstract Window Toolkit)** but is more powerful, flexible, and **lightweight**.

◈ Features of Swing

✅ **Lightweight** – Does not depend on native OS components.
✅ **Rich Set of Components** – Includes buttons, tables, trees, sliders, and more.
✅ **Pluggable Look and Feel** – Supports multiple themes (Windows, Metal, Nimbus, etc.).
✅ **Event-Driven Programming** – Uses Listeners to handle user interactions.
✅ **MVC Architecture** – Separates UI (View), Data (Model), and Logic (Controller).
✅ **Platform-Independent** – Runs on **Windows, macOS, Linux** without changes.

◈ Swing vs. AWT vs. JavaFX

Feature	Swing (`javax.swing`)	AWT (`java.awt`)	JavaFX (`javafx`)
Lightweight?	✅ Yes	❌ No (Heavy)	✅ Yes
More Components?	✅ Yes	❌ No	✅ Yes (Modern UI)
Customizable?	✅ Yes	❌ No	✅ Yes (CSS, XML)
Multi-threaded?	❌ No (Single-threaded)	❌ No	✅ Yes
Recommended for New Projects?	✅ Yes	❌ No	✅ Yes

◈ First Swing Program (Simple Window)

Every Swing application **must run on the Event Dispatch Thread (EDT)** using `SwingUtilities.invokeLater()`.

✅ Example: Creating a Basic Swing Window

```
import javax.swing.*;

public class SwingDemo {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame("Swing Demo"); // Create window
            frame.setSize(400, 300);  // Set size
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Exit on
close
            frame.setVisible(true);  // Show window
        });
    }
}
```

✓ **`JFrame`** – Creates the main window.
✓ **`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`** – Ensures program exits when the

window is closed.

✓ **`SwingUtilities.invokeLater()`** – Ensures thread safety.

◈ Common Swing Components

Component	Description
`JFrame`	Main window
`JPanel`	Container for grouping components
`JLabel`	Displays text or images
`JButton`	Clickable button
`JTextField`	Single-line text input
`JTextArea`	Multi-line text input
`JCheckBox`	Checkbox for multiple selections
`JRadioButton`	Radio button (single selection)
`JComboBox`	Dropdown (Combo Box)
`JList`	List of items
`JTable`	Table to display data
`JScrollPane`	Scrollable component
`JMenuBar`	Menu bar with options

◈ Event Handling in Swing

Swing applications are **event-driven**. User actions (clicking a button, typing in a text box) trigger **events**, which are handled using **listeners**.

☑ Example: Handling Button Click Event

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Example");
```

```
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null); // Manual positioning

        JButton button = new JButton("Click Me");
        button.setBounds(100, 80, 100, 30);

        JLabel label = new JLabel("Button not clicked.");
        label.setBounds(90, 30, 200, 30);

        // Event handling using lambda expression
        button.addActionListener(e -> label.setText("Button Clicked!"));

        frame.add(button);
        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ **`ActionListener`** – Detects button clicks.
✓ **Lambda (`e -> {}`)** – Shorter way to handle events.

◈ Layout Managers in Swing

Swing provides different **layout managers** to arrange components.

Layout Manager	Description
FlowLayout	Arranges components **left to right**
BorderLayout	Divides into **NORTH, SOUTH, EAST, WEST, CENTER**
GridLayout	Arranges components in **rows and columns**
BoxLayout	Arranges components in a **single row or column**

☑ Example: GridLayout

```
import javax.swing.*;
import java.awt.*;

public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Grid Layout Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new GridLayout(2, 2)); // 2 Rows, 2 Columns

        frame.add(new JButton("Button 1"));
        frame.add(new JButton("Button 2"));
        frame.add(new JButton("Button 3"));
```

```
        frame.add(new JButton("Button 4"));

        frame.setVisible(true);
    }
}
```

✓ **GridLayout(2,2)** → Creates a **2x2 grid**.
✓ Components automatically **resize** to fit the grid.

◈ Swing Dialog Boxes (`JOptionPane`)

Swing provides built-in dialog boxes for **messages, inputs, and confirmations**.

✅ Example: Message Dialog

```
JOptionPane.showMessageDialog(null, "Hello, Swing!");
```

✅ Example: Input Dialog

```
String name = JOptionPane.showInputDialog("Enter your name:");
```

✅ Example: Confirmation Dialog

```
int choice = JOptionPane.showConfirmDialog(null, "Do you want to continue?");
if (choice == JOptionPane.YES_OPTION) {
    System.out.println("User clicked YES");
}
```

✓ Returns `YES_OPTION` (`0`), `NO_OPTION` (`1`), or `CANCEL_OPTION` (`2`).

◈ Changing Look and Feel

Swing allows customizing the **Look and Feel (L&F)**.

✅ Example: Set Windows Look and Feel

```
try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
} catch (Exception e) {
    e.printStackTrace();
}
```

✓ Uses the **native OS theme** (Windows, macOS, Linux).

◈ Summary

Feature	Description
Swing Package	`javax.swing`

Feature	Description
Main Window	`JFrame`
Event Handling	`ActionListener`, `KeyListener`, etc.
Layout Managers	`FlowLayout`, `BorderLayout`, `GridLayout`, etc.
Dialogs	`JOptionPane`
Custom Look & Feel	`UIManager.setLookAndFeel()`

✨ Conclusion

✓ **Swing** is Java's standard GUI library for desktop applications.
✓ Uses **event-driven programming** to handle user interactions.
✓ Provides **rich components and layouts** for UI design.
✓ Supports **cross-platform** and **custom themes**.

Would you like an **example with menus (`JMenuBar`)** or a **complete GUI project**? 🚀

🔷 MVC Architecture in Java (Model-View-Controller)

🔷 What is MVC?

MVC (**Model-View-Controller**) is a **design pattern** used in software development to separate an application's logic into **three interconnected components**:

1. **Model** – Represents **data & business logic**
2. **View** – Handles the **UI (User Interface)**
3. **Controller** – Manages **user interactions and updates Model/View**

This **separation of concerns** improves **code maintainability, scalability, and testability**.

🔷 Components of MVC

Component	Description	Example in Java
Model	Manages **data, logic, and rules**	A class that represents a **Student, Product, or User**
View	Displays **UI elements** to the user	`JFrame`, `JLabel`, `JButton` in **Swing**

Component	Description	Example in Java
Controller	Handles **user inputs** and updates the Model/View	`ActionListener` for button clicks

---

🔷 How MVC Works

1. **User interacts** with the **View** (clicks a button, enters text).
2. **Controller processes** the input and **updates Model**.
3. **Model updates View**, reflecting **new data on UI**.

This **ensures separation of concerns** where:
✓ **Model** doesn't depend on UI.
✓ **View** doesn't handle business logic.
✓ **Controller** connects Model & View.

🔷 Implementing MVC in Java (Swing Example)

✅ Step 1: Create the Model (Data & Logic)

```
// Model Class (Holds Data)
public class Student {
    private String name;
    private int age;

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

    public String getName() { return name; }
    public int getAge() { return age; }

    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
}
```

✓ Holds **data** (student name, age).
✓ Provides **getters & setters**.

✅ Step 2: Create the View (User Interface)

```
import javax.swing.*;

public class StudentView {
    private JFrame frame;
    private JTextField nameField, ageField;
```

```
    private JButton updateButton;

    public StudentView() {
        frame = new JFrame("Student MVC Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        JLabel nameLabel = new JLabel("Name:");
        nameLabel.setBounds(30, 30, 80, 25);
        frame.add(nameLabel);

        nameField = new JTextField();
        nameField.setBounds(100, 30, 150, 25);
        frame.add(nameField);

        JLabel ageLabel = new JLabel("Age:");
        ageLabel.setBounds(30, 70, 80, 25);
        frame.add(ageLabel);

        ageField = new JTextField();
        ageField.setBounds(100, 70, 150, 25);
        frame.add(ageField);

        updateButton = new JButton("Update");
        updateButton.setBounds(100, 110, 100, 30);
        frame.add(updateButton);

        frame.setVisible(true);
    }

    public String getNameInput() { return nameField.getText(); }
    public int getAgeInput() { return Integer.parseInt(ageField.getText()); }
    public JButton getUpdateButton() { return updateButton; }

    public void setStudentData(String name, int age) {
        nameField.setText(name);
        ageField.setText(String.valueOf(age));
    }
}
```

✓ **Swing UI** (`JFrame`, `JLabel`, `JTextField`, `JButton`).
✓ Methods to **get user input** and **set UI data**.
✓ The **button click event is handled in the Controller**.

✅ Step 3: Create the Controller (Handles User Input & Updates Model)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view) {
        this.model = model;
```

```
        this.view = view;

        // Add button click listener
        this.view.getUpdateButton().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                model.setName(view.getNameInput());
                model.setAge(view.getAgeInput());
                view.setStudentData(model.getName(), model.getAge());
            }
        });

        // Display initial data
        view.setStudentData(model.getName(), model.getAge());
    }
}
```

✓ **Handles button click events** (`ActionListener`).
✓ Updates **Model (data) when user enters new values**.
✓ Refreshes **View with updated data**.

☑ Step 4: Run the MVC Application

```
public class MVCDemo {
    public static void main(String[] args) {
        Student model = new Student("Alice", 20);  // Create Model
        StudentView view = new StudentView();      // Create View
        new StudentController(model, view);        // Connect Model & View
via Controller
    }
}
```

✓ Creates **Model, View, and Controller objects**.
✓ Links everything together.

◈ Output of the MVC Application

📄 **A Swing window appears with:**
✓ **Text fields** for **name and age**.
✓ **An "Update" button**.
✓ When the user **enters new data and clicks "Update"**,
🔹 The **Model (Student)** is updated,
🔹 The **View refreshes** with new data.

◈ Advantages of MVC in Java

✅ **Separation of Concerns** – Code is modular and **easy to maintain**.
✅ **Scalability** – Easy to add new **features** (e.g., database, API).
✅ **Testability** – Model and Controller can be tested **independently**.
✅ **Code Reusability** – View and Model **can be reused** in different applications.

◈ MVC vs. Other Architectures

Architecture	Description	Used In
MVC	Separates Model, View, and Controller	**Desktop & Web Apps**
MVVM (Model-View-ViewModel)	Used in **JavaFX, Android**, **Data Binding**	**Modern UI Frameworks**
MVP (Model-View-Presenter)	Presenter handles business logic & updates View	**Android, Swing**

✨ Conclusion

✓ **MVC** is a widely used **design pattern** in Java applications.
✓ It helps keep the code **clean, maintainable, and scalable**.
✓ **Swing, JavaFX, and Web Apps** benefit from MVC.
✓ Ideal for applications requiring **User Interface (UI) & Business Logic separation**.

Would you like an example using **JTable for displaying data** or **database integration (JDBC)** with MVC? 🚀

◈ GUI Programming with Swing Components in Java

◈ Introduction

Swing is a Java GUI framework (`javax.swing` package) that provides a rich set of **components** to build desktop applications. These components include **buttons, text fields, labels, menus, tables**, and more.

◈ Swing Components Overview

Swing components are **lightweight**, meaning they do not rely on the native OS GUI. Some commonly used Swing components:

Component	Description
`JFrame`	Main window (top-level container)
`JPanel`	A container for grouping components
`JLabel`	Displays text or images
`JButton`	Clickable button
`JTextField`	Single-line text input
`JTextArea`	Multi-line text input
`JPasswordField`	Secure password input
`JCheckBox`	Checkbox for multiple selections
`JRadioButton`	Radio button (single selection)
`JComboBox`	Dropdown (Combo Box)
`JList`	List of items
`JTable`	Displays tabular data
`JMenuBar`	Menu bar for application menus
`JScrollPane`	Adds scrollbars to components
`JProgressBar`	Displays progress of a task
`JSlider`	Allows users to select values from a range
`JDialog`	Popup dialog box

◈ Swing Components with Examples

Let’s explore Swing components with Java examples.

✅ 1. JFrame (Main Window)

`JFrame` is the **main window** for Swing applications.

```
import javax.swing.*;

public class JFrameExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JFrame Example"); // Create Window
        frame.setSize(400, 300);  // Set window size
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Close
operation
        frame.setVisible(true);  // Show the window
    }
}
```

✓ `JFrame.EXIT_ON_CLOSE` ensures the program **closes on exit**.

✅ 2. JLabel (Display Text or Images)

```
import javax.swing.*;

public class JLabelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JLabel Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        JLabel label = new JLabel("Hello, Swing!");
        label.setBounds(100, 50, 150, 30); // Position

        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ `JLabel` is **non-editable** and used for **display purposes**.

✅ 3. JButton (Button Click Event)

```
import javax.swing.*;
import java.awt.event.*;

public class JButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JButton Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
```

```
        JButton button = new JButton("Click Me");
        button.setBounds(100, 50, 100, 40);

        JLabel label = new JLabel("Button not clicked");
        label.setBounds(90, 100, 200, 30);

        button.addActionListener(e -> label.setText("Button Clicked!")); //
Event Handling

        frame.add(button);
        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ `ActionListener` handles **button clicks**.

☑ 4. JTextField (Single-Line Text Input)

```
import javax.swing.*;

public class JTextFieldExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTextField Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        JLabel label = new JLabel("Enter Name:");
        label.setBounds(20, 30, 100, 30);

        JTextField textField = new JTextField();
        textField.setBounds(120, 30, 150, 30);

        frame.add(label);
        frame.add(textField);
        frame.setVisible(true);
    }
}
```

✓ `JTextField` allows **user input**.

☑ 5. JCheckBox (Multiple Selections)

```
import javax.swing.*;

public class JCheckBoxExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JCheckBox Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        JCheckBox checkBox1 = new JCheckBox("Option 1");
```

```
        checkBox1.setBounds(50, 50, 100, 30);

        JCheckBox checkBox2 = new JCheckBox("Option 2");
        checkBox2.setBounds(50, 80, 100, 30);

        frame.add(checkBox1);
        frame.add(checkBox2);
        frame.setVisible(true);
    }
}
```

✓ `JCheckBox` allows **multiple selections**.

☑ 6. JRadioButton (Single Selection)

```
import javax.swing.*;

public class JRadioButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JRadioButton Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        JRadioButton radio1 = new JRadioButton("Male");
        radio1.setBounds(50, 50, 100, 30);

        JRadioButton radio2 = new JRadioButton("Female");
        radio2.setBounds(50, 80, 100, 30);

        ButtonGroup group = new ButtonGroup(); // Ensures only one is
selected
        group.add(radio1);
        group.add(radio2);

        frame.add(radio1);
        frame.add(radio2);
        frame.setVisible(true);
    }
}
```

✓ `ButtonGroup` ensures **only one radio button can be selected**.

☑ 7. JComboBox (Dropdown)

```
import javax.swing.*;

public class JComboBoxExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JComboBox Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
```

```
        String[] choices = {"Java", "Python", "C++"};
        JComboBox<String> comboBox = new JComboBox<>(choices);
        comboBox.setBounds(100, 50, 100, 30);

        frame.add(comboBox);
        frame.setVisible(true);
    }
}
```

✓ `JComboBox` provides a **dropdown selection**.

☑ 8. JTable (Table for Data Display)

```
import javax.swing.*;

public class JTableExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTable Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        String[][] data = { {"1", "Alice", "20"}, {"2", "Bob", "22"} };
        String[] columns = {"ID", "Name", "Age"};

        JTable table = new JTable(data, columns);
        JScrollPane scrollPane = new JScrollPane(table);
        frame.add(scrollPane);

        frame.setVisible(true);
    }
}
```

✓ `JTable` is used for **displaying tabular data**.

◈ Summary

Component	Use Case
`JFrame`	Main window
`JButton`	Button clicks
`JLabel`	Display text/images
`JTextField`	Single-line input
`JCheckBox`	Multiple selections
`JRadioButton`	Single selection

Component	Use Case
`JComboBox`	Dropdown menu
`JTable`	Display table data

Would you like to see **event handling** or **layout management** next? 🚀

🔷 GUI Programming with Swing Containers in Java

🔷 Introduction

In **Swing**, **containers** are components that **hold and organize other components** (buttons, labels, text fields, etc.). Containers help in **structuring the GUI** by defining how components are placed and interact.

Swing provides **three main types of containers**:

1. **Top-Level Containers**: JFrame, JDialog, JApplet
2. **Intermediate Containers**: JPanel, JScrollPane, JSplitPane
3. **Lightweight Containers**: JLayeredPane, JRootPane

🔷 1. Top-Level Containers

These are **self-contained windows** that act as the **main application windows**.

☑️ JFrame (Main Window)

`JFrame` is the **main window** that holds other components.

```
import javax.swing.*;

public class JFrameExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JFrame Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

✓ **`EXIT_ON_CLOSE`** ensures the program exits when the window is closed.

☑ JDialog (Popup Window)

`JDialog` is used for **pop-up windows**, such as confirmation dialogs.

```
import javax.swing.*;

public class JDialogExample {
    public static void main(String[] args) {
        JDialog dialog = new JDialog();
        dialog.setTitle("JDialog Example");
        dialog.setSize(300, 150);
        dialog.setModal(true); // Blocks interaction with the main window
        dialog.add(new JLabel("This is a dialog box."));
        dialog.setVisible(true);
    }
}
```

✓ `setModal(true)` makes the dialog **block user interaction** with other windows until it is closed.

☑ JApplet (Deprecated, Used for Web Applications)

`JApplet` was used for **Java Applets**, but it is now deprecated.

◈ 2. Intermediate Containers

These **group components together** and help in organizing the UI.

☑ JPanel (Flexible Container)

`JPanel` is a **general-purpose container** used to group components.

```
import javax.swing.*;

public class JPanelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JPanel Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel(); // Create panel
        JLabel label = new JLabel("Hello from JPanel!");
        JButton button = new JButton("Click Me");

        panel.add(label);
        panel.add(button);

        frame.add(panel);
        frame.setVisible(true);
```

```
    }
}
```

✓ `JPanel` is used for **grouping components together** inside a window.

☑ JScrollPane (Adding Scrollbars)

`JScrollPane` is used to **add scrollbars** to components.

```
import javax.swing.*;

public class JScrollPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JScrollPane Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTextArea textArea = new JTextArea(10, 30); // Large text area
        JScrollPane scrollPane = new JScrollPane(textArea); // Add scrolling

        frame.add(scrollPane);
        frame.setVisible(true);
    }
}
```

✓ `JScrollPane` is useful for **scrolling text areas, tables, or large components**.

☑ JSplitPane (Divides Window into Two)

`JSplitPane` splits a window into **two resizable areas**.

```
import javax.swing.*;

public class JSplitPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JSplitPane Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton leftButton = new JButton("Left Side");
        JButton rightButton = new JButton("Right Side");

        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
leftButton, rightButton);
        splitPane.setDividerLocation(150);

        frame.add(splitPane);
        frame.setVisible(true);
    }
}
```

✓ `JSplitPane` allows **resizing two sections** of a window.

🔷 3. Lightweight Containers

These provide **advanced customization options**.

☑️ JLayeredPane (Stacking Components)

`JLayeredPane` allows **stacking multiple components on different layers**.

```
import javax.swing.*;

public class JLayeredPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JLayeredPane Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLayeredPane layeredPane = new JLayeredPane();

        JLabel label1 = new JLabel("Back Layer");
        label1.setBounds(50, 50, 100, 30);

        JLabel label2 = new JLabel("Front Layer");
        label2.setBounds(70, 70, 100, 30);

        layeredPane.add(label1, JLayeredPane.DEFAULT_LAYER);
        layeredPane.add(label2, JLayeredPane.PALETTE_LAYER);

        frame.add(layeredPane);
        frame.setVisible(true);
    }
}
```

✓ `JLayeredPane` **controls component stacking order**.

☑️ JRootPane (Base Pane for JFrame, JDialog)

`JRootPane` is the **base container** that holds `JMenuBar`, content pane, and glass pane.

```
import javax.swing.*;

public class JRootPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JRootPane Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JRootPane rootPane = frame.getRootPane();
        rootPane.getContentPane().add(new JLabel("Hello, RootPane!"));
```

```
        frame.setVisible(true);
    }
}
```

✓ `JRootPane` is used internally by `JFrame` and `JDialog`.

◈ Summary of Swing Containers

Container	Type	Usage
JFrame	Top-Level	Main application window
JDialog	Top-Level	Popup dialog windows
JPanel	Intermediate	Groups components together
JScrollPane	Intermediate	Adds scrollbars to components
JSplitPane	Intermediate	Splits the window into two resizable areas
JLayeredPane	Lightweight	Stacks components in layers
JRootPane	Lightweight	Base pane for JFrame, JDialog

◈ Conclusion

- **Top-Level Containers**: Main windows like `JFrame`, `JDialog`.
- **Intermediate Containers**: Used for organizing components (`JPanel`, `JScrollPane`).
- **Lightweight Containers**: Provide additional customization (`JLayeredPane`, `JRootPane`).

Would you like to see **Layout Managers** next? 🚀

◈ Understanding Layout Managers in Java Swing

◈ What is a Layout Manager?

A **Layout Manager** in Java **automatically arranges components** (buttons, text fields, labels, etc.) in a container (like `JFrame` or `JPanel`).
Instead of manually setting positions, a Layout Manager handles the placement and resizing of components dynamically.

☑ Why Use Layout Managers?

- **Responsive UI:** Components adjust automatically when resizing the window.

- **Platform Independence:** Works on different screen resolutions.
- **Ease of Development:** No need for manual positioning (`setBounds()`).

◈ Types of Layout Managers

Java provides several built-in Layout Managers:

Layout Manager	Usage
FlowLayout	Aligns components in a row, wrapping when needed.
BorderLayout	Divides container into five regions (North, South, East, West, Center).
GridLayout	Organizes components in a grid (rows & columns).
BoxLayout	Arranges components in a single row or column.
GridBagLayout	Flexible grid-based layout with precise control.
CardLayout	Switches between multiple layouts like a deck of cards.

◈ 1. FlowLayout (Default for JPanel)

- Arranges components **left to right**, wrapping to the next row if needed.
- Alignment: **LEFT, CENTER (default), RIGHT**

```
import javax.swing.*;
import java.awt.*;

public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FlowLayout Example");
        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));
        panel.add(new JButton("Button 1"));
        panel.add(new JButton("Button 2"));
        panel.add(new JButton("Button 3"));

        frame.add(panel);
        frame.setVisible(true);
    }
}
```

✓ **Components are arranged horizontally** and wrap when necessary.

🔷 2. BorderLayout (Default for JFrame)

- Divides the container into **five regions**:
  🔷 **NORTH**, **SOUTH**, **EAST**, **WEST**, **CENTER**
- Each region can hold **only one** component.

```
import javax.swing.*;
import java.awt.*;

public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BorderLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new BorderLayout());

        frame.add(new JButton("NORTH"), BorderLayout.NORTH);
        frame.add(new JButton("SOUTH"), BorderLayout.SOUTH);
        frame.add(new JButton("EAST"), BorderLayout.EAST);
        frame.add(new JButton("WEST"), BorderLayout.WEST);
        frame.add(new JButton("CENTER"), BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

✓ Best for structuring UI into different sections.

🔷 3. GridLayout (Equal-Sized Grid)

- Arranges components in a **table-like format** (rows × columns).
- Components have **equal size**.

```
import javax.swing.*;
import java.awt.*;

public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new GridLayout(2, 3, 10, 10)); // 2 rows, 3 columns,
10px spacing

        for (int i = 1; i <= 6; i++) {
            frame.add(new JButton("Button " + i));
        }

        frame.setVisible(true);
    }
}
```

✓ **Useful for calculators, forms, and tables.**

🔷 4. BoxLayout (Row or Column Alignment)

- Arranges components **horizontally or vertically**.
- Components keep their **preferred sizes**.

```
import javax.swing.*;
import java.awt.*;

public class BoxLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BoxLayout Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS)); // Vertical
layout

        panel.add(new JButton("Button 1"));
        panel.add(Box.createVerticalStrut(10)); // Add spacing
        panel.add(new JButton("Button 2"));

        frame.add(panel);
        frame.setVisible(true);
    }
}
```

✓ **Great for vertical menus or aligned toolbars.**

🔷 5. GridBagLayout (Advanced Grid Layout)

- Like `GridLayout`, but with **flexible row & column sizes**.
- Allows **precise control** of component placement.

```
import javax.swing.*;
import java.awt.*;

public class GridBagLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridBagLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();

        gbc.gridx = 0; gbc.gridy = 0;
        frame.add(new JButton("Button 1"), gbc);
```

```
        gbc.gridx = 1; gbc.gridy = 0;
        frame.add(new JButton("Button 2"), gbc);

        gbc.gridx = 0; gbc.gridy = 1;
        gbc.gridwidth = 2; // Span two columns
        frame.add(new JButton("Wide Button"), gbc);

        frame.setVisible(true);
    }
}
```

✓ Best for complex forms and flexible layouts.

🔷 6. CardLayout (Switching Between Panels)

- Allows **switching between multiple panels**.
- Useful for **wizards, forms, and tab-like structures**.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CardLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("CardLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel cardPanel = new JPanel(new CardLayout());
        CardLayout cl = (CardLayout) cardPanel.getLayout();

        JPanel panel1 = new JPanel();
        panel1.add(new JLabel("Panel 1"));
        JPanel panel2 = new JPanel();
        panel2.add(new JLabel("Panel 2"));

        cardPanel.add(panel1, "Card1");
        cardPanel.add(panel2, "Card2");

        JButton switchButton = new JButton("Switch");
        switchButton.addActionListener(e -> cl.next(cardPanel));

        frame.setLayout(new BorderLayout());
        frame.add(cardPanel, BorderLayout.CENTER);
        frame.add(switchButton, BorderLayout.SOUTH);

        frame.setVisible(true);
    }
}
```

✓ Best for multi-step forms or tab-like navigation.

◈ Summary of Layout Managers

Layout Manager	Best Use Case
FlowLayout	Simple alignment of buttons, labels, etc.
BorderLayout	Dividing window into sections (e.g., header, footer).
GridLayout	Creating tables, grids, calculators.
BoxLayout	Vertical or horizontal menus.
GridBagLayout	Complex forms with flexible layout.
CardLayout	Multi-step forms, tabs, wizards.

◈ Conclusion

- **Use `FlowLayout` for simple alignment**.
- **Use `BorderLayout` for structured UIs**.
- **Use `GridLayout` for equal-sized elements**.
- **Use `GridBagLayout` for advanced layouts**.
- **Use `CardLayout` for multi-panel navigation**.

Would you like an example of a **custom Layout Manager** next? 🚀

FlowLayout, BorderLayout, GridLayout, CardLayout, and GridBagLayout in Java Swing, with examples.

1️⃣ FlowLayout (Default for JPanel)

- Arranges components **horizontally**.
- Wraps to the next row when needed.
- Aligns components: **LEFT, CENTER (default), RIGHT**.
- Spacing can be set between components.

✅ Example:

```
import javax.swing.*;
import java.awt.*;

public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FlowLayout Example");
        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));
// Alignment, hGap, vGap

        panel.add(new JButton("Button 1"));
        panel.add(new JButton("Button 2"));
        panel.add(new JButton("Button 3"));

        frame.add(panel);
        frame.setVisible(true);
    }
}
```

✓ Best for simple UI elements like toolbars or button groups.

---

2 BorderLayout (Default for JFrame)

- Divides the container into **five regions**:
  - **NORTH**, **SOUTH**, **EAST**, **WEST**, **CENTER**
- Each region can **hold only one component**.
- **Center region expands automatically**.

☑ Example:

```
import javax.swing.*;
import java.awt.*;

public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BorderLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new BorderLayout());

        frame.add(new JButton("NORTH"), BorderLayout.NORTH);
        frame.add(new JButton("SOUTH"), BorderLayout.SOUTH);
        frame.add(new JButton("EAST"), BorderLayout.EAST);
        frame.add(new JButton("WEST"), BorderLayout.WEST);
        frame.add(new JButton("CENTER"), BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

✓ Best for structuring UI into header, footer, sidebar, and main content.

---

3 GridLayout

- Arranges components in a **table-like structure** (rows × columns).
- All components have **equal size**.
- Spaces between rows and columns can be set.

✅ Example:

```
import javax.swing.*;
import java.awt.*;

public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new GridLayout(2, 3, 10, 10)); // 2 rows, 3 columns,
spacing 10px

        for (int i = 1; i <= 6; i++) {
            frame.add(new JButton("Button " + i));
        }

        frame.setVisible(true);
    }
}
```

✓ Best for calculators, forms, and tables.

4️⃣ CardLayout

- Allows switching between **multiple panels** (like a deck of cards).
- Used in **wizards, forms, or tab-like interfaces**.

✅ Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CardLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("CardLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel cardPanel = new JPanel(new CardLayout());
        CardLayout cl = (CardLayout) cardPanel.getLayout();

        JPanel panel1 = new JPanel();
        panel1.add(new JLabel("Panel 1"));
        JPanel panel2 = new JPanel();
        panel2.add(new JLabel("Panel 2"));

        cardPanel.add(panel1, "Card1");
        cardPanel.add(panel2, "Card2");

        JButton switchButton = new JButton("Switch");
        switchButton.addActionListener(e -> cl.next(cardPanel));

        frame.setLayout(new BorderLayout());
```

```
        frame.add(cardPanel, BorderLayout.CENTER);
        frame.add(switchButton, BorderLayout.SOUTH);

        frame.setVisible(true);
    }
}
```

✓ Best for multi-step forms, settings pages, or navigation menus.

---

5 GridBagLayout (Advanced Grid Layout)

- A **flexible** layout that allows components to **span multiple rows/columns**.
- Used for **complex UI designs**.
- Uses **GridBagConstraints** for positioning.

☑ Example:

```
import javax.swing.*;
import java.awt.*;

public class GridBagLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridBagLayout Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5); // Padding between components

        gbc.gridx = 0; gbc.gridy = 0;
        frame.add(new JButton("Button 1"), gbc);

        gbc.gridx = 1; gbc.gridy = 0;
        frame.add(new JButton("Button 2"), gbc);

        gbc.gridx = 0; gbc.gridy = 1;
        gbc.gridwidth = 2; // Span two columns
        frame.add(new JButton("Wide Button"), gbc);

        frame.setVisible(true);
    }
}
```

✓ Best for complex forms, dashboards, or mixed layouts.

◈ Comparison of Layout Managers

Layout Manager	Best Use Case
FlowLayout	Simple UI elements like toolbars, buttons.

Layout Manager	Best Use Case
BorderLayout	Structuring UI (header, sidebar, main content).
GridLayout	Creating tables, grids, or calculators.
CardLayout	Multi-step forms, tabs, wizards.
GridBagLayout	Complex layouts with precise positioning.

◈ Conclusion

- **Use `FlowLayout`** for simple alignments.
- **Use `BorderLayout`** to organize UI into sections.
- **Use `GridLayout`** for equal-sized elements.
- **Use `CardLayout`** for switching between multiple panels.
- **Use `GridBagLayout`** for complex, flexible UIs.

Would you like an example combining multiple layouts in a single UI? 🚀

Event Handling in Java (Swing & AWT)

Java **event handling** allows programs to respond to user interactions (clicks, key presses, etc.). It follows the **Event Delegation Model**, where an event source generates an event, and an event listener handles it.

◈ Event Handling Model in Java

Java's event handling follows three main steps:

1. **Event Source** → Component (e.g., `JButton`, `JTextField`) generates an event.
2. **Event Object** → An instance of `ActionEvent`, `MouseEvent`, `KeyEvent`, etc.
3. **Event Listener** → Interface that defines methods to handle events.

Component (Source)	Event Class	Listener Interface
Button (`JButton`)	`ActionEvent`	`ActionListener`
Mouse (`JFrame`, `JLabel`)	`MouseEvent`	`MouseListener`, `MouseMotionListener`

Component (Source)	Event Class	Listener Interface
Keyboard (`JTextField`)	`KeyEvent`	`KeyListener`
Window (`JFrame`)	`WindowEvent`	`WindowListener`

🔷 Handling Button Clicks (`ActionListener`)

✅ Example:

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonClickExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Click Event");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("Click Me");

        // Adding ActionListener
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame,
"Button Clicked!"));

        frame.add(button);
        frame.setVisible(true);
    }
}
```

✓ **When clicked, the button shows a message box.**
✓ Uses **Lambda Expression** (`e -> {}`) for concise event handling.

🔷 Handling Mouse Events (`MouseListener`)

- Detects mouse clicks, enters, exits, presses, and releases.

✅ Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Mouse Event Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Hover or Click Me",
SwingConstants.CENTER);
        frame.add(label);
```

```
        label.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                label.setText("Mouse Clicked!");
            }

            @Override
            public void mouseEntered(MouseEvent e) {
                label.setForeground(Color.RED);
            }

            @Override
            public void mouseExited(MouseEvent e) {
                label.setForeground(Color.BLACK);
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **Detects mouse clicks, hover (`mouseEntered`), and exit (`mouseExited`).**
✓ Uses **`MouseAdapter`** to avoid implementing all methods.

🔷 Handling Keyboard Events (`KeyListener`)

- Detects key presses, releases, and typing.

☑ Example:

```
import javax.swing.*;
import java.awt.event.*;

public class KeyEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Key Event Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTextField textField = new JTextField();
        JLabel label = new JLabel("Type something...");

        textField.addKeyListener(new KeyAdapter() {
            @Override
            public void keyTyped(KeyEvent e) {
                label.setText("Typed: " + e.getKeyChar());
            }
        });

        frame.add(textField, "North");
        frame.add(label, "South");
        frame.setVisible(true);
    }
}
```

✓ **Updates label when a key is typed in the text field.**
✓ Uses **`KeyAdapter`** for simplicity.

◈ Handling Window Events (`WindowListener`)

- Detects window open, close, minimize, etc.

☑ Example:

```
import javax.swing.*;
import java.awt.event.*;

public class WindowEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Window Event Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                int confirm = JOptionPane.showConfirmDialog(frame, "Are you
sure you want to exit?");
                if (confirm == JOptionPane.YES_OPTION) {
                    frame.dispose();
                }
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **Asks for confirmation before closing the window.**
✓ Uses **`WindowAdapter`** for cleaner code.

◈ Handling Multiple Events with One Listener

We can use a **single listener** to handle multiple events.

☑ Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MultiEventExample implements ActionListener {
    JButton button1, button2;
    JLabel label;

    public MultiEventExample() {
```

```
        JFrame frame = new JFrame("Multiple Events");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        button1 = new JButton("Hello");
        button2 = new JButton("Goodbye");
        label = new JLabel("Click a button", SwingConstants.CENTER);

        button1.addActionListener(this);
        button2.addActionListener(this);

        frame.setLayout(new FlowLayout());
        frame.add(button1);
        frame.add(button2);
        frame.add(label);

        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == button1) {
            label.setText("Hello!");
        } else if (e.getSource() == button2) {
            label.setText("Goodbye!");
        }
    }

    public static void main(String[] args) {
        new MultiEventExample();
    }
}
```

✓ **Handles multiple buttons in a single `ActionListener`.**
✓ Uses `getSource()` to identify the clicked button.

◈ Summary: Event Handling Interfaces

Event Type	Listener Interface	Method to Implement
Button Click	`ActionListener`	`actionPerformed(ActionEvent e)`
Mouse Events	`MouseListener`	`mouseClicked`, `mouseEntered`, `mouseExited`
Key Events	`KeyListener`	`keyTyped`, `keyPressed`, `keyReleased`
Window Events	`WindowListener`	`windowClosing`, `windowOpened`, etc.

◈ Conclusion

- **Use `ActionListener`** for button clicks.

- **Use `MouseListener`** for hover and click events.
- **Use `KeyListener`** to detect key presses.
- **Use `WindowListener`** for window actions.

Would you like a **custom event example** next? 🚀

Delegation Event Model in Java

The **Delegation Event Model** is Java's approach to handling events. It was introduced in **Java 1.1** and is used in **Swing** and **AWT** to handle user interactions efficiently.

🔷 What is the Delegation Event Model?

In this model:

1. **An event source generates an event** (e.g., a button click).
2. **A listener waits for the event** and processes it when triggered.
3. **The listener is registered with the source** using an **event listener interface**.

🔷 Components of the Delegation Event Model

1. **Event Source** → The component that generates the event (e.g., `JButton`, `JTextField`).
2. **Event Object** → Encapsulates event details (e.g., `ActionEvent`, `MouseEvent`).
3. **Event Listener** → Interface that defines methods to handle the event (`ActionListener`, `MouseListener`).
4. **Event Registration** → The process of linking the event source to the event listener using `addActionListener()` or similar methods.

🔷 Steps to Implement Delegation Event Model

1. Implement the required **listener interface**.
2. Override the **event-handling method** from the interface.
3. Register the **listener with the event source**.

🔷 Example: Button Click Event Handling

```
import javax.swing.*;
import java.awt.event.*;

public class DelegationEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Delegation Event Model");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        JButton button = new JButton("Click Me");

        // Step 1: Register an ActionListener
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, "Button Clicked!");
            }
        });

        frame.add(button);
        frame.setVisible(true);
    }
}
```

✓ **Uses `ActionListener` to handle button clicks.**
✓ **Registers the listener using `addActionListener()`.**

◈ Event Object and Listener Interfaces

1 Event Object (`java.util.EventObject`)

All event classes in Java extend `EventObject`. Some common event classes:

Event Type	Event Class	Generated By
Action Event	`ActionEvent`	Buttons, menus, text fields
Mouse Event	`MouseEvent`	Mouse actions (click, enter, exit)
Key Event	`KeyEvent`	Keyboard actions
Window Event	`WindowEvent`	Window open, close, minimize

---

2 Event Listener Interfaces

Java provides predefined interfaces that must be implemented for handling events.

Listener Interface	Event Type	Method to Implement
`ActionListener`	Button clicks	`actionPerformed(ActionEvent e)`
`MouseListener`	Mouse clicks, enters, exits	`mouseClicked`, `mouseEntered`, etc.
`KeyListener`	Key presses	`keyTyped`, `keyPressed`, etc.

Listener Interface	Event Type	Method to Implement
`WindowListener`	Window open/close	`windowClosing`, `windowOpened`, etc.

◈ Example: Handling Multiple Events

✅ Handling Button Click & Mouse Events Together

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MultiEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Multi-Event Handling");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("Click or Hover Me");

        // ActionListener for button click
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame,
"Button Clicked!"));

        // MouseListener for mouse events
        button.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseEntered(MouseEvent e) {
                button.setForeground(Color.RED);
            }

            @Override
            public void mouseExited(MouseEvent e) {
                button.setForeground(Color.BLACK);
            }
        });

        frame.add(button);
        frame.setVisible(true);
    }
}
```

✓ **Uses `ActionListener` for clicks.**
✓ **Uses `MouseListener` to change color on hover.**

◈ Custom Event Handling (Creating Custom Events)

You can define **custom events** by extending `EventObject` and creating a custom listener.

✅ Example: Creating a Custom Event

```
import java.util.*;
```

```
// Custom Event Class
class CustomEvent extends EventObject {
    public CustomEvent(Object source) {
        super(source);
    }
}

// Custom Listener Interface
interface CustomEventListener extends EventListener {
    void handleCustomEvent(CustomEvent e);
}

// Custom Event Source
class CustomEventSource {
    private List<CustomEventListener> listeners = new ArrayList<>();

    public void addCustomEventListener(CustomEventListener listener) {
        listeners.add(listener);
    }

    public void triggerEvent() {
        CustomEvent event = new CustomEvent(this);
        for (CustomEventListener listener : listeners) {
            listener.handleCustomEvent(event);
        }
    }
}

// Main Class
public class CustomEventDemo {
    public static void main(String[] args) {
        CustomEventSource source = new CustomEventSource();

        // Registering a listener
        source.addCustomEventListener(e -> System.out.println("Custom event
triggered!"));

        // Trigger the event
        source.triggerEvent();
    }
}
```

✓ Defines a custom event, listener, and event source.
✓ Uses an `ArrayList` to store multiple listeners.

◈ Advantages of Delegation Event Model

✅ **Decouples event generation and handling** (increases flexibility).
✅ **Supports multiple listeners** (multiple objects can respond to the same event).
✅ **Improves performance** (only notified listeners handle events).

◈ Summary

1. **Event Source** (JButton, JTextField, JFrame) generates an event.
2. **Event Object** (`ActionEvent`, `MouseEvent`, etc.) encapsulates event details.
3. **Event Listener** (`ActionListener`, `MouseListener`) handles the event.
4. **Event Registration** (`addActionListener()`, `addMouseListener()`) links the event source to the listener.

Would you like a **GUI-based example with multiple components**? 🚀

Events in Java

In Java, an **event** is an action triggered by the user (e.g., clicking a button, pressing a key) or the system (e.g., window closing). Java handles events using the **Event Delegation Model**, where an **event source** generates an event and an **event listener** handles it.

◈ Types of Events in Java

Java provides built-in event classes in the `java.awt.event` and `javax.swing.event` packages. Below are the main types of events:

Event Type	Event Class	Triggered By
Action Event	`ActionEvent`	Button clicks, menu selections, text field submissions
Mouse Event	`MouseEvent`	Mouse clicks, movement, entering/exiting components
Key Event	`KeyEvent`	Key presses, releases, and typing
Window Event	`WindowEvent`	Opening, closing, minimizing a window
Focus Event	`FocusEvent`	Gaining or losing focus (e.g., text field selection)
Item Event	`ItemEvent`	Selecting checkboxes, radio buttons, combo boxes
Text Event	`TextEvent`	Modifying text in a text component
Adjustment Event	`AdjustmentEvent`	Scrollbars adjustments

◈ Event Handling Process

To handle an event in Java:

1. **Create an Event Source** (e.g., `JButton`, `JTextField`).
2. **Implement the appropriate Listener Interface** (e.g., `ActionListener`, `MouseListener`).
3. **Register the Listener** to the event source using `addActionListener()` or other methods.

🔷 Example: Handling Button Click Event (`ActionListener`)

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonClickExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Click Event");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("Click Me");

        // Registering an ActionListener
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame,
"Button Clicked!"));

        frame.add(button);
        frame.setVisible(true);
    }
}
```

✓ Uses `ActionListener` to handle button clicks.
✓ Registers the listener using `addActionListener()`.

🔷 Handling Different Types of Events

1️⃣ Mouse Events (`MouseListener`)

Handles mouse actions like clicks, entering, and exiting.

✅ Example: Detecting Mouse Clicks & Hover

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Mouse Event Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Hover or Click Me",
SwingConstants.CENTER);
        frame.add(label);

        label.addMouseListener(new MouseAdapter() {
            @Override
```

```
            public void mouseClicked(MouseEvent e) {
                label.setText("Mouse Clicked!");
            }

            @Override
            public void mouseEntered(MouseEvent e) {
                label.setForeground(Color.RED);
            }

            @Override
            public void mouseExited(MouseEvent e) {
                label.setForeground(Color.BLACK);
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **Detects mouse clicks (`mouseClicked`) and hover (`mouseEntered`, `mouseExited`).**
✓ Uses **`MouseAdapter`** to avoid implementing unnecessary methods.

---

2 Keyboard Events (`KeyListener`)

Handles key presses, releases, and typing.

☑ Example: Detecting Key Press

```
import javax.swing.*;
import java.awt.event.*;

public class KeyEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Key Event Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTextField textField = new JTextField();
        JLabel label = new JLabel("Type something...");

        textField.addKeyListener(new KeyAdapter() {
            @Override
            public void keyTyped(KeyEvent e) {
                label.setText("Typed: " + e.getKeyChar());
            }
        });

        frame.add(textField, "North");
        frame.add(label, "South");
        frame.setVisible(true);
    }
}
```

✓ **Updates label when a key is typed in the text field.**
✓ Uses **`KeyAdapter`** to simplify event handling.

3️⃣ Window Events (`WindowListener`)

Handles window actions like opening, closing, minimizing.

✅ Example: Confirm Before Closing a Window

```
import javax.swing.*;
import java.awt.event.*;

public class WindowEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Window Event Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                int confirm = JOptionPane.showConfirmDialog(frame, "Are you
sure you want to exit?");
                if (confirm == JOptionPane.YES_OPTION) {
                    frame.dispose();
                }
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **Asks for confirmation before closing the window.**
✓ Uses **`WindowAdapter`** for cleaner code.

🔷 Handling Multiple Events with One Listener

We can use a **single listener** to handle multiple events.

✅ Example: Handling Button Clicks & Mouse Hover in One Class

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MultiEventExample implements ActionListener {
    JButton button1, button2;
    JLabel label;
```

```
    public MultiEventExample() {
        JFrame frame = new JFrame("Multiple Events");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        button1 = new JButton("Hello");
        button2 = new JButton("Goodbye");
        label = new JLabel("Click a button", SwingConstants.CENTER);

        button1.addActionListener(this);
        button2.addActionListener(this);

        frame.setLayout(new FlowLayout());
        frame.add(button1);
        frame.add(button2);
        frame.add(label);

        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == button1) {
            label.setText("Hello!");
        } else if (e.getSource() == button2) {
            label.setText("Goodbye!");
        }
    }

    public static void main(String[] args) {
        new MultiEventExample();
    }
}
```

✓ **Handles multiple buttons in a single `ActionListener`.**
✓ Uses `getSource()` to identify the clicked button.

◈ Summary

Event Type	Listener Interface	Method to Implement
Button Click	`ActionListener`	`actionPerformed(ActionEvent e)`
Mouse Events	`MouseListener`	`mouseClicked`, `mouseEntered`, `mouseExited`
Key Events	`KeyListener`	`keyTyped`, `keyPressed`, `keyReleased`
Window Events	`WindowListener`	`windowClosing`, `windowOpened`, etc.

🔷 Conclusion

- **Use `ActionListener`** for button clicks.
- **Use `MouseListener`** for mouse events.
- **Use `KeyListener`** for keyboard input.
- **Use `WindowListener`** for window-related actions.

Would you like **a more advanced GUI example** with multiple event listeners? 🚀

Event Sources in Java

An **Event Source** in Java is an object that **generates events** when a user interacts with it. For example, a **button click, key press, or mouse movement** can trigger an event from an event source.

🔷 What is an Event Source?

An **event source** is any **GUI component** or system event that can generate an event. When an event occurs, it is passed to an **event listener** that handles it.

✅ **Examples of Event Sources**:

- **GUI Components** (Buttons, Text Fields, Checkboxes)
- **System Events** (Window Closing, Focus Change)
- **Custom Events** (User-defined triggers)

🔷 Common Event Sources in Java

Java provides multiple event sources through **Swing** and **AWT** components.

Event Source	Event Type	Listener Interface	Method to Register Listener
`JButton` (Button)	`ActionEvent`	`ActionListener`	`addActionListener()`
`JTextField` (Text Field)	`ActionEvent`, `KeyEvent`	`ActionListener`, `KeyListener`	`addActionListener()`, `addKeyListener()`
`JCheckBox` (Checkbox)	`ItemEvent`	`ItemListener`	`addItemListener()`

Event Source	Event Type	Listener Interface	Method to Register Listener
`JRadioButton` (Radio Button)	`ItemEvent`	`ItemListener`	`addItemListener()`
`JList` (List)	`ListSelectionEvent`	`ListSelectionListener`	`addListSelectionListener()`
`JComboBox` (Dropdown)	`ItemEvent`, `ActionEvent`	`ItemListener`, `ActionListener`	`addItemListener()`, `addActionListener()`
`JFrame` (Window)	`WindowEvent`	`WindowListener`	`addWindowListener()`
`JPanel`, `JLabel`	`MouseEvent`	`MouseListener`, `MouseMotionListener`	`addMouseListener()`, `addMouseMotionListener()`

◈ Example 1: Button Click as an Event Source

A `JButton` acts as an **event source**, generating an **ActionEvent** when clicked.

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Event Source");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("Click Me");

        // Register an ActionListener
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame,
"Button Clicked!"));

        frame.add(button);
        frame.setVisible(true);
    }
}
```

✓ JButton acts as an event source.
✓ ActionListener captures the button click event.

◈ Example 2: Text Field as an Event Source

A `JTextField` generates events when the user **presses "Enter"** or types a key.

```
import javax.swing.*;
import java.awt.event.*;

public class TextFieldEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Text Field Event Source");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTextField textField = new JTextField(20);
        JLabel label = new JLabel("Type something...");

        // Register an ActionListener for "Enter" key
        textField.addActionListener(e -> label.setText("Entered: " +
textField.getText()));

        frame.setLayout(new java.awt.FlowLayout());
        frame.add(textField);
        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ JTextField acts as an event source.
✓ Triggers `ActionEvent` when Enter is pressed.

◈ Example 3: Checkbox as an Event Source

A `JCheckBox` generates an **ItemEvent** when **checked or unchecked**.

```
import javax.swing.*;
import java.awt.event.*;

public class CheckBoxEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Checkbox Event Source");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JCheckBox checkBox = new JCheckBox("Enable Feature");
        JLabel label = new JLabel("Feature is OFF");

        // Register an ItemListener
        checkBox.addItemListener(e -> {
            if (e.getStateChange() == ItemEvent.SELECTED) {
                label.setText("Feature is ON");
            } else {
                label.setText("Feature is OFF");
            }
```

```
        });

        frame.setLayout(new java.awt.FlowLayout());
        frame.add(checkBox);
        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ JCheckBox acts as an event source.
✓ Triggers `ItemEvent` when checked/unchecked.

◈ Example 4: Window Closing as an Event Source

A `JFrame` can generate a **WindowEvent** when **closed**.

```
import javax.swing.*;
import java.awt.event.*;

public class WindowEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Window Event Source");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        // Register a WindowListener
        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                int choice = JOptionPane.showConfirmDialog(frame, "Are you
sure you want to exit?");
                if (choice == JOptionPane.YES_OPTION) {
                    frame.dispose();
                }
            }
        });

        frame.setVisible(true);
    }
}
```

✓ JFrame acts as an event source.
✓ Triggers `WindowEvent` when closing the window.

◈ Summary

Event Source	Triggers Event Type	Listener Interface
JButton	ActionEvent	ActionListener

Event Source	Triggers Event Type	Listener Interface
`JTextField`	`ActionEvent`, `KeyEvent`	`ActionListener`, `KeyListener`
`JCheckBox`	`ItemEvent`	`ItemListener`
`JFrame`	`WindowEvent`	`WindowListener`

◈ Conclusion

- **An Event Source is any object that generates events.**
- **Each event source has a corresponding event type and listener interface.**
- **To handle events, register the appropriate listener with the event source.**

Would you like an example of **handling multiple event sources in one program**? 🚀

Event Listeners in Java

◈ What is an Event Listener?

An **event listener** in Java is an interface that listens for specific events (e.g., button clicks, mouse movements, key presses) and executes code in response.

📌 How Event Listeners Work

1. **An event occurs** (e.g., a button is clicked).
2. **The event source generates the event** (e.g., `JButton`).
3. **An event listener detects the event** and executes a method to handle it.

◈ List of Event Listeners in Java

Java provides various event listener interfaces in the `java.awt.event` and `javax.swing.event` packages.

Event Listener Interface	Event Type	Event Source Example	Method to Implement
`ActionListener`	`ActionEvent`	`JButton`, `JTextField`, `JMenuItem`	`actionPerformed(ActionEvent e)`

Event Listener Interface	Event Type	Event Source Example	Method to Implement
MouseListener	MouseEvent	JLabel, JButton, JPanel	mouseClicked, mousePressed, mouseReleased, mouseEntered, mouseExited
MouseMotionListener	MouseMotionEvent	JPanel, JLabel	mouseMoved, mouseDragged
KeyListener	KeyEvent	JTextField, JFrame	keyTyped, keyPressed, keyReleased
ItemListener	ItemEvent	JCheckBox, JRadioButton, JComboBox	itemStateChanged(ItemEvent e)
WindowListener	WindowEvent	JFrame	windowOpened, windowClosing, windowClosed, windowActivated, windowDeactivated
FocusListener	FocusEvent	JTextField, JButton	focusGained, focusLost
ListSelectionListener	ListSelectionEvent	JList	valueChanged(ListSelectionEvent e)

◈ Example 1: Handling Button Clicks (`ActionListener`)

A **button (`JButton`)** generates an **ActionEvent**, which is handled by an **ActionListener**.

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Listener Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("Click Me");

        // Registering an ActionListener
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, "Button Clicked!");
            }
        });

        frame.add(button);
```

```
        frame.setVisible(true);
    }
}
```

✓ **`ActionListener` detects button clicks**
✓ **Executes `actionPerformed()` when clicked**

◈ Example 2: Handling Mouse Events (`MouseListener`)

A **label (`JLabel`)** changes color when the mouse enters or exits.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Mouse Listener Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Hover Over Me", SwingConstants.CENTER);
        frame.add(label);

        // Registering a MouseListener
        label.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseEntered(MouseEvent e) {
                label.setForeground(Color.RED);
            }

            @Override
            public void mouseExited(MouseEvent e) {
                label.setForeground(Color.BLACK);
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **`MouseListener` detects mouse enter/exit events**
✓ **Uses `MouseAdapter` to simplify the code**

◈ Example 3: Handling Keyboard Input (`KeyListener`)

A **text field (`JTextField`)** detects key presses.

```
import javax.swing.*;
import java.awt.event.*;
```

```
public class KeyListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Key Listener Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTextField textField = new JTextField();
        JLabel label = new JLabel("Type something...");

        textField.addKeyListener(new KeyAdapter() {
            @Override
            public void keyTyped(KeyEvent e) {
                label.setText("Typed: " + e.getKeyChar());
            }
        });

        frame.add(textField, "North");
        frame.add(label, "South");
        frame.setVisible(true);
    }
}
```

✓ Detects key presses using `KeyListener`
✓ Displays the typed character in a label

◈ Example 4: Handling Item Selection (`ItemListener`)

A **checkbox (`JCheckBox`)** toggles a label when checked/unchecked.

```
import javax.swing.*;
import java.awt.event.*;

public class CheckBoxListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Item Listener Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JCheckBox checkBox = new JCheckBox("Enable Feature");
        JLabel label = new JLabel("Feature is OFF");

        // Registering an ItemListener
        checkBox.addItemListener(e -> {
            if (e.getStateChange() == ItemEvent.SELECTED) {
                label.setText("Feature is ON");
            } else {
                label.setText("Feature is OFF");
            }
        });

        frame.setLayout(new java.awt.FlowLayout());
        frame.add(checkBox);
        frame.add(label);
        frame.setVisible(true);
```

```
    }
}
```

✓ **Uses `ItemListener` to detect checkbox state change**

◈ Example 5: Handling Window Closing (`WindowListener`)

A **window (`JFrame`)** asks for confirmation before closing.

```
import javax.swing.*;
import java.awt.event.*;

public class WindowListenerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Window Listener Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                int choice = JOptionPane.showConfirmDialog(frame, "Are you
sure you want to exit?");
                if (choice == JOptionPane.YES_OPTION) {
                    frame.dispose();
                }
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **Uses `WindowListener` to handle window closing events**
✓ **Confirms before closing the application**

◈ Summary

Event Listener	Used For	Method Implemented
`ActionListener`	Button clicks, menu selections	`actionPerformed(ActionEvent e)`
`MouseListener`	Mouse clicks, movement	`mouseClicked`, `mouseEntered`, `mouseExited`
`KeyListener`	Key presses, typing	`keyTyped`, `keyPressed`, `keyReleased`
`ItemListener`	Checkboxes, radio buttons	`itemStateChanged(ItemEvent e)`

Event Listener	Used For	Method Implemented
`WindowListener`	Window opening, closing	`windowClosing`, `windowOpened`

◈ Conclusion

- **Listeners detect and respond to events.**
- **Each listener has a corresponding event source and method.**
- **Using `Adapter` classes (e.g., `MouseAdapter`) helps avoid implementing unnecessary methods.**

Would you like an example combining multiple event listeners in one program? 🚀

Event Classes in Java

◈ What Are Event Classes?

In Java, event classes are part of the **Java Event Handling mechanism** and are used to **represent different types of events**. They are part of the `java.awt.event` package and are generated by event sources (like buttons, text fields, and windows).

Each event class contains information about the event, such as **which component triggered it, the type of action, and additional event details**.

◈ Event Class Hierarchy

All event classes extend the base class `java.util.EventObject`.
The hierarchy is as follows:

```
java.util.EventObject  (Base Class)
   ├── java.awt.AWTEvent  (Superclass of GUI-related events)
        ├── java.awt.event.ActionEvent
        ├── java.awt.event.MouseEvent
        ├── java.awt.event.KeyEvent
        ├── java.awt.event.WindowEvent
        ├── java.awt.event.ItemEvent
        ├── java.awt.event.FocusEvent
        ├── java.awt.event.ComponentEvent
        ├── java.awt.event.ContainerEvent
        ├── java.awt.event.AdjustmentEvent
        ├── java.awt.event.TextEvent
```

◈ Common Event Classes in Java

Event Class	Generated By	Used With
`ActionEvent`	Button clicks, menu item selection	`JButton`, `JMenuItem`, `JTextField`
`MouseEvent`	Mouse clicks, movements, drags	`JPanel`, `JButton`, `JLabel`
`KeyEvent`	Key presses/releases	`JTextField`, `JFrame`, `JPanel`
`ItemEvent`	Checkbox, radio button state change	`JCheckBox`, `JRadioButton`, `JComboBox`
`WindowEvent`	Window open/close	`JFrame`, `JDialog`
`FocusEvent`	Focus gained/lost	`JTextField`, `JButton`
`ComponentEvent`	Component resized, moved, hidden	`JPanel`, `JFrame`
`ContainerEvent`	Components added/removed	`JPanel`, `JFrame`
`AdjustmentEvent`	Scrollbar position change	`JScrollBar`
`TextEvent`	Text changes in a component	`JTextField`, `JTextArea`

◈ Example 1: Handling `ActionEvent` (Button Click)

A `JButton` generates an **`ActionEvent`** when clicked.

```
import javax.swing.*;
import java.awt.event.*;

public class ActionEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ActionEvent Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("Click Me");

        // Handling ActionEvent
        button.addActionListener(e -> {
            JOptionPane.showMessageDialog(frame, "Button Clicked!");
        });

        frame.add(button);
        frame.setVisible(true);
    }
}
```

✓ **Event:** `ActionEvent`
✓ **Generated by:** `JButton`

🔷 Example 2: Handling `MouseEvent` (Mouse Click)

A `JLabel` changes color when clicked.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("MouseEvent Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Click Me", SwingConstants.CENTER);
        frame.add(label);

        // Handling MouseEvent
        label.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                label.setForeground(Color.RED);
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **Event:** `MouseEvent`
✓ **Generated by:** `JLabel` **when clicked**

🔷 Example 3: Handling `KeyEvent` (Key Press)

A `JTextField` detects when a key is typed.

```
import javax.swing.*;
import java.awt.event.*;

public class KeyEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("KeyEvent Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTextField textField = new JTextField();
        JLabel label = new JLabel("Type something...");
```

```
        // Handling KeyEvent
        textField.addKeyListener(new KeyAdapter() {
            @Override
            public void keyTyped(KeyEvent e) {
                label.setText("Typed: " + e.getKeyChar());
            }
        });

        frame.add(textField, "North");
        frame.add(label, "South");
        frame.setVisible(true);
    }
}
```

✓ **Event: `KeyEvent`**
✓ **Generated by: `JTextField` when a key is pressed**

◈ Example 4: Handling `WindowEvent` (Window Closing)

A `JFrame` asks for confirmation before closing.

```
import javax.swing.*;
import java.awt.event.*;

public class WindowEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("WindowEvent Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                int choice = JOptionPane.showConfirmDialog(frame, "Are you
sure you want to exit?");
                if (choice == JOptionPane.YES_OPTION) {
                    frame.dispose();
                }
            }
        });

        frame.setVisible(true);
    }
}
```

✓ **Event: `WindowEvent`**
✓ **Generated by: `JFrame` when closing**

◈ Example 5: Handling `ItemEvent` (Checkbox State Change)

A `JCheckBox` toggles a label when checked/unchecked.

```
import javax.swing.*;
import java.awt.event.*;

public class ItemEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ItemEvent Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JCheckBox checkBox = new JCheckBox("Enable Feature");
        JLabel label = new JLabel("Feature is OFF");

        // Handling ItemEvent
        checkBox.addItemListener(e -> {
            if (e.getStateChange() == ItemEvent.SELECTED) {
                label.setText("Feature is ON");
            } else {
                label.setText("Feature is OFF");
            }
        });

        frame.setLayout(new java.awt.FlowLayout());
        frame.add(checkBox);
        frame.add(label);
        frame.setVisible(true);
    }
}
```

✓ **Event:** **`ItemEvent`**
✓ **Generated by:** **`JCheckBox`** **when state changes**

◈ Summary of Event Classes

Event Class	Generated By	Handled Using
`ActionEvent`	Buttons, Menus	`ActionListener`
`MouseEvent`	Mouse actions	`MouseListener`, `MouseMotionListener`
`KeyEvent`	Keyboard input	`KeyListener`
`ItemEvent`	Checkbox, RadioButton	`ItemListener`
`WindowEvent`	Window open/close	`WindowListener`
`FocusEvent`	Focus changes	`FocusListener`

Event Class	Generated By	Handled Using
`AdjustmentEvent`	Scrollbars	`AdjustmentListener`
`TextEvent`	TextField changes	`TextListener`

◈ Conclusion

- **Event classes represent different types of user interactions.**
- **Each event is handled using a corresponding listener.**
- **Java provides built-in event classes like `ActionEvent`, `MouseEvent`, and `KeyEvent`.**

Would you like an example combining multiple event classes in one program? 🚀

Handling Mouse and Keyboard Events in Java

◈ Introduction

Java provides event handling mechanisms for **mouse and keyboard interactions** using the **AWT (Abstract Window Toolkit) Event Model**. These events are handled using **listeners**, which are interfaces containing methods to respond to different actions.

◈ Handling Mouse Events

Mouse events are captured using the `MouseListener` and `MouseMotionListener` interfaces. They handle actions such as **clicking, pressing, releasing, entering, exiting, moving, and dragging** the mouse.

Mouse Event Classes

Event Class	Description
`MouseEvent`	Represents mouse actions like clicks, movements, and drags.

Mouse Event Methods

Method	Description
`mouseClicked(MouseEvent e)`	Called when the mouse is clicked.
`mousePressed(MouseEvent e)`	Called when a mouse button is pressed.
`mouseReleased(MouseEvent e)`	Called when a pressed button is released.

Method	Description
`mouseEntered(MouseEvent e)`	Called when the mouse enters a component.
`mouseExited(MouseEvent e)`	Called when the mouse exits a component.
`mouseMoved(MouseEvent e)`	Called when the mouse moves within a component.
`mouseDragged(MouseEvent e)`	Called when the mouse is dragged (pressed and moved).

Example: Handling Mouse Events

This program displays a `JLabel` that updates when the user **clicks or moves** the mouse.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseEventExample extends JFrame implements MouseListener,
MouseMotionListener {
    JLabel label;

    public MouseEventExample() {
        setTitle("Mouse Event Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        label = new JLabel("Perform Mouse Actions");
        add(label);

        addMouseListener(this);
        addMouseMotionListener(this);

        setVisible(true);
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        label.setText("Mouse Clicked at X: " + e.getX() + " Y: " + e.getY());
    }

    @Override
    public void mousePressed(MouseEvent e) {
        label.setText("Mouse Pressed");
    }

    @Override
    public void mouseReleased(MouseEvent e) {
        label.setText("Mouse Released");
    }

    @Override
    public void mouseEntered(MouseEvent e) {
        label.setText("Mouse Entered the Window");
    }
```

```
    @Override
    public void mouseExited(MouseEvent e) {
        label.setText("Mouse Exited the Window");
    }

    @Override
    public void mouseMoved(MouseEvent e) {
        label.setText("Mouse Moved to X: " + e.getX() + " Y: " + e.getY());
    }

    @Override
    public void mouseDragged(MouseEvent e) {
        label.setText("Mouse Dragged");
    }

    public static void main(String[] args) {
        new MouseEventExample();
    }
}
```

✓ **Event: Mouse clicks, movement, and dragging**
✓ **Handled by:** `MouseListener`, `MouseMotionListener`
✓ **Displayed on:** `JLabel`

◈ Handling Keyboard Events

Keyboard events are handled using the `KeyListener` interface, which detects key presses, releases, and typing.

Keyboard Event Classes

Event Class	Description
`KeyEvent`	Represents keyboard actions like key press and release.

Keyboard Event Methods

Method	Description
`keyTyped(KeyEvent e)`	Called when a key is typed (press & release).
`keyPressed(KeyEvent e)`	Called when a key is pressed.
`keyReleased(KeyEvent e)`	Called when a key is released.

Example: Handling Keyboard Events

This program captures keyboard input and displays which key is pressed.

```
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;

public class KeyEventExample extends JFrame implements KeyListener {
    JLabel label;

    public KeyEventExample() {
        setTitle("Key Event Example");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        label = new JLabel("Type something...");
        add(label);

        addKeyListener(this);
        setFocusable(true); // Required for key events to work
        setVisible(true);
    }

    @Override
    public void keyTyped(KeyEvent e) {
        label.setText("Key Typed: " + e.getKeyChar());
    }

    @Override
    public void keyPressed(KeyEvent e) {
        label.setText("Key Pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
    }

    @Override
    public void keyReleased(KeyEvent e) {
        label.setText("Key Released: " +
KeyEvent.getKeyText(e.getKeyCode()));
    }

    public static void main(String[] args) {
        new KeyEventExample();
    }
}
```

✓ **Event: Key typing, pressing, and releasing**
✓ **Handled by:** `KeyListener`
✓ **Displayed on:** `JLabel`

◈ Combined Example: Handling Mouse and Keyboard Events

This program responds to **both mouse and keyboard events**.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseAndKeyEventExample extends JFrame implements MouseListener,
KeyListener {
```

```
    JLabel label;

    public MouseAndKeyEventExample() {
        setTitle("Mouse & Keyboard Event Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        label = new JLabel("Perform Mouse or Keyboard Actions");
        add(label);

        addMouseListener(this);
        addKeyListener(this);
        setFocusable(true);
        setVisible(true);
    }

    // Mouse Events
    @Override
    public void mouseClicked(MouseEvent e) {
        label.setText("Mouse Clicked at X: " + e.getX() + " Y: " + e.getY());
    }

    @Override
    public void mousePressed(MouseEvent e) {}

    @Override
    public void mouseReleased(MouseEvent e) {}

    @Override
    public void mouseEntered(MouseEvent e) {
        label.setText("Mouse Entered");
    }

    @Override
    public void mouseExited(MouseEvent e) {
        label.setText("Mouse Exited");
    }

    // Keyboard Events
    @Override
    public void keyTyped(KeyEvent e) {
        label.setText("Key Typed: " + e.getKeyChar());
    }

    @Override
    public void keyPressed(KeyEvent e) {
        label.setText("Key Pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
    }

    @Override
    public void keyReleased(KeyEvent e) {}

    public static void main(String[] args) {
        new MouseAndKeyEventExample();
    }
}
```

✓ **Handles both:** `MouseListener` and `KeyListener`
✓ **Updates label based on user actions**

◈ Summary

Event Type	Listener Interface	Common Methods
Mouse Events	`MouseListener`	`mouseClicked()`, `mousePressed()`, `mouseReleased()`, `mouseEntered()`, `mouseExited()`
Mouse Motion Events	`MouseMotionListener`	`mouseMoved()`, `mouseDragged()`
Keyboard Events	`KeyListener`	`keyTyped()`, `keyPressed()`, `keyReleased()`

◈ Conclusion

- Mouse events are handled using `MouseListener` and `MouseMotionListener`.
- Keyboard events are handled using `KeyListener`.
- `KeyEvent` allows detecting specific key presses (`KeyEvent.getKeyText(e.getKeyCode())`).
- **Combination of both** allows full user interaction.

Would you like to extend this with **game-like interactions or hotkeys**? 🚀

Adapter Classes in Java

◈ Introduction

Adapter classes in Java are **abstract classes** that provide **default implementations** for interface methods. They help in handling event-driven programming by allowing developers to override only the required methods instead of implementing all the methods of an interface.

Why Use Adapter Classes?

- Interfaces like `MouseListener`, `KeyListener`, etc., have multiple methods.
- If you **only need one or two methods**, you would still have to implement all.
- **Adapter classes solve this** by providing default implementations so you can override only what you need.

◈ Adapter Classes in Java's AWT & Swing

Java provides adapter classes for event listeners in **`java.awt.event`** package:

Adapter Class	Interface Implemented	Used For
`MouseAdapter`	`MouseListener`, `MouseMotionListener`	Mouse events (click, move, drag, etc.)
`KeyAdapter`	`KeyListener`	Keyboard events (key press, release)
`WindowAdapter`	`WindowListener`	Window events (open, close, minimize)
`FocusAdapter`	`FocusListener`	Focus events (gaining/losing focus)
`ComponentAdapter`	`ComponentListener`	Component events (resizing, showing)

◈ Example: Using `MouseAdapter`

Instead of implementing all five methods of `MouseListener`, we use `MouseAdapter` and override only `mouseClicked()`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseAdapterExample extends JFrame {
    JLabel label;

    public MouseAdapterExample() {
        setTitle("Mouse Adapter Example");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        label = new JLabel("Click anywhere in the window");
        add(label);

        addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                label.setText("Mouse Clicked at X: " + e.getX() + " Y: " +
e.getY());
            }
        });

        setVisible(true);
    }
```

```
    public static void main(String[] args) {
        new MouseAdapterExample();
    }
}
```

✓ **Uses `MouseAdapter` instead of `MouseListener`**
✓ **Overrides only `mouseClicked()`**
✓ **Shorter, cleaner code**

🔷 Example: Using `KeyAdapter`

We override only `keyPressed()` instead of implementing all `KeyListener` methods.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KeyAdapterExample extends JFrame {
    JLabel label;

    public KeyAdapterExample() {
        setTitle("Key Adapter Example");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        label = new JLabel("Press any key...");
        add(label);

        addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                label.setText("Key Pressed: " +
KeyEvent.getKeyText(e.getKeyCode()));
            }
        });

        setFocusable(true);
        setVisible(true);
    }

    public static void main(String[] args) {
        new KeyAdapterExample();
    }
}
```

✓ **Uses `KeyAdapter` instead of `KeyListener`**
✓ **Overrides only `keyPressed()`**

🔷 Example: Using `WindowAdapter`

We override only `windowClosing()` to confirm exit.

```
import javax.swing.*;
import java.awt.event.*;

public class WindowAdapterExample extends JFrame {
    public WindowAdapterExample() {
        setTitle("Window Adapter Example");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                int confirm = JOptionPane.showConfirmDialog(
                        null, "Are you sure you want to exit?", "Exit
Confirmation",
                        JOptionPane.YES_NO_OPTION);
                if (confirm == JOptionPane.YES_OPTION) {
                    dispose();
                }
            }
        });

        setVisible(true);
    }

    public static void main(String[] args) {
        new WindowAdapterExample();
    }
}
```

✓ **Uses `WindowAdapter` to handle `windowClosing()`**
✓ **Displays an exit confirmation dialog**

🔷 Benefits of Using Adapter Classes

✅ **Avoids unnecessary method implementations**
✅ **Simplifies event handling code**
✅ **Improves readability and maintainability**

🔷 Summary

Adapter Class	Interface Implemented	Example Use
`MouseAdapter`	`MouseListener`, `MouseMotionListener`	Handling mouse clicks

Adapter Class	Interface Implemented	Example Use
`KeyAdapter`	`KeyListener`	Handling key presses
`WindowAdapter`	`WindowListener`	Handling window close event
`FocusAdapter`	`FocusListener`	Handling focus gained/lost
`ComponentAdapter`	`ComponentListener`	Handling component resizing

Would you like an example with multiple adapters combined? 🚀

Inner Classes in Java

◈ Introduction

In Java, **inner classes** are classes defined **inside another class**. They help in logically grouping classes that are only used within a specific context, improving encapsulation and readability.

◈ Types of Inner Classes

Java provides **four** types of inner classes:

Type	Definition
Member Inner Class	A class defined inside another class (non-static)
Static Nested Class	A static class inside another class
Local Inner Class	A class defined inside a method
Anonymous Inner Class	A class without a name, declared and instantiated in one step

---

◈ 1. Member Inner Class

- Defined **inside another class**, but **outside any method**.
- Can **access private members** of the outer class.

Example: Member Inner Class

```
class Outer {
    private String message = "Hello from Outer Class";

    class Inner {
```

```
        void display() {
            System.out.println(message); // Access outer class private member
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner(); // Create Inner class object
        inner.display();
    }
}
```

✓ Accesses outer class's private data
✓ Created using `outerObject.new Inner()`

◈ 2. Static Nested Class

- Declared using `static` keyword.
- **Cannot** access **non-static** members of the outer class directly.
- **Created without an instance** of the outer class.

Example: Static Nested Class

```
class Outer {
    static String message = "Hello from Static Nested Class";

    static class Nested {
        void display() {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        Outer.Nested nested = new Outer.Nested(); // No outer object needed
        nested.display();
    }
}
```

✓ No need for an outer class instance
✓ Only accesses static members

◈ 3. Local Inner Class

- **Defined inside a method**.
- **Can only be used inside that method**.
- **Cannot have `static` members**.

Example: Local Inner Class

```
class Outer {
```

```
    void show() {
        class LocalInner {
            void display() {
                System.out.println("Hello from Local Inner Class");
            }
        }

        LocalInner local = new LocalInner();
        local.display();
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.show();
    }
}
```

✓ Defined inside a method
✓ Only accessible within the method

🔷 4. Anonymous Inner Class

- A **class without a name**.
- Created for **one-time use**.
- Declared and **instantiated in one step**.

Example: Anonymous Inner Class (with Interface)

```
interface Greeting {
    void sayHello();
}

public class AnonymousInnerClass {
    public static void main(String[] args) {
        Greeting greeting = new Greeting() { // Anonymous class
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner Class");
            }
        };
        greeting.sayHello();
    }
}
```

✓ Used when a class is needed only once
✓ Implements an interface without a separate class

◈ Summary

Inner Class Type	Defined	Access	Usage
Member Inner Class	Inside another class	Can access outer class members	Group related classes
Static Nested Class	Inside another class, declared `static`	Can access only `static` members	Utility/helper classes
Local Inner Class	Inside a method	Only inside that method	Encapsulation within methods
Anonymous Inner Class	Inside method/block	Cannot be reused	Used for event handling

Would you like an example combining **inner classes and GUI programming**? 🚀

Anonymous Inner Classes in Java

◈ What is an Anonymous Inner Class?

An **anonymous inner class** is a class **without a name**, declared and instantiated in a **single step**. It is commonly used when you need to create a subclass or implement an interface **for one-time use**.

◈ Characteristics of Anonymous Inner Classes

- ✅ **No explicit class name**
- ✅ **Declared and instantiated simultaneously**
- ✅ **Cannot have constructors**
- ✅ **Can extend a class or implement an interface**
- ✅ **Mostly used in event handling and short-lived objects**

◈ Types of Anonymous Inner Classes

1. **Extending a Class**
2. **Implementing an Interface**
3. **Used in Method Arguments**

🔷 1. Anonymous Inner Class Extending a Class

- You can create an anonymous subclass of a **concrete or abstract class**.

Example: Extending a Class

```
abstract class Greeting {
    abstract void sayHello();
}

public class AnonymousClassExample {
    public static void main(String[] args) {
        // Anonymous class extending Greeting
        Greeting greeting = new Greeting() {
            @Override
            void sayHello() {
                System.out.println("Hello from Anonymous Inner Class!");
            }
        };

        greeting.sayHello();
    }
}
```

✓ Creates an instance of `Greeting` without defining a separate subclass
✓ Overrides `sayHello()` directly

🔷 2. Anonymous Inner Class Implementing an Interface

- You can create an anonymous class that **implements an interface** and provides its own implementation.

Example: Implementing an Interface

```
interface Animal {
    void makeSound();
}

public class AnonymousClassExample {
    public static void main(String[] args) {
        // Anonymous class implementing Animal interface
        Animal dog = new Animal() {
            @Override
            public void makeSound() {
                System.out.println("Dog says: Woof Woof!");
            }
        };

        dog.makeSound();
    }
}
```

✓ Implements `Animal` without creating a named class

🔷 3. Anonymous Inner Class as a Method Argument

- You can define an anonymous inner class **inside a method argument** for one-time use.

Example: Passing as an Argument

```
interface Printer {
    void printMessage(String message);
}

public class AnonymousClassExample {
    public static void main(String[] args) {
        printHello(new Printer() { // Anonymous inner class
            @Override
            public void printMessage(String message) {
                System.out.println(message);
            }
        });
    }

    static void printHello(Printer printer) {
        printer.printMessage("Hello from Anonymous Class in Method
Argument!");
    }
}
```

✓ No need for an extra class
✓ Declares the class inside the method call itself

🔷 4. Anonymous Inner Class in Event Handling (Swing)

- Used frequently in **GUI programming** for handling button clicks and other events.

Example: Swing Button Click Listener

```
import javax.swing.*;
import java.awt.event.*;

public class AnonymousClassSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Anonymous Inner Class Example");
        JButton button = new JButton("Click Me");

        // Using Anonymous Inner Class for event handling
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button Clicked!");
            }
        });

        frame.add(button);
        frame.setSize(300, 200);
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

✓ **No need to create a separate listener class**
✓ **Directly handles button clicks**

◈ Summary

Type	Example	Usage
Extending a Class	`new ParentClass() {}`	Quick subclassing
Implementing an Interface	`new InterfaceName() {}`	One-time interface implementation
Method Argument	`method(new Interface() { ... })`	Passing behavior dynamically
Event Handling (Swing)	`button.addActionListener(new ActionListener() {...})`	GUI and user interactions

◈ When to Use Anonymous Inner Classes?

✓ When you **need a one-time use** implementation
✓ When you **don't want to create a separate class**
✓ In **event-driven programming** (e.g., GUI, threads)
✓ To **reduce boilerplate code**

Would you like a **Lambda Expression** example as well? Since Java 8, they offer a cleaner way to handle functional interfaces! 🚀

LAB PROGRAMS

Write a Java program to print Fibonacci series using for loop.

```
import java.util.Scanner;

public class FibonacciSeries {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Input: Number of terms

        System.out.print("Enter the number of terms: ");

        int n = scanner.nextInt();

        int first = 0, second = 1;

        System.out.println("Fibonacci Series up to " + n + " terms:");

          for (int i = 0; i < n; i++) {

            System.out.print(first + " ");

            // Compute next term

            int next = first + second;

            first = second;

            second = next;

        }

        scanner.close();

    }

}
```

Output :-

```
Enter the number of terms: 10

Fibonacci Series up to 10 terms:

0 1 1 2 3 5 8 13 21 34
```

Write a Java program to calculate multiplication of 2 matrices.

```
import java.util.Scanner;

public class MatrixMultiplication {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Input dimensions for first matrix

        System.out.print("Enter rows and columns for first matrix: ");

        int rowsA = scanner.nextInt();

        int colsA = scanner.nextInt();


        // Input dimensions for second matrix

        System.out.print("Enter rows and columns for second matrix: ");

        int rowsB = scanner.nextInt();

        int colsB = scanner.nextInt();


        // Check if multiplication is possible

        if (colsA != rowsB) {

            System.out.println("Matrix multiplication not possible. Columns of A must match rows of B.");

            return;

        }


        // Input first matrix

        int[][] A = new int[rowsA][colsA];

        System.out.println("Enter elements of first matrix:");

        for (int i = 0; i < rowsA; i++) {

            for (int j = 0; j < colsA; j++) {

                A[i][j] = scanner.nextInt();
```

```
    }
}

// Input second matrix
int[][] B = new int[rowsB][colsB];
System.out.println("Enter elements of second matrix:");
for (int i = 0; i < rowsB; i++) {
    for (int j = 0; j < colsB; j++) {
        B[i][j] = scanner.nextInt();
    }
}

// Resultant matrix
int[][] C = new int[rowsA][colsB];

// Matrix multiplication
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        for (int k = 0; k < colsA; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

// Display result
System.out.println("Resultant Matrix:");
for (int i = 0; i < rowsA; i++) {
```

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

        scanner.close();
    }
}
```

Output :-

```
Enter rows and columns for first matrix: 2 3
Enter rows and columns for second matrix: 3 2
Enter elements of first matrix:
1 2 3
4 5 6
Enter elements of second matrix:
7 8
9 10
11 12
Resultant Matrix:
58 64
139 154
```

Create a class Rectangle. The class has attributes length and width. It should have methods that calculate the perimeter and area of the rectangle. It should have read Attributes method to read length and width from user.

```
import java.util.Scanner;


class Rectangle {
    // Attributes
    private double length;
    private double width;

    // Method to read attributes from user
    public void readAttributes() {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the length of the rectangle: ");
        length = scanner.nextDouble();

        System.out.print("Enter the width of the rectangle: ");
        width = scanner.nextDouble();

        scanner.close();
    }

    // Method to calculate area
    public double calculateArea() {
```

```
        return length * width;
    }

    // Method to calculate perimeter
    public double calculatePerimeter() {
        return 2 * (length + width);
    }

    // Method to display results
    public void display() {
        System.out.println("Rectangle Details:");
        System.out.println("Length: " + length);
        System.out.println("Width: " + width);
        System.out.println("Area: " + calculateArea());
        System.out.println("Perimeter: " + calculatePerimeter());
    }
}

public class RectangleDemo {
    public static void main(String[] args) {
        // Creating an object of Rectangle
        Rectangle rect = new Rectangle();

        // Reading attributes
        rect.readAttributes();

        // Display results
```

```
        rect.display();
    }
}
```

Output :-

```
Enter the length of the rectangle: 10
Enter the width of the rectangle: 5
Rectangle Details:
Length: 10.0
Width: 5.0
Area: 50.0
Perimeter: 30.0
```

Write a Java program that implements method overloading

```
class MathOperations {

    // Method 1: Add two integers

    int add(int a, int b) {

        return a + b;

    }


    // Method 2: Add three integers

    int add(int a, int b, int c) {

        return a + b + c;

    }


    // Method 3: Add two double numbers

    double add(double a, double b) {

        return a + b;

    }


    // Method 4: Concatenate two strings

    String add(String a, String b) {

        return a + b;

    }

}


public class MethodOverloadingDemo {

    public static void main(String[] args) {

        MathOperations math = new MathOperations();
```

```
        // Calling different overloaded methods
        System.out.println("Addition of two integers: " + math.add(10, 20));
        System.out.println("Addition of three integers: " + math.add(5, 10, 15));
        System.out.println("Addition of two doubles: " + math.add(3.5, 2.2));
        System.out.println("Concatenation of strings: " + math.add("Hello, ", "World!"));
    }
}
```

Output :-

```
Addition of two integers: 30
Addition of three integers: 30
Addition of two doubles: 5.7
Concatenation of strings: Hello, World!
```

Write a Java program for sorting a given list of names in ascending order.

```
import java.util.Scanner;

import java.util.Arrays;


public class NameSorter {

   public static void main(String[] args) {

      Scanner scanner = new Scanner(System.in);


      // Input: Number of names

      System.out.print("Enter the number of names: ");

      int n = scanner.nextInt();

      scanner.nextLine(); // Consume the newline


      // Array to store names

      String[] names = new String[n];


      // Reading names from user

      System.out.println("Enter " + n + " names:");

      for (int i = 0; i < n; i++) {

         names[i] = scanner.nextLine();

      }


      // Sorting names in ascending order

      Arrays.sort(names);
```

```
        // Displaying sorted names
        System.out.println("\nSorted Names in Ascending Order:");
        for (String name : names) {
            System.out.println(name);
        }

        scanner.close();
    }
}
```

Output :-

```
Enter the number of names: 5
Enter 5 names:
John
Alice
Bob
Charlie
David

Sorted Names in Ascending Order:
Alice
Bob
Charlie
David
John
```

Write a Java program that displays the number of characters, lines and words in a text file.

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;


public class FileStats {

   public static void main(String[] args) {

      // File name (change this to your file path)

      String fileName = "sample.txt";


      int charCount = 0;

      int wordCount = 0;

      int lineCount = 0;


      try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {

         String line;


         while ((line = reader.readLine()) != null) {

            lineCount++;  // Count lines

            charCount += line.length();  // Count characters (excluding newlines)


            // Split line into words using spaces

            String[] words = line.trim().split("\\s+");

            wordCount += words.length;

         }
```

```
            // Display results
            System.out.println("File: " + fileName);
            System.out.println("Number of lines: " + lineCount);
            System.out.println("Number of words: " + wordCount);
            System.out.println("Number of characters: " + charCount);

        } catch (IOException e) {
            System.out.println("Error reading the file: " + e.getMessage());
        }
    }
}
```

Output :-

sample.txt

Hello world.

Java programming is fun.

File handling is important.

File: sample.txt

Number of lines: 3

Number of words: 8

Number of characters: 54

Write a Java program to implement various types of inheritance

i. Single ii. Multi-Level iii. Hierarchical iv. Hybrid

```
// **1. Single Inheritance**

class Animal {

    void sound() {

        System.out.println("Animals make sounds.");

    }

}


class Dog extends Animal {  // Dog inherits from Animal

    void bark() {

        System.out.println("Dog barks.");

    }

}


// **2. Multi-Level Inheritance**

class Puppy extends Dog {  // Puppy inherits from Dog

    void weep() {

        System.out.println("Puppy weeps.");

    }

}


// **3. Hierarchical Inheritance**

class Cat extends Animal {  // Cat also inherits from Animal

    void meow() {

        System.out.println("Cat meows.");
```

```
    }
}

// **4. Hybrid Inheritance (Using Interfaces)**
interface Wild {
    void habitat();
}

class Tiger extends Animal implements Wild {  // Tiger inherits Animal and implements Wild
    public void habitat() {
        System.out.println("Tiger lives in the jungle.");
    }
    void roar() {
        System.out.println("Tiger roars.");
    }
}

// **Main Class to Test All Inheritance Types**
public class InheritanceDemo {
    public static void main(String[] args) {
        // **Single Inheritance**
        System.out.println("**Single Inheritance**");
        Dog dog = new Dog();
        dog.sound();
        dog.bark();

        // **Multi-Level Inheritance**
```

```
        System.out.println("\n**Multi-Level Inheritance**");
        Puppy puppy = new Puppy();
        puppy.sound();
        puppy.bark();
        puppy.weep();

        // **Hierarchical Inheritance**
        System.out.println("\n**Hierarchical Inheritance**");
        Cat cat = new Cat();
        cat.sound();
        cat.meow();

        // **Hybrid Inheritance (via Interface)**
        System.out.println("\n**Hybrid Inheritance**");
        Tiger tiger = new Tiger();
        tiger.sound();
        tiger.roar();
        tiger.habitat();
    }
}
```

Output :-

```
**Single Inheritance**
Animals make sounds.
Dog barks.

**Multi-Level Inheritance**
Animals make sounds.
```

Dog barks.

Puppy weeps.

Hierarchical Inheritance

Animals make sounds.

Cat meows.

Hybrid Inheritance

Animals make sounds.

Tiger roars.

Tiger lives in the jungle.

Write a java program to implement runtime polymorphism.

```
// Parent class
class Animal {
    void makeSound() {  // Method to be overridden
        System.out.println("Animals make different sounds.");
    }
}

// Subclass 1: Dog
class Dog extends Animal {
    @Override
    void makeSound() {  // Overriding parent method
        System.out.println("Dog barks.");
    }
}

// Subclass 2: Cat
class Cat extends Animal {
    @Override
    void makeSound() {  // Overriding parent method
        System.out.println("Cat meows.");
    }
}

// Subclass 3: Cow
class Cow extends Animal {
    @Override
```

```
    void makeSound() {  // Overriding parent method
        System.out.println("Cow moos.");
    }
}

// Main class to demonstrate runtime polymorphism
public class RuntimePolymorphismDemo {
    public static void main(String[] args) {
        // Parent class reference holding subclass objects
        Animal myAnimal;

        myAnimal = new Dog();
        myAnimal.makeSound();  // Calls Dog's makeSound()

        myAnimal = new Cat();
        myAnimal.makeSound();  // Calls Cat's makeSound()

        myAnimal = new Cow();
        myAnimal.makeSound();  // Calls Cow's makeSound()
    }
}
```

Output :-

```
Dog barks.
Cat meows.
Cow moos.
```

Write a Java program which accepts withdraw amount from the user and throws an exception “In Sufficient Funds” when withdraw amount more than available amount.

```
import java.util.Scanner;


// Custom exception for insufficient funds
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}


// Bank Account class
class BankAccount {
    private double balance;


    // Constructor to initialize balance
    public BankAccount(double balance) {
        this.balance = balance;
    }


    // Method to withdraw money
    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient Funds! Available balance: " + balance);
        } else {
```

```
            balance -= amount;
            System.out.println("Withdrawal successful! Remaining balance: " + balance);
        }
    }

    // Method to display current balance
    public void displayBalance() {
        System.out.println("Current balance: " + balance);
    }
}

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

        // Initialize account with some balance
        BankAccount account = new BankAccount(5000);

        System.out.println("Welcome to the Bank");
        account.displayBalance();

        // Get withdrawal amount from user
        System.out.print("Enter amount to withdraw: ");
        double withdrawAmount = scanner.nextDouble();

        try {
```

```
            account.withdraw(withdrawAmount);
        } catch (InsufficientFundsException e) {
            System.out.println("Error: " + e.getMessage());
        }

        scanner.close();
    }
}
```

Output :-

```
Welcome to the Bank
Current balance: 5000.0
Enter amount to withdraw: 2000
Withdrawal successful! Remaining balance: 3000.0

Welcome to the Bank
Current balance: 5000.0
Enter amount to withdraw: 7000
Error: Insufficient Funds! Available balance: 5000.0
```

Write a Java program to create three threads and that displays “good morning”, for every one second, “hello” for every 2 seconds and “welcome” for every 3 seconds by using extending Thread class.

```
// Thread 1: Prints "Good Morning" every 1 second
class GoodMorningThread extends Thread {
    public void run() {
        try {
            while (true) {
                System.out.println("Good Morning");
                Thread.sleep(1000); // 1 second delay
            }
        } catch (InterruptedException e) {
            System.out.println("GoodMorningThread interrupted.");
        }
    }
}

// Thread 2: Prints "Hello" every 2 seconds
class HelloThread extends Thread {
    public void run() {
        try {
            while (true) {
                System.out.println("Hello");
                Thread.sleep(2000); // 2 seconds delay
            }
        } catch (InterruptedException e) {
```

```
            System.out.println("HelloThread interrupted.");
        }
    }
}

// Thread 3: Prints "Welcome" every 3 seconds
class WelcomeThread extends Thread {
    public void run() {
        try {
            while (true) {
                System.out.println("Welcome");
                Thread.sleep(3000); // 3 seconds delay
            }
        } catch (InterruptedException e) {
            System.out.println("WelcomeThread interrupted.");
        }
    }
}

// Main class
public class MultiThreadDemo {
    public static void main(String[] args) {
        // Creating thread objects
        GoodMorningThread t1 = new GoodMorningThread();
        HelloThread t2 = new HelloThread();
        WelcomeThread t3 = new WelcomeThread();
```

```
        // Starting the threads
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output :-

```
Good Morning
Hello
Good Morning
Welcome
Good Morning
Hello
Good Morning
Good Morning
Welcome
Hello
...
```

Write a Java program that creates three threads. First thread displays “OOPS”, the second thread displays “Through” and the third thread Displays “JAVA” by using Runnable interface.

```
// Implementing Runnable Interface for multi-threading

class PrintMessage implements Runnable {

    private String message;

    private int delay;


    // Constructor to initialize message and delay

    public PrintMessage(String message, int delay) {

        this.message = message;

        this.delay = delay;

    }


    public void run() {

        try {

            while (true) {

                System.out.println(message);

                Thread.sleep(delay);  // Pause the thread for given milliseconds

            }

        } catch (InterruptedException e) {

            System.out.println(Thread.currentThread().getName() + " interrupted.");

        }

    }

}
```

```
// Main class
public class RunnableThreadDemo {
    public static void main(String[] args) {
        // Creating Runnable objects with different messages
        Runnable r1 = new PrintMessage("OOPS", 1000);      // 1-second delay
        Runnable r2 = new PrintMessage("Through", 1500);   // 1.5-second delay
        Runnable r3 = new PrintMessage("JAVA", 2000);      // 2-second delay

        // Creating Threads and assigning Runnable objects
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        Thread t3 = new Thread(r3);

        // Starting the threads
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output :-

```
OOPS
Through
JAVA
OOPS
Through
OOPS
JAVA
Through
OOPS
...
```

Implement a Java program for handling mouse events when the mouse entered, exited, clicked, pressed, released, dragged and moved in the client area.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;


// Class that handles mouse events

class MouseEventDemo extends JFrame implements MouseListener, MouseMotionListener {

   private JLabel statusLabel;


   // Constructor to set up the GUI

   public MouseEventDemo() {

      setTitle("Mouse Event Demo");

      setSize(400, 300);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      setLayout(new BorderLayout());


      // Label to display mouse events

      statusLabel = new JLabel("Perform mouse actions in the window", JLabel.CENTER);

      add(statusLabel, BorderLayout.SOUTH);


      // Add mouse listeners to the frame

      addMouseListener(this);

      addMouseMotionListener(this);
```

```
    setVisible(true);
}

// MouseListener Methods
public void mouseEntered(MouseEvent e) {
    statusLabel.setText("Mouse Entered the Window");
}

public void mouseExited(MouseEvent e) {
    statusLabel.setText("Mouse Exited the Window");
}

public void mouseClicked(MouseEvent e) {
    statusLabel.setText("Mouse Clicked at (" + e.getX() + ", " + e.getY() + ")");
}

public void mousePressed(MouseEvent e) {
    statusLabel.setText("Mouse Pressed at (" + e.getX() + ", " + e.getY() + ")");
}

public void mouseReleased(MouseEvent e) {
    statusLabel.setText("Mouse Released at (" + e.getX() + ", " + e.getY() + ")");
}

// MouseMotionListener Methods
public void mouseMoved(MouseEvent e) {
    statusLabel.setText("Mouse Moved to (" + e.getX() + ", " + e.getY() + ")");
```

```
    }

    public void mouseDragged(MouseEvent e) {
        statusLabel.setText("Mouse Dragged to (" + e.getX() + ", " + e.getY() + ")");
    }

    // Main method to run the program
    public static void main(String[] args) {
        new MouseEventDemo();
    }
}
```

Output :-

- **Mouse Clicked at (100, 150)**
- **Mouse Moved to (250, 180)**
- **Mouse Dragged to (300, 200)**
- **Mouse Entered the Window**
- **Mouse Exited the Window**

Implement a Java program for handling key events when the key board is pressed, released, typed.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;


// Class that handles key events

class KeyEventDemo extends JFrame implements KeyListener {

   private JLabel statusLabel;


   // Constructor to set up GUI

   public KeyEventDemo() {

      setTitle("Key Event Demo");

      setSize(400, 200);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      setLayout(new BorderLayout());


      // Label to display key events

      statusLabel = new JLabel("Press any key...", JLabel.CENTER);

      add(statusLabel, BorderLayout.CENTER);


      // Add KeyListener to JFrame

      addKeyListener(this);

      setFocusable(true);

      setVisible(true);

   }
```

```
    // KeyListener methods
    public void keyPressed(KeyEvent e) {
        statusLabel.setText("Key Pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
    }

    public void keyReleased(KeyEvent e) {
        statusLabel.setText("Key Released: " + KeyEvent.getKeyText(e.getKeyCode()));
    }

    public void keyTyped(KeyEvent e) {
        statusLabel.setText("Key Typed: " + e.getKeyChar());
    }

    // Main method to run the program
    public static void main(String[] args) {
        new KeyEventDemo();
    }
}
```

Output :-

```
Key Pressed: A
Key Released: A
Key Typed: a

Key Pressed: Enter
Key Released: Enter
```

Write a Java swing program that reads two numbers from two separate text fields and display sum of two numbers in third text field when button “add” is pressed.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;


// Class for the addition GUI

class AdditionGUI extends JFrame implements ActionListener {

    private JTextField num1Field, num2Field, resultField;

    private JButton addButton;


    // Constructor to set up GUI

    public AdditionGUI() {

        setTitle("Addition Calculator");

        setSize(350, 200);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new GridLayout(4, 2, 5, 5));


        // Create Labels and TextFields

        add(new JLabel("Number 1:"));

        num1Field = new JTextField();

        add(num1Field);


        add(new JLabel("Number 2:"));

        num2Field = new JTextField();

        add(num2Field);
```

```
    add(new JLabel("Result:"));
    resultField = new JTextField();
    resultField.setEditable(false); // Make result field non-editable
    add(resultField);

    // Add button
    addButton = new JButton("Add");
    addButton.addActionListener(this);
    add(addButton);

    setVisible(true);
}

// Handle button click event
public void actionPerformed(ActionEvent e) {
    try {
        double num1 = Double.parseDouble(num1Field.getText());
        double num2 = Double.parseDouble(num2Field.getText());
        double sum = num1 + num2;

        resultField.setText(String.valueOf(sum)); // Display result
    } catch (NumberFormatException ex) {
        resultField.setText("Invalid Input");
    }
}
```

```
    // Main method to run the program
    public static void main(String[] args) {
        new AdditionGUI();
    }
}
```

Output :-

Number 1: 15

Number 2: 25

[Add] → Result: 40

Number 1: abc

Number 2: 20

[Add] → Result: Invalid Input

Write a Java program to design student registration form using Swing Controls. The form which having the following fields and button SAVE

Form Fields are: Name, RNO, Mailid, Gender, Branch, Address.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;


// Class for the Student Registration Form

class StudentRegistrationForm extends JFrame implements ActionListener {

    private JTextField nameField, rnoField, mailField;

    private JRadioButton maleRadio, femaleRadio;

    private JComboBox<String> branchBox;

    private JTextArea addressArea;

    private JButton saveButton;


    // Constructor to set up GUI

    public StudentRegistrationForm() {

        setTitle("Student Registration Form");

        setSize(400, 400);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new GridLayout(7, 2, 5, 5));


        // Name

        add(new JLabel("Name:"));

        nameField = new JTextField();
```

```
add(nameField);

// Roll Number
add(new JLabel("RNO:"));
rnoField = new JTextField();
add(rnoField);

// Mail ID
add(new JLabel("Mail ID:"));
mailField = new JTextField();
add(mailField);

// Gender
add(new JLabel("Gender:"));
JPanel genderPanel = new JPanel();
maleRadio = new JRadioButton("Male");
femaleRadio = new JRadioButton("Female");
ButtonGroup genderGroup = new ButtonGroup();
genderGroup.add(maleRadio);
genderGroup.add(femaleRadio);
genderPanel.add(maleRadio);
genderPanel.add(femaleRadio);
add(genderPanel);

// Branch (Drop-down)
add(new JLabel("Branch:"));
String[] branches = { "CSE", "ECE", "EEE", "MECH", "CIVIL" };
```

```
        branchBox = new JComboBox<>(branches);
        add(branchBox);

        // Address
        add(new JLabel("Address:"));
        addressArea = new JTextArea(3, 20);
        add(new JScrollPane(addressArea));

        // Save Button
        saveButton = new JButton("SAVE");
        saveButton.addActionListener(this);
        add(saveButton);

        setVisible(true);
    }

    // ActionListener for SAVE Button
    public void actionPerformed(ActionEvent e) {
        String name = nameField.getText();
        String rno = rnoField.getText();
        String mail = mailField.getText();
        String gender = maleRadio.isSelected() ? "Male" : (femaleRadio.isSelected() ? "Female" : "Not
Selected");
        String branch = (String) branchBox.getSelectedItem();
        String address = addressArea.getText();

        // Display input details
```

```
        JOptionPane.showMessageDialog(this,
            "Student Details:\n" +
            "Name: " + name + "\n" +
            "RNO: " + rno + "\n" +
            "Mail ID: " + mail + "\n" +
            "Gender: " + gender + "\n" +
            "Branch: " + branch + "\n" +
            "Address: " + address);
    }

    // Main method to run the program
    public static void main(String[] args) {
        new StudentRegistrationForm();
    }
}
```

Output :-

Name: John Doe
RNO: 12345
Mail ID: johndoe@gmail.com
Gender: Male
Branch: CSE
Address: 123, Main Street, NY
[SAVE]

Student Details:

Name: John Doe
RNO: 12345
Mail ID: johndoe@gmail.com
Gender: Male
Branch: CSE
Address: 123, Main Street, NY

Viva - Questions

Here are **VIVA questions and answers** based on **UNIT-I: OOP Concepts and Java Programming**

---

1. Object-Oriented Programming (OOP) Concepts

Q1: What is Object-Oriented Programming (OOP)?

A: Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of objects. It allows data and behavior to be encapsulated within objects, promoting modularity and reusability.

Q2: How does OOP differ from procedural programming?

A:

Feature	Procedural Programming	Object-Oriented Programming
Approach	Follows a step-by-step approach	Based on objects and classes
Data Access	Data is accessible throughout the program	Data is encapsulated within objects
Reusability	Low code reusability	High code reusability through inheritance
Examples	C, Pascal	Java, C++

Q3: What are the four main principles of OOP?

A:

1. **Encapsulation** – Hiding data using access modifiers (private, protected, public).
2. **Abstraction** – Hiding implementation details and exposing only the necessary functionality.
3. **Inheritance** – Allowing a class to derive properties from another class.
4. **Polymorphism** – Ability to take multiple forms (method overloading & overriding).

Q4: What is encapsulation?

A: Encapsulation is the technique of wrapping data (variables) and code (methods) together in a single unit, restricting direct access from outside the class. Example:

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

Q5: What is inheritance?

A: Inheritance allows a class (child) to acquire properties and behaviors of another class (parent). It helps in code reusability.

Example:

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

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

Q6: What is polymorphism?

A: Polymorphism means "many forms". It allows methods to be used in different ways.

- **Method Overloading** (Same method name, different parameters)
- **Method Overriding** (Redefining inherited method in the child class)

Example:

```
class Shape {
    void draw() { System.out.println("Drawing a shape"); }
}
class Circle extends Shape {
    void draw() { System.out.println("Drawing a circle"); }  // Overriding
}
```

Q7: What is abstraction?

A: Abstraction hides the internal details and shows only the necessary parts of an object. It is implemented using **abstract classes** and **interfaces**.

Example:

```
abstract class Vehicle {
    abstract void start();
}

class Car extends Vehicle {
    void start() { System.out.println("Car starts with a key"); }
}
```

---

2. Java Basics

Q8: What are the key features of Java?

A: Java is

- **Platform-Independent** (Runs on JVM)

- **Object-Oriented**
- **Secure** (No pointers)
- **Robust** (Exception handling, Garbage collection)
- **Multithreaded**

Q9: What is the Java Virtual Machine (JVM)?

A: JVM is an interpreter that converts **Java bytecode** into machine code for execution. It makes Java platform-independent.

Q10: What is the difference between JDK, JRE, and JVM?

Component	Description
JDK (Java Development Kit)	Contains tools for Java development (JRE + Compiler)
JRE (Java Runtime Environment)	Contains JVM + libraries required to run Java programs
JVM (Java Virtual Machine)	Converts bytecode into machine code for execution

Q11: What is bytecode in Java?

A: Bytecode is an intermediate code generated after Java compilation. It is executed by the JVM.

---

3. Data Types, Variables & Operators

Q12: What are the different data types in Java?

A:

- **Primitive Types**: `int, float, char, boolean, double, byte, short, long`
- **Non-Primitive Types**: `String, Arrays, Classes, Interfaces`

Q13: What is type conversion?

A: Type conversion automatically converts a smaller data type into a larger type. Example:

```
int num = 10;
double d = num;  // Implicit conversion
```

Q14: What is type casting?

A: Type casting explicitly converts one data type to another. Example:

```
double d = 10.5;
int num = (int) d;  // Explicit casting
```

---

4. Input & Output in Java

Q15: How do you take input in Java?

A: Using `Scanner` class. Example:

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
int num = sc.nextInt();
```

Q16: What is the difference between `System.out.print()`, `System.out.println()`, and `System.out.printf()`?

Method	Description
`print()`	Prints text without a newline
`println()`	Prints text with a newline
`printf()`	Formats output using placeholders (like C's `printf`)

Q17: How do you format output in Java?

A: Using `String.format()` or `System.out.printf()`. Example:

```
System.out.printf("Price: %.2f", 99.456);
```

Output: `Price: 99.46`

---

5. Control Statements

Q18: What are control statements in Java?

A: Control statements are used for decision-making and loops.

Q19: What are the different types of control statements in Java?

A:

1. **Decision-Making**: `if-else`, `switch`
2. **Loops**: `for`, `while`, `do-while`
3. **Branching**: `break`, `continue`, `return`

Q20: What is the difference between `if-else` and `switch`?

Feature	if-else	switch
Type	Checks conditions	Works with values (int, char, String)

Feature	if-else	switch
Performance	Slower for multiple cases	Faster for multiple fixed values

Example:

```
int num = 2;
switch(num) {
    case 1: System.out.println("One"); break;
    case 2: System.out.println("Two"); break;
}
```

Q21: What is the difference between `break` and `continue`?

A:

- `break`: Exits the loop completely.
- `continue`: Skips the current iteration and moves to the next iteration.

Example:

```
for(int i = 1; i <= 5; i++) {
    if(i == 3) continue;  // Skips 3
    System.out.println(i);
}
```

Output: `1 2 4 5`

---

1. Arrays and Command Line Arguments

Q1: What is an array in Java?

A: An array is a collection of elements of the same data type stored in contiguous memory locations. Example:

```
int[] arr = {10, 20, 30, 40};
```

Q2: How do you declare and initialize an array in Java?

A:

```
int[] numbers = new int[5]; // Declaration and initialization
numbers[0] = 10;
```

Q3: What are command-line arguments in Java?

A: Command-line arguments are inputs passed to the `main` method when running a Java program. Example:

```
public class Test {
    public static void main(String args[]) {
        System.out.println("Argument: " + args[0]);
    }
}
```

Run as:

```
java Test Hello
```

Output: `Argument: Hello`

---

2. Strings and String Class Methods

Q4: What is a String in Java?

A: A String is an immutable sequence of characters stored as a `String` object.

Q5: What are some common String methods?

A:

Method	Description	Example
`length()`	Returns string length	`"hello".length()` → `5`
`charAt()`	Gets a character at an index	`"hello".charAt(1)` → `e`
`substring()`	Extracts part of a string	`"hello".substring(1,4)` → `ell`
`equals()`	Compares strings (case-sensitive)	`"hello".equals("HELLO")` → `false`
`toUpperCase()`	Converts to uppercase	`"hello".toUpperCase()` → `HELLO`
`concat()`	Joins two strings	`"hello".concat(" world")` → `hello world`

---

3. Classes & Objects

Q6: What is a class in Java?

A: A class is a blueprint for objects. Example:

```
class Car {
    String model;
}
```

Q7: What is an object?

A: An object is an instance of a class. Example:

```
Car c = new Car();
```

Q8: What is a constructor?

A: A constructor is a special method used to initialize objects. Example:

```
class Car {
    String model;
    Car(String m) { model = m; } // Constructor
}
```

Q9: What is the 'this' keyword?

A: It refers to the current object's instance variables. Example:

```
class Car {
    String model;
    Car(String model) { this.model = model; }
}
```

Q10: What are static methods and fields?

A: Static members belong to the class, not objects. Example:

```
class Test {
    static int count = 0; // Static field
    static void show() { System.out.println(count); } // Static method
}
```

Q11: What is method overloading?

A: Method overloading allows multiple methods with the same name but different parameters. Example:

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

---

4. Inheritance

Q12: What is inheritance in Java?

A: Inheritance allows a class to acquire properties and methods of another class.

Q13: How do you implement inheritance?

A:

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

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

Q14: What is the 'super' keyword?

A: `super` is used to refer to the parent class. Example:

```
class Animal {
    Animal() { System.out.println("Animal constructor"); }
}
class Dog extends Animal {
    Dog() { super(); System.out.println("Dog constructor"); }
}
```

Q15: How do you prevent inheritance?

A: Using the `final` keyword.

```
final class A {} // Cannot be inherited
```

Q16: What is the Object class?

A: `Object` is the parent of all Java classes. Common methods: `toString()`, `equals()`, `hashCode()`.

---

5. Polymorphism

Q17: What is method overriding?

A: When a subclass provides a specific implementation of a method already defined in its parent class. Example:

```
class Parent {
    void show() { System.out.println("Parent method"); }
}
class Child extends Parent {
    void show() { System.out.println("Child method"); } // Overriding
}
```

Q18: What is dynamic binding?

A: Dynamic method dispatch happens when a method is called at runtime. Example:

```
Parent obj = new Child(); // Dynamic binding
obj.show(); // Calls overridden method in Child class
```

Q19: What are abstract classes and methods?

A:

- **Abstract class:** Cannot be instantiated, may contain abstract methods.
- **Abstract method:** A method without a body, must be implemented by subclasses.

Example:

```
abstract class Animal {
    abstract void makeSound(); // Abstract method
}
class Dog extends Animal {
    void makeSound() { System.out.println("Bark"); }
}
```

Here are **VIVA questions with answers** based on **UNIT-III: Interfaces, Packages, and Exception Handling**.

---

1. Interfaces

Q1: What is an interface in Java?

A: An interface is a collection of abstract methods and constants. It is used to achieve multiple inheritance in Java.

Q2: How does an interface differ from an abstract class?

Feature	Abstract Class	Interface
Methods	Can have abstract & concrete methods	Only abstract methods (until Java 8)
Variables	Can have instance variables	Only static and final variables
Constructors	Can have constructors	Cannot have constructors
Inheritance	Supports single inheritance	Supports multiple inheritance

Q3: How do you define an interface?

A:

```
interface Animal {
    void makeSound(); // Abstract method
}
```

Q4: How do you implement an interface?

A:

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

Q5: Can an interface extend another interface?

A: Yes, an interface can extend another interface. Example:

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

```
interface B extends A { void display(); }
```

Q6: How do you access an implementation through an interface reference?

A:

```
Animal obj = new Dog(); // Accessing through interface reference
obj.makeSound();
```

---

2. Packages

Q7: What is a package in Java?

A: A package is a collection of related classes and interfaces. It is used for **encapsulation and reusability**.

Q8: How do you define and use a package?

A:

- **Define a package:**

```
package mypackage;
public class MyClass { public void show() { System.out.println("Hello"); } }
```

- **Use the package:**

```
import mypackage.MyClass;
MyClass obj = new MyClass();
obj.show();
```

Q9: What is CLASSPATH in Java?

A: `CLASSPATH` is an environment variable that tells Java where to find compiled class files.

Q10: How do you import a package?

A:

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

Q11: What is the difference between `import package.*` and `import package.ClassName`?

A:

- `import package.*` imports all classes from the package.
- `import package.ClassName` imports only the specified class.

---

3. Exception Handling

Q12: What is exception handling in Java?

A: Exception handling is a mechanism to handle runtime errors and prevent program crashes.

Q13: What are the benefits of exception handling?

A:

- Improves program robustness
- Separates error-handling code
- Prevents abnormal program termination

Q14: What is the difference between checked and unchecked exceptions?

Type	Checked Exception	Unchecked Exception
When	Checked at compile-time	Checked at runtime
Examples	`IOException`, `SQLException`	`NullPointerException`, `ArithmeticException`

Q15: What is the exception hierarchy in Java?

A:

```
Throwable
  |-- Exception
  |     |-- IOException
  |     |-- SQLException
  |     |-- RuntimeException
  |           |-- NullPointerException
  |           |-- ArithmeticException
  |
  |-- Error (e.g., OutOfMemoryError)
```

Q16: How do you use `try` and `catch`?

A:

```
try {
    int a = 5 / 0; // ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
}
```

Q17: What is the use of `throw`?

A: `throw` is used to manually throw an exception. Example:

```
throw new ArithmeticException("Custom Exception");
```

Q18: What is the use of `throws`?

A: `throws` declares exceptions in a method signature. Example:

```
void myMethod() throws IOException { ... }
```

Q19: What is the `finally` block?

A: The `finally` block always executes, whether an exception occurs or not. Example:

```
try { int a = 5 / 0; }
catch (ArithmeticException e) { System.out.println("Error"); }
finally { System.out.println("This will always execute"); }
```

Q20: What is exception rethrowing?

A: Rethrowing an exception allows the exception to be handled at a higher level. Example:

```
void myMethod() throws IOException {
    try { throw new IOException("Error"); }
    catch (IOException e) { System.out.println("Caught and rethrown"); throw
e; }
}
```

Q21: What are built-in exceptions in Java?

A:

Exception	Description
`ArithmeticException`	Division by zero
`NullPointerException`	Accessing a null object
`ArrayIndexOutOfBoundsException`	Accessing an invalid array index
`FileNotFoundException`	File not found

Q22: How do you create a custom exception in Java?

A:

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

These **VIVA questions and answers** cover **Interfaces, Packages, and Exception Handling**. Do you need more detailed examples or explanations? 😊

Here are **VIVA questions with answers** based on **UNIT-IV: Multithreading & Stream-Based I/O (java.io)**.

1. Multithreading

Q1: What is multithreading in Java?

A: Multithreading is the ability of a program to execute multiple threads concurrently, improving performance.

Q2: What is the difference between a process and a thread?

Feature	Process	Thread
Definition	A program in execution	A lightweight subprocess
Memory	Each process has separate memory	Threads share the same memory
Communication	Inter-process communication required	Threads communicate easily
Example	Running multiple applications	Running multiple tasks in one application

Q3: What are the states in a thread life cycle?

A:

1. **New** – Thread created but not started
2. **Runnable** – Ready to run
3. **Blocked** – Waiting for a resource
4. **Waiting** – Waiting indefinitely
5. **Timed Waiting** – Waiting for a specific time
6. **Terminated** – Thread execution completed

Q4: How do you create a thread in Java?

A: Using **Thread class** or **Runnable interface**.

1 Extending Thread class:

```
class MyThread extends Thread {
    public void run() { System.out.println("Thread running"); }
}
MyThread t = new MyThread();
t.start();
```

2 Implementing Runnable interface:

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

Q5: How do you interrupt a thread?

A: Using `interrupt()` method.

```
t.interrupt();
```

Q6: What are thread priorities?

A: Thread priorities range from **1 (MIN_PRIORITY)** to **10 (MAX_PRIORITY)**. Default is **5 (NORM_PRIORITY)**.

```
t.setPriority(Thread.MAX_PRIORITY);
```

Q7: What is synchronization? Why is it needed?

A: Synchronization prevents multiple threads from accessing a shared resource simultaneously, avoiding data inconsistency.

Q8: How do you synchronize a method?

A: Using the `synchronized` keyword.

```
synchronized void print() { ... }
```

Q9: What is inter-thread communication?

A: It allows threads to communicate using `wait()`, `notify()`, and `notifyAll()`.

```
synchronized(obj) {
    obj.wait(); // Thread waits
    obj.notify(); // Wakes up a waiting thread
}
```

---

2. Stream-Based I/O (java.io)

Q10: What is the difference between byte streams and character streams?

Feature	Byte Stream (`InputStream`, `OutputStream`)	Character Stream (`Reader`, `Writer`)
Data type	Reads/writes **binary data**	Reads/writes **text data**
Example	`FileInputStream`, `FileOutputStream`	`FileReader`, `FileWriter`

Q11: How do you read input from the console in Java?

A: Using `Scanner` or `BufferedReader`.

```
Scanner sc = new Scanner(System.in);
String input = sc.nextLine();
```

Q12: How do you write output to the console?

A: Using `System.out.print()` or `System.out.printf()`.

Q13: What is the File class?

A: `File` represents file and directory pathnames in Java. Example:

```
File f = new File("test.txt");
```

Q14: How do you check if a file exists?

A:

```
if (f.exists()) System.out.println("File exists");
```

Q15: How do you read a file in Java?

A: Using `FileReader`.

```
FileReader fr = new FileReader("file.txt");
int ch;
while ((ch = fr.read()) != -1) { System.out.print((char) ch); }
```

Q16: How do you write to a file in Java?

A: Using `FileWriter`.

```
FileWriter fw = new FileWriter("file.txt");
fw.write("Hello, world!");
fw.close();
```

Q17: What is the Console class in Java?

A: `Console` is used for interactive input/output without `Scanner`.

```
Console console = System.console();
String name = console.readLine("Enter name: ");
```

---

3. Serialization

Q18: What is serialization in Java?

A: Serialization converts an object into a byte stream to save or transfer it.

Q19: How do you make a class serializable?

A: By implementing `Serializable` interface.

```
class Person implements Serializable {
    String name;
    int age;
}
```

Q20: How do you serialize an object?

A: Using `ObjectOutputStream`.

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data.ser"));
out.writeObject(new Person("John", 25));
out.close();
```

Q21: How do you deserialize an object?

A: Using `ObjectInputStream`.

```
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("data.ser"));
Person p = (Person) in.readObject();
in.close();
```

These **VIVA questions and answers** cover **Multithreading & Stream-Based I/O**. Let me know if you need further explanations or examples! 😊

Here are **VIVA questions with answers** based on **UNIT-V: GUI Programming with Swing & Event Handling**.

---

1. GUI Programming with Swing

Q1: What is Swing in Java?

A: Swing is a part of Java's **javax.swing** package that provides a rich set of GUI components like buttons, text fields, tables, etc.

Q2: What is the difference between Swing and AWT?

Feature	AWT (Abstract Window Toolkit)	Swing
Lightweight	No (depends on OS)	Yes (pure Java)
Performance	Slower	Faster
Components	Fewer	More advanced
Example	`Button`, `TextField`	`JButton`, `JTextField`

Q3: What is MVC architecture in Swing?

A: Swing follows the **Model-View-Controller (MVC)** pattern:

- **Model** – Stores data
- **View** – Displays data
- **Controller** – Handles user interaction

Example: `JTable` uses **TableModel (Model), JTable (View), ListSelectionListener (Controller)**.

Q4: What are Swing containers?

A: Containers hold GUI components.

- **Top-Level Containers:** `JFrame`, `JDialog`, `JApplet`
- **Intermediate Containers:** `JPanel`, `JSplitPane`, `JScrollPane`
- **Lightweight Containers:** Derived from `JComponent`

Q5: How do you create a simple Swing application?

A:

```
import javax.swing.*;
public class MySwingApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Example");
        JButton button = new JButton("Click Me");
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

---

2. Layout Managers

Q6: What are layout managers in Java?

A: Layout managers arrange components inside a container.

Q7: What are the types of layout managers in Java?

Layout	Description
FlowLayout	Default layout, arranges components in a row
BorderLayout	Divides container into North, South, East, West, Center
GridLayout	Arranges components in a grid (rows & columns)
CardLayout	Allows switching between different components
GridBagLayout	Flexible grid-based layout with different sizes

Q8: How do you use FlowLayout?

```
JPanel panel = new JPanel(new FlowLayout());
```

Q9: How do you use BorderLayout?

```
JFrame frame = new JFrame();
frame.setLayout(new BorderLayout());
frame.add(new JButton("North"), BorderLayout.NORTH);
```

---

3. Event Handling

Q10: What is event handling in Java?

A: Event handling allows a program to respond to user interactions like clicks, key presses, etc.

Q11: What is the Delegation Event Model?

A: In this model:

- **Event Source** – Generates the event (e.g., `JButton`)
- **Event Listener** – Listens and responds to the event (e.g., `ActionListener`)
- **Event Object** – Contains event details (e.g., `ActionEvent`)

Q12: What are different types of event listeners in Java?

Listener	Description
`ActionListener`	Handles button clicks
`MouseListener`	Handles mouse events
`KeyListener`	Handles keyboard events
`WindowListener`	Handles window events

Q13: How do you handle a button click event?

```
JButton button = new JButton("Click Me");
button.addActionListener(e -> System.out.println("Button Clicked!"));
```

Q14: What are event classes in Java?

Event Class	Description
`ActionEvent`	Button clicks, menu actions
`MouseEvent`	Mouse clicks, movements
`KeyEvent`	Keyboard key press, release
`WindowEvent`	Window open, close, minimize

---

4. Mouse & Keyboard Events

Q15: How do you handle mouse events in Java?

A: Using `MouseListener`.

```
button.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse Clicked");
    }
});
```

Q16: How do you handle keyboard events in Java?

A: Using `KeyListener`.

```
textField.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        System.out.println("Key Pressed: " + e.getKeyChar());
    }
});
```

---

5. Adapter & Inner Classes

Q17: What is an adapter class?

A: Adapter classes provide empty implementations of listener methods, so we can override only the required methods.

Q18: How do you use an adapter class?

```
button.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) { System.out.println("Mouse
Pressed"); }
});
```

Q19: What is an inner class in Java?

A: A class defined inside another class. Used for better encapsulation.

Q20: What is an anonymous inner class?

A: A class without a name, used to define event handlers inline.

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button Clicked");
    }
});
```

These **VIVA questions and answers** cover **Swing, Layout Managers, and Event Handling**. Let me know if you need more details or examples! 😊

IMPORTANT QUESTIONS

UNIT - 1

Short Answer Questions (SAQs)

1. Introduction to OOP Concepts & Paradigms

1. What is Object-Oriented Programming (OOP)?
2. How is OOP different from procedural programming?
3. What are the four main principles of OOP?
4. Define **class** and **object** with examples.
5. What is encapsulation? Provide an example.
6. How does inheritance work in Java?
7. What is polymorphism? Explain its types.
8. What is abstraction? How is it implemented in Java?
9. What is the difference between method overloading and method overriding?
10. Why is Java called a **purely object-oriented language**?

2. Java Programming Basics

11. What is the Java Virtual Machine (JVM)?
12. What is JDK, JRE, and JVM? How are they related?
13. Explain Java's platform-independent nature.
14. What are the different data types in Java?
15. What is the scope and lifetime of a variable?
16. How do you declare a constant in Java?
17. What is the difference between implicit and explicit type casting?
18. How do you take input from the user in Java using `Scanner`?
19. Explain `System.out.printf()` with an example.
20. How does Java handle type conversion?

3. Control Statements

21. What are the different types of control statements in Java?
22. How does an `if-else` statement work?
23. What is the syntax of a `switch` statement in Java?
24. What are the different types of loops in Java?
25. What is the difference between `break` and `continue`?
26. What is the purpose of a `do-while` loop?
27. Explain the use of `for-each` loop in Java.
28. What is the difference between an entry-controlled and an exit-controlled loop?
29. How does the `while` loop work?
30. What is an infinite loop? How can you create one?

Long Answer Questions (LAQs)

1. Object-Oriented Concepts & Paradigms

1. Explain the **difference between procedural and object-oriented programming** with examples.

2. Discuss the **four pillars of OOP** with real-world examples.
3. What is **inheritance**? Explain **types of inheritance** in Java with code examples.
4. What is **polymorphism**? Explain **method overloading and method overriding** with examples.
5. How does **encapsulation** improve security in Java? Provide an example.
6. Explain **abstraction** in Java. How is it implemented using **abstract classes and interfaces**?
7. What is the **difference between an abstract class and an interface**?
8. How does **Java support multiple inheritance** using interfaces?
9. Explain the **difference between static and instance variables** in Java.
10. What is the role of the **"this" keyword** in Java? Provide examples.

2. Java Programming Basics

11. Explain the **Java environment (JDK, JRE, JVM)** in detail.
12. Describe the **Java compilation and execution process** step by step.
13. What are **primitive data types** in Java? Explain with examples.
14. What is **type conversion and casting**? Explain with examples.
15. Describe **different types of variables in Java** with examples.
16. What are **constants in Java**? How do you declare them?
17. Explain the **Scanner class** with a program to read input from the user.
18. What is **formatted output**? Explain `System.out.printf()` and `String.format()` with examples.
19. How do you handle **input and output operations** in Java?
20. What is the **difference between printf() and print()**?

3. Control Statements

21. Explain **if-else statements** with syntax and examples.
22. What is a **switch-case statement**? How is it different from `if-else`?
23. Explain **different types of loops in Java** with examples.
24. Describe the **for-each loop** and how it is used in arrays.
25. What is **the difference between break, continue, and return statements**?
26. Explain the **difference between while, do-while, and for loops** with examples.
27. What are **nested loops**? Provide an example.
28. How does **loop control flow** work in Java?
29. Explain **jump statements in Java** with examples.
30. How do you use a **for loop to iterate over an array**?

UNIT – II

Here are **Short Answer Questions (SAQs) and Long Answer Questions (LAQs)** based on **UNIT-II: Arrays, Command Line Arguments, Strings, Classes & Objects, Inheritance, and Polymorphism**.

Short Answer Questions (SAQs)

1. Arrays & Command Line Arguments

1. What is an array in Java? How do you declare and initialize an array?

2. What are the different types of arrays in Java?
3. What is the difference between a one-dimensional and a two-dimensional array?
4. How do you find the length of an array in Java?
5. Explain the use of `Arrays.toString()` method.
6. How do you pass an array to a method in Java?
7. What are command-line arguments in Java? How are they used?
8. How can you access command-line arguments in a Java program?
9. What happens if no command-line arguments are passed?
10. Write a simple program to print all command-line arguments.

2. Strings & String Class Methods

11. What is a String in Java? How is it different from a character array?
12. What are immutable Strings in Java?
13. What is the difference between `String`, `StringBuffer`, and `StringBuilder`?
14. What are some common String methods in Java?
15. How do you compare two Strings in Java?
16. What is the difference between `==` and `.equals()` when comparing Strings?
17. How do you convert a String to uppercase or lowercase?
18. What is the purpose of `substring()` method?
19. How do you check if a String is empty or null?
20. What is the use of the `split()` method in String?

3. Classes & Objects

21. What is a class in Java? How is it different from an object?
22. How do you create an object in Java?
23. What is the significance of the `new` keyword in Java?
24. What are instance variables and methods?
25. What is the difference between local, instance, and static variables?
26. How do you pass parameters to methods in Java?
27. What is a static method? How is it different from an instance method?
28. What is a constructor? How is it different from a method?
29. What is the use of the `this` keyword in Java?
30. What is method overloading?

4. Inheritance

31. What is inheritance in Java?
32. What are the different types of inheritance in Java?
33. What is the difference between single and multilevel inheritance?
34. What is a superclass and subclass?
35. What are the access modifiers in Java?
36. What is the `super` keyword? How is it used in Java?
37. How can you prevent inheritance in Java?
38. What is a final class? How does it prevent inheritance?
39. What is the `Object` class in Java? What are its methods?
40. What is method overriding? How is it different from method overloading?

5. Polymorphism

41. What is polymorphism? What are its types?
42. What is dynamic method dispatch?
43. What is runtime polymorphism? Provide an example.
44. What is the difference between static and dynamic binding?
45. What are abstract classes and methods?
46. How do abstract classes support polymorphism?
47. What is the difference between an abstract class and an interface?
48. Can we create an object of an abstract class? Why or why not?
49. How does Java achieve **runtime polymorphism**?
50. What are the advantages of using polymorphism?

Long Answer Questions (LAQs)

1. Arrays & Command Line Arguments

1. What is an array in Java? Explain how to declare, initialize, and access array elements with examples.
2. Explain **multidimensional arrays** in Java with an example.
3. How can arrays be passed to methods in Java? Provide examples.
4. Write a program to find the **largest element in an array**.
5. What are **command-line arguments**? How are they used in Java? Provide an example.

2. Strings & String Class Methods

6. Explain **mutable and immutable Strings** with examples.
7. What is **StringBuffer** and **StringBuilder**? Explain the differences between them.
8. Discuss various **String methods** such as `charAt()`, `substring()`, `indexOf()`, `replace()`, and `split()`.
9. Write a Java program to **count the number of words in a String**.
10. Explain **how to reverse a String in Java** using different approaches.

3. Classes & Objects

11. Explain **how to create and use classes and objects in Java**. Provide a code example.
12. What is the difference between **instance variables, local variables, and static variables**?
13. Explain **constructors in Java**. What are the different types of constructors?
14. How does the **"this" keyword** work in Java? Explain with examples.
15. What is **method overloading**? How does Java resolve overloaded methods?

4. Inheritance

16. Explain **inheritance in Java**. How does it promote code reusability?
17. Discuss **types of inheritance** in Java with examples.
18. What is the **super keyword**? How is it used to access superclass members?
19. Explain **how access modifiers (private, protected, public) affect inheritance**.

20. What is the `Object` class? Explain its common methods like `toString()`, `equals()`, and `hashCode()`.

5. Polymorphism

21. What is **polymorphism**? Explain **compile-time and runtime polymorphism** with examples.
22. Explain **method overriding**. How does it differ from method overloading?
23. What is **dynamic method dispatch**? How does Java determine which method to call at runtime?
24. Explain **abstract classes and methods**. How are they used in Java?
25. What are the differences between **an abstract class and an interface**?

UNIT – III

Here are **Short Answer Questions (SAQs) and Long Answer Questions (LAQs)** for **UNIT-III: Interfaces, Packages, and Exception Handling**.

Short Answer Questions (SAQs)

1. Interfaces

1. What is an interface in Java?
2. How is an interface different from an abstract class?
3. How do you declare an interface in Java?
4. How do you implement an interface in Java?
5. Can a Java class implement multiple interfaces? Explain with an example.
6. What are default methods in an interface?
7. What are static methods in an interface?
8. Can an interface have constructors? Why or why not?
9. How can we access implementations through interface references?
10. Can an interface extend another interface? Explain with an example.

2. Packages

11. What is a package in Java?
12. How do you create a package in Java?
13. How can you access a class from a different package?
14. What is the purpose of the `import` keyword?
15. What is `CLASSPATH` in Java?
16. How does Java handle package naming conflicts?
17. What are built-in packages in Java? Give examples.
18. What is the difference between `import package.*` and `import package.className`?
19. Can a Java class belong to multiple packages? Why or why not?
20. What is the `java.lang` package, and why is it special?

3. Exception Handling

21. What is an exception in Java?

22. What is the difference between an error and an exception?
23. What are the benefits of exception handling?
24. What is the difference between checked and unchecked exceptions?
25. What is exception propagation?
26. What are `try`, `catch`, `finally`, `throw`, and `throws`?
27. Can a `finally` block exist without a `try` block? Why or why not?
28. What is the difference between `throw` and `throws`?
29. What are built-in exceptions in Java? Provide examples.
30. How do you create a custom exception in Java?

Long Answer Questions (LAQs)

1. Interfaces

1. What is an **interface** in Java? How is it different from an **abstract class**?
2. How do you **declare and implement an interface**? Provide an example.
3. Explain **default and static methods** in interfaces with examples.
4. What is **multiple inheritance**? How does Java achieve it using interfaces?
5. What are the advantages of using interfaces in Java?

2. Packages

6. What is a **package**? Explain how to **create and use packages** in Java with an example.
7. What is the purpose of the `CLASSPATH` variable? How do you set it?
8. Explain the **difference between built-in and user-defined packages**.
9. How does **package access control** work in Java? Explain with an example.
10. What are the advantages of using packages in Java?

3. Exception Handling

11. What is **exception handling**? Why is it needed in Java?
12. Explain the **hierarchy of exceptions** in Java.
13. What is the difference between **checked and unchecked exceptions**? Give examples.
14. Explain the **try-catch-finally** mechanism in exception handling.
15. What is **rethrowing an exception**? Provide an example.
16. Explain the **difference between throw and throws** with examples.
17. What are **built-in exceptions** in Java? Explain any five.
18. How do you create a **custom exception** in Java? Provide an example.
19. Explain the **role of multiple catch blocks** in exception handling.
20. Discuss the **importance of the finally block**. When is it executed?

Here are **Short Answer Questions (SAQs) and Long Answer Questions (LAQs)** for **UNIT-IV: Multithreading and Stream-Based I/O (java.io)**.

Short Answer Questions (SAQs)

1. Multithreading

1. What is multithreading in Java?
2. How is multithreading different from multiprocessing?
3. What are the benefits of multithreading?
4. What are the states of a thread in Java?
5. What is the **thread life cycle** in Java?
6. How do you create a thread in Java?
7. What is the difference between extending `Thread` class and implementing `Runnable` interface?
8. What is the purpose of the `start()` method in threads?
9. What happens if we call the `run()` method directly instead of `start()`?
10. What is thread priority in Java? How is it set?
11. What is the default priority of a thread in Java?
12. How do you pause a thread in Java?
13. How do you stop a running thread in Java?
14. What is thread synchronization? Why is it needed?
15. What is the `synchronized` keyword? Provide an example.
16. What are **race conditions** in multithreading? How do you avoid them?
17. What is the difference between `wait()` and `sleep()` in Java?
18. What is inter-thread communication? How is it achieved in Java?
19. What is a **deadlock** in multithreading?
20. What is the purpose of `join()` method in Java?

2. Stream-Based I/O (`java.io`)

21. What are byte streams and character streams in Java?
22. What is the difference between `InputStream` and `Reader`?
23. What is the difference between `OutputStream` and `Writer`?
24. How do you read input from the console in Java?
25. How do you write output to the console in Java?
26. What is the `File` class in Java? What is it used for?
27. How do you create a new file in Java?
28. How do you read a file in Java?
29. How do you write to a file in Java?
30. What are the differences between `FileReader` and `BufferedReader`?
31. What is the `Console` class in Java? How is it used?
32. What is serialization in Java?
33. What is the purpose of the `Serializable` interface?
34. What is object deserialization?
35. What is the difference between transient and non-transient variables in serialization?

36. What is `ObjectOutputStream` and `ObjectInputStream`?
37. What happens if a class does not implement `Serializable` but an object of it is serialized?
38. How do you prevent a variable from being serialized?
39. What is `EOFException` in Java? When does it occur?
40. What are the advantages of using buffered streams in Java?

Long Answer Questions (LAQs)

1. Multithreading

1. What is **multithreading**? Explain its advantages and real-world applications.
2. How do you create and run threads in Java? Explain both **Thread class** and **Runnable interface** methods.
3. What is the **thread life cycle** in Java? Explain each state with a diagram.
4. Explain the **different thread priorities** in Java. How does priority affect execution?
5. What is **thread synchronization**? Explain the `synchronized` keyword with an example.
6. What is **inter-thread communication**? How do `wait()`, `notify()`, and `notifyAll()` methods work?
7. Explain **deadlocks in multithreading**. How can they be avoided?
8. What is **thread pooling**? How does Java provide thread pooling?
9. Explain **join(), sleep(), yield(), and interrupt()** methods in Java multithreading.
10. What is **race condition**? How can you prevent it using synchronization?

2. Stream-Based I/O (`java.io`)

11. Explain **byte streams and character streams** in Java. How do they differ?
12. What are **buffered streams**? Explain their advantages in Java I/O.
13. Explain how to **read and write files** in Java using `FileReader` and `FileWriter`.
14. What is the purpose of the **File class**? Explain its important methods.
15. Write a Java program to **copy the contents of one file to another** using streams.
16. Explain **serialization and deserialization** in Java with an example.
17. What are **transient variables**? How do they affect serialization?
18. What is the **difference between PrintWriter and BufferedWriter**?
19. Explain **reading input using Scanner and Console class** with examples.
20. What are **exception handling techniques in file handling**? How do we handle `FileNotFoundException` and `IOException`?

UNIT – V

Here are **Short Answer Questions (SAQs) and Long Answer Questions (LAQs)** for **UNIT-V: GUI Programming with Swing and Event Handling**.

Short Answer Questions (SAQs)

1. GUI Programming with Swing

1. What is Swing in Java? How is it different from AWT?
2. What is the **MVC architecture** in Java Swing?
3. What are Swing components? Name some commonly used Swing components.
4. What is a **JFrame**? How is it different from `Frame` in AWT?
5. What are **Swing containers**? Name the top-level containers.
6. What is the difference between **lightweight and heavyweight components** in Java?
7. What is `JPanel` in Swing?
8. How do you create a **button** in Swing? Which class is used?
9. What is the difference between `JTextField` and `JTextArea`?
10. How do you create a **label** in Swing?
11. What is a **layout manager** in Swing? Why is it used?
12. What is **FlowLayout**? How does it arrange components?
13. What is **BorderLayout**? Explain its regions.
14. What is **GridLayout**? How does it arrange components?
15. What is **CardLayout**? Where is it useful?
16. What is **GridBagLayout**? How is it different from GridLayout?
17. How can we set a layout manager for a container in Swing?
18. What is the default layout of `JFrame`?
19. How do you make a Swing application **exit on close**?
20. What is the `setVisible(true)` method in Swing?

2. Event Handling

21. What is **event handling** in Java?
22. What is the **Delegation Event Model**?
23. What are **event sources**? Give examples.
24. What are **event listeners**? How do they work?
25. What are **event classes**? Name some important event classes.
26. How do you handle **button click events** in Java Swing?
27. How do you handle **keyboard events** in Java Swing?
28. How do you handle **mouse events** in Java Swing?
29. What is an **Adapter class**? Why is it used?
30. What is an **anonymous inner class**? How is it used in event handling?
31. What is an **inner class**? How is it different from an anonymous inner class?
32. What are the advantages of using **anonymous inner classes** in event handling?
33. What is the `ActionListener` interface? How is it implemented?
34. What is the difference between `MouseListener` and `MouseMotionListener`?
35. What are `KeyListener` and `KeyAdapter`? How are they used?
36. What is the `WindowListener` interface?
37. What are `FocusListener` and `ItemListener`? How do they work?
38. What is `ActionEvent`? Which Swing components generate this event?
39. What is the difference between `setText()` and `getText()` methods in `JTextField`?
40. How do you register an event listener for a Swing component?

Long Answer Questions (LAQs)

1. GUI Programming with Swing

1. What is **Java Swing**? How is it different from AWT?
2. Explain **MVC architecture** in Java Swing with an example.
3. What are **top-level containers** in Swing? Explain `JFrame`, `JDialog`, and `JApplet`.
4. Explain **Swing components** such as `JButton`, `JLabel`, `JTextField`, `JTextArea`, `JCheckBox`, and `JRadioButton` with examples.
5. What is a **layout manager**? Explain different types of layout managers in Swing.
6. Explain **FlowLayout, BorderLayout, and GridLayout** with examples.
7. What is **CardLayout**? How is it used in GUI design?
8. Explain **GridBagLayout** and how it differs from GridLayout.
9. Write a Java program to **create a simple Swing GUI** with a button, label, and text field.
10. Explain **JTabbedPane, JScrollPane, and JSplitPane** with examples.

2. Event Handling

11. What is **event handling**? Explain the **Delegation Event Model** with an example.
12. Explain **event sources and event listeners** in Java Swing.
13. What is **ActionListener**? Explain how to handle button click events in Swing.
14. Explain **MouseListener and MouseMotionListener** with examples.
15. Explain **KeyListener and KeyAdapter** with an example.
16. Write a Java program to **handle keyboard events** in a Swing application.
17. What is an **Adapter class**? How does it simplify event handling?
18. What is an **anonymous inner class**? How is it used in event handling?
19. Write a Java program to **create a Swing application that handles mouse events**.
20. Explain the **difference between focus events, action events, and key events**.
