Introduction to Java

What is Java?

Java is a **high-level**, **object-oriented**, **class-based** programming language designed to have **minimal implementation dependencies**. Developed by **Sun Microsystems** and now owned by **Oracle Corporation**, Java enables developers to write code once and run it anywhere — thanks to its **platform independence**.

Java is widely used for **web applications**, **mobile development (Android)**, **desktop software**, **embedded systems**, and **enterprise solutions**.

Key Features of Java

- **Platform Independent**
  Java programs are compiled into bytecode, which can run on any device equipped with the Java Virtual Machine (JVM).
- **Object-Oriented**
  Everything in Java revolves around objects and classes, promoting modularity and code reuse.
- **Robust and Secure**
  Java includes strong memory management, exception handling, and runtime checks to eliminate common programming errors.
- **Simple and Familiar**
  Its syntax is clean and derived from C/C++, making it easier for developers with a C background to learn.
- **Multithreaded**
  Java supports multithreading, enabling programs to perform multiple tasks simultaneously.
- **Distributed**
  Java allows the development of applications that can run on networks and support remote communication.
- **High Performance**
  Although Java is interpreted, Just-In-Time (JIT) compilers help improve performance by converting bytecode to native machine code at runtime.
- **Dynamic and Extensible**
  Java supports dynamic loading of classes and libraries, which allows new features to be integrated without modifying existing code.

Core Components of Java

1. **Java Development Kit (JDK)**
   A full-featured software development kit including compiler, debugger, and tools needed to develop Java applications.
2. **Java Runtime Environment (JRE)**
   Contains the libraries and components to run Java programs but does not include development tools.
3. **Java Virtual Machine (JVM)**
   Executes the compiled Java bytecode and provides a platform-independent runtime environment.

Java Program Structure

Even though we are avoiding code, it's important to understand that a basic Java program includes:

- **Class Declaration**: Everything in Java is written inside a class.
- **Main Method**: The entry point of any Java application.
- **Statements and Blocks**: Java uses statements (instructions) and blocks (groups of statements) to define logic.

Java Paradigms & Principles

- **Encapsulation**: Bundling of data and methods that operate on the data within one unit.
- **Abstraction**: Hiding internal details and showing only necessary features.
- **Inheritance**: Mechanism where one class acquires properties of another.
- **Polymorphism**: Ability to perform a single action in different ways.

Real-World Applications of Java

- **Android App Development**
- **Web Applications (Servlets, JSP, Spring Boot)**
- **Enterprise Applications (Banking, CRM)**
- **Embedded Systems**
- **Scientific Applications**
- **Big Data (Hadoop)**
- **Game Development**

Java Editions

1. **Java Standard Edition (Java SE)**
   Core functionalities of Java for general-purpose use.
2. **Java Enterprise Edition (Java EE)**
   Advanced tools and APIs for developing large-scale, distributed systems.
3. **Java Micro Edition (Java ME)**
   For developing applications on embedded and mobile devices.
4. **JavaFX**
   For developing rich internet applications with a modern UI.

Conclusion

Java remains one of the **most powerful**, **reliable**, and **widely used** programming languages in the world. Its **platform independence**, **security**, and **robust ecosystem** make it a top choice for developers across industries — from startups to global enterprises.

Whether you're building mobile apps, web portals, or complex enterprise solutions, Java provides the **tools**, **frameworks**, and **community** to support development at any scale.

Setting Up Java Environment

To start developing Java applications, you need to set up the Java Development Environment properly.

Installing JDK (Java Development Kit)

The JDK provides all the tools you need to write, compile, and run Java programs.

Steps to install JDK:

1. Visit the official Oracle website or OpenJDK:
   https://www.oracle.com/java/technologies/downloads/?er=221886
2. Download the latest version of JDK suitable for your OS (Windows/macOS/Linux).
3. Run the installer and follow the instructions.
4. After installation, check if Java is installed using the terminal/command prompt:

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

Setting Environment Variables (Windows)

After installation, you must configure environment variables:

1. **Set JAVA_HOME:**
   - Go to System Properties > Environment Variables.
   - Click on **New** under System variables.
   - Name: JAVA_HOME
     Value: C:\Program Files\Java\jdk-<version>
2. **Add bin directory to Path:**
   - Edit the Path variable.
   - Add:
     C:\Program Files\Java\jdk-<version>\bin
3. Confirm setup:

```
echo %JAVA_HOME%
java -version
```

You're ready to compile and run Java programs!

Installing an IDE

An IDE (Integrated Development Environment) simplifies writing, compiling, and debugging Java code.

Popular IDEs for Java:

IDE	Features
IntelliJ IDEA	Best for professional development; great auto-completion and refactoring tools
Eclipse	Lightweight and widely used in enterprises
VS Code	Simple and extensible; requires Java extensions

Recommended for beginners: VS Code or IntelliJ IDEA Community Edition

Compilation and Execution Process

Use the terminal or command prompt:

Your file_name and the class_name should be same

1. Navigate to the folder containing your .java file:

cd C:\JavaProjects

2. Compile the Java file using javac:

   javac File_name/class_name.java

   This creates a File_name.class bytecode file.

3. Run the compiled program using java:

   java File_name

Output: Given output prints here.

1. Variables in Java

A **variable** in Java is a named memory location that holds a value. Variables store data that can be used and modified during the program's execution.

Characteristics:

- Each variable has a **type**, which determines the size and layout of the variable's memory.
- A variable must be **declared** before it is used.
- Java enforces **strong typing**, meaning you must specify the data type when declaring a variable.

Types of Variables:

- **Local Variables**: Declared inside methods or blocks and exist only within that scope.
- **Instance Variables**: Declared inside a class but outside any method. Each object of the class has its own copy.
- **Static Variables**: Declared with the static keyword and shared among all instances of a class.

2. Data Types in Java

Java is a **statically typed** language, which means every variable must be declared with a data type. Java supports two categories of data types:

Primitive Data Types

These are predefined and store simple values:

Data Type	Size	Description
byte	1 byte	Whole numbers from -128 to 127
short	2 bytes	Whole numbers from -32,768 to 32,767
int	4 bytes	Most commonly used integer type
long	8 bytes	Large whole numbers
float	4 bytes	Decimal numbers (less precision)
double	8 bytes	Decimal numbers (more precision)
char	2 bytes	A single character (Unicode)
boolean	1 bit	true or false

Non-Primitive (Reference) Data Types

These are used to store complex types or references to objects:

- **String**: Represents sequences of characters.
- **Arrays**: A collection of variables of the same type.
- **Classes**: User-defined data types that group variables and methods.
- **Interfaces and Enums**: Define behaviors and fixed sets of constants, respectively.

3. Comments in Java

Comments are non-executable text in a program that help explain code to humans. Java supports three types of comments:

Single-line Comments

- Start with //
- Used for short notes or explanations on the same line.

Multi-line Comments

- Begin with /* and end with */
- Used to describe larger blocks or disable multiple lines temporarily.

Documentation Comments

- Begin with /** and end with */
- Used to generate external documentation (like Javadoc).
- Typically used to describe classes, methods, and parameters.

4. Identifiers and Naming Rules

- An **identifier** is the name used for variables, methods, classes, etc.
- Must begin with a letter (A–Z or a–z), currency character ($), or an underscore (_).
- Cannot begin with a digit or use reserved keywords.
- Identifiers are **case-sensitive**.
- Common naming conventions:
    - **CamelCase** for variables and methods (myVariableName)
    - **PascalCase** for class names (MyClassName)

5. Literals in Java

Literals represent constant values assigned to variables.

- **Integer literals**: e.g., 10, -25
- **Floating-point literals**: e.g., 3.14, -0.001
- **Character literals**: e.g., 'a', 'Z'
- **String literals**: e.g., "Hello", "Java"
- **Boolean literals**: true, false

6. Type Casting

Type casting is converting one data type into another.

Two types:

- **Implicit Casting (Widening)**: Automatically converts smaller to larger data types (e.g., int to float).
- **Explicit Casting (Narrowing)**: Manually converts larger to smaller types (e.g., double to int).

7. Constants in Java

Constants are fixed values that do not change during program execution.

- Declared using the final keyword.
- Once assigned, their value cannot be modified.

Example conceptually:

- A constant named MAX_SPEED could be declared to store the maximum speed limit of a vehicle and will remain unchanged throughout the program.

8. Java Keywords

Java contains **reserved words** that have special meanings and cannot be used as variable names or identifiers.

Examples include:

- class, public, static, void, if, else, while, for, switch, return, new, try, catch, finally, import, package, interface, extends, implements, super, this, and many others.

These are part of Java's syntax and define the structure and behavior of code.

Writing your first java program…….

Java First Program: Hello World

Code Example

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

Line-by-Line Explanation

public class HelloWorld

- **public**: An access modifier. It means this class can be accessed from anywhere.
- **class**: A keyword used to define a class in Java.
- **HelloWorld**: The name of the class. In Java, the file name must match the public class name. So the file should be named HelloWorld.java.

This line declares a class named HelloWorld, which serves as the blueprint for the program.

{

- This opening brace marks the **start of the class body**. Everything between this { and its matching } belongs to the HelloWorld class.

public static void main(String[] args)

This line defines the **main method**. It's the **entry point** of any standalone Java application.

- **public**: The method is accessible from anywhere. Required so the JVM can call it.
- **static**: The method belongs to the class and does not require creating an object to call it.
- **void**: This method does not return any value.
- **main**: The name of the method. Java looks for this exact method signature to start the program.
- **String[] args**: A parameter that allows command-line arguments to be passed as an array of String values.

This method is what the **Java Virtual Machine (JVM)** calls to start the execution of the program.

{

- This opening brace marks the **start of the main method body**.

System.out.println("Hello, World!");

This line **prints text to the console**.

Breakdown:

- **System**: A built-in class from the java.lang package.
- **out**: A static object (instance of PrintStream) in the System class, connected to the console.
- **println()**: A method of the PrintStream class that prints the text and moves the cursor to the next line.
- **"Hello, World!"**: A string literal that gets printed to the screen.

The output of this line will be:

CopyEdit

Hello, World!

}

- This closing brace marks the **end of the main method**.

}

- This closing brace marks the **end of the class HelloWorld**.

Java Operators

Java supports several types of operators used to perform operations on variables and values. Below is a breakdown of each operator type with examples.

Category	Example Symbols
Arithmetic	+ - * / %
Relational	== != > < >= <=
Logical	`&&
Assignment	= += -= *= /= %=
Unary	+ - ++ --
Bitwise	`&
Ternary	? :

1. Arithmetic Operators

These are used for basic mathematical operations.

Operator	Meaning	Example	Result
+	Addition	10 + 5	15
-	Subtraction	10 - 5	5
*	Multiplication	10 * 5	50
/	Division	10 / 2	5
%	Modulus (Remainder)	10 % 3	1

Ex:

```
public class ArithmeticExample {
   public static void main(String[] args) {
      int a = 10, b = 3;
      System.out.println("Addition: " + (a + b));      // 13
      System.out.println("Subtraction: " + (a - b));   // 7
      System.out.println("Multiplication: " + (a * b));// 30
      System.out.println("Division: " + (a / b));      // 3
```

```
        System.out.println("Modulus: " + (a % b));      // 1
    }
}
```

2. Relational (Comparison) Operators

Used to compare two values. Returns true or false.

Operator	Meaning	Example	Result
==	Equal to	5 == 5	true
!=	Not equal to	5 != 3	true
>	Greater than	7 > 4	true
<	Less than	3 < 6	true
>=	Greater than or equal to	5 >= 5	true
<=	Less than or equal to	4 <= 6	true

Ex:

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

3. Logical Operators

Used for combining two or more boolean expressions.

Operator	Meaning	Example	Result
&&	Logical AND	true && false	false

Operator	Meaning	Example	Result
`		`	Logical OR
!	Logical NOT	!true	false

EX:

```
public class LogicalExample {
   public static void main(String[] args) {
      int age = 25;
      boolean isStudent = false;

      System.out.println(age > 18 && age < 30); // true
      System.out.println(age < 18 || isStudent); // false
      System.out.println(!isStudent); // true
   }
}
```

4. Assignment Operators

Used to assign values to variables.

Operator	Meaning	Example	Equivalent
=	Assign	a = 5	—
+=	Add and assign	a += 3	a = a + 3
-=	Subtract and assign	a -= 2	a = a - 2
*=	Multiply and assign	a *= 4	a = a * 4
/=	Divide and assign	a /= 5	a = a / 5
%=	Modulo and assign	a %=10	a = a % 10

EX:

```
public class AssignmentExample {
   public static void main(String[] args) {
      int a = 5;
      a += 3;  // a = a + 3 => 8
      a -= 2;  // a = a - 2 => 6
```

```
        a *= 2;  // a = a * 2 => 12
        a /= 4;  // a = a / 4 => 3
        a %= 2;  // a = a % 2 => 1
        System.out.println("Final value of a: " + a); // 1
    }
}
```

5. Unary Operators

Operate on a single operand.

Operator	Description
+	Unary plus
-	Unary minus
++	Increment
--	Decrement

Ex:

```
public class UnaryExample {
    public static void main(String[] args) {
        int a = 5;
        int b = +a;     // Unary plus (no change)
        int c = -a;     // Unary minus
        int d = a++;    // Post-increment: use then increase
        int e = ++a;    // Pre-increment: increase then use

        System.out.println("b: " + b); // 5
        System.out.println("c: " + c); // -5
        System.out.println("d: " + d); // 5
        System.out.println("e: " + e); // 7
    }
}
```

6. Bitwise Operators

Used to perform operations on binary representations.

EX:

```
public class BitwiseExample {
   public static void main(String[] args) {
      int a = 5;  // 0101 in binary
      int b = 3;  // 0011 in binary
      System.out.println(a & b);  // 1 (0001)
      System.out.println(a | b);  // 7 (0111)
      System.out.println(a ^ b);  // 6 (0110)
      System.out.println(~a);     // -6 (inverts all bits)
      System.out.println(a << 1); // 10 (01010)
      System.out.println(a >> 1); // 2 (0010)
      System.out.println(a >>> 1);// 2 (logical right shift)
   }
}
```

7. Ternary Operator

Used as a shorthand for if-else conditions.

Operator	**Description**
? :	Ternary (condition ? true : false)

EX:

```
public class TernaryExample {
   public static void main(String[] args) {
      int age = 18;
      String result = (age >= 18) ? "Adult" : "Minor";
      System.out.println(result); // "Adult"
   }
}
```

Conditional Statements in Java

Conditional statements allow your program to make decisions based on **conditions** (true or false).

Types of Conditional Statements:

1. if statement
2. if-else statement
3. if-else-if ladder
4. switch statement

if Statement

Used when you want to execute a block of code **only if** a condition is true.

```
int age = 20;

if (age >= 18) {
   System.out.println("You are eligible to vote.");
}
```

Output:

You are eligible to vote.

if-else Statement

Used when you want to execute **one block if true** and **another if false**.

```
int number = 5;

if (number % 2 == 0) {
   System.out.println("Even number");
} else {
   System.out.println("Odd number");
}
```

Output:

Odd number

if-else-if Ladder

Used when you have **multiple conditions** to check.

```
int marks = 75;

if (marks >= 90) {
   System.out.println("Grade: A");
```

```
} else if (marks >= 80) {
   System.out.println("Grade: B");
} else if (marks >= 70) {
   System.out.println("Grade: C");
} else {
   System.out.println("Grade: F");
}
```

Output:

Grade: C

switch Statement

Used when you have multiple possible values for a variable. It's cleaner than using many if-else-if.

```
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("Other Day");
}
```

Output:

Wednesday

Note: break is important to prevent fall-through (executing all cases after a match).

Statement	Use Case
if	Single condition check
if-else	Either this or that
if-else-if	Multiple condition checks
switch	Clean alternative for multiple exact value checks

Looping Statements in Java

Loops are used to execute a block of code repeatedly as long as a given condition is true.

Types of Loops in Java:

1. for loop
2. while loop
3. do-while loop
4. Enhanced for-each loop (for arrays/collections)

for Loop

Used when the number of iterations is known in advance.

```
// Print numbers from 1 to 5
for (int i = 1; i <= 5; i++) {
   System.out.println("Number: " + i);
}
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Structure:

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

while Loop

Used when the number of iterations is **not known** in advance. It checks the condition **before** executing the loop.

```
int i = 1;
```

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

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

do-while Loop

Similar to while, but it checks the condition **after** the loop has executed at least once.

```
int i = 1;

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

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Enhanced for-each Loop

Used for iterating through arrays or collections.

```
int[] numbers = {10, 20, 30, 40};
for (int num : numbers) {
   System.out.println("Element: " + num);
}
```

Output:

```
Element: 10
```

Element: 20
Element: 30
Element: 40

Loop Type	When to Use
for	When you know the exact number of iterations
while	When the number of iterations is unknown
do-while	When you need to execute at least once
for-each	Simplified loop for arrays/collections

Jump Statements in Java

Jump statements are used to **alter the flow of execution** based on certain conditions. Java supports three main jump statements:

break

continue

return

break Statement

The break statement is used to **exit** a loop or switch statement **prematurely**.

Example: Breaking a for loop

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) {
      break;  // Exit loop when i is 5
   }
   System.out.println("i = " + i);
}
```

Output:

i = 1
i = 2
i = 3
i = 4

Use case: Exiting early when a condition is met.

continue Statement

The continue statement **skips the current iteration** and continues with the next one.

Example: Skipping even numbers

```
for (int i = 1; i <= 5; i++) {
   if (i % 2 == 0) {
      continue;  // Skip even numbers
   }
   System.out.println("i = " + i);
}
```

Output:

```
i = 1
i = 3
i = 5
```

Use case: Skipping specific cases without exiting the loop.

return Statement

The return statement is used to **exit from a method** and optionally **return a value**.

Example 1: Return from a void method

```
public class Demo {
   public static void checkAge(int age) {
      if (age < 18) {
         System.out.println("Underage");
         return;  // Exit the method early
      }
      System.out.println("Eligible to vote");
   }

   public static void main(String[] args) {
      checkAge(16);
      checkAge(20);
   }
}
```

Output:

```
Underage
```

Eligible to vote

Example 2: Returning a value

```
public class MathOperations {
   public static int add(int a, int b) {
      return a + b;  // Return the sum
   }

   public static void main(String[] args) {
      int result = add(3, 5);
      System.out.println("Sum: " + result);
   }
}
```

Output:

Sum: 8

Use case: Exit methods and return control to the calling method (optionally with a result).

Statement	Purpose	Works Inside
break	Exit from loop/switch early	for, while, do-while, switch
continue	Skip current iteration and continue loop	for, while, do-while
return	Exit from a method (with/without value)	Any method

Access Modifiers in Java

Access modifiers in Java are used to **define the visibility/scope** of classes, methods, and variables. Java has four main access modifiers:

Modifier	Same Class	Same Package	Subclass (Diff. Package)	Other Packages
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
(default)	Yes	Yes	No	No
private	Yes	No	No	No

1. public Access Modifier

- Accessible **from anywhere**.
- Used when you want the class, method, or variable to be accessible globally.

```
// File: Main.java
public class Main {
   public int number = 100;

   public void display() {
      System.out.println("Public method called");
   }
}
// File: Another.java
public class Another {
   public static void main(String[] args) {
      Main obj = new Main();
      obj.display();          // ✅Accessible
      System.out.println(obj.number); // ✅Accessible
   }
}
```

2. private Access Modifier

- Accessible **only within the same class**.
- Not accessible from outside the class.

```
public class Main {
   private int number = 50;

   private void display() {
      System.out.println("Private method called");
   }

   public static void main(String[] args) {
      Main obj = new Main();
      obj.display();          // ✅Accessible inside same class
      System.out.println(obj.number); // ✅Accessible
   }
}
public class Another {
   public static void main(String[] args) {
      Main obj = new Main();
      // obj.display();      // ❌Not allowed
      // System.out.println(obj.number); // ❌Not allowed
   }
}
```

3. protected Access Modifier

- Accessible **within the same package** or **in subclasses (even in different packages)**.

```
// File: Animal.java
package animals;
public class Animal {
   protected String type = "Dog";

   protected void sound() {
      System.out.println("Animal makes sound");
   }
}
// File: Dog.java (in same or subclass package)
package pets;
import animals.Animal;

public class Dog extends Animal {
   public void bark() {
      System.out.println(type);   // ✅Accessible
      sound();                    // ✅Accessible
   }
}
```

4. Default Access Modifier (No Keyword)

- When no modifier is specified, it is **package-private** (i.e., accessible **only within the same package**).

```
// File: Main.java
class Main {
   int age = 20; // default

   void show() {  // default
      System.out.println("Default access method");
   }
}
// File: Test.java
class Test {
   public static void main(String[] args) {
      Main obj = new Main();
      obj.show();         // ✅Accessible (same package)
      System.out.println(obj.age); // ✅Accessible
   }
}
```

If you try to access Main from another package, it will give an error

What are Wrapper Classes?

Java is an **object-oriented language**, but its **primitive types** (like int, char, etc.) are **not objects**. To use primitives as **objects**, Java provides **Wrapper Classes** in java.lang package.

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

Autoboxing

Autoboxing is the automatic conversion of a **primitive** to its **corresponding wrapper object**.

Example:

```
public class AutoBoxingDemo {
   public static void main(String[] args) {
      int a = 50;

      Integer wrapper = a;  // Autoboxing: int to Integer
      System.out.println("Wrapper object: " + wrapper);
   }
}
```

Useful when working with **collections**, which require objects.

```
import java.util.ArrayList;

public class Example {
   public static void main(String[] args) {
      ArrayList<Integer> list = new ArrayList<>();
      list.add(5);  // int is autoboxed to Integer
      list.add(10);

      System.out.println(list);  // Output: [5, 10]
   }
}
```

Unboxing

Unboxing is the automatic conversion of a **wrapper object** back to its **primitive** type.

Example:

```
public class UnboxingDemo {
   public static void main(String[] args) {
      Integer obj = new Integer(100);

      int value = obj;  // Unboxing: Integer to int
      System.out.println("Unboxed value: " + value);
   }
}
```

Concept	Description	Example
Wrapper	Object version of a primitive	Integer i = new Integer(5);
Autoboxing	Primitive → Object automatically	Integer i = 5;
Unboxing	Object → Primitive automatically	int x = i;

What is an Array?

An **array** in Java is a **collection of similar data types stored in a contiguous memory location**. Arrays are used to store **multiple values** in a single variable, instead of declaring separate variables for each value.

One-Dimensional Array

Declaration and Initialization:

```
int[] numbers = new int[5]; // Declaration of an array with 5 elements
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

Example:

```
public class OneDArrayExample {
   public static void main(String[] args) {
      // Declare and initialize array
      int[] numbers = {10, 20, 30, 40, 50};
```

```
        // Access and print elements using traditional for loop
        System.out.println("Using normal for loop:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }
    }
}
```

Multi-Dimensional Array

A **multi-dimensional array** is essentially an array of arrays.

Example: 2D Array

```
public class TwoDArrayExample {
    public static void main(String[] args) {
        // Declare and initialize 2D array (matrix)
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6}
        };

        // Print 2D array
        System.out.println("2D Array Output:");
        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(); // new line after each row
        }
    }
}
```

Array Operations

Common operations:

1. **Find length of array** – arr.length
2. **Access element** – arr[index]
3. **Update value** – arr[index] = newValue
4. **Loop through array** – for, while, or for-each

Enhanced for-loop (for-each)

Used to **simplify looping** through arrays or collections.

Example:

```
public class ForEachExample {
   public static void main(String[] args) {
      String[] fruits = {"Apple", "Banana", "Cherry"};

      // Using for-each loop
      System.out.println("Fruits using for-each:");
      for (String fruit : fruits) {
         System.out.println(fruit);
      }
   }
}
```

Note: You can't get the index directly in a for-each loop.

Concept	Syntax Example	Use Case
Declare 1D Array	int[] arr = new int[5];	Store list of integers
Declare 2D Array	int[][] matrix = new int[2][3];	Matrix or table-like data
Access element	arr[0], matrix[1][2]	Get element at position
Modify element	arr[2] = 99;	Change array data
Loop (for loop)	for (int i = 0; i < arr.length; i++)	Index-based processing
Loop (for-each)	for (int x : arr)	Simpler, no index needed

String in Java

- A **String** is an **immutable** (cannot be changed) sequence of characters.
- Defined using **double quotes** (" ").
- It's a class in java.lang package.

Declaration:

```
String name = "Mazid";
```

Common String Methods with Examples

Method	Description	Example Output
length()	Returns the number of characters	5
charAt(index)	Returns character at specific index	'a'
substring(start)	Returns substring from start index	"zid"
substring(a, b)	Returns substring from a to b-1	"azi"
equals()	Compares strings (case-sensitive)	true / false
equalsIgnoreCase()	Compares strings ignoring case	true / false
toLowerCase()	Converts all chars to lowercase	"mazid"
toUpperCase()	Converts all chars to uppercase	"MAZID"
trim()	Removes leading/trailing spaces	"Mazid"
contains()	Checks if substring exists	true / false
replace(a, b)	Replaces character a with b	"M@zid"
split(" ")	Splits string by a delimiter (space, comma)	Array of substrings

Example with All Methods:

```
public class StringExample {
   public static void main(String[] args) {
      String name = "  Mazid Mohammad  ";

      System.out.println("Original: '" + name + "'");
      System.out.println("Trimmed: '" + name.trim() + "'");
      System.out.println("Length: " + name.length());
      System.out.println("Char at index 2: " + name.charAt(2));
      System.out.println("Substring(3): " + name.substring(3));
      System.out.println("Substring(2, 6): " + name.substring(2, 6));
      System.out.println("Equals: " + name.trim().equals("Mazid Mohammad"));
      System.out.println("Ignore Case Equals: " + name.trim().equalsIgnoreCase("mazid mohammad"));
      System.out.println("Lowercase: " + name.toLowerCase());
      System.out.println("Uppercase: " + name.toUpperCase());
      System.out.println("Contains 'Moh': " + name.contains("Moh"));
      System.out.println("Replace 'a' with '@': " + name.replace('a', '@'));

      String[] parts = name.trim().split(" ");
      System.out.println("Split result:");
      for (String part : parts) {
         System.out.println(part);
      }
   }
}
```

StringBuilder

- **Mutable** string (can change content).
- **Faster** than String in many operations (especially loops).

- **Not thread-safe.**

Example:

```
public class StringBuilderExample {
   public static void main(String[] args) {
      StringBuilder sb = new StringBuilder("Hello");

      sb.append(" World");
      sb.insert(0, "Say ");
      sb.replace(4, 7, "Hello");
      sb.delete(0, 4);
      sb.reverse();

      System.out.println("Result: " + sb); // Output: "dlroW olleH"
   }
}
```

StringBuffer

- Same as StringBuilder but **thread-safe** (synchronized).
- Slightly **slower** due to synchronization.

Example:

```
public class StringBufferExample {
   public static void main(String[] args) {
      StringBuffer sb = new StringBuffer("Java");

      sb.append(" is awesome!");
      sb.insert(0, "Learning ");
      sb.replace(9, 13, "Java");
      sb.delete(0, 9);

      System.out.println("Final: " + sb); // Output: "Java is awesome!"
   }
}
```

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread-safe	No	No	Yes
Performance	Slower	Fast	Slower than Builder
Use Case	Simple, read-only strings	Efficient manipulation in single thread	Multi-threaded environments

Object-Oriented Programming (OOP) in Java

Java is a **pure object-oriented programming language** (except for primitive types). OOP allows us to structure programs in a more modular, reusable, and logical way using **objects** and **classes**.

1. Class and Object

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

Example:

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

   void drive() {
      System.out.println("The car is driving at " + speed + " km/h.");
   }
}

public class Main {
   public static void main(String[] args) {
      Car myCar = new Car();     // Creating an object
      myCar.color = "Red";
      myCar.speed = 120;
      myCar.drive();
   }
}
```

2. Encapsulation

- Encapsulation is the **bundling of data and methods** into a single unit (class).
- It hides the internal details and only exposes necessary parts using **getters and setters**.

Example:

```
class Person {
   private String name;  // Private = hidden from outside

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

   public String getName() {
      return name;      // Getter
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");
        System.out.println("Name: " + p.getName());
    }
}
```

3. Inheritance

Inheritance in Java is a mechanism where one class (**child class**) acquires the properties (fields) and behaviors (methods) of another class (**parent class**).
This promotes **code reuse** and **method overriding**.

Syntax:

```
class Parent {
    // fields and methods
}

class Child extends Parent {
    // additional fields and methods
}
```

Types of Inheritance in Java

Java supports the following inheritance types:

Single Inheritance

One class inherits from one superclass.

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

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

```
public class Main {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.sound();  // Inherited from Animal
      d.bark();   // Defined in Dog
   }
}
```

Multilevel Inheritance

A class is derived from a class that is also derived from another class.

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

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

class Puppy extends Dog {
   void weep() {
      System.out.println("Puppy weeps");
   }
}

public class Main {
   public static void main(String[] args) {
      Puppy p = new Puppy();
      p.eat();   // from Animal
      p.bark();  // from Dog
      p.weep();  // from Puppy
   }
}
```

Hierarchical Inheritance

Multiple subclasses inherit from the same superclass.

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

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

class Cat extends Animal {
   void meow() {
      System.out.println("Cat meows");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.sound();
      d.bark();

      Cat c = new Cat();
      c.sound();
      c.meow();
   }
}
```

Multiple Inheritance (via Interfaces)

Java **does not support** multiple inheritance using **classes** to avoid ambiguity.
But it **does support** multiple inheritance using **interfaces**.

```
interface Flyable {
   void fly();
}

interface Swimmable {
   void swim();
}

class Duck implements Flyable, Swimmable {
   public void fly() {
      System.out.println("Duck flies");
   }

   public void swim() {
      System.out.println("Duck swims");
   }
}

public class Main {
   public static void main(String[] args) {
      Duck d = new Duck();
      d.fly();
```

```
        d.swim();
    }
}
```

Hybrid Inheritance

Hybrid = Combination of multiple inheritance types (e.g., multiple + multilevel).
Can be **simulated** using **interfaces**.

Example:

4. Polymorphism

- **Polymorphism** means "many forms".
- It allows a method to behave differently based on the object.
- Two types: **Compile-time (Method Overloading)** and **Runtime (Method Overriding)**.

Method Overloading (Compile-time):

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

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

public class Main {
    public static void main(String[] args) {
        MathOperation m = new MathOperation();
        System.out.println(m.add(5, 3));       // int
        System.out.println(m.add(5.5, 3.2));    // double
    }
}
```

Method Overriding (Runtime):

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

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

public class Main {
    public static void main(String[] args) {
        Animal a = new Cat();  // Upcasting
        a.sound();  // Calls Cat's version
    }
}
```

5. Abstraction

- **Abstraction** hides complex details and shows only the essential features.
- Achieved using **abstract classes** and **interfaces**.

Abstract Class Example:

```
abstract class Shape {
    abstract void draw();  // Abstract method
}

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

public class Main {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.draw();
    }
}
```

What is an Interface in Java?

An **interface** in Java is like a **contract**. It is a completely abstract class that contains:

- Only **abstract methods** (until Java 7).
- From Java 8 onwards, it can also have **default** and **static** methods.
- From Java 9 onwards, **private** methods are allowed in interfaces.

Key Points:

- An interface **cannot have instance variables** (only constants).
- All methods are **public and abstract** by default (unless marked default/static).
- A class implements an interface using the implements keyword.
- A class can **implement multiple interfaces** (solving multiple inheritance issues).

Syntax:

```
interface InterfaceName {
   void method1();
   void method2();
}
java
CopyEdit
class ClassName implements InterfaceName {
   public void method1() {
      // Implementation
   }

   public void method2() {
      // Implementation
   }
}
```

Basic Example

```
interface Animal {
   void eat();
   void sleep();
}

class Dog implements Animal {
   public void eat() {
      System.out.println("Dog eats meat");
   }

   public void sleep() {
      System.out.println("Dog sleeps at night");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.eat();
      d.sleep();
   }
}
```

Output:

Dog eats meat
Dog sleeps at night

Multiple Interface Implementation

```
interface Printable {
   void print();
}

interface Showable {
   void show();
}

class Document implements Printable, Showable {
   public void print() {
      System.out.println("Printing document");
   }

   public void show() {
      System.out.println("Showing document");
   }
}

public class Main {
   public static void main(String[] args) {
      Document doc = new Document();
      doc.print();
      doc.show();
   }
}
```

Interface with Default Method (Java 8+)

```
interface Vehicle {
   void start();

   default void fuelType() {
      System.out.println("Petrol");
   }
}

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

public class Main {
```

```
    public static void main(String[] args) {
        Car c = new Car();
        c.start();
        c.fuelType();  // Default method
    }
}
```

Abstract Class

An **abstract class** in Java:

- Is a **class that cannot be instantiated** (you cannot create objects of it directly).
- Can have **abstract methods** (methods without a body).
- Can also have **concrete methods** (normal methods with implementation).
- Is used as a **base class** for other classes to extend and implement the abstract methods.

Why Use Abstract Classes?

- To provide a **common base** with shared behavior and **force certain methods** to be implemented in child classes.
- Supports **partial abstraction** (unlike interface which supports full abstraction).

Syntax:

```
abstract class Animal {
    abstract void sound();  // Abstract method

    void sleep() {        // Concrete method
        System.out.println("Animal is sleeping");
    }
}
```

Example:

```
abstract class Animal {
    abstract void sound();  // Abstract method

    void sleep() {
        System.out.println("Animal sleeps...");
    }
}

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

```
}

class Cat extends Animal {
   void sound() {
      System.out.println("Cat meows");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal a1 = new Dog();
      a1.sound(); // Dog barks
      a1.sleep(); // Animal sleeps...

      Animal a2 = new Cat();
      a2.sound(); // Cat meows
      a2.sleep(); // Animal sleeps...
   }
}
```

1. Constructors in Java

What is a Constructor?

- A **constructor** is a special method that is called **when an object is created**.
- It is used to **initialize objects**.
- It has **no return type**, **not even void**.
- It must have the **same name** as the class.

Types of Constructors:

1. **Default Constructor** – No parameters.
2. **Parameterized Constructor** – Accepts parameters to initialize fields.
3. **Copy Constructor** (manually written) – Initializes object using another object (not built-in like C++).

Example:

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

   // Default Constructor
   Student() {
      name = "Unknown";
      age = 0;
   }
```

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

    void display() {
        System.out.println(name + " is " + age + " years old.");
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();          // Default constructor
        Student s2 = new Student("Mazid", 21); // Parameterized constructor

        s1.display(); // Unknown is 0 years old.
        s2.display(); // Mazid is 21 years old.
    }
}
```

2. this Keyword in Java

What is this?

- Refers to the **current object**.
- Used to:
    - Access current class fields/methods.
    - **Differentiate** between instance variables and parameters.
    - Call another constructor inside the same class.

Example:

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

    Car(String model, int year) {
        this.model = model; // refers to instance variable
        this.year = year;
    }

    void display() {
        System.out.println("Model: " + this.model + ", Year: " + this.year);
    }
}
```

3. super Keyword in Java

What is super?

- Refers to the **immediate parent class**.
- Used to:
  - Call the **parent class constructor**.
  - Access **parent class methods or variables** if they are overridden.

Example:

```
class Animal {
   String type = "Animal";

   Animal() {
      System.out.println("Animal constructor called");
   }

   void eat() {
      System.out.println("Animal eats food");
   }
}

class Dog extends Animal {
   String type = "Dog";

   Dog() {
      super(); // calls parent constructor
      System.out.println("Dog constructor called");
   }

   void showType() {
      System.out.println("Type: " + super.type); // Animal
      System.out.println("Type: " + this.type);  // Dog
   }

   void eat() {
      super.eat(); // calls Animal's eat method
      System.out.println("Dog eats bones");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.showType();
      d.eat();
   }
}
```

4. final Keyword in Java

Uses of final:

Used with	Meaning
Variable	Cannot change the value (constant)
Method	Cannot be overridden in a subclass
Class	Cannot be inherited (no subclass can extend it)

Examples:

Final Variable:

```
class Circle {
   final double PI = 3.14159;

   double area(double radius) {
      return PI * radius * radius;
   }
}
```

Final Method:

```
class A {
   final void show() {
      System.out.println("This cannot be overridden");
   }
}

class B extends A {
   // void show() {} ✕Compilation error
}
```

Final Class:

```
final class ConstantValues {
   // cannot be extended
}

// class MyClass extends ConstantValues {} ✕Error
```

Concept	Use	Example
Constructor	Initializes object fields	Student(String name)
this	Refers to current class/object	this.name = name;
super	Refers to parent class or constructor	super();, super.method();
final	Prevents modification (variable, method, class)	final double PI = 3.14;

What is a Package in Java?

A **package** is a namespace that organizes a set of related classes and interfaces.

Why use packages?

- To **avoid class name conflicts**
- To **organize classes** logically (like folders)
- To control **access protection**
- To make **reusability and maintainability** easier

Types of Packages

1. **Built-in packages** (like java.util, java.io, etc.)
2. **User-defined packages** (packages you create)

Creating a Package in Java

Suppose you want to create a package named mypackage.

Step 1: Create a Class inside a Package

File: MyClass.java

```
package mypackage;  // Define the package

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

Save this file inside a folder named mypackage.

Folder structure:

```
project_folder/
│
└──mypackage/
   │  MyClass.java
```

Step 2: Compile the Package Class

Open terminal/command prompt and navigate to the project_folder:

javac mypackage/MyClass.java

Importing and Using a Package

Now, create another class outside the package to use MyClass.

File: Main.java

```
import mypackage.MyClass;  // Importing the class from package

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

Step 3: Compile and Run

Compile:

javac Main.java

Run:

java Main

Output:

Hello from MyClass in mypackage!

Using import Variants

Syntax	Meaning
import packagename.ClassName;	Import a specific class
import packagename.*;	Import all classes from a package
import static ...	Used to import static members (Java 5+)

What is Exception Handling in Java?

Exception Handling is a mechanism to handle runtime errors so the normal flow of the program can be maintained.

Java provides five keywords for exception handling:

- try
- catch
- finally
- throw
- throws

1. try-catch Block

Syntax:

```
try {
  // code that may throw an exception
} catch(ExceptionType e) {
  // code to handle the exception
}
```

Example:

```
public class TryCatchExample {
   public static void main(String[] args) {
      try {
         int result = 10 / 0;  // This will throw ArithmeticException
      } catch (ArithmeticException e) {
         System.out.println("Cannot divide by zero!");
      }
   }
}
```

Output:

```
Cannot divide by zero!
```

2. finally Block

- **Always executes** whether or not an exception occurs.
- Used for cleanup like closing files, releasing resources, etc.

Example:

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

```
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]);  // ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index error!");
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

Output:

```
Array index error!
This will always execute.
```

3. throw Keyword

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

Example:

```
public class ThrowExample {
    public static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Age must be 18 or older!");
        } else {
            System.out.println("You are eligible.");
        }
    }

    public static void main(String[] args) {
        checkAge(15);
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Age must be 18 or older!
```

4. throws Keyword

- Used in method declaration to indicate that the method **might throw** an exception.

Example:

```
import java.io.*;

public class ThrowsExample {
    public static void readFile() throws IOException {
```

```
        FileReader fr = new FileReader("somefile.txt");
        fr.close();
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("File not found or error while reading file.");
        }
    }
}
```

Keyword	Description	Use Case	Can be Overridden?
final	A **modifier** used with variables, methods, or classes.	Prevent value changes or inheritance.	✗
finally	A **block** used with try-catch that **always executes**.	Clean-up code (e.g., closing files).	✗
finalize	A **method** from Object class, called by GC before object is destroyed.	To clean up before GC destroys an object.	✓(but deprecated)

Collection Framework in Java:

1. **ArrayList**
2. **LinkedList**
3. **HashSet**
4. **HashMap**

Each collection has its own **use case**, **structure**, and **performance characteristics**. Let's go one by one with simple examples and explanations. □

1. ArrayList

- **Resizable array** (like Python list).
- Maintains **insertion order**.
- Allows **duplicate elements**.

Example:

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();

        fruits.add("Apple");
```

```
        fruits.add("Banana");
        fruits.add("Apple");  // duplicate allowed

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

Output:

```
Apple
Banana
Apple
```

2. LinkedList

- Doubly-linked list implementation.
- Good for **frequent insertions/deletions**.
- Maintains **insertion order**.
- Slower than ArrayList in **random access**.

Example:

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> animals = new LinkedList<>();

        animals.add("Dog");
        animals.add("Cat");
        animals.addFirst("Horse");  // adds to the beginning

        for (String animal : animals) {
            System.out.println(animal);
        }
    }
}
```

Output:

```
Horse
Dog
Cat
```

3. HashSet

- Implements **Set** interface.

- **No duplicates allowed**.
- No guaranteed order of elements.

Example:

```
import java.util.HashSet;

public class HashSetExample {
   public static void main(String[] args) {
      HashSet<String> colors = new HashSet<>();

      colors.add("Red");
      colors.add("Blue");
      colors.add("Red");  // duplicate - ignored

      for (String color : colors) {
         System.out.println(color);
      }
   }
}
```

Output (Order is NOT guaranteed):

```
Red
Blue
```

4. HashMap

- Key-value pair collection.
- Keys are **unique**, values can be **duplicate**.
- No order guaranteed.

Example:

```
import java.util.HashMap;

public class HashMapExample {
   public static void main(String[] args) {
      HashMap<String, Integer> marks = new HashMap<>();

      marks.put("John", 90);
      marks.put("Emma", 85);
      marks.put("John", 95);  // overrides existing value

      for (String key : marks.keySet()) {
         System.out.println(key + ": " + marks.get(key));
      }
   }
}
```

Output:

```
Emma: 85
John: 95
```

What is a Thread?

A **thread** is a lightweight, independent path of execution within a program.
In Java, **multithreading** allows multiple threads to run concurrently — either doing independent tasks or sharing work.

Think of threads as workers in a team. Each can do a part of the task at the same time.

Why Use Threads?

- To perform **multiple tasks at once** (e.g., downloading files while browsing).
- To make applications **faster and more responsive**.
- To **maximize CPU usage** by running tasks in parallel.

Ways to Create Threads in Java:

1. By Extending the Thread Class

2. By Implementing the Runnable Interface

Example 1: Extending Thread Class

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Running thread: " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Create a thread
        t1.start();                   // Start the thread

        for (int i = 1; i <= 3; i++) {
            System.out.println("Main thread: " + i);
        }
    }
}
```

Output (Sample):

```
Main thread: 1
Running thread: 1
Main thread: 2
Running thread: 2
Main thread: 3
Running thread: 3
```

Note: Output may vary due to **thread scheduling**.

Example 2: Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Runnable thread: " + i);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        MyRunnable obj = new MyRunnable();
        Thread t1 = new Thread(obj); // Pass Runnable to Thread
        t1.start();              // Start the thread

        for (int i = 1; i <= 3; i++) {
            System.out.println("Main thread: " + i);
        }
    }
}
```

Thread Lifecycle:

1. **New** – Thread is created.
2. **Runnable** – Thread is ready to run.
3. **Running** – Thread is executing.
4. **Waiting/Sleeping** – Thread is temporarily inactive.
5. **Dead** – Thread has finished execution.

Common Thread Methods:

Method	Description
start()	Starts the thread
run()	Contains code to run in the thread
sleep(ms)	Pauses thread for given milliseconds
join()	Waits for a thread to finish
isAlive()	Checks if the thread is still running

What is Multithreading?

Multithreading is a feature in Java that allows **concurrent execution of two or more threads**. Each thread is a lightweight sub-process, and they share the same memory space.

It is used to:

- Perform multiple tasks simultaneously.
- Improve performance of applications.
- Efficiently utilize CPU resources.

How to Create Threads in Java?

There are **two ways** to create a thread in Java:

1. **Extending the `Thread` class**
2. **Implementing the `Runnable` interface**

1. Using `Thread` Class

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread running: " + i);
        }
    }
}

public class MainThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();  // Creating thread
        t1.start();                    // Starting thread

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

2. Using `Runnable` Interface

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Runnable thread: " + i);
        }
    }
}
```

```
public class RunnableExample {
   public static void main(String[] args) {
      MyRunnable obj = new MyRunnable();
      Thread t1 = new Thread(obj);  // Passing Runnable to Thread
      t1.start();                   // Start the thread

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

Output Example (interleaved execution):

```
Main thread: 1
Runnable thread: 1
Main thread: 2
Runnable thread: 2
Runnable thread: 3
Main thread: 3
Runnable thread: 4
Main thread: 4
Runnable thread: 5
Main thread: 5
```

The output can vary due to **thread scheduling** by the OS — this is the nature of concurrency.

Key Methods of `Thread` Class:

Method	Description
start()	Starts a new thread
run()	Contains the code to be executed
sleep(ms)	Puts thread to sleep
join()	Waits for a thread to die
isAlive()	Checks if thread is still running
setPriority()	Sets thread priority (1 to 10)

Benefits of Multithreading:

- Efficient CPU utilization
- Saves time (parallel execution)
- Useful for real-time systems (games, servers, etc.)

Java File I/O Basics

Java provides several classes in the java.io and java.util packages to perform input and output operations on files.

1. File Class

Used to represent a file or directory path (not to read/write data itself).

Example: Check if file exists

```
import java.io.File;

public class FileExample {
   public static void main(String[] args) {
      File file = new File("example.txt");

      if (file.exists()) {
         System.out.println("File exists");
      } else {
         System.out.println("File does not exist");
      }
   }
}
```

2. Reading a File Using Scanner

Example:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFileScanner {
   public static void main(String[] args) throws FileNotFoundException {
      File file = new File("input.txt");
      Scanner sc = new Scanner(file);

      while (sc.hasNextLine()) {
         String line = sc.nextLine();
         System.out.println(line);
      }

      sc.close();
   }
}
```

3. Reading a File Using BufferedReader

Example:

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

public class ReadFileBuffered {
   public static void main(String[] args) throws IOException {
      BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
      String line;

      while ((line = reader.readLine()) != null) {
         System.out.println(line);
      }

      reader.close();
   }
}
```

4. Writing to a File Using FileWriter

Example:

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteFile {
   public static void main(String[] args) throws IOException {
      FileWriter writer = new FileWriter("output.txt");
      writer.write("Hello, this is a sample file.\n");
      writer.write("Java makes file writing easy!");

      writer.close();
      System.out.println("File written successfully.");
   }
}
```

5. Writing to a File Using BufferedWriter

Example:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileBuffered {
   public static void main(String[] args) throws IOException {
```

```
        BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));

        writer.write("BufferedWriter example.");
        writer.newLine();  // Inserts a new line
        writer.write("Second line using BufferedWriter.");

        writer.close();
        System.out.println("Buffered writing done!");
    }
}
```

Comparison of Reader Classes

Class	Use Case	Pros
Scanner	Reading line-by-line or word-by-word	Easy parsing
BufferedReader	Fast, efficient reading	Good for large files
FileWriter	Simple file writing	Lightweight
BufferedWriter	Writing efficiently	Better for large outputs

Tip:

Always close file resources after use using close() or better use **try-with-resources** to handle exceptions and auto-close.

```
try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Thank You

And I'll keep on updating this file as per the requirements of the organizations