Topic 1: Introduction to Java

Overview:
Java is a high-level, object-oriented programming language developed by Sun Microsystems in 1995. It was designed to be **platform-independent**, enabling applications to run on any operating system that has a Java Virtual Machine (JVM). Java's simplicity, reliability, and wide library support have made it one of the most popular programming languages worldwide.

History of Java:

- Java was initially called **Oak**, created by James Gosling in the early 1990s.
- Oak was intended for embedded devices but later evolved into a general-purpose programming language.
- Renamed **Java** in 1995, inspired by Java coffee.
- Java 1.0 introduced platform independence and applets.
- Later versions enhanced GUI (Swing), multithreading, networking, and enterprise applications (Java EE/Jakarta EE).

Features of Java:

1. **Platform Independence:** Compile once, run anywhere using JVM.
2. **Object-Oriented:** Supports encapsulation, inheritance, polymorphism, abstraction.
3. **Robustness:** Strong memory management, exception handling, type checking.
4. **Security:** Prevents direct memory access; secure for networked applications.
5. **Multithreading:** Enables concurrent execution for performance.
6. **High Performance:** JIT compiler improves bytecode execution.
7. **Distributed Computing:** Networking APIs for RMI, sockets, web services.
8. **Rich API:** Libraries for GUI, networking, database, XML, etc.

JVM, JRE, and JDK:

- **JVM (Java Virtual Machine):** Executes bytecode; enables platform independence.
- **JRE (Java Runtime Environment):** JVM + core libraries; required to run Java programs.
- **JDK (Java Development Kit):** JRE + development tools (compiler, debugger, doc generator).

Java Program Structure:

- Every Java program contains **packages, classes, methods, variables, statements**.
- **Entry Point:** main() method.
- Java is **case-sensitive**; Main and main are different.
- Example of a basic program:

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

Setting up Java Environment:

1. Install **JDK** (Oracle or OpenJDK).
2. Set environment variables:
    - JAVA_HOME → path to JDK
    - PATH → include bin folder
3. Choose IDE: Eclipse, IntelliJ IDEA, NetBeans

Example Explanation:

- Save file as HelloWorld.java
- Compile: javac HelloWorld.java
- Run: java HelloWorld
- Output: Hello, World!

Why Java is Popular:

- Platform-independent execution
- Large community support
- Suitable for web, mobile, and enterprise apps
- Memory management handled automatically
- Beginner-friendly syntax

Topic 2: Data Types, Variables, and Operators

Overview

Java supports **primitive and non-primitive data types**. Variables store data values and must be declared with a type before use. Operators perform operations on variables and values, including arithmetic, relational, logical, assignment, unary, and bitwise operations.

---

1. Data Types in Java

Primitive Data Types:

1. int – stores integers (e.g., 10, -5)
2. float – stores decimal numbers (e.g., 3.14f)
3. double – stores larger decimal numbers (e.g., 3.141592)
4. char – stores single characters (e.g., 'A')
5. boolean – stores true or false
6. byte – stores small integers (-128 to 127)
7. short – stores small integers (-32,768 to 32,767)
8. long – stores large integers (e.g., 100000L)

Non-Primitive Data Types:

- Strings, Arrays, Classes, Interfaces

Example:

int age = 25;

float price = 19.99f;

char grade = 'A';

boolean isJavaFun = true;

---

2. Variables

- Variables are **containers for data values**.
- Must be declared with a **type**.
- Can be initialized at declaration or later.
- **Naming rules:**
    - Must begin with a letter, _ or $

- Cannot be a keyword
- Case-sensitive

Example:

```
int x; // declaration

x = 10; // initialization

int y = 20; // declaration + initialization

System.out.println(x + y); // Output: 30
```

---

3. Operators in Java

Arithmetic Operators: +, -, *, /, %
Example:

```
int a = 10, b = 3;

System.out.println(a + b); // 13

System.out.println(a % b); // 1
```

Relational Operators: >, <, >=, <=, ==, !=
Example:

```
int a = 10, b = 5;

System.out.println(a > b); // true

System.out.println(a == b); // false
```

Logical Operators: &&, ||, !
Example:

```
boolean x = true, y = false;

System.out.println(x && y); // false

System.out.println(!x); // false
```

Assignment Operators: =, +=, -=, *=, /=
Example:

```
int a = 5;

a += 3; // a = a + 3 => 8
```

Unary Operators: ++, --, +, -
Example:

```
int a = 5;

a++; // a = 6
```

Bitwise Operators: &, |, ^, ~, <<, >>
Example:

```
int a = 5; // 0101

int b = 3; // 0011

System.out.println(a & b); // 1 (0101 & 0011)
```

---

4. Type Conversion and Type Casting

- **Implicit Conversion (Widening):** Automatic conversion from smaller to larger type
- int a = 10;
- double b = a; // int -> double
- **Explicit Conversion (Narrowing):** Manual conversion using casting
- double a = 9.78;
- int b = (int)a; // double -> int
- System.out.println(b); // 9

---

5. Example: Calculator Program

```
import java.util.Scanner;


public class Calculator {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter first number: ");
      int a = sc.nextInt();
      System.out.print("Enter second number: ");
      int b = sc.nextInt();
```

```
        System.out.println("Sum: " + (a + b));
        System.out.println("Difference: " + (a - b));
        System.out.println("Product: " + (a * b));
        System.out.println("Quotient: " + (a / b));
        System.out.println("Remainder: " + (a % b));
    }
}
```

Output Example:

Enter first number: 10

Enter second number: 3

Sum: 13

Difference: 7

Product: 30

Quotient: 3

Remainder: 1

Topic 3: Control Flow Statements

Overview

Control flow statements in Java determine the **order in which statements are executed** in a program. They allow **decision-making** and **looping**, which is essential for dynamic and flexible programs. Java provides **conditional statements**, **loops**, and **jump statements** to control the flow.

---

1. If-Else Statements

- The **if** statement evaluates a boolean condition. If the condition is true, the code block executes.
- The **else** statement executes when the if condition is false.
- **Syntax:**

```
if (condition) {
```

```
    // statements
} else {
    // statements
}
```

Example: Check Even or Odd

```
import java.util.Scanner;


public class EvenOdd {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = sc.nextInt();
        if (num % 2 == 0) {
            System.out.println(num + " is even.");
        } else {
            System.out.println(num + " is odd.");
        }
    }
}
```

Output Example:

```
Enter a number: 7
7 is odd.
```

---

2. Nested If-Else

- You can have **if-else statements inside another if-else** for complex conditions.
- **Example:** Find the largest of three numbers

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

```
public class LargestNumber {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter three numbers: ");
        int a = sc.nextInt();
        int b = sc.nextInt();
        int c = sc.nextInt();

        if (a >= b && a >= c) {
            System.out.println(a + " is the largest.");
        } else if (b >= a && b >= c) {
            System.out.println(b + " is the largest.");
        } else {
            System.out.println(c + " is the largest.");
        }
    }
}
```

Output Example:

Enter three numbers: 10 20 15

20 is the largest.

---

3. Switch Case Statements

- The **switch statement** is used to select one of many code blocks.
- Useful when there are **multiple conditions based on a single variable**.
- **Syntax:**

```
switch(expression) {
    case value1:
        // statements
```

```
        break;
    case value2:
        // statements
        break;
    default:
        // statements
}
```

Example: Simple Menu

```
import java.util.Scanner;

public class Menu {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("1. Add\n2. Subtract\n3. Multiply\n4. Divide");
        System.out.print("Enter your choice: ");
        int choice = sc.nextInt();

        switch (choice) {
            case 1: System.out.println("Addition selected"); break;
            case 2: System.out.println("Subtraction selected"); break;
            case 3: System.out.println("Multiplication selected"); break;
            case 4: System.out.println("Division selected"); break;
            default: System.out.println("Invalid choice");
        }
    }
}
```

Output Example:

```
Enter your choice: 3
```

Multiplication selected

---

4. Loops

Loops execute a block of code **repeatedly** as long as a condition is true. Java provides **for, while, and do-while loops**.

4.1 For Loop

- Repeats a block a **specific number of times**.

```
for(initialization; condition; increment/decrement) {
   // statements
}
```

Example: Print 1 to 5

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

4.2 While Loop

- Repeats **while a condition is true**.

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

4.3 Do-While Loop

- Executes **at least once**, then checks the condition.

```
int i = 1;
do {
   System.out.println(i);
```

```
    i++;
} while(i <= 5);
```

---

5. Break and Continue

- **Break:** Exits the loop immediately.
- **Continue:** Skips the current iteration and continues with the next iteration.

Example: Skip 3 and stop at 5

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

Output:

1

2

4

---

6. Fibonacci Series Example

Print first 10 Fibonacci numbers

```
int n1 = 0, n2 = 1, n3, count = 10;
System.out.print(n1 + " " + n2 + " ");
for(int i = 2; i < count; i++) {
    n3 = n1 + n2;
    System.out.print(n3 + " ");
    n1 = n2;
    n2 = n3;
}
```

Output:

0 1 1 2 3 5 8 13 21 34

Topic 4: Classes and Objects

Overview

Java is an **object-oriented programming language (OOP)**. Everything in Java is treated as an **object** or a **class**. A **class** is a blueprint for creating objects, and an **object** is an instance of a class. Classes contain **attributes (fields)** and **methods (functions)** that define object behavior.

---

1. Defining a Class and Object

- **Class:** A template that defines **properties** (variables) and **behavior** (methods).
- **Object:** A real instance created from a class using the **new** keyword.

Syntax:

```
class ClassName {
   // variables (attributes)
   // methods (functions)
}
```

Example: Student Class

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

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

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

```
        Student s1 = new Student();
        s1.name = "Alice";
        s1.age = 20;
        s1.display();
    }
}
```

Output:

Name: Alice

Age: 20

---

2. Constructors

- A **constructor** initializes objects. It has the **same name as the class** and no return type.
- **Default Constructor:** Automatically called if no constructor is defined.
- **Parameterized Constructor:** Initializes object with provided values.

Example: Parameterized Constructor

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

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

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

```
}

public class TestStudent {
    public static void main(String[] args) {
        Student s1 = new Student("Bob", 22);
        s1.display();
    }
}
```

Output:

```
Name: Bob, Age: 22
```

---

3. Constructor Overloading

- Multiple constructors can exist in a class with **different parameters**.

Example:

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

    Student() {  // default
        name = "Unknown";
        age = 0;
    }

    Student(String n) { // parameterized
        name = n;
        age = 18;
    }
```

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

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

public class TestStudent {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Charlie");
        Student s3 = new Student("David", 25);

        s1.display(); // Unknown 0
        s2.display(); // Charlie 18
        s3.display(); // David 25
    }
}
```

---

4. The this Keyword

- **this** refers to the **current object**.
- It is used to differentiate **instance variables from local variables**.

Example:

```
class Student {
    String name;
```

```
    int age;

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

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

---

5. Getters and Setters (Encapsulation)

- **Encapsulation** hides data using **private variables** and accesses it via **getter and setter methods**.

Example:

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

    public void setName(String n) { name = n; }
    public void setAge(int a) { age = a; }
    public String getName() { return name; }
    public int getAge() { return age; }
}

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

```
        Student s = new Student();
        s.setName("Eva");
        s.setAge(21);
        System.out.println(s.getName() + " " + s.getAge());
    }
}
```

Output:

Eva 21

---

6. Object Creation and Access

- Objects are created using the **new keyword**.
- Access object variables and methods using **dot . operator**.

Example:

```
Student s = new Student("Fiona", 23);
s.display();
```

Summary

- A **class** is a blueprint, an **object** is an instance.
- **Constructors** initialize objects.
- **Overloading** allows multiple constructors.
- **this keyword** refers to the current object.
- **Encapsulation** protects data using private variables and getter/setter methods.
- Objects are the **core of Java's OOP features**.

Topic 5: Methods in Java

Overview

Methods in Java are **blocks of code that perform specific tasks**. They help **divide a program into smaller, reusable pieces**, improving modularity and readability. Methods can accept **parameters** and return values. Java supports **method overloading**, **static methods**, and **recursion**.

---

1. Defining Methods

- A **method** consists of a **method signature** and **body**.
- **Syntax:**

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

Example: Simple Method

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

   public static void main(String[] args) {
      Calculator calc = new Calculator();
      int sum = calc.add(10, 20);
      System.out.println("Sum: " + sum);
   }
}
```

Output:

```
Sum: 30
```

---

2. Method Parameters and Return Types

- **Parameters:** Values passed to a method to process.
- **Return Type:** Type of value the method returns (int, double, String, etc.)
- If no value is returned, use void.

Example: Multiplication Method

```
class Calculator {
    void multiply(int a, int b) {
        System.out.println("Product: " + (a * b));
    }

    public static void main(String[] args) {
        Calculator c = new Calculator();
        c.multiply(5, 4); // prints Product: 20
    }
}
```

---

3. Method Overloading

- **Overloading**: Multiple methods with the **same name but different parameters**.
- Parameters can differ in **number or type**.

Example: Overloaded max Method

```
class Calculator {
    int max(int a, int b) {
        return (a > b) ? a : b;
    }

    double max(double a, double b) {
        return (a > b) ? a : b;
    }

    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.max(10, 20)); // 20
        System.out.println(c.max(3.5, 2.5)); // 3.5
```

```
    }
}
```

---

4. Static Methods and Variables

- **Static methods** belong to the class rather than an object.
- **Static variables** are shared among all objects of the class.
- Access using ClassName.methodName() or directly inside the class.

Example: Static Method

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

    public static void main(String[] args) {
        System.out.println("Square: " + MathUtil.square(5)); // 25
    }
}
```

Example: Static Variable

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

    Counter() {
        count++;
    }

    static void displayCount() {
        System.out.println("Objects created: " + count);
    }
```

```
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter.displayCount(); // 2
    }
}
```

---

5. Recursion

- A method can call **itself**, called **recursion**.
- Often used for **factorials, Fibonacci, and tree traversal**.

Example: Factorial Using Recursion

```
class Factorial {
    int factorial(int n) {
        if(n == 0) return 1;
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        Factorial f = new Factorial();
        System.out.println("Factorial of 5: " + f.factorial(5)); // 120
    }
}
```

---

6. Example: Maximum of Three Numbers Using Method

```
class MaxOfThree {
    int max(int a, int b, int c) {
        int max = (a > b) ? a : b;
```

```
        return (max > c) ? max : c;
    }

    public static void main(String[] args) {
        MaxOfThree m = new MaxOfThree();
        System.out.println("Maximum: " + m.max(10, 20, 15)); // 20
    }
}
```

Summary

- Methods **improve modularity** and **code reuse**.
- **Method overloading** allows multiple methods with the same name.
- **Static methods/variables** belong to the class, not objects.
- **Recursion** is a powerful technique to solve problems repeatedly.
- Using methods, complex problems are **broken into smaller, manageable tasks**.

6. Object-Oriented Programming (OOP) Concepts

Theory:
Java is an **Object-Oriented Programming language** (OOP). OOP is a programming paradigm based on the concept of **objects**, which can contain data (fields/attributes) and code (methods). OOP aims to organize software as a collection of reusable objects.

Main Concepts:

1. **Class and Object**
    - **Class**: Blueprint for objects.
    - **Object**: Instance of a class.
2. class Car {
3. String color;
4. int speed;
5.
6. void display() {

```
        System.out.println("Car color: " + color + ", speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.color = "Red";
        c1.speed = 120;
        c1.display();
    }
}
```

19. **Encapsulation**
    - Wrapping data (variables) and code (methods) together and controlling access using **private, public, protected**.

```
class BankAccount {
    private double balance;

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

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

31. **Inheritance**
    - Enables a class to acquire properties and behaviors of another class (extends).

32. **Polymorphism**
    - Ability of an object to take multiple forms.
    - **Compile-time** (method overloading) and **Run-time** (method overriding).
33. **Abstraction**
    - Hiding internal implementation and showing only functionality. Achieved using **abstract classes** and **interfaces**.

---

7. Constructors and Destructors

Theory:

- **Constructor**: Special method to initialize objects. It has **no return type** and the name is same as the class.
- **Destructor**: In Java, there is **no explicit destructor**, garbage collector (finalize()) cleans memory automatically.

Types of Constructors:

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor** (manual in Java)

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: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Pintu", 28);

        s1.display();
        s2.display();
    }
}
```

---

8. Arrays and Strings

Arrays:

- Collection of elements of **same type** stored at contiguous memory locations.

```
int[] numbers = {10, 20, 30, 40};
for(int i=0; i<numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Strings:

- Immutable sequence of characters.
- Created using **String literal** or **new keyword**.

```
String s1 = "Hello";
String s2 = new String("World");

System.out.println(s1 + " " + s2); // Output: Hello World
```

---

9. Inheritance and Polymorphism

Inheritance:

- Reuse existing class (parent) in another class (child).
- Syntax: class Child extends Parent {}

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

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.eat();
      d.bark();
```

```
    }
}
```

Polymorphism:

1. **Compile-time (Method Overloading)**

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

2. **Run-time (Method Overriding)**

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

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

---

10. Interfaces and Abstract Classes

Abstract Class:

- Cannot create object directly. Can have **abstract** and **non-abstract** methods.

```
abstract class Shape {
    abstract void area();
}

class Circle extends Shape {
    int radius = 5;
    void area() { System.out.println("Area: " + (3.14 * radius * radius)); }
}
```

Interface:

- 100% abstraction. All methods are **abstract by default**.
- Multiple interfaces can be implemented.

```
interface Drawable {
   void draw();
}

class Rectangle implements Drawable {
   public void draw() { System.out.println("Drawing Rectangle"); }
}
```

---

11. Packages and Access Modifiers

Packages:

- Group of related classes/interfaces.
- import keyword is used to access classes from other packages.

```
package mypack;

public class Demo {
   public void show() { System.out.println("Hello from package"); }
}
```

Access Modifiers:

Modifier	Within Class	Within Package	Subclass	Outside Package
private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

---

12. Exception Handling

Theory:

- Mechanism to handle runtime errors and prevent abnormal termination.
- Types:
    1. **Checked Exception** (Compile-time)
    2. **Unchecked Exception** (Runtime)

Keywords: try, catch, finally, throw, throws

```
public class Main {
   public static void main(String[] args) {
      try {
         int a = 10/0;  // ArithmeticException
      } catch (ArithmeticException e) {
         System.out.println("Error: " + e);
      } finally {
         System.out.println("Finally block executed");
      }
   }
}
```

Custom Exception Example:

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


class Test {
   void check(int age) throws MyException {
      if(age < 18) throw new MyException("Not allowed");
   }
}
```

---

13. Multithreading

Theory:

- Java allows **concurrent execution** of two or more threads.
- Threads can be created using:
    1. **Thread class**
    2. **Runnable interface**

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


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

Runnable Example:

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


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

---

14. File Handling

Theory:

- Java provides **java.io** and **java.nio** for file handling.
- Operations: Read, Write, Append, Delete.

```
import java.io.*;


public class Main {
   public static void main(String[] args) throws IOException {
      FileWriter fw = new FileWriter("test.txt");
      fw.write("Hello Java File");
      fw.close();


      BufferedReader br = new BufferedReader(new FileReader("test.txt"));
      String line;
      while((line = br.readLine()) != null) {
         System.out.println(line);
      }
      br.close();
   }
}
```

---

15. Collections Framework

Theory:

- Predefined classes/interfaces for storing and manipulating **groups of objects**.

Main Interfaces:

1. **List** - Ordered, allows duplicates (ArrayList, LinkedList)
2. **Set** - Unordered, no duplicates (HashSet, LinkedHashSet, TreeSet)

3. **Map** - Key-Value pairs (HashMap, TreeMap)

Example:

```
import java.util.*;


public class Main {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>();
      list.add("Apple");
      list.add("Banana");
      list.add("Apple");  // Duplicate allowed
      System.out.println(list);


      HashSet<String> set = new HashSet<>(list); // Removes duplicates
      System.out.println(set);
   }
}
```

---

16. Java Input/Output (I/O)

Theory:

- Handles communication between **program and outside world** (keyboard, file, network).
- **Streams:** Sequence of data.

Types:

1. **Byte Streams** (InputStream, OutputStream) - Handle binary data.
2. **Character Streams** (Reader, Writer) - Handle text data.

Example (Console Input using Scanner):

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

```
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter name:");
        String name = sc.nextLine();
        System.out.println("Hello " + name);
        sc.close();
    }
}
```