Java Programming Notes

---

1. Introduction to Java

1.1 What is Java?

Java is a **high-level, class-based, object-oriented** programming language developed at **Sun Microsystems (1995)** by **James Gosling**.

- It is designed to be **simple, secure, and portable**.
- Java programs are compiled into **bytecode**, which runs on the **Java Virtual Machine (JVM)**.

💡 **Analogy:** Think of Java like a "universal adapter." You write code once, and it can run on any operating system that has JVM installed.

1.2 History of Java

- 1991 → Project "Oak" started (for embedded devices).
- 1995 → Officially released as **Java**.
- 2009 → Sun Microsystems acquired by Oracle.
- 2017+ → Java follows a **6-month release cycle**.

1.3 Features of Java

Feature	Explanation	Example
Simple	Syntax is easy (similar to C++ but no pointers).	System.out.println("Hello Java");
Object-Oriented	Based on classes/objects.	Classes, Inheritance
Portable	Bytecode can run anywhere.	Compile once, run everywhere
Robust	Strong memory management + exceptions.	try-catch blocks
Secure	No explicit pointers, bytecode verifier.	Java Applets (sandboxed)

Feature	Explanation	Example
Multithreaded	Multiple tasks at the same time.	Thread class
Distributed	Supports networking (RMI, sockets).	Java EE

1.4 How Java Works

1. **Code (.java file)** → Written by developer.
2. **Compiler (javac)** → Converts into **Bytecode (.class file)**.
3. **JVM** → Executes bytecode on any platform.

🖼️ *(Imagine a diagram: Source Code → Compiler → Bytecode → JVM → Output)*

---

2. Introduction to Java Programming Environment

2.1 Components

- **JDK (Java Development Kit):** Tools to develop programs.
- **JRE (Java Runtime Environment):** Required to run Java apps.
- **JVM (Java Virtual Machine):** Executes bytecode.

2.2 Writing First Program

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

Steps to Run:

1. Save file as Hello.java
2. Compile: javac Hello.java
3. Run: java Hello

📌 Output:

Hello, World!

2.3 Java Editions

- **Java SE** → Standard Edition (desktop, console apps).
- **Java EE** → Enterprise Edition (web apps, servers).
- **Java ME** → Micro Edition (mobile devices).

---

3. Fundamentals of Java Programming

3.1 Structure of a Java Program

1. **Package declaration** (optional)
2. **Import statements**
3. **Class definition**
4. **Main method**

Example:

```
package mypackage;

import java.util.Scanner;


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

3.2 Variables

- **Local Variables** → inside methods.

- **Instance Variables** → defined inside class but outside methods.
- **Static Variables** → shared among all objects.

Example:

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

   void display() {
      int localVar = 30;     // Local
      System.out.println(localVar);
   }
}
```

3.3 Data Types

Type	Size	Range
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2^31 to 2^31-1
long	8 bytes	-2^63 to 2^63-1
float	4 bytes	6-7 decimal digits
double	8 bytes	15 decimal digits
char	2 bytes	Unicode (0–65535)
boolean	1 bit	true/false

3.4 Operators

- **Arithmetic** → + - * / %
- **Relational** → < <= > >= == !=

- **Logical** → && || !
- **Assignment** → = += -= *=
- **Increment/Decrement** → ++ --

💡 Example:

```
int a = 10, b = 5;
System.out.println(a+b);  // 15
System.out.println(a>b);  // true
System.out.println(a==b); // false
```

---

4. Control Structures

4.1 If-Else

```
int num = 10;
if (num > 0)
   System.out.println("Positive");
else
   System.out.println("Negative");
```

4.2 Switch

```
int day = 3;
switch(day) {
   case 1: System.out.println("Monday"); break;
   case 2: System.out.println("Tuesday"); break;
   default: System.out.println("Other day");
}
```

4.3 Loops

- **For Loop**

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

- **While Loop**

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

- **Do-While Loop**

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

4.4 Jump Statements

- break → exits loop.
- continue → skips iteration.
- return → exits method.

Example:

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

Input Fundamentals and Data Types

5.1 User Input in Java

In Java, input is generally handled by the **Scanner class** (from java.util package).

Example:

```
import java.util.Scanner;


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


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


      System.out.println("Enter your name:");
      String name = sc.next();


      System.out.println("Hello " + name + ", Age: " + age);
   }
}
```

📌 Input methods:

- nextInt() → integers
- nextDouble() → floating-point
- nextLine() → full line of text
- next() → single word

5.2 Command-Line Input

Command-line arguments are passed as a **String array** in the main method.

```
class CLDemo {
   public static void main(String[] args) {
      for(int i=0; i<args.length; i++) {
```

```
        System.out.println("Arg " + i + ": " + args[i]);
    }
  }
}
```

Run:

java CLDemo Hello World 123

Output:

Arg 0: Hello

Arg 1: World

Arg 2: 123

5.3 Primitive Data Types Recap

Type	Example	Use Case
int	int x = 25;	Counting, IDs
double	double pi = 3.14;	Precise decimal values
char	char grade = 'A';	Single characters
boolean	boolean flag = true;	Conditions
String (non-primitive)	String name="Java";	Text handling

---

6. Object-Oriented Programming in Java

Java follows **OOP principles**:

6.1 Classes and Objects

A **class** is a blueprint, and an **object** is an instance.

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

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

class Test {
  public static void main(String[] args) {
    Student s1 = new Student();
    s1.name = "Rahul";
    s1.age = 20;
    s1.display();
  }
}
```

6.2 Four Pillars of OOP

1. **Encapsulation** → Wrapping data & methods (using classes, private variables, getters & setters).
2. **Abstraction** → Hiding implementation (interfaces, abstract classes).
3. **Inheritance** → Reuse by deriving classes (extends).
4. **Polymorphism** → Multiple forms (method overloading/overriding).

6.3 Constructors

```
class Car {
  String model;
  Car(String m) {
    model = m;
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Car c = new Car("Tesla");
    System.out.println(c.model);
  }
}
```

---

7. Command-Line Arguments

Covered in **Section 5.2**, but some **real-world uses**:

- Passing file names: java MyApp input.txt
- Configurations: java ServerApp -port 8080

Example:

```
class Sum {
  public static void main(String[] args) {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    System.out.println("Sum: " + (a+b));
  }
}
```

Run:

java Sum 10 20

Output:

Sum: 30

---

8. Integrated Development Environment (IDE)

8.1 Popular Java IDEs

- **Eclipse** → Widely used, enterprise-friendly.
- **IntelliJ IDEA** → Smart code completion, powerful refactoring.

- **NetBeans** → Open-source, easy GUI building.
- **VS Code (with Java extensions)** → Lightweight alternative.

8.2 Benefits of IDE

- Code completion (auto-suggestions).
- Debugging tools.
- Integrated compiler & runtime.
- Project management.
- GUI design tools.

---

9. Inner Classes

9.1 Types of Inner Classes

1. **Member Inner Class**

```
class Outer {
  class Inner {
    void msg() { System.out.println("Hello from Inner!"); }
  }
}
class Test {
  public static void main(String[] args) {
    Outer o = new Outer();
    Outer.Inner in = o.new Inner();
    in.msg();
  }
}
```

2. **Static Nested Class**

```
class Outer {
  static class Inner {
```

```
    void msg() { System.out.println("Static Inner!"); }
  }
}
```

3. **Anonymous Inner Class**

```
abstract class Animal {
  abstract void sound();
}
class Test {
  public static void main(String[] args) {
    Animal dog = new Animal() {
      void sound() { System.out.println("Bark!"); }
    };
    dog.sound();
  }
}
```

---

10. Types of Inheritance

10.1 Single Inheritance

```
class A {
  void show() { System.out.println("A"); }
}
class B extends A {
  void display() { System.out.println("B"); }
}
```

10.2 Multilevel Inheritance

```
class A { void show() { System.out.println("A"); } }
class B extends A { void display() { System.out.println("B"); } }
class C extends B { void print() { System.out.println("C"); } }
```

10.3 Hierarchical Inheritance

```
class A { void show() { System.out.println("A"); } }

class B extends A { void display() { System.out.println("B"); } }

class C extends A { void print() { System.out.println("C"); } }
```

📌 Note: Java does **not** support **multiple inheritance with classes** to avoid ambiguity (diamond problem). Instead, it uses **interfaces**.

---

11. Abstract Class and Inheritance

11.1 Abstract Class

An abstract class **cannot be instantiated**, but can have abstract & concrete methods.

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

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

---

12. Polymorphism

12.1 Compile-Time (Method Overloading)

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

12.2 Runtime (Method Overriding)

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

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

---

13. Packages in Java

13.1 Defining a Package

```
package mypackage;


public class MyClass {
   public void msg() { System.out.println("Hello from package!"); }
}
```

13.2 Using a Package

```
import mypackage.MyClass;


class Test {
   public static void main(String[] args) {
      MyClass m = new MyClass();
      m.msg();
   }
}
```

14. Interfaces in Java

14.1 What is an Interface?

- An **interface** in Java is a blueprint of a class.
- It contains **abstract methods** (implicitly public abstract) and constants (implicitly public static final).
- A class **implements** an interface.

📌 Java supports **multiple inheritance** through interfaces.

14.2 Example

```
interface Vehicle {
   void start();
}
```

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

class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike starts with a button!");
    }
}

class Test {
    public static void main(String[] args) {
        Vehicle v = new Car();
        v.start();
    }
}
```

14.3 Default and Static Methods in Interfaces (Java 8+)

```
interface Calculator {
    void add(int a, int b);

    default void greet() {
        System.out.println("Hello from Calculator!");
    }

    static void info() {
```

```
        System.out.println("Calculator Interface (Static method)");
    }
}
```

---

15. Exception Handling

15.1 What is an Exception?

- An **exception** is an unwanted event that disrupts program execution.
- Example: division by zero, file not found, invalid input.

15.2 Types of Exceptions

1. **Checked Exceptions** → Must be handled at compile time (e.g., IOException).
2. **Unchecked Exceptions** → Occur at runtime (e.g., NullPointerException).
3. **Errors** → Serious issues (e.g., OutOfMemoryError).

15.3 Syntax

```
try {
    int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Error: " + e);
} finally {
    System.out.println("Finally block always executes");
}
```

15.4 Throw and Throws

```
class Test {
    static void checkAge(int age) throws Exception {
        if(age < 18)
            throw new Exception("Not eligible to vote");
```

```
    }
    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

---

16. Multithreading in Java

16.1 What is Multithreading?

- Multithreading = executing multiple tasks **simultaneously**.
- Each task = a **thread**.

16.2 Creating a Thread

Two ways:

1. Extending Thread class
2. Implementing Runnable interface

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running...");
    }
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

16.3 Runnable Example

```
class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Runnable thread running...");
   }
   public static void main(String[] args) {
      Thread t = new Thread(new MyRunnable());
      t.start();
   }
}
```

16.4 Thread Methods

- start() → starts thread
- sleep(ms) → pauses execution
- join() → waits for a thread to finish
- setPriority(int) → sets priority (1 to 10)

---

17. Collections Framework

17.1 What is Collection?

- A **collection** is a group of objects.
- Java Collections Framework provides **classes & interfaces** for storing and manipulating groups of data.

17.2 Interfaces

- List → ordered collection (ArrayList, LinkedList)
- Set → unique elements (HashSet, TreeSet)
- Map → key-value pairs (HashMap, TreeMap)

17.3 Example (ArrayList)

```
import java.util.*;


class Test {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>();
      list.add("Java");
      list.add("Python");
      list.add("C++");

      for(String s : list)
         System.out.println(s);
   }
}
```

17.4 Example (HashMap)

```
import java.util.*;


class Test {
   public static void main(String[] args) {
      HashMap<Integer, String> map = new HashMap<>();
      map.put(1, "Apple");
      map.put(2, "Banana");
      map.put(3, "Cherry");

      for(Map.Entry<Integer, String> entry : map.entrySet()) {
         System.out.println(entry.getKey() + " -> " + entry.getValue());
```

```
    }
  }
}
```

---

18. Generics in Java

18.1 What are Generics?

- Generics allow classes, interfaces, and methods to operate on **any data type** while maintaining type safety.

```
class Box<T> {
  T value;
  void set(T val) { value = val; }
  T get() { return value; }
}

class Test {
  public static void main(String[] args) {
    Box<Integer> b1 = new Box<>();
    b1.set(10);
    System.out.println(b1.get());

    Box<String> b2 = new Box<>();
    b2.set("Hello");
    System.out.println(b2.get());
  }
}
```

---

19. Streams and Lambda Expressions (Java 8+)

19.1 Lambda Expression

```
interface Greeting {
   void sayHello();
}

class Test {
   public static void main(String[] args) {
      Greeting g = () -> System.out.println("Hello, Lambda!");
      g.sayHello();
   }
}
```

19.2 Streams API

```
import java.util.*;

class Test {
   public static void main(String[] args) {
      List<Integer> nums = Arrays.asList(1,2,3,4,5);

      nums.stream()
         .filter(n -> n % 2 == 0)
         .map(n -> n * n)
         .forEach(System.out::println);
   }
}
```

Output:

4

16

---

20. JDBC (Java Database Connectivity)

20.1 Steps to Connect Java with Database

1. Import java.sql.*
2. Load Driver class (Class.forName)
3. Establish Connection (DriverManager.getConnection)
4. Create Statement/PreparedStatement
5. Execute SQL Query
6. Process ResultSet
7. Close connection

20.2 Example

```
import java.sql.*;


class DBTest {
   public static void main(String[] args) {
      try {
         Class.forName("com.mysql.cj.jdbc.Driver");
         Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/testdb", "root", "password");


         Statement stmt = con.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM students");


         while(rs.next())
            System.out.println(rs.getInt(1) + " " + rs.getString(2));


         con.close();
      } catch(Exception e) {
```

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

21. Serialization and Deserialization

21.1 What is Serialization?

- Serialization = converting an object into a **byte stream** (so it can be saved to a file, sent over a network, etc.).
- Deserialization = converting the byte stream back into an object.

21.2 Example

```
import java.io.*;

class Student implements Serializable {
  int id;
  String name;
  public Student(int id, String name) {
    this.id = id;
    this.name = name;
  }
}

class Test {
  public static void main(String[] args) {
    try {
      // Serialization
      Student s1 = new Student(101, "Alice");
      FileOutputStream fos = new FileOutputStream("student.ser");
```

```
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(s1);
        oos.close();
        fos.close();
        System.out.println("Object Serialized");

        // Deserialization
        FileInputStream fis = new FileInputStream("student.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Student s2 = (Student) ois.readObject();
        System.out.println("Deserialized: " + s2.id + " " + s2.name);

      } catch(Exception e) {
        e.printStackTrace();
      }
   }
}
```

---

22. Networking in Java

22.1 Introduction

- Java provides java.net package for networking.
- Common classes:
    - Socket, ServerSocket
    - InetAddress
    - URL

22.2 Example: Client-Server Communication

Server

```
import java.net.*;
import java.io.*;

class Server {
   public static void main(String[] args) throws Exception {
      ServerSocket ss = new ServerSocket(6666);
      Socket s = ss.accept();
      DataInputStream dis = new DataInputStream(s.getInputStream());
      String str = dis.readUTF();
      System.out.println("Message = " + str);
      ss.close();
   }
}
```

Client

```
import java.net.*;
import java.io.*;

class Client {
   public static void main(String[] args) throws Exception {
      Socket s = new Socket("localhost", 6666);
      DataOutputStream dout = new DataOutputStream(s.getOutputStream());
      dout.writeUTF("Hello Server!");
      dout.flush();
      dout.close();
      s.close();
   }
}
```

---

23. Java GUI Programming (Swing & JavaFX)

23.1 Swing

- Swing is part of Java Foundation Classes (JFC).
- Used to create **desktop applications**.

Example: Simple JFrame

```
import javax.swing.*;


class MyFrame {
   public static void main(String[] args) {
      JFrame f = new JFrame("My App");
      JButton b = new JButton("Click Me!");
      b.setBounds(50, 100, 100, 40);
      f.add(b);
      f.setSize(300, 300);
      f.setLayout(null);
      f.setVisible(true);
   }
}
```

23.2 JavaFX (Modern GUI)

- Introduced in Java 8.
- Uses **FXML** and supports **CSS-like styling**.

---

24. Advanced OOP: Design Patterns in Java

24.1 Singleton Pattern

Ensures only one instance of a class exists.

```
class Singleton {
   private static Singleton instance;
```

```
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

24.2 Factory Pattern

Used to create objects without specifying exact class.

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

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

class Square implements Shape {
    public void draw() { System.out.println("Drawing Square"); }
}

class ShapeFactory {
    public Shape getShape(String type) {
        if(type.equalsIgnoreCase("CIRCLE")) return new Circle();
        if(type.equalsIgnoreCase("SQUARE")) return new Square();
        return null;
    }
}
```

```
class Test {
   public static void main(String[] args) {
      ShapeFactory factory = new ShapeFactory();
      Shape s1 = factory.getShape("CIRCLE");
      s1.draw();
   }
}
```

---

25. Java and Web Applications

25.1 Java Servlets

- Servlets are Java programs that run on a web server.
- Used for handling **HTTP requests and responses**.

25.2 JSP (JavaServer Pages)

- Easier way to create dynamic web content by embedding Java into HTML.

25.3 Spring Framework (Brief)

- Modern enterprise applications use **Spring Boot**.
- Provides features for:
    - Dependency Injection
    - Web Applications
    - Security
    - REST APIs

---

26. Java and Android

- **Java is the primary language for Android app development** (along with Kotlin).
- Example: Android MainActivity.java uses Java code to control UI.

---

27. Real-World Applications of Java

1. **Banking Systems** → transaction management, ATM software.
2. **E-commerce** → Amazon, Flipkart backend uses Java.
3. **Big Data (Hadoop, Spark)** → written in Java/Scala.
4. **Android Apps** → WhatsApp, Instagram (initial versions in Java).
5. **Enterprise Applications** → ERP, CRM systems.

---

28. Best Practices in Java

1. Follow **naming conventions** (ClassName, methodName).
2. Always **close resources** (files, DB connections).
3. Use **exceptions properly**, avoid empty catch blocks.
4. Use **generics** for type safety.
5. Apply **SOLID principles** in OOP.

---

29. Case Study: Java in an Online Shopping System

Modules

- User Authentication → Login, Signup (Servlet + JDBC)
- Product Catalog → Stored in Database, displayed with JSP
- Shopping Cart → Implemented using **Collections (ArrayList)**
- Payment Processing → Java libraries with security (SSL)
- Multithreading → Handles multiple users concurrently

📌 Demonstrates how **OOP, collections, JDBC, multithreading, and exception handling** all combine in a real-world project.

---

30. Summary

- Java evolved from a simple language for interactive devices to a **powerful enterprise and mobile platform**.
- Key strengths: **platform independence, OOP, security, scalability**.

- Applications: **Desktop apps, Web, Mobile, Cloud, AI, IoT, Big Data**.