Java: A Brief Overview

Introduction

- Developed by James Gosling at Sun Microsystems (1995); later acquired by Oracle.
- Named after "Java coffee" from an Indonesian island.
- A simple, object-oriented, and platform-independent language.
- **WORA**: Write Once, Run Anywhere.

Applications

- Mobile (Android apps), Desktop, Web, Servers, Games, Databases, and more.

Key Features

1. **Platform Independence**: Compile once, run anywhere via JVM.
2. **Object-Oriented**: Based on abstraction, encapsulation, inheritance, and polymorphism.
3. **Simple**: Avoids complexities like pointers and operator overloading.
4. **Robust**: Strong error-checking.
5. **Multithreading**: Allows concurrent task execution.
6. **Portable**: Bytecode can run on any platform.
7. **High Performance**: Optimized execution with Just-In-Time (JIT) compiler.
8. **Dynamic**: Flexible to add classes and methods.

Classes and Objects

- Classes are blueprints for objects containing data fields and methods.
- Programs are collections of classes.

Java API

- A library of prewritten classes for input, database handling, etc.
- Divided into packages; use import to include.
- **JDK, JRE, JVM**: Core components for development and execution.

Java Program Lifecycle

1. **Edit**: Write source code (.java).
2. **Compile**: Convert to bytecode (.class) via javac.
3. **Load**: Class loader transfers bytecode to memory.
4. **Verify**: Bytecode verifier checks security.
5. **Execute**: JVM interprets bytecode into machine language.

Basic Syntax
Example:

```
public class FirstProgram {
   public static void main(String[] args) {
      System.out.println("Mishon Das Arnab");
   }
}
```

- public: Access modifier.
- class: Declares class name.
- static: Makes methods accessible without an object.
- main(): Program entry point.
- System.out.println(): Prints output.

Comments

- Single-line: //
- Multi-line: /* ... */

Scopes

- Defined by { } braces to specify the visibility and lifetime of variables and methods.

Output Formatting

- Use System.out.printf() for formatted output.

Example Program

```
public class Main {
    public static void main(String[] args) {
        int a = 10, b = 20;
        System.out.println("Sum=" + (a + b));
        System.out.println("Product=" + (a * b));
    }
}
```

Lecture 2

Identifiers in Java

1. **Rules for Identifiers**:
    - Can contain letters, digits, underscores (_), and dollar signs ($).
    - Must begin with a letter.
    - Case-sensitive.
    - Cannot contain spaces or start with a digit.
    - Cannot use reserved keywords (e.g., int, boolean).
2. **Valid Identifiers**:
    - MyVariable, myvariable, _myvariable, $myvariable, sum_of_array.
3. **Invalid Identifiers**:
    - My Variable (space), 123geeks (starts with a digit), a+c (invalid symbol).

Variables in Java

1. **Variable Types**:
    - String: Text ("Hello").

- int: Integers (123).
- float: Decimal numbers (19.99f).
- char: Single characters ('A').
- boolean: True/False values.

2. **Declaration and Initialization**:

```
int myNum = 5;
float myFloatNum = 5.99f;
char myLetter = 'D';
boolean myBool = true;
String myText = "Hello";
```

3. **Multiple Declarations**:

```
int x = 5, y = 6, z = 50;
System.out.println(x + y + z);
```

Primitive Data Types

1. **Integer Types**:
   - byte: 1 byte (-128 to 127).
   - short: 2 bytes (-32,768 to 32,767).
   - int: 4 bytes (-2,147,483,648 to 2,147,483,647).
   - long: 8 bytes (-9 quintillion to 9 quintillion).
2. **Floating-Point Types**:
   - float: 4 bytes (up to 6-7 decimals).
   - double: 8 bytes (up to 15 decimals).
3. **Other Types**:
   - boolean: 1 bit (true or false).
   - char: 2 bytes (single characters or Unicode).

Constants and Naming Conventions

1. **Constants**:

- Use final keyword.
- Example: final double PI = 3.14159;.

2. **Naming Conventions**:
   - Variables/methods: Lowercase (radius, computeArea).
   - Classes: Capitalize each word (ComputeArea).
   - Constants: All caps with underscores (MAX_VALUE).

Key Points

- Java is strongly typed, requiring type declaration.
- Use double for precision in floating-point calculations.
- All statements end with a semicolon (;).
-

Lecture-3

1. **Scanner Class**: Used to read input from users.

```
import java.util.Scanner;

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

2. **Common Methods**:
   - nextInt(): Reads an integer.
   - nextFloat(): Reads a float.
   - nextLine(): Reads an entire line (with spaces).
   - next(): Reads a word (stops at whitespace).
3. **Example**:

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

System.out.print("Enter your name: ");

String name = input.nextLine();

System.out.println("I am Mishon and Your name is: " + name);
```

input.close();

4. **next() vs. nextLine()**:
    - next(): Stops at whitespace.
    - nextLine(): Reads the entire line.

Java Type Casting

1. **Widening Casting** (automatic):
    - Converts smaller to larger types: byte -> short -> int -> long -> float -> double.

```
int num = 555;
double d = num;
System.out.println(d); // 555.0
```

2. **Narrowing Casting** (manual):
    - Converts larger to smaller types: double -> float -> long -> int -> short -> byte.

```
double num = 11.1234;
int n = (int) num; // Truncates value
System.out.println(n); // 11
```

3. **Example**:

```
double d = 3.6;
int x = (int) Math.round(d); // 4
```

Math Library

1. **Common Methods**:
    - Math.max(a, b): Returns the larger value.
    - Math.sqrt(x): Square root.
    - Math.abs(x): Absolute value.
    - Math.pow(x, y): Power calculation.
2. **Example**:

```
System.out.println(Math.max(5, 10)); // 10
System.out.println(Math.sqrt(64)); // 8.0
```

```
System.out.println(Math.abs(-4.7)); // 4.7
System.out.println(Math.pow(2.5, 2)); // 6.25
```

Lecture-4

 Literals:

- **Numeric**: Constants like 34, 1_000_000, or 5.0.
- **Floating-Point**: Includes standard (42.4362) and scientific notation (424362E-4).
- **Boolean**: true and false (not convertible to numeric values).
- **Character**: Single characters like 'a', '\n'.
- **String**: Enclosed in double quotes, e.g., "hello world".

 Unary Operators:

- Pre-increment (++var) and Post-increment (var++).
- Pre-decrement (--var) and Post-decrement (var--).

 Binary Operators:

- **Arithmetic**: Addition, subtraction, multiplication, division, modulus.
- **Relational**: Used for comparisons, return boolean values.
- **Logical**: Logical AND (&&) and OR (||).
- **Assignment**: Assign values using = or compound operators like +=.

 Ternary Operator:

- Shorthand for if-else: condition ? ifTrue : ifFalse.
- Example:

```
int result = (a > b) ? a : b;
```

 Bitwise Operators:

- Operate on individual bits (e.g., &, |, ^).

 Precedence and Associativity:

- Defines the order in which operators are evaluated.

 Problem Example:

- Convert seconds to minutes and seconds:

```
int minutes = seconds / 60;

int remainingSeconds = seconds % 60;
```

 Evaluation of Expressions:

- Java follows standard arithmetic rules for evaluating expressions.

Lecture-5

1. Flow of Control

- The execution of statements in a program follows a **sequential order** unless controlled.
- Selection statements help in **decision-making** by evaluating **boolean expressions** (true/false).

2. Selection (Conditional) Statements

Java supports three selection statements:

- **if statement** – Executes a block of code if a condition is true.
- **if-else statement** – Provides two execution paths based on the condition.
- **switch statement** – Allows multiple execution paths based on matching values.

3. The if Statement

- Syntax:

```
if (condition) {
   statementBlock;
}
```

- Uses **relational operators** (==, !=, <, >, <=, >=) for condition checking.
- Example:

```
if (sum > MAX)
   delta = sum - MAX;
System.out.println("The sum is " + sum);
```

Logical Operators in if Statements

- **! (NOT)** – Reverses a boolean value.
- **&& (AND)** – Both conditions must be true.
- **|| (OR)** – At least one condition must be true.
- **^ (XOR)** – Only one of the two conditions must be true.

4. The if-else Statement

- Syntax:

```
if (condition) {
   statementBlock1;
}
else {
   statementBlock2;
}
```

- Example (Grading System):

```
if (score >= 90)
   System.out.print("A");
else if (score >= 80)
   System.out.print("B");
else if (score >= 70)
   System.out.print("C");
else
   System.out.print("F");
```

5. The Conditional (?:) Operator

- A **ternary operator** that acts as a shorthand for if-else.
- Syntax:

```
result = (condition) ? value1 : value2;
```

- Example:

```
larger = (num1 > num2) ? num1 : num2;
```

6. The switch Statement

- Alternative to if-else when checking multiple possible values.
- Syntax:

```
switch (expression) {
   case value1:
      statementBlock1;
      break;
   case value2:
      statementBlock2;
      break;
   default:
      statementBlockDefault;
}
```

- Example (Day of the Week):

```
switch (day) {
   case 1: case 2: case 3: case 4: case 5:
      System.out.println("Weekday");
      break;
   case 6: case 7:
      System.out.println("Weekend");
      break;
   default:
      System.out.println("Invalid Day");
}
```

7. Useful Hints for Writing Selection Statements

- **Indentation matters** – Helps in readability, though ignored by the compiler.
- **Avoid misplaced semicolons** in if conditions.
- **Use braces {}** for multi-line statements inside if, else, and switch cases.
- **Operator precedence** affects the evaluation of expressions.

Lecture-6

1. Introduction to Loops

- Loops allow repeated execution of a block of code.
- Example: Printing "Welcome to Java!" **1000 times** manually is impractical.
- Loops **automate repetition**, making the code efficient and manageable.

2. Types of Loops in Java

Java provides **three types of loops**:

1. **while loop** – Repeats while a condition remains true.
2. **do-while loop** – Executes at least once before checking the condition.
3. **for loop** – Runs a fixed number of times, typically used for **counting iterations**.

---

3. The while Loop

- Syntax:

```
while (condition) {
   statementBlock;
}
```

- **Condition is checked first** before executing the loop body.
- Executes **zero or more times** depending on the condition.

Example

```
int count = 0;
while (count < 5) {
   System.out.println("Welcome to Java!");
   count++;
}
```

✅ Runs **5 times**, printing the message.

Sentinel Value in while Loops

- A **sentinel value** (special input) is used to **terminate** the loop.
- Example: Asking for grades until the user enters 9999 to stop.

---

4. The do-while Loop

- Ensures the loop body executes **at least once**, even if the condition is false.
- Syntax:

```
do {
   statementBlock;
}
while (condition);
```

Example

```
int count = 0;
do {
   count++;
   System.out.println(count);
} while (count < 5);
```

✅ Prints **1 to 5**, ensuring execution at least once.

---

5. The for Loop

- Best used when the number of iterations is **known beforehand**.
- Syntax:

```
for (initialization; condition; increment) {
   statementBlock;
}
```

Example

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

✅ Runs exactly **5 times**, printing numbers **1 to 5**.

Converting for loop to while loop

A for loop can be rewritten using a while loop:

```
int i = 1;
```

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

---

6. Infinite Loops

- Occur when the loop **never terminates** due to incorrect condition or missing update.
- Example of an **infinite loop** (logical error):

```
int count = 1;
while (count <= 25) {
   System.out.println(count);
   count = count - 1; // Mistake: count never increases!
}
```

- **Fix**: Ensure the loop condition becomes false eventually.

---

7. Nested Loops

- A loop inside another loop.
- Used in **pattern printing, matrices, and complex logic**.

Example: Printing a Triangle of Stars

```
for (int row = 1; row <= 5; row++) {
   for (int star = 1; star <= row; star++) {
      System.out.print("*");
   }
   System.out.println();
}
```

✅ Prints:

```
*
**
***
```

****

*****

---

8. break and continue Statements

Used to **control loop execution**:

- **break**: Exits the loop immediately.
- **continue**: Skips the current iteration and moves to the next.

Example: Using break

```
int sum = 0, number = 0;
while (number < 20) {
   number++;
   sum += number;
   if (sum >= 100)
 break;
}
System.out.println("Stopped at: " + number);
```

✅ Stops when sum reaches 100.

Example: Using continue

```
for (int i = 1; i <= 10; i++) {
   if (i == 5 || i == 6) continue; // Skip 5 and 6
   System.out.println(i);
}
```

✅ **Skips** printing 5 and 6.

Conclusion

- Loops **reduce redundancy** and **improve efficiency**.
- Use while for **unknown iterations**, do-while when **at least one iteration is needed**, and for when **counting iterations**.
- Avoid **infinite loops** by ensuring conditions become false.

- break and continue help in **controlling loop execution** effectively.

Lecture -7

1. Introduction to Methods

- A **method** is a block of code that performs a specific task.
- Helps in **code reusability**, **modularity**, and **reducing complexity**.

Example Without Methods

```
int sum = 0;
for (int i = 1; i <= 10; i++)
sum += i;
System.out.println("Sum from 1 to 10: " + sum);

sum = 0;
for (int i = 20; i <= 30; i++) sum += i;
System.out.println("Sum from 20 to 30: " + sum);
```

🚫 **Problem**: Code repetition.

Solution Using Methods

```
public static int sum(int num1, int num2) {
   int sum = 0;
   for (int i = num1; i <= num2; i++) sum += i;
   return sum;
}

public static void main(String[] args) {
   System.out.println("Sum from 1 to 10: " + sum(1, 10));
   System.out.println("Sum from 20 to 30: " + sum(20, 30));
}
```

✅ **Advantage**: Reusable and organized code.

---

2. Declaring a Method

Syntax:

```
<access_modifier> <return_type> <method_name>(parameters) {
   // method body
}
```

- **Access Modifier**: (public, private, protected).
- **Return Type**: The data type of the return value (int, double, void).
- **Method Name**: Identifier for the method.
- **Parameters**: Inputs passed to the method.

---

3. Calling Methods

Example: Finding the Maximum Value

```
public static int max(int num1, int num2) {
   return (num1 > num2) ? num1 : num2;
}


public static void main(String[] args) {
   int result = max(5, 2);
   System.out.println("Max value is: " + result);
}
```

🚀 **Concepts**:

- **Method Invocation** – Calls the method.
- **Return Value** – Stores the result.

---

4. void Methods (No Return Value)

- A method that performs an action but does not return a value.

Example: Even or Odd Check

```
public static void evenOdd(int n) {
```

```
    if (n % 2 == 0) System.out.println(n + " is Even.");
    else System.out.println(n + " is Odd.");
}

public static void main(String[] args) {
    evenOdd(10);
}
```

✅ **Prints "10 is Even."**

---

5. Passing Parameters

- **Formal Parameters** – Variables declared in the method.
- **Actual Parameters** – Values passed during method invocation.

Example: Printing a Message Multiple Times

```
public static void printMessage(String message, int n) {
    for (int i = 0; i < n; i++) {
        System.out.println(message);
    }
}

public static void main(String[] args) {
    printMessage("Welcome to Java", 5);
}
```

✅ **Prints "Welcome to Java" five times.**

---

6. Pass-by-Value in Java

- Java **passes arguments by value**, meaning the original variable **remains unchanged**.

Example: Swap Method (Incorrect Result)

```
public static void swap(int n1, int n2) {
    int temp = n1;
```

```
    n1 = n2;
    n2 = temp;
}

public static void main(String[] args) {
    int num1 = 11, num2 = 200;
    swap(num1, num2);
    System.out.println("After swap: num1 = " + num1 + ", num2 = " + num2);
}
```

🚫 **Does NOT swap values** because Java **passes values, not references**.

---

7. Method Overloading (Same Name, Different Parameters)

- A method can have multiple versions with different parameter types.

Example: Overloading max Method

```
public static int max(int num1, int num2) {
    return (num1 > num2) ? num1 : num2;
}

public static double max(double num1, double num2) {
    return (num1 > num2) ? num1 : num2;
}
```

✅ **Allows calling max(5, 10) or max(5.5, 10.3).**

🚫 **Ambiguous Invocation Error** occurs when two methods have unclear matches.

---

8. Scope of Local Variables

- **Local Variable**: Declared inside a method and only accessible there.
- Java **does not allow** two variables with the same name inside nested blocks.

Example (Error Case)

```
public static void incorrectMethod() {
```

```
    int x = 1;
    for (int i = 1; i < 10; i++) {
        int x = 0; // ❌ Error: x is already declared
    }
}
```

🚫 **Fix**: Use different variable names.

---

9. Using Methods from Other Classes

Java allows **method reuse across classes**.

Example: Using Math Class Methods

```
double result = Math.pow(2, 3);  // 2^3 = 8
double squareRoot = Math.sqrt(25); // Square root of 25 = 5
```

✅ **Built-in utility methods** improve efficiency.

---

10. Call Stack & Method Execution

- **Runtime Stack** stores **active method calls**.
- Methods execute **last in, first out (LIFO)** order.

Example: Call Stack Execution

```
public static int max(int a, int b) {
    return (a > b) ? a : b;
}

public static void main(String[] args) {
    int k = max(5, 2);
    System.out.println(k); // 5
}
```

🚀 **Execution Order**:
1️⃣ main() calls max()

2. max() returns the value
3. main() prints the result

---

11. Modularizing Code with Methods

- **Methods divide a large program into smaller, manageable parts.**
- Improves **readability, reusability, and debugging**.

Conclusion

✓ Methods **reduce redundancy** and **improve reusability**.
✓ Java uses **pass-by-value**, so original variables are unchanged.
✓ **Method overloading** allows multiple methods with the same name.
✓ **Scope rules** prevent variables from being redefined in the same block.
✓ **Call stack** tracks method execution order.

Lecture-

1. Introduction to Arrays

- An **array** is a data structure that stores **multiple values of the same type**.
- **Single-dimensional arrays** store elements in **one row**.

Example:

```
int[] numbers = new int[5];  // Declaring an array

numbers[0] = 10;  // Assigning a value

System.out.println(numbers[0]);  // Accessing a value
```

---

2. Declaring & Creating Arrays

- **Declaration:**

```
int[] myArray;
```

- **Memory Allocation:**

```
myArray = new int[10];  // Allocating memory for 10 integers
```

- **Combined Declaration & Allocation:**

```
int[] myArray = new int[10];
```

---

3. Array Length & Default Values

- **Array size is fixed** once created.
- Use arrayName.length to get the size.

```
int size = myArray.length;
```

- **Default Values:**
    - 0 for numeric types (int, double, float, etc.)
    - false for boolean
    - '\u0000' (null character) for char

---

4. Array Indexing & Initialization

- **Index starts at 0** and goes up to array.length - 1.
- **Assigning values using an index:**

```
myArray[0] = 25;
```

- **Shorthand Initialization:**

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

---

5. Processing Arrays (Common Operations)

5.1 Initializing Arrays with Input

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

int[] myArray = new int[5];


for (int i = 0; i < myArray.length; i++) {
   myArray[i] = input.nextInt();
}
```

5.2 Initializing with Random Values

```
for (int i = 0; i < myArray.length; i++) {
   myArray[i] = (int) (Math.random() * 100);
```

```
}
```

5.3 Printing Array Elements

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

5.4 Summing All Elements

```
int sum = 0;
for (int num : myArray) {
   sum += num;
}
```

5.5 Finding the Largest Element

```
int max = myArray[0];
for (int i = 1; i < myArray.length; i++) {
   if (myArray[i] > max) {
      max = myArray[i];
   }
}
```

---

6. Enhanced for Loop (For-Each Loop)

- Introduced in **Java 1.5** for **easier traversal** of arrays.

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

---

7. Copying Arrays

7.1 Using a Loop

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
```

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

7.2 Using System.arraycopy()

```
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

---

8. Searching in Arrays

8.1 Linear Search

- **Searches sequentially** from first to last element.
- Returns **index of element** if found, **-1** otherwise.

```
public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == key) return i;
    }
    return -1;
}
```

8.2 Binary Search (For Sorted Arrays Only)

- **Divides the array** into halves to search efficiently.

```
public static int binarySearch(int[] list, int key) {
    int low = 0, high = list.length - 1;

    while (high >= low) {
        int mid = (low + high) / 2;
        if (key < list[mid]) high = mid - 1;
        else if (key == list[mid]) return mid;
        else low = mid + 1;
    }
```

```
    return -1;  // Not found
}
```

---

9. Sorting Arrays

9.1 Selection Sort Algorithm

- **Finds the smallest element** and swaps it to the correct position.

```
for (int i = 0; i < list.length; i++) {
    int minIndex = i;
    for (int j = i + 1; j < list.length; j++) {
        if (list[j] < list[minIndex]) {
            minIndex = j;
        }
    }
    // Swap elements
    int temp = list[i];
    list[i] = list[minIndex];
    list[minIndex] = temp;
}
```

9.2 Using Arrays.sort()

- **Built-in sorting method** in Java.

```
import java.util.Arrays;
int[] numbers = {6, 4, 1, 9, 2};
Arrays.sort(numbers);
```

---

10. Conclusion

✓ Arrays store multiple values efficiently.
✓ Elements are accessed using indexes (starting from 0).
✓ Common operations include **traversal, copying, searching, and sorting.**
✓ For-each loops simplify array iteration.

✓ **Binary search is more efficient than linear search for sorted arrays.**
✓ **Java provides built-in utilities like Arrays.sort() and Arrays.binarySearch().**