Introduction to Java

Java is a high-level, object-oriented programming language developed by **Sun Microsystems** (now owned by **Oracle Corporation**) and released in 1995. It is designed to have as few implementation dependencies as possible, making it versatile and widely used for developing various types of software. Here's an overview:

Key Features of Java:

1. **Platform Independence**:
2. Java programs are compiled into **bytecode**, which can run on any system equipped with the Java Virtual Machine (**JVM**). This is often referred to as the "Write Once, Run Anywhere" (WORA) capability.
3. **Object-Oriented**:
4. Java is based on the principles of object-oriented programming (OOP), meaning it uses concepts like classes, objects, inheritance, polymorphism, encapsulation, and abstraction.

Notes on Data Types in Java: int, float, char, and double

1. int (Integer)

- **Definition**:
- The int data type is used to store whole numbers (positive or negative) without decimal points.
- **Size**:
- 4 bytes (32 bits).
- **Value Range**:
- -2,147,483,648 to 2,147,483,647

2. float (Floating-Point)

- **Definition**:
- The float data type is used to store fractional numbers (decimals) with single precision.
- **Size**:
- 4 bytes (32 bits).
- **Value Range**:
- Approximately ±3.4 * 10^38 (7 decimal digits of precision).

3. char (Character)

- **Definition**:
- The char data type is used to store a single character.
- **Size**:
- 2 bytes (16 bits) because it uses the **Unicode** character set.

4. double (Double-Precision Floating-Point)

- **Definition**:
- The double data type is used to store fractional numbers (decimals) with double precision.
- **Size**:
- 8 bytes (64 bits).
- **Value Range**:
- Approximately ±1.7 * 10^308 (15-16 decimal digits of precision).

What is a Variable?
A variable is like a container in which you can store data, such as numbers or text, and use it later in your program. Think of it like a box with a name. The type of data you store in the box (number, text, etc.) depends on the "type" of the variable.

Comments in Java
Comments in Java are used to make the code more readable and to explain what the code does. They are ignored by the compiler, so they do not affect the execution of the program.
Types of Comments in Java
Java supports three types of comments:
1. Single-Line Comments

- Start with //.
- Used for brief explanations or notes about a single line of code.

Java Workbench

https://www.programiz.com/java-programming/online-compiler/

String and method

What is a String in Java?

- A **String** in Java is a sequence of characters, like "Hello" or "Java Programming".
- Strings in Java are immutable, meaning their value cannot be changed after they are created.

Common String Methods in Java
1. length()

- **Purpose**: Finds the number of characters in a string.
- **Use Case**: Determine how long a string is.

2. trim()

- **Purpose**: Removes leading and trailing spaces from a string.
- **Use Case**: Clean up user input or data before processing.

3. toUpperCase() and toLowerCase()

- **Purpose**: Converts all characters in the string to uppercase or lowercase.
- **Use Case**: Normalize data for comparison or formatting.

4. charAt(index)

- **Purpose**: Retrieves the character at a specific position in the string (0-based index).
- **Use Case**: Access individual characters in a string.

5. substring(startIndex) and substring(startIndex, endIndex)

- **Purpose**: Extracts a portion of the string.
- **Use Case**: Get specific parts of a string, like a word or phrase.

6. replace(oldChar, newChar)

- **Purpose**: Replaces all occurrences of a character with another character.
- **Use Case**: Modify strings, like correcting typos or formatting data.

7. contains(sequence)

- **Purpose**: Checks if the string contains a specific sequence of characters.
- **Use Case**: Search for specific keywords in text.

8. startsWith(prefix) and endsWith(suffix)

- **Purpose**: Checks if the string starts or ends with a specific substring.
- **Use Case**: Validate input or identify patterns.

Java Operators and Operands

1. What are Operators and Operands?

- **Operators** are symbols that perform operations on variables and values.
- **Operands** are the values or variables that the operators act upon.

For example, in int a = 10 + 5;, + is the operator, while 10 and 5 are operands.
2. Types of Operators in Java
1. Arithmetic Operators
Used for basic mathematical operations.

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)

Modulus (%) Operator in Java
Definition:
The **modulus operator (%)** in Java returns the **remainder** after dividing one number by another.
Syntax:
a % b → This gives the remainder when a is divided by b.
Example:

- 10 % 3 = 1 (Since 10 ÷ 3 gives quotient 3 and remainder 1).
- 15 % 4 = 3 (Since 15 ÷ 4 gives quotient 3 and remainder 3).
- 20 % 5 = 0 (Since 20 ÷ 5 gives quotient 4 and remainder 0).

Example: 10 + 5 results in 15.

2. Relational (Comparison) Operators
Used to compare values, returning true or false.

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)

- <= (Less than or equal to)

Example: 10 > 5 returns true.

3. Logical Operators
Used to perform logical operations, usually in decision-making.

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

Example: (10 > 5) && (5 > 3) returns true.

Types of Logical Operators in Java
1. Logical AND (&&)

- Returns **true** if **both** conditions are true.
- Returns **false** if **any one** condition is false.

Example:

- (10 > 5) && (5 > 3) → **true** (Both conditions are true)
- (10 > 5) && (5 < 3) → **false** (One condition is false)

2. Logical OR (||)

- Returns **true** if **at least one** condition is true.
- Returns **false** if **both** conditions are false.

Example:

- (10 > 5) || (5 < 3) → **true** (One condition is true)
- (10 < 5) || (5 < 3) → **false** (Both conditions are false)

Types of Type Casting in Java

1. Implicit Type Casting (Widening)

- Automatically performed by Java when converting a **smaller data type to a larger data type**.
- No data loss occurs.
- Done automatically by the compiler.

Example:

- byte → short → int → long → float → double

2. Explicit Type Casting (Narrowing)

- Done manually by the programmer.
- Converts **a larger data type into a smaller one**.
- May result in **data loss**.
- Syntax: (targetType) value

Example:

- double → float → long → int → short → byte

Classwork question

Classwork Question on Type Casting in Java
Question:
Write a Java program that:

1. **Declares a double variable and assigns it a value of 75.95.**
2. **Performs implicit type casting to convert the double into a float, then into a long, and finally into an int.**
3. **Prints the values at each step.**
4. **Performs explicit type casting to convert the int into a short and then into a byte.**
5. **Prints the final values.**

Control Statements in Java

if Statement

- **Purpose: Executes a block of code only when a specific condition is true.**
- **Flow: If the condition is true, the code block executes. Otherwise, it is skipped.**
- **Usage: Suitable for simple conditions with a single action.**

if-else Statement

- **Purpose: Provides two paths of execution based on a boolean condition.**
- **Flow:**
- **If the condition is true, the if block executes.**
- **If the condition is false, the else block executes.**
- **Usage: Suitable for binary decisions where only two outcomes are possible.**

if-else-if Ladder

- **Purpose: Handles multiple conditions sequentially.**
- **Flow:**
- **Evaluates each condition in order.**
- **Executes the block associated with the first true condition.**
- **If no conditions are true, the else block (optional) executes.**
- **Usage: Ideal for complex decision-making scenarios with multiple conditions.**

Switch Case

- **Purpose: Handles multiple possible values for a single variable.**
- **Flow:**
- **Evaluates the expression and matches it against multiple cases.**
- **Executes the code block associated with the matching case.**
- **The default case executes if no case matches.**
- **Usage: Best used when evaluating discrete values, such as enums, characters, or integers.**

Java for Loop

- **Explanation: What is a for loop? A for loop in Java is a control structure used to repeat a block of code a specific number of times. It is most useful when the number of repetitions is known in advance.**

  ```
  for (initialization; condition; update) {

     // code to be executed

  }
  ```

Components:

1. **Initialization – sets up a starting value (usually a counter variable).**
2. **Condition – tested before each loop; the loop runs only if this is true.**
3. **Update – modifies the loop variable after each iteration.**

Example (explained simply):

Imagine you want to print numbers from 1 to 5.

- **You start at 1.**
- **You keep printing the number while it's less than or equal to 5.**
- **After each print, you go to the next number (add 1).**

This repeats 5 times, and you see:

```
1
2
3
4
5
```

When should you use a for loop?

- **When you know exactly how many times something should happen.**
- **When you're counting through numbers.**
- **When you go through items in a list or array.**

Why is it useful?
It saves time. Instead of writing the same line of code many times, you let the loop repeat it for you. It also makes your code shorter and easier to read.
Let me know if you'd like this explained with a real-life example (like climbing stairs or filling bottles).

What is a while loop?

- A while loop is used in Java to repeat a block of code as long as a condition is true. You use it when you don't always know how many times something will repeat, but you know the condition that should keep it going.

  How does it work?

1. Java checks the condition.
2. If the condition is true, the code inside the loop runs.
3. After the code runs, it goes back and checks the condition again.
4. This repeats until the condition becomes false.

If the condition is false from the start, the code inside the loop will never run.
Real-life example:
Imagine you're filling a glass of water.

- While the glass is not full, keep pouring.
- Once it's full, stop.

The loop continues only while the condition (glass not full) is true.
When to use a while loop?

- When you don't know exactly how many times the code needs to run.
- When you want to keep checking a condition after each step.
- When you're waiting for something to happen (like user input or a specific event).

Introduction to Break and Continue in Java

- Break Statement – Definition and Usage
- The break statement is used when you want to stop a loop completely before it normally ends.
- This is helpful when you're looping through data and want to exit early once a condition is true.

Example:
Imagine you're checking numbers from 1 to 10, but want to stop when you reach 5.
As soon as the number is 5, you use break, and the loop stops.
So it would print:
1
2
3
4
Then it stops.
Continue Statement – Definition and Usage
The continue statement is used when you want to skip one turn of the loop, but still continue with the rest.
Instead of ending the loop, it just jumps to the next round.
Example:
If you're printing numbers from 1 to 5 and want to skip number 3, you use continue when the number is 3.
It would print:
1
2
(skip 3)
4
5
The loop keeps going, but 3 is skipped.
Differences Between Break and Continue (Explained Simply)

- **break stops the entire loop. Once it runs, nothing else in the loop will happen.**
- **continue skips only one time in the loop. The loop will still keep running afterward.**

Think of a break like pressing the stop button.
Think of continue like saying, "Skip this o
ne, move to the next."
Infinite Loops in Java
What is an Infinite Loop?
An infinite loop is a loop that never ends because its condition is always true, or there's nothing inside the loop to stop it.
The program keeps repeating forever unless you close it manually or stop it with break.
Types of Infinite Loops

1. **Using while(true)**
2. **This is a loop where the condition is always true, so it never stops by itself.**
3. **It can be useful when you're waiting for something, like user input, but you must use break to exit it at some point.**
4. **Missing update in the loop**
5. **Sometimes, you forget to change the variable inside the loop.**
6. **For example, if you're using i = 1 and checking i <= 5 but never increasing i, the loop runs forever.**
7. **Wrong condition**
8. **If your condition is written in a way that it never becomes false, the loop will never stop.**

How to Handle Infinite Loops

- **Always check your loop condition carefully.**
- **Make sure the variable in the condition is being changed.**
- **Use break if you want to exit based on some rule.**
- **Use print statements to debug what's happening inside the loop.**
- **If your program gets stuck, use your IDE's stop button or press Ctrl + C in the console to stop it.**

Arrays in Java

An array is a data structure in Java that stores a fixed-size sequence of elements of the same data type.

Declaring and Initializing an Array

You can declare an array by specifying the data type and square brackets. Initialization can be done using the new keyword.

Example:
int[] numbers = new int[5];
String[] names = {"Alice", "Bob", "Charlie"};

Accessing Array Elements

Array elements are accessed using their index, starting from 0.
Example:
To access the first element: numbers[0] = 10;
To print it: System.out.println(numbers[0]);

Example: Working with Arrays

Declare an array of marks:
int[] marks = {80, 90, 75};
You can loop through it to calculate the average or print each mark.

Types of Arrays in Java

1. **One-Dimensional Arrays**
2. **Two-Dimensional Arrays**

3. **Multi-Dimensional Arrays**

One-Dimensional Arrays Example

A single row of elements:
int[] ages = {20, 21, 22, 23};
You can use a loop to access each age.

Two-Dimensional Arrays Example

Like a table with rows and columns:
int[][] matrix = {{1, 2}, {3, 4}};
To access value at the first row, second column: matrix[0][1] gives 2.

Multi-Dimensional Arrays

Arrays with more than two dimensions. Less common, but useful in advanced scenarios such as 3D space.
Example:
int[][][] cube = new int[3][3][3];
Each element is accessed with three indexes.

Introduction to ArrayList in Java

ArrayList is a class in Java from the java.util package that provides a dynamic array which grows as needed. Unlike arrays, ArrayList can change size at runtime.

Working with ArrayList

You must import java.util.ArrayList to use it. ArrayList stores objects, not primitive types directly.
Example:
ArrayList<String> cities = new ArrayList<>();

ArrayList Syntax

To create an ArrayList:
ArrayList<DataType> name = new ArrayList<>();
Example:
ArrayList<Integer> list = new ArrayList<>();

ArrayList Methods and Operations

Common methods include:

- **add(value) – adds an element**
- **get(index) – retrieves an element**
- **set(index, value) – updates an element**
- **remove(index) – removes an element**
- **size() – returns the number of elements**
- **clear() – removes all elements**
- **contains(value) – checks if a value exists**

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming is a programming paradigm based on the concept of "objects." Objects represent real-world entities and contain both data (fields) and behavior (methods). OOP focuses on four main principles: encapsulation, inheritance, polymorphism, and abstraction.

Classes and Objects in Java

A class is a blueprint for creating objects. It defines properties and behaviors that the objects created from it will have. An object is an instance of a class. When you create an object, you allocate memory and can use the methods and fields defined in the class.

Example – Classes and Objects

For example, consider a Car class with properties like color and model, and methods like start() and stop(). When you create a specific Car object like myCar, it can have values assigned to its properties and can call its methods.

Example – Working with Arrays

You can also use arrays of objects in Java. For example, an array of Student objects can store multiple student records. Each element in the array represents a different instance of the Student class.

Types of Objects

Objects in Java can be:

- **Instance Objects: Created using the new keyword from a class.**
- **Anonymous Objects: Created without storing the object reference in a variable.**
- **Singleton Objects: A design pattern where only one instance of a class is created.**
- **Immutable Objects: Objects whose state cannot be changed after creation (e.g., String in Java).**

Access Modifiers in Java

Access modifiers control the visibility of classes, methods, and variables. Java has four main types:

- **private: Accessible only within the class.**
- **default (no modifier): Accessible within the same package.**
- **protected: Accessible within the same package and by subclasses.**
- **public: Accessible from anywhere.**

Encapsulation

Encapsulation is the practice of hiding internal details of an object and only exposing what is necessary. It involves bundling data (variables) and methods that operate on that data into a single unit or class and restricting direct access to some components.

Key Characteristics of Encapsulation

- **Data hiding using private access modifier.**
- **Public methods (getters and setters) used to access and modify private fields.**
- **Improves modularity and maintainability.**
- **Increases security and control over data.**

Inheritance

Inheritance allows a class (subclass) to inherit fields and methods from another class (superclass). This promotes code reuse and establishes a parent-child relationship between classes.

Single Inheritance

In single inheritance, a subclass inherits from one superclass. This is the most common form and is supported in Java.

Multilevel Inheritance

In multilevel inheritance, a class inherits from a subclass, which in turn inherits from another class. This forms a chain of inheritance.

Hierarchical Inheritance

In hierarchical inheritance, multiple subclasses inherit from a single superclass. Each subclass can use the features of the common parent class.

Polymorphism

Polymorphism means "many forms." In Java, it allows objects to take on multiple forms depending on how they are used. It helps achieve flexibility and reusability in code.
There are two types:

Compile-time Polymorphism (Method Overloading)

Occurs when multiple methods in the same class have the same name but different parameters. The method call is resolved during compilation.
Example:

Two add() methods, one for integers and one for doubles.

Runtime Polymorphism (Method Overriding)

Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method call is resolved at runtime based on the object type.
Example:
A superclass Animal has a method sound(). Subclasses Dog and Cat override this method with specific behavior.

Types of Objects

Types of Objects

- **Instance Object: Created using the new keyword. Each has its own state.**
- **Anonymous Object: Created and used without assigning to a variable.**
- **Singleton Object: Only one instance exists in the entire application.**
- **Immutable Object: Its state cannot be changed after creation (like String in Java).**

Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the essential features to the user.
In Java, abstraction is achieved using:

1. **Abstract Classes**
2. **Interfaces**

Abstract Class

- **Cannot be instantiated directly.**
- **Can have both abstract (unimplemented) and non-abstract (implemented) methods.**
- **Used when classes share a common base but have some different implementations.**

Interface

Introduction to Instance Variables

Instance variables are variables that are declared inside a class but outside any method. They are associated with an object and store the state or properties of that object. Each object of the class has its own copy of the instance variables.

Key Features of Instance Variables

- **They are unique to each object.**
- **They are declared within a class but outside methods.**
- **They are initialized when an object is created.**
- **They can have different values for different objects of the same class.**
- **They are accessed using the object name.**

Example of Instance Variables

Imagine a Car class. The color and model of each car can be different. These properties can be stored as instance variables so each Car object maintains its own color and model.

Introduction to Constructors

A constructor is a special method in a class that is automatically called when an object is created. Its main purpose is to initialize the object's instance variables.

Default Constructor

A default constructor is a constructor that takes no parameters. It assigns default values to the instance variables.

Parameterized Constructor

A parameterized constructor takes arguments so that specific values can be passed during object creation, allowing more control over initialization.

Destructor

A destructor is a special method that is called when an object is destroyed. It is used to perform cleanup activities, like closing files or releasing resources.

Difference between Constructor and Destructor

- **A constructor initializes an object, while a destructor destroys it.**
- **A constructor is called when the object is created; a destructor is called when the object is deleted or goes out of scope.**
- **A constructor can be overloaded (have multiple versions), but a destructor cannot.**
- **A constructor usually takes parameters (if it's parameterized), while a destructor does not.**