JAVA

What is Java?

Java is a high-level, object-oriented programming language known for its platform independence (Write Once, Run Anywhere). It's used for building a wide range of applications, from enterprise systems to mobile apps.

Key Points

1. Java is case-sensitive.
2. Every statement ends with a semicolon (;).
3. Comments are used to explain code (// for single-line, /* */ for multi-line).
4. Indentation is not mandatory but improves readability.

Java: A Look at its Past, Present, and Future

Java's Origins (Early 1990s):
Developed by Sun Microsystems for consumer electronics (TVs, VCRs).
Goals: Small, fast, efficient, and portable across various devices.
Used internally at Sun but gained wider attention with HotJava.

HotJava (1994):
A web browser written in Java, showcasing its potential for web applications.
Sparked interest in Java for web development.

Java Development Kit (JDK) Beta (1995):
Provided tools for developing Java applets and applications.
Initially available for Sun Solaris and Windows platforms.
Subject to change before official release.

Challenges (Mid-1990s):
Beta status of JDK meant potential changes in future versions.
HotJava versions weren't compatible with each other.

Looking Forward (Late 1990s):
Sun expected to release a final JDK and a compatible HotJava version.
Other companies announced Java support in their web browsers (e.g., Netscape).
Increased developer tools and support anticipated.

What is a Variable?

1. Think of a variable as a named container that holds a value.
2. You can store different types of data in these containers, and you can use the name of the container to access and manipulate the data.

Naming VariablesRules:Must

1. start with a letter or underscore.
2. Can only contain letters, numbers, and underscores.Case-sensitive (age, Age, and AGE are different).
3. Cannot be a reserved word (like if, else, class, etc.).

Best practices:

a. Use meaningful names (e.g., age, firstName, totalScore).
b. Start variable names with lowercase letters.
c. Use camel case for multi-word names (e.g., customerName).

Understanding Data Types and Literals in Java

Java is a **strongly typed language**, meaning you must specify the data type of a variable before using it. This helps the compiler catch potential errors early on.

Primitive Data Types:

byte: Stores whole numbers from -128 to 127.

short: Stores whole numbers from -32768 to 32767.

int: Stores whole numbers within a larger range than short.

long: Stores very large whole numbers.

float: Stores single-precision floating-point numbers (numbers with decimal points).

double: Stores double-precision floating-point numbers (more precise than float).

char: Stores a single character (like 'a', 'B', '$').

boolean: Stores either true or false.

Literals
Literals are the actual values assigned to variables.

Integer literals: Numbers without decimal points (e.g., 42, -10).

Floating-point literals: Numbers with decimal points (e.g., 3.14, -0.5).

Character literals: Single characters enclosed in single quotes (e.g., 'a', '$').

String literals: Sequences of characters enclosed in double quotes (e.g., "Hello").

Boolean literals: true or false.

Key Points
- Choose the appropriate data type based on the range of values you need to store.
- Use meaningful variable names to improve code readability.
- Be aware of the difference between integer and floating-point numbers.
- Understand the concept of literals as constant values

Example:-

```
int age = 30; // Integer literal 30 assigned to an int variable
double pi = 3.14159; // Floating-point literal assigned to a double variable
char firstLetter = 'A'; // Character literal assigned to a char variable
boolean isAdult = true; // Boolean literal assigned to a boolean variable
```

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

Class: Car
Think of a car as a class. It's a blueprint or template that defines what a car is. It includes properties like color, make, model, and behaviors like starting, stopping, accelerating.
Object: Your Car
Your specific car is an object of the Car class. It has its own unique color, make, and model. You can perform actions on it like starting the engine or applying brakes.
Instance Variables:
The color, make, and model of your car are instance variables. Each car will have its own set of these values.
Methods:
Starting the car, accelerating, braking, and turning are methods. These are the actions or behaviors a car can perform.
To summarize:
Car class: Defines the general characteristics and behaviors of a car.
Your car: A specific instance of the Car class with its own details.
Instance variables: Car's specific attributes (color, make, model).
Methods: Car's actions (start, stop, accelerate)

Basic Syntax:

Case Sensitivity:

Java is case sensitive.

Example: Hello and hello are different identifiers.

```
8
9  public class Main
10 {
11     public static void main(String[] args) {
12         System.out.println("Hello World");
13     }
14 }
```

```
Hello World
```

Class Names:

The first letter of class names should be in Upper Case.

If a class name consists of multiple words, each word's first letter should be in Upper Case.

Example: MyFirstJavaClass.

Method Names:

Method names should start with a Lower Case letter.

If a method name consists of multiple words, each inner word's first letter should be in Upper Case.

Example: public void myMethodName().

Program File Name:

The program file name should exactly match the class name.

Save the file with the class name and append .java to the end.

Example: If the class name is MyFirstJavaProgram, the file should be saved as MyFirstJavaProgram.java.

Main Method:

Java program processing starts from the main() method.

The main() method is mandatory in every Java program.

Example: public static void main(String args[]).

Strings, Objects, and Enums:

String: A sequence of characters (text) and is an object.

Object: A thing that holds data and methods to manipulate that data. Objects are created from classes.

Class: A blueprint for creating objects.

Enum: A special type of class that defines a set of constants (fixed values).

Subroutines:

Subroutine: A set of instructions that perform a task. Also known as a method.

Static Subroutine: A subroutine that belongs to the class itself, not to objects created from the class.

Operator in Java

Other Important Points

Operator Precedence: Java follows a specific order for evaluating expressions. For example, multiplication and division have higher precedence than addition and subtraction.

Unary Operators: Increment and Decrement:

Java also provides ++ (increment) and -- (decrement) operators:

x++ increments x by 1 after the current expression is evaluated.

++x increments x by 1 before the current expression is evaluated.

x-- decrements x by 1 after the current expression is evaluated.

--x decrements x by 1 before the current expression is evaluated.

Strings as Objects

In Java, strings are treated as objects. This means they have properties and methods associated with them.

Properties: The sequence of characters that make up the string.

Methods: Built-in functions to manipulate and extract information from the string.

Common String Methods

Here's a breakdown of some essential string methods:

- **length():** Returns the number of characters in the string.
- **charAt(index):** Returns the character at the specified index (starting from 0).
- **substring(beginIndex, endIndex):** Extracts a substring from the given indices.
- **indexOf(char):** Returns the index of the first occurrence of the specified character.
- **equals(String other):** Compares two strings for equality (case-sensitive).
- **equalsIgnoreCase(String other):** Compares two strings for equality, ignoring case.
- **toUpperCase():** Converts all characters to uppercase.
- **toLowerCase():** Converts all characters to lowercase.
- **trim():** Removes leading and trailing whitespace.
- **concat(String str):** Concatenates two strings.

String Arithmetic in Java

In Java, you can use the + operator to perform string concatenation, which means joining multiple strings or other types of data into one string. Here's a breakdown of how this works:

String Concatenation with + Operator

When you use the + operator with strings, it concatenates the strings together. If you include non-string data types in the concatenation, Java automatically converts them to their string representation. This is a powerful feature for creating and formatting output.

```
String name = "Alice";
String color = "blue";

// Concatenating strings with variables
System.out.println(name + " is a " + color + " beetle");
// Output: Alice is a blue beetle
```

Automatic Conversion

If you concatenate a string with a non-string type (like an integer or a double), Java converts the non-string type to its string representation. For instance:

```
int age = 25;
System.out.println("Age: " + age);     // Output: Age: 25
```

String Immutability

An important concept is that strings are **immutable** in Java. This means once a string is created, its contents cannot be changed. Any operation that appears to modify a string actually creates a new string. For instance, when you use toUpperCase(), a new string with uppercase characters is created, leaving the original string unchanged.

What is an Enum?

An enum (enumeration) is a special data type that represents a set of named constants. It's a way to define a variable with a limited set of possible values.

Why Use Enums?

- **Readability:** Enums make code more readable and maintainable by providing descriptive names for constants.
- **Type Safety:** Prevents accidental assignment of incorrect values to variables.
- **Efficiency:** Enums can be more efficient than using integer constants.

```
enum Season { SPRING, SUMMER, FALL, WINTER }
```

Key Points

- Enum constants are implicitly public, static, and final.
- You can use ordinal() method to get the index of an enum constant (starting from 0).
- Enums can be used in switch statements.
- Enums are internally implemented as classes.

Summary of Text Input and Output in Java

This section discusses text input and output functionalities in Java, with a focus on the TextIO class for non-GUI programs.

Key Points:

- Java's built-in support for input is limited (System.in).
- TextIO class (not part of standard Java) simplifies input/output for non-GUI programs.
- TextIO provides functions for reading various data types (int, double, boolean, char, String).
- getln functions read an entire line of input.
- get functions read a value and leave remaining data in the input buffer for later reading.
- TextIO offers additional functionalities like getAnyChar, peek, for advanced input control.

Benefits of TextIO:

- Easier input compared to using System.in directly.
- More control over input behavior.
- Can be used alongside System.out for output.

Limitations:

- Requires the TextIO.java file to be available to the program.
- Not part of the standard Java library.

Example:

The provided code snippet demonstrates how to use TextIO to read investment amount and interest rate from the user and calculate the final investment value.

Formatted Output with printf

Java provides the printf method for formatted output, offering greater control over how data is displayed.

Key Points:

- **printf method:** Used for formatted output.
- **Format string:** Specifies the format of the output.
- **Format specifiers:** Indicate how values should be formatted.

Width: Specifies the minimum field width.%10d: Right-justifies an integer in a field of 10 characters.
Precision: For floating-point numbers, specifies the number of digits after the decimal point.%.2f: Displays two decimal places.
Flags: Modify the output format (e.g., - for left-justification, 0 for padding with zeros).

Additional Notes

The TextIO.putf method provides similar functionality to System.out.printf.
You can use multiple format specifiers in a single format string.
For more complex formatting, explore the java.util.Formatter class.
By understanding these concepts, you can create well-formatted and readable output in your Java programs.

TextIO for File Handling

File Output:

TextIO.writeFile("filename.txt"): Redirects output to a file.
TextIO.writeUserSelectedFile(): Allows user to select the output file.
TextIO.writeStandardOutput(): Restores output to the console.

File Input:

TextIO.readFile("filename.txt"): Reads input from a file.
TextIO.readUserSelectedFile(): Allows user to select the input file.
TextIO.readStandardInput(): Restores input to the console.

Important Considerations:

- Overwriting Files: Be cautious when writing to a file as existing content will be erased.
- Error Handling: Handle potential file-related errors (e.g., file not found).
- Efficiency: For large files, consider using more efficient file I/O methods.

Example:

- The provided code demonstrates how to:
- Gather user information.
- Create a profile file.
- Write user data to the file.

Additional Notes

TextIO is a simplified approach to file I/O.
For more complex file operations, you'll need to use Java's built-in File and FileReader/FileWriter classes.
Always close files after use to release system resources.

By using TextIO, you can easily handle file input and output tasks in your Java programs.

Arrays, Conditionals, and Loops

Arrays in Java: A Breakdown

This passage dives into arrays, a fundamental building block for data storage in Java programs.

Key Points:

Arrays: Ordered collections of elements of the same type.

Declaration:

Specify the type and size (number of elements) using [].

Example: int[] numbers = new int[10]; (array of 10 integers)

Initialization:

Allocate memory with new and specify size.

Initialize elements directly within curly braces.

Example: String[] colors = {"red", "green", "blue"};

Accessing Elements:

Use the array variable and subscript (index) starting from 0.

Example: int firstNumber = numbers[0]; (gets the first element)

Length:

Use the length property to find the number of elements.

Example: int arrayLength = numbers.length;

Arrays in Java

An array is a collection of elements of the same data type. It's a fundamental data structure used to store and organize data efficiently.

Declaring an Array

To declare an array, you specify the data type followed by square brackets []:

```
int[] numbers; // Declares an array of integers
String[] names; // Declares an array of strings
```

Creating an Array

To create an array, you use the new keyword and specify the size of the array

```
numbers = new int[5]; // Creates an array of 5 integers
names = new String[3]; // Creates an array of 3 strings
```

Accessing Array Elements

Array elements are accessed using an index, which starts from 0.

```
numbers[0] = 10; // Assigns 10 to the first element
String name1 = names[1]; // Accesses the second element
```

- Array size is fixed once created.
- Array indices start from 0.
- Accessing an index out of bounds will result in an ArrayIndexOutOfBoundsException.
- Arrays can store both primitive data types and objects.

Important Notes:

Array size is fixed upon creation.

Subscript out-of-bounds access leads to errors.

Arrays hold references to objects, not the objects themselves (for object arrays).

Multidimensional Arrays (Not Supported Directly):

Achieve similar functionality by creating arrays of arrays.

Additional Considerations:

Java offers more advanced data structures like ArrayList for dynamic resizing.

Understanding arrays is crucial for many Java programming concepts.

Understanding Block Statements in Java

A block in Java is a group of statements enclosed within curly braces {}. It's essentially a way to group multiple statements into a single unit.

Purpose of Blocks

- **Grouping related statements:** Encapsulates logically connected code.
- **Creating a new scope:** Variables declared within a block are only accessible within that block (local scope).

Variables declared within a block are not accessible outside that block.
Blocks can be nested.
Blocks are used extensively in control flow statements (if, else, for, while, etc.).

Common Use Cases

- **Conditional statements:** The if and else blocks.
- **Loops:** The body of for, while, and do-while loops.
- **Methods:** The body of a method is a block.
- **Classes:** The body of a class is a block.

If Statements

Purpose: Execute code based on a boolean condition.

Syntax:

if (condition) {
// Code to execute if condition is true}

```java
switch (variable) {
    case value1:
        // Code to execute if variable == value1
        break;
    case value2:
        // Code to execute if variable == value2
        break;
    case value3:
        // Code to execute if variable == value3
        break;
    // More cases as needed
    default:
        // Code to execute if no cases match
}
```

switch Statements in Java

A switch statement is a control structure in Java that simplifies handling multiple potential values for a variable. It allows you to test a variable against a list of values, and execute different code based on which value matches. This can be more readable and efficient than using multiple if-else statements for many possible values.

Loops in Java

Understanding For Loops in Java

Purpose

- Execute a block of code repeatedly until a condition becomes false.
- Often used for iterating through a specific number of times.
- Can be adapted for various looping scenarios.

```
for (initialization; test; update) {
  // statements to execute (loop body)
}
```

initialization: Typically, a variable declaration or assignment to initialize a loop counter.

test: A boolean expression that determines if the loop continues.

update: An expression that modifies the loop counter, often an increment or decrement.

loop body: The statements to be executed repeatedly within the loop.

```java
int sum = 0;
for (int i = 1; i <= 5; i++) {
  sum += i; // Add current value of i to the sum
}
System.out.println("The sum of 1 to 5 is: " + sum); // Output: 15
```

- int sum = 0; initializes a variable sum to store the accumulated value.
- for (int i = 1; i <= 5; i++) defines the loop:int i = 1; initializes the loop counter i to 1.
- i <= 5; is the condition that checks if i is less than or equal to 5 (true in the first iteration).
- i++ increments i by 1 after each iteration.
- Inside the loop body: sum += i; adds the current value of i to the sum.
- The loop iterates 5 times (1 + 2 + 3 + 4 + 5).Finally, System.out.println(...) prints the calculated sum.

Key Points

- Curly braces are optional for single-line loop bodies, but recommended for readability.
- initialization, test, or update can be empty statements (;), but use them cautiously.
- For loops are versatile and can be adapted for various looping needs.

While and Do-While Loops in Java

While and do-while loops are control flow statements used for repeated execution of code blocks. Understanding their differences helps you choose the appropriate loop for your needs.

```
while (condition) {
    // statements to execute (loop body)
}
```

- The condition is a boolean expression that determines loop continuation.
- If condition is true, the loop body executes . After each iteration, the condition is re-evaluated.
- The loop repeats as long as condition remains true.

```
int count = 1;
while (count <= 5) {
    System.out.println("Count: " + count);
    count++;
}
```

Do-While Loop

```
do {
    // statements to execute (loop body)
} while (condition);
```

- The loop body executes at least once, regardless of the initial condition value.
- After the body executes, the condition is checked.
- If condition is true, the loop repeats.

Choosing Between While and Do-While

- Use a while loop when you want the loop to continue as long as a condition is true, but the condition might not necessarily be true initially.
- Use a do-while loop when you want the loop body to execute at least once, regardless of the initial condition. This is useful for user input validation or initialization routines.

❖ Both while and do-while loops can be used with break and continue statements to control loop execution.

❖ For loops are often preferred for simple iteration due to their concise syntax

Break and Continue Statements in Java

Break Statement

Purpose: Terminates the execution of a loop prematurely.

Behavior:

- Exits the innermost loop.
- Control transfers to the statement immediately after the loop.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
```

This code will print numbers from 0 to 4, and then the loop will be terminated when i reaches 5.

Continue Statement

Purpose:
Skips the remaining part of the current loop iteration and jumps to the next iteration.

Behavior:

- The loop doesn't terminate.
- Execution continues with the next iteration.

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println(i);
}
```

This code will print numbers from 0 to 2 and 4, skipping 3.

- break and continue can be used in for, while, and do-while loops.break is used to exit the loop entirely.
- continue is used to skip the current iteration and move to the next one.
- These statements can improve code efficiency and readability in certain scenarios.

Labeled Loops in Java

Labeled loops provide a way to control the flow of nested loops more precisely using break and continue statements.

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

label:- Any valid identifier, You can also use labels with while and do-while loops.

Break and Continue with Labels

- break label: Exits the loop with the specified label.
- continue label: Skips the remaining code in the current iteration of the loop with the specified label and jumps to the next iteration.

```
outer:
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 5; j++) {
    if (i * j == 8) {
      break outer;  // Exit both outer and inner loops
    }
    System.out.println("i: " + i + ", j: " + j);
  }
}

System.out.println("End of loops");
```

```
i: 0, j: 0
i: 0, j: 1
i: 0, j: 2
i: 0, j: 3
```

Continuing an Outer Loop

```
outer:
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 5; j++) {
    if (j == 2) {
      continue outer;  // Skip remaining code in outer loop and restart
    }
    System.out.println("i: " + i + ", j: " + j);
  }
  System.out.println("Outer loop iteration: " + i);
}

System.out.println("End of loops");
```

```
i: 0, j: 0
i: 0, j: 1
Outer loop iteration: 0
i: 1, j: 0
i: 1, j: 1
Outer loop iteration: 1
i: 2, j: 0
i: 2, j: 1
Outer loop iteration: 2
End of loops
```

When j becomes 2, the continue outer statement skips the remaining code in the outer loop and jumps to the next iteration.

Key Points

- Labeled loops offer more control over loop execution.
- Use them sparingly to avoid making code harder to read.
- Consider refactoring nested loops if labeled loops become too complex.

Methods in Java

A method in Java is a block of code that performs a specific task. It's essentially a function that encapsulates a set of instructions.

```
public static return_type method_name(parameters) {
    // method body
}
```

- **public**: Access modifier (can be replaced by other modifiers like private, protected, etc.)

  static: Keyword indicating the method can be called without creating an object of the class.
- **return_type**: The data type of the value returned by the method. If the method doesn't return a value, use void.
- **method_name**: The name of the method, following Java naming conventions.
- **parameters**: Optional list of variables passed to the method.
- **method body**: The code that defines the method's behavior

```
        int sum = a + b;
        return sum;
    }

    public static void main(String[] args) {
        int num1 = 5;
        int num2
 = 10;
        int result = addNumbers(num1, num2);
        System.out.println("Sum
 is: " + result);
    }
}
```

Key Points

- Methods promote code reusability.
- They improve code organization and readability.
- Methods can take parameters and return values.
- Methods are defined within classes.

Types of Methods

Static methods: Belong to the class itself, not to specific objects.

Instance methods: Belong to objects of a class.

Constructor methods: Special methods used to initialize objects.

Abstract methods: Declared without implementation, used in abstract classes and interfaces.

Additional Notes

- Methods can be overloaded, meaning multiple methods can have the same name but different parameters.
- Recursive methods can call themselves.
- Java provides many built-in methods in its standard libraries.

Object-Oriented Programming and Java

Objects and Classes

Objects are the fundamental building blocks of OOP. They represent real-world entities with their own properties (attributes) and behaviors (methods). For example, a Car object might have attributes like color, model, year, and methods like start, stop, accelerate.

Classes are blueprints for creating objects. They define the structure and behavior of objects. A Car class would define the attributes and methods common to all cars.

Encapsulation

Encapsulation is the bundling of data (attributes) and methods that operate on that data within a single unit (an object). This protects data from accidental modification and improves code organization.

Inheritance

Inheritance allows you to create new classes (subclasses) based on existing classes (superclasses). Subclasses inherit the attributes and methods of the superclass, and can add or modify them. This promotes code reusability and hierarchical relationships between classes.

Key Terms

Superclass (Parent class): The existing class from which properties and behaviors are inherited.

Subclass (Child class): The new class that inherits from the superclass.

Inheritance: The mechanism by which a subclass acquires the characteristics of a superclass.

How Inheritance Works

- A subclass implicitly contains all the non-private members (fields and methods) of its superclass.
- Subclasses can add new members and override existing methods from the superclass.
- The extends keyword is used to establish an inheritance relationship

Benefits of Inheritance

Code Reusability: Avoids duplicate code by inheriting common properties and methods.

Hierarchical Classification: Organizes classes into a logical structure.

Polymorphism: Enables treating objects of different types as if they were of the same type.

Additional Considerations

Single Inheritance: Java supports single inheritance, meaning a class can extend only one direct superclass.

Multilevel Inheritance: Inheritance can be chained, creating a hierarchy of classes.

Hierarchical Inheritance: Multiple subclasses can inherit from a single superclass.

Overriding: Subclasses can modify the implementation of inherited methods.

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

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

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();  // Inherited from Animal
        dog.bark();        // Specific to Dog
    }
}
```

Explanation:

- The `Animal` class defines a general `makeSound()` method.
- The `Dog` class extends `Animal`, inheriting the `makeSound()` method.
- The `Dog` class also has its own specific method, `bark()`.
- In the `main` method, a `Dog` object is created.
- We can call both the inherited `makeSound()` method and the specific `bark()` method on the `Dog` object.

Key Points:

- The `Dog` class inherits the `makeSound()` method from the `Animal` class.
- This demonstrates the concept of code reusability and hierarchical relationships in OOP.
- This example provides a basic understanding of inheritance in Java.

Inheritance and Methods

Method Overriding:

- A subclass can provide a specific implementation for a method that is already defined in its superclass.
- This is achieved by using the @Override annotation (optional but recommended).
- The overridden method must have the same signature (name, parameters, and return type) as the superclass method.

Method Overloading:

- This is not directly related to inheritance but is often mentioned in the context of methods in subclasses.
- It involves having multiple methods with the same name but different parameters in the same class.

Method Resolution:

- When a method is called on a subclass object, Java follows a process called *dynamic binding* to determine the appropriate method to execute.
- It starts by searching for the method in the subclass.
- If not found, it looks in the superclass and continues up the inheritance hierarchy until the method is found.

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

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");

    }
}
```

Polymorphism

Polymorphism means "many forms." It allows objects of different types to be treated as if they were of the same type. There are two main types of polymorphism:

- **Compile-time polymorphism (Overloading):** Different methods with the same name but different parameters.
- **Runtime polymorphism (Overriding):** Subclasses providing their own implementation of methods inherited from the superclass.

Abstraction

Abstraction focuses on essential features while hiding unnecessary details. It simplifies complex systems by providing a higher-level view. Abstract classes and interfaces are key concepts in abstraction.

Abstract Classes: Can have both abstract and concrete methods. Cannot be instantiated directly.

Interfaces: Define a contract of methods without implementation, used for achieving abstraction and polymorphism.

Objects are the concrete realizations of classes.
Classes provide the framework for creating objects.A class can have many instances (objects).Objects interact with each other through methods.

Working with Objects

a