1. JAVA FUNDAMENTALS REFRESHER

• VARIABLES & DATA TYPES

○ What is a Variable?

A variable is a container with a specific name that holds data of a particular type. It allows programmers to store, modify, and retrieve data efficiently.

○ Declaration of Variables

Before using a variable, you must declare it by specifying its type and name:

```
int number;
double price;
char grade;
String message;
```

○ Initialization

Assigning an initial value to a variable:

```
int number = 10;
double price = 99.99;
char grade = 'A';
String message = "Hello!";
```

○ Data Types in Java

Variables can hold different types of data. Java has two main categories:

▪ Primitive Data Types

- **`byte`**: 8-bit signed integer (-128 to 127)
- **`short`**: 16-bit signed integer
- **`int`**: 32-bit signed integer
- **`long`**: 64-bit signed integer
- **`float`**: 32-bit floating-point
- **`double`**: 64-bit floating-point
- **`char`**: Single 16-bit Unicode character
- **`boolean`**: `true` or `false`

▪ Reference Data Types

- **`String`**: Represents text
- Custom classes and objects

○ Variable Scope

Scope defines where a variable can be accessed:

- **Local variables**: Declared inside methods, constructors, or blocks; accessible only within that block.
- **Instance variables**: Declared inside a class but outside any method; associated with an object.
- **Class variables (static variables)**: Declared with the **`static`** keyword; shared among all objects of the class.

○ Variable Naming Rules

- Must start with a letter, dollar sign `$`, or underscore `_`.
- Subsequent characters can be letters, digits, `$`, or `_`.
- Cannot be a Java reserved keyword.
- Should be meaningful and follow naming conventions (`camelCase` for variables).

- Example

```
public class VariableExample {
    public static void main(String[] args) {
        int age = 50; // local variable
        double salary = 54321.50; // local variable
        boolean isEmployed = true; // local variable
        String name = "Dilip Malani"; // local variable

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
        System.out.println("Employed: " + isEmployed);
    }
}
```

- Constants

  Variables declared as `final` cannot be changed after initialization:

```
final double PI = 3.14159;
```

• OPERATORS

- Types of Operators in Java

  Java operators can be broadly classified into several categories:

  - Arithmetic Operators

    Used for basic math calculations:

    - + (Addition): Adds two operands.
    - − (Subtraction): Subtracts second operand from first.
    - * (Multiplication): Multiplies two operands.
    - / (Division): Divides numerator by denominator.
    - % (Modulus): Returns remainder of division.

```
int firstNumber = 15;
int secondNumber = 12;
System.out.println(firstNumber + secondNumber); // 27
System.out.println(firstNumber % secondNumber); // 3
```

  - Relational (Comparison) Operators

    Compare two values:

    - == (Equal): Checks if two values are equal.
    - != (Not equal): Checks if two values are not equal.
    - > (Greater than)
    - < (Less than)
    - >= (Greater than or equal to)
    - <= (Less than or equal to)

```
int firstNumber = 15, secondNumber = 25;
System.out.println(firstNumber < secondNumber); // true
```

  - Logical Operators

    Combine multiple boolean expressions:

    - && (Logical AND): True if both expressions are `true`.
    - || (Logical OR): True if at least one expression is `true`.

```
boolean firstValue = true, secondValue = false;
System.out.println(firstValue && secondValue); // false
System.out.println(!firstValue); // false
```

- **Bitwise Operators**

  Operate at the bit level, mainly used with integers:
  - `&` (AND)
  - `|` (OR)
  - `^` (XOR)
  - `~` (Complement)
  - `<<` (Left shift)
  - `>>` (Right shift)
  - `>>>` (Unsigned right shift)

```
int firstNumber = 5; // 0101 in binary
int secondNumber = 3; // 0011 in binary
System.out.println(firstNumber & secondNumber); // 1 (0001)
```

- **Assignment Operators**

  Assign values and modify existing values:
  - `=` assigns value.
  - Compound operators like `+=`, `-=`, `*=`, `/=`, `%=` combine operation and assignment.

```
int number = 5;
number += 3; // number = number + 3; now number is 8
```

- **Unary Operators**

  Operate on a single operand:
  - `+` (Unary plus): Indicates positive value.
  - `-` (Unary minus): Negates value.
  - `++` (Increment): Increase value by 1.
  - `--` (Decrement): Decrease value by 1.
  - `!` (Logical complement): Inverts boolean value.

```
int number = 5;
System.out.println(++number); // 6 (pre-increment)
System.out.println(number--); // 6, then a becomes 5 (post-decrement)
```

- **Ternary Operator**

  Shorthand for `if-else`:
  - condition `?` expression1 `:` expression2;

```
int max = (firstNumber > secondNumber) ? firstNumber : secondNumber;
```

- **instanceof Operator**

  Checks whether object is an instance of a specific class:

```
String message = "hello";
System.out.println(message instanceof String); // true
```

- **Operator Precedence and Associativity**

  Operator precedence determines the order in which operators are evaluated in an expression. For example, multiplication has higher precedence than addition.

```
int result = 3 + 4 * 5; // 3 + (4*5) = 23
```

  Associativity defines the order for operators with the same precedence (left-to-right or right-to-left). For most operators like `+`, `-`, `*`, etc., associativity is left-to-right.

- CONDITIONAL STATEMENTS

What are Conditional Statements?

Conditional statements allow your Java program to make decisions by executing certain blocks of code only if specific conditions are true. They enable dynamic behavior based on runtime data, making programs flexible and intelligent.

Basic Concept

A **conditional statement** evaluates a boolean expression (an expression that results in `true` or `false`). Based on the evaluation, different parts of code are executed.

Common Conditional Statements in Java

Java provides several types of conditional statements:

`if` Statement

```
if (condition) {
    // code to execute if condition is true
}
```

- Checks a boolean expression.
- If the condition is `true`, executes the block inside `{}`.
- If `false`, skips the block.

```
int age = 20;
if (age >= 18) {
    System.out.println("You are an adult.");
```

`if-else` Statement

```
if (condition) {
    // code if condition is true
} else {
    // code if condition is false
}
```

- Checks a condition.
- Executes the first block if condition is `true`.
- Executes the `else` block if condition is `false`.

```
int number = 10;
if (number % 2 == 0) {
    System.out.println("Even");
} else {
    System.out.println("Odd");
}
```

`else-if` Ladder

```
if (condition1) {
    // code if condition1 is true
} else if (condition2) {
    // code if condition2 is true
} else {
    // code if all above are false
}
```

- Checks multiple conditions sequentially.
- The first `true` condition's block executes.
- If none are `true`, the `else` block executes.

```
int score = 85;
if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 75) {
    System.out.println("Grade: B");
} else {
    System.out.println("Grade: C");
}
```

- `switch` Statement

```
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code if no case matches
}
```

  - Evaluates an expression once.
  - Compares it against multiple `case` values.
  - Executes matching `case`.
  - `break` prevents fall-through; if omitted, execution continues to next case.
  - `default` executes if none of the cases match.

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

Boolean Expressions and Operators

Conditional statements rely on boolean expressions. These often involve:

- Comparison operators:
  `==`, `!=`, `>`, `<`, `>=`, `<=`
- Logical operators:
  `&&` (AND), `||` (OR), `!` (NOT)

```
if (age >= 18 && hasID) {
    // both conditions must be true
}
```

Nested Conditional Statements

You can nest `if` statements inside each other for complex decision-making:

```
if (condition1) {
    if (condition2) {
        // code
    }
}
```

- Example

```
public class ConditionalExample {
    public static void main(String[] args) {
        int score = 85;
        int age = 20;

        if (score >= 90) {
            System.out.println("Excellent score!");
        } else if (score >= 75 && age >= 18) {
            System.out.println("Good job!");
        } else {
            System.out.println("Needs Improvement");
        }
    }
}
```

• LOOPS

- What Are Loops?

Loops are control flow statements that allow you to execute a block of code repeatedly based on a condition. They are essential for tasks that require iteration, such as processing items in a list, performing calculations multiple times, or waiting for a condition to become false.

- Types of Loops in Java

Java provides three main types of loops:

- `for` Loop

```
for (initialization; condition; update) {
    // statements to execute
}
```

  - **initialization**: Usually used to initialize a counter variable.
  - **condition**: Evaluated before each iteration; if `true`, loop continues.
  - **update**: Executed after each iteration; typically updates the loop counter.

```
for (int counter = 0; counter < 5; counter++) {
    System.out.println("counter = " + counter);
}
```

This prints numbers 0 to 4.

- `while` Loop

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

  - The condition is checked before each iteration.
  - If `true`, the loop body executes; if `false`, the loop terminates.

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

- `do-while` Loop

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

- The loop body executes at least once before the condition is tested.
- After executing the body, the condition is checked; if true, loop continues.

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

Loop Control Statements

Loops can be controlled using special statements:

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

```
for (int counter = 0; counter < 10; counter++) {
    if (counter == 5) {
        break; // exit loop when counter equals 5
    }
    if (counter % 2 == 0) {
        continue; // skip even numbers
    }
    System.out.println(counter);
}
```

Nested Loops

Loops inside other loops are called nested loops. They are useful for multi-dimensional data structures like matrices.

```
for (int firstCounter = 1; firstCounter <= 3; firstCounter++) {
    for (int secondCounter = 1; secondCounter <= 3; secondCounter++) {
        System.out.println("firstCounter = " + firstCounter +
                           ", secondCounter = " + secondCounter);
    }
}
```

Infinite Loops

A loop that never terminates unless externally interrupted. Be cautious with the loop condition.

```
while (true) {
    // infinite loop
}
```