**DAY-2**

**Basic terminologies in Java Program**

**1. Class:** The class is a blueprint (plan) of the instance of a class (object). It can be defined as a logical template that share common properties and methods.

* Example1: Blueprint of the house is class.

**2.** **Object**: The object is an instance of a class. It is an entity that has behavior and state.

* Example: Dog, Cat, Monkey etc. are the object of “Animal” class.
* **Behavior:** Running on the road.

**3.** **Method**: The behavior of an object is the method.

* **Example**: The fuel indicator indicates the amount of fuel left in the car.

**4. Instance variables**: Every object has its own unique set of instance variables. The state of an object is generally created by the values that are assigned to these instance variables.

## **Steps to Implement Java Program**

Implementation of a Java application program involves the following step. They include:

1. Creating the program
2. Compiling the program
3. Running the program

**class** HelloWorld {

**public** **static** **void** main(String args[])

    {

        System.out.print("Hello, World");

    }

}

#### **1. Class Definition**

This line uses the keyword **class**to declare that a new class is being defined.

Class HelloWorld {  
 //  
 //Statements  
}

#### **2. HelloWorld**

It is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace “**{**” and the closing curly brace “**}**“.

#### **3. main Method**

In the Java programming language, every application must contain a main method. The main function(method) is the entry point of your Java application, and it’s mandatory in a Java program. whose signature in Java is:

public static void main(String[] args)

#### Explanation of the above syntax

* ***public****: So that*[*JVM*](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)*can execute the method from anywhere.*
* ***static****: The main method is to be called without an object. The modifiers are public and static can be written in either order.*
* ***void****: The main method doesn’t return anything.*
* ***main()****: Name configured in the JVM. The main method must be inside the class definition. The compiler executes the codes starting always from the main function.*
* ***String[]****: The main method accepts a single argument, i.e., an array of elements of type String.*

Like in C/C++, the main method is the entry point for your application and will subsequently invoke all the other methods required by your program.

System.out.print("Hello, World");

This line outputs the string “Hello, World” followed by a new line on the screen. Output is accomplished by the built-in print( ) method. The **System** is a predefined class that provides access to the system and **out** is the variable of type output stream connected to the console.

**Important Points**

* *The name of the class defined by the program is HelloWorld, which is the same as the name of the file (HelloWorld.java). This is not a coincidence. In Java, all codes must reside inside a class, and there is at most one public class which contains the main() method.*
* *By convention, the name of the main class (a class that contains the main method) should match the name of the file that holds the program.*
* *Every Java program must have a class definition that matches the filename (class name and file name should be same)*

# Compilation and Execution of a Java Program



### Java Compilation and Execution

#### Compilation

1. **Parse:** Converts .java files into an Abstract Syntax Tree (AST).
2. **Enter:** Adds definitions to the symbol table.
3. **Process Annotations:** Handles annotations if requested.
4. **Attribute:** Performs name resolution, type checking, and constant folding.
5. **Flow:** Conducts data flow analysis, checking assignments and reachability.
6. **Desugar:** Simplifies the AST by removing syntactic sugar.
7. **Generate:** Creates .class files with bytecode.

#### Execution

1. **Class Loader:** Loads .class files into memory. It includes:
   * **Primordial Class Loader:** Default loader embedded in JVM.
   * **Non-Primordial Class Loader:** User-defined for custom loading.

Class r = loadClass(String className, boolean resolveIt);

1. **Bytecode Verifier:** Ensures bytecode safety by checking:
   * Variables are initialized.
   * Method calls match object types.
   * Private data and method access rules.
   * Local variables are within runtime stack bounds.
   * No runtime stack overflow.
2. **Just-In-Time (JIT) Compiler:** Converts bytecode to machine code for execution, enhancing performance by reducing repeated bytecode interpretation.

# **Data Types**

Java is a statically typed and strongly typed language, meaning every piece of data has a specific type that must be defined before using it. There are two main categories of data types in Java: primitive and non-primitive.

**1. boolean**

* **Description**: The boolean data type is used to store true or false values. It is primarily used for conditional statements and control flow in Java.
* **Default Value**: false
* **Size**: While the exact size of a boolean is not explicitly defined in the Java specification and can be JVM-dependent, it is typically represented using 1 bit.
* **Example**:

boolean isTrue = true;

* **Usage**:

boolean isJavaFun = true;

if (isJavaFun) {

System.out.println("Java is fun!");

}

**2. byte**

* **Description**: The byte data type is an 8-bit signed integer. It is useful for saving memory in large arrays where memory savings are important.
* **Default Value**: 0
* **Size**: 1 byte (8 bits)
* **Range**: -128 to 127
* **Example**:

byte byteValue = 100;

* **Usage**:

byte age = 25;

System.out.println("Age: " + age);

**3. short**

* **Description**: The short data type is a 16-bit signed integer. Like byte, it can be used to save memory in large arrays.
* **Default Value**: 0
* **Size**: 2 bytes (16 bits)
* **Range**: -32,768 to 32,767
* **Example**:

short shortValue = 1000;

* **Usage**:

short distance = 150;

System.out.println("Distance: " + distance);

**4. int**

* **Description**: The int data type is a 32-bit signed integer. It is the most commonly used integer type.
* **Default Value**: 0
* **Size**: 4 bytes (32 bits)
* **Range**: -2,147,483,648 to 2,147,483,647
* **Example**:

int intValue = 100000;

* **Usage**:

int salary = 50000;

System.out.println("Salary: " + salary);

**5. long**

* **Description**: The long data type is a 64-bit signed integer. It is used when a wider range than int is needed.
* **Default Value**: 0L
* **Size**: 8 bytes (64 bits)
* **Range**: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
* **Example**:

long longValue = 100000L;

* **Usage**:

long population = 7800000000L;

System.out.println("World Population: " + population);

**6. float**

* **Description**: The float data type is a single-precision 32-bit IEEE 754 floating-point. It is used for fractional numbers with less precision compared to double.
* **Default Value**: 0.0f
* **Size**: 4 bytes (32 bits)
* **Range**: Approximately 7 decimal digits of precision
* **Example**:

float floatValue = 10.5f;

* **Usage**:

float temperature = 36.6f;

System.out.println("Temperature: " + temperature);

**7. double**

* **Description**: The double data type is a double-precision 64-bit IEEE 754 floating-point. It is used for decimal values with higher precision.
* **Default Value**: 0.0d
* **Size**: 8 bytes (64 bits)
* **Range**: Approximately 16 decimal digits of precision
* **Example**:

double doubleValue = 10.5;

* **Usage**:

double pi = 3.141592653589793;

System.out.println("Pi: " + pi);

**8. char**

* **Description**: The char data type is a single 16-bit Unicode character. It is used to store characters.
* **Default Value**: '\u0000' (null character)
* **Size**: 2 bytes (16 bits)
* **Range**: 0 to 65,535 (Unicode values)
* **Example**:

char charValue = 'A';

* **Usage**:

char grade = 'A';

System.out.println("Grade: " + grade);

**Non-Primitive / (Reference Types) / Object type Data Types :**

**Non-primitive data types** in Java are more complex types that store references (memory addresses) to the actual data rather than the data itself. These include:

1. **Strings**
   * **Description:** Strings are sequences of characters.
   * **Example:**

String s = "Hello";

String s1 = new String("World");

1. **Classes**
   * **Description:** A blueprint for creating objects. It defines properties (attributes) and behaviors (methods).
   * **Example:**

public class Car {

String color;

void drive() {

// Driving behavior

}

}

1. **Objects**
   * **Description:** Instances of classes representing real-world entities with state (attributes) and behavior (methods).
   * **Example:**

Car myCar = new Car();

myCar.color = "Red";

myCar.drive();

1. **Interfaces**
   * **Description:** A contract that a class can implement. It specifies what methods a class must have but not how they should work.
   * **Example:**

interface Animal {

void eat();

}

class Dog implements Animal {

public void eat() {

// Eating behavior

}

}

1. **Arrays**
   * **Description:** A collection of variables of the same type.
   * **Example:**

int[] numbers = {1, 2, 3, 4, 5};

int length = numbers.length; // Access array length

**Q. Difference between Primitive and Object type Data Type.**

# **Java Identifiers**

In Java, identifiers are used for identification purposes. Java Identifiers can be a class name, method name, variable name, or label.

### Example of Java Identifiers

public class Test  
{  
 public static void main(String[] args)  
 {  
 int a = 20;  
 }  
}

In the above Java code, we have 5 identifiers namely:

* **Test**: class name.
* **main**: method name.
* **String**: predefined class name.
* **args**: variable name.
* **a**: variable name.

### Java Variables

**Variables** in Java are containers that hold data values during program execution. Each variable has a specific data type that defines the type of value it can store.

#### Declaration

To declare a variable, specify the data type followed by the variable name:

int age;

String name;

#### Initialization

Variables can be initialized when declared or later in the program:

int age = 25;

name = "John";

#### Types of Variables

1. **Local Variables**
   * Declared inside a method, constructor, or block.
   * Created and destroyed within the block scope.
   * Must be initialized before use.
   * Example:

void method() {

int localVar = 10; // Local variable

}

1. **Instance Variables**
   * Declared in a class but outside any method, constructor, or block.
   * Created when an object is instantiated and destroyed when the object is destroyed.
   * Can have access specifiers (default is package-private).
   * Default values depend on the data type (e.g., null for objects, 0 for int).
   * Example:

public class MyClass {

int instanceVar; // Instance variable

}

1. **Static Variables**
   * Declared with the static keyword inside a class, but outside any method, constructor, or block.
   * Only one copy per class, shared among all instances.
   * Created at the start of program execution and destroyed at the end.
   * Default values depend on the data type (similar to instance variables).
   * Example:

public class MyClass {

static int staticVar; // Static variable

}

***Different Types of Operators***

**1. Arithmetic Operators**

Arithmetic operators perform basic mathematical operations.

* **Multiplication (\*)**: Multiplies two operands.

int result = a \* b; // result is the product of a and b

* **Division (/)**: Divides the first operand by the second.

int result = a / b; // result is the quotient of a divided by b

* **Modulo (%)**: Finds the remainder when the first operand is divided by the second.

int result = a % b; // result is the remainder of a divided by b

* **Addition (+)**: Adds two operands.

int result = a + b; // result is the sum of a and b

* **Subtraction (-)**: Subtracts the second operand from the first.

int result = a - b; // result is the difference between a and b

**2. Unary Operators**

Unary operators operate on a single operand to perform various tasks.

* **Unary minus (-)**: Negates an expression.

int result = -a; // result is the negation of a

* **Unary plus (+)**: Indicates a positive value.

int result = +a; // result is the value of a

* **Increment (++)**: Increases the value of an operand by 1. It has two forms:
  + **Post-increment (a++)**: Returns the value before incrementing.

int result = a++; // result is the value of a before incrementing

* + **Pre-increment (++a)**: Returns the value after incrementing.

int result = ++a; // result is the value of a after incrementing

* **Decrement (--)**: Decreases the value of an operand by 1. It has two forms:
  + **Post-decrement (a--)**: Returns the value before decrementing.

int result = a--; // result is the value of a before decrementing

* + **Pre-decrement (--a)**: Returns the value after decrementing.

int result = --a; // result is the value of a after decrementing

* **Logical NOT (!)**: Inverts the value of a boolean expression.

boolean result = !isTrue; // result is the logical negation of isTrue

**3. Assignment Operators**

Assignment operators assign values to variables. They also support compound operations.

* **Basic assignment (=)**: Assigns the right-hand operand to the left-hand variable.

int f = 7; // f is assigned the value 7

* **Addition assignment (+=)**: Adds the right-hand operand to the left-hand variable and assigns the result to the left-hand variable.

f += 3; // f is now 10

* **Subtraction assignment (-=)**: Subtracts the right-hand operand from the left-hand variable and assigns the result to the left-hand variable.

f -= 3; // f is now 4

* **Multiplication assignment (\*=)**: Multiplies the left-hand variable by the right-hand operand and assigns the result to the left-hand variable.

f \*= 3; // f is now 21

* **Division assignment (/=)**: Divides the left-hand variable by the right-hand operand and assigns the result to the left-hand variable.

f /= 3; // f is now 2

* **Modulo assignment (%=)**: Applies modulo operation on the left-hand variable with the right-hand operand and assigns the result to the left-hand variable.

f %= 3; // f is now 1

**4. Relational Operators**

Relational operators compare two values and return a boolean result.

* **Equal to (==)**: Checks if two operands are equal.

boolean result = (a == b); // result is true if a is equal to b

* **Not equal to (!=)**: Checks if two operands are not equal.

boolean result = (a != b); // result is true if a is not equal to b

* **Less than (<)**: Checks if the first operand is less than the second.

boolean result = (a < b); // result is true if a is less than b

* **Less than or equal to (<=)**: Checks if the first operand is less than or equal to the second.

boolean result = (a <= b); // result is true if a is less than or equal to b

* **Greater than (>)**: Checks if the first operand is greater than the second.

boolean result = (a > b); // result is true if a is greater than b

* **Greater than or equal to (>=)**: Checks if the first operand is greater than or equal to the second.

boolean result = (a >= b); // result is true if a is greater than or equal to b

**5. Logical Operators**

Logical operators are used to combine multiple boolean expressions.

* **Logical AND (&&)**: Returns true if both operands are true.

boolean result = (x && y); // result is true if both x and y are true

* **Logical OR (||)**: Returns true if at least one operand is true.

boolean result = (x || y); // result is true if either x or y is true

* **Logical NOT (!)**: Inverts the value of a boolean expression.

boolean result = !x; // result is the logical negation of x

**6. Ternary Operator**

The ternary operator is a shorthand for the if-else statement.

* **Syntax**: condition ? if true : if false

int result = (a > b) ? a : b; // result is a if a is greater than b, otherwise result is b

**7. Bitwise Operators**

Bitwise operators perform operations on individual bits of integer types.

* **Bitwise AND (&)**: Performs a bitwise AND operation.

int result = a & b; // result is a bitwise AND of a and b

* **Bitwise OR (|)**: Performs a bitwise OR operation.

int result = a | b; // result is a bitwise OR of a and b

* **Bitwise XOR (^)**: Performs a bitwise XOR operation.

int result = a ^ b; // result is a bitwise XOR of a and b

* **Bitwise Complement (~)**: Inverts all the bits of the operand.

int result = ~a; // result is the bitwise complement of a

* **Left shift (<<)**: Shifts bits to the left.

int result = a << 1; // result is a shifted left by 1 bit

* **Signed right shift (>>)**: Shifts bits to the right, preserving the sign bit.

int result = a >> 1; // result is a shifted right by 1 bit, with sign bit preserved

* **Unsigned right shift (>>>)**: Shifts bits to the right, filling with zeroes.

int result = a >>> 1; // result is a shifted right by 1 bit, with zeroes filled in

**8. Shift Operators**

Shift operators are used to shift bits left or right.

* **Left shift (<<)**: Shifts bits to the left.

int result = a << 1; // result is a shifted left by 1 bit

* **Signed right shift (>>)**: Shifts bits to the right, preserving the sign bit.

int result = a >> 1; // result is a shifted right by 1 bit, with sign bit preserved

* **Unsigned right shift (>>>)**: Shifts bits to the right, filling with zeroes.

int result = a >>> 1; // result is a shifted right by 1 bit, with zeroes filled in

s