UNIT 1

Overview of JAVA: The genesis of java, history of java, java virtual machine (JVM), java development kit (JDK), source files, jar files, compiling and running of files, byte code, platform independency, data types, literals, variables, constants, array and its types, operators, conditional and looping statements, various packages, introduction of class, objects and methods, nested and inner class, string handling, constructor and its types.

The Genesis of Java

Java is one of the most popular and widely used programming languages in the world. It is a general-purpose, object-oriented, platform-independent, and high-performance language that can run on any device that supports a Java virtual machine (JVM).

Java was developed by **James Gosling** and his team at **Sun Microsystems** in the early 1990s. Initially called *Oak*, it was designed for embedded systems and small appliances. Later renamed Java, it became popular for web and enterprise applications. Java introduced the concept of "**write once, run anywhere**" due to its platform independence. Its reliability, simplicity, security, and object-oriented features made it suitable for Internet-based applications. Java evolved rapidly and is now maintained by Oracle Corporation. It is widely used in mobile apps (especially Android), enterprise systems, embedded systems, and web applications.

History of Java

Java's history dates back to 1991 when the "Green Team" began working on a new language for digital devices like TVs and set-top boxes. In 1995, Java was officially launched as a core component of Sun Microsystems's Java platform. Its first release was Java 1.0, which allowed developers to create applets for the web. Since then, Java has undergone many updates (Java SE, EE, ME), improving performance, security, and functionality. Major versions include Java 5 (with generics), Java 8 (with lambda expressions), and newer versions up to Java 17 and beyond. Java's backward compatibility and strong community support have made it one of the most enduring languages.

Java Virtual Machine (JVM)

The **Java Virtual Machine (JVM)** is a critical component of the Java Runtime Environment (JRE). It allows Java programs to be executed on any device or operating system. The JVM loads `.class` files (compiled byte code), verifies code for safety, and executes it. It also manages memory via **garbage collection**, handles exceptions, and ensures secure execution of code. JVM is responsible for converting platform-independent bytecode into machine code that the host system can understand. Each operating system has its own JVM implementation, but all can run the same bytecode, enabling Java's cross-platform nature.

Java Development Kit (JDK)

The **Java Development Kit (JDK)** is a complete software development kit provided by Oracle (or OpenJDK). It includes everything a developer needs to write, compile, and run Java applications. Key components of JDK include the **Java Compiler (javac)**, **JVM**, **JRE**, debugging tools, and libraries. The JDK provides development tools for creating applets and applications. It also comes with tools like `javadoc`, `javap`, `jar`, and `javah`. The JDK is platform-specific and must be installed to begin Java programming. There are different JDK versions (SE, EE, ME) based on the type of development.

Source Files and Class Files

In Java, programs are written in **source files** with the `.java` extension. These contain human-readable code written by the programmer. The Java Compiler (`javac`) converts these source files into **class files** (with `.class` extension), which contain **bytecode**. Bytecode is an intermediate representation that can be executed by the JVM. A single source file can produce multiple class files if it contains multiple classes. For example, if a file named `HelloWorld.java` contains a class `HelloWorld`, compiling it generates `HelloWorld.class`, which is then executed using `java HelloWorld`.

Compiling and Running Java Files

Java code must first be compiled before it can be executed. The compilation is done using the `javac` command:

```
javac MyProgram.java
```

This generates a `MyProgram.class` bytecode file. To run the program, use:

```
java MyProgram
```

Note that you do not use the `.class` extension while running the program. Errors in the code (syntax, logic, etc.) must be resolved before successful compilation. The Java compiler checks for type errors, syntax rules, and more. Java provides a robust exception-handling mechanism to deal with runtime issues. This compile-run process is crucial to Java's secure and efficient execution model.

Byte Code

Byte code is the intermediate code generated by the Java compiler. It is a set of instructions understandable by the Java Virtual Machine (JVM), not by the physical processor. Bytecode makes Java platform-independent. Each `.class` file contains bytecode and can be executed on any system with a JVM. This abstraction allows Java programs to be "write once, run anywhere." Bytecode is more compact and efficient than source code, helping improve performance. It also enhances security since the JVM can verify and control what bytecode does during execution. Tools like `javap` can be used to inspect bytecode instructions.

Platform Independency

Java's **platform independence** means that Java code can run on any system that has a JVM, regardless of the underlying hardware or OS. Unlike compiled languages like C or C++, which produce platform-specific binaries, Java compiles code to bytecode. This bytecode can be interpreted by the JVM on any platform — Windows, Mac, Linux, etc. Thus, a Java program compiled on Windows can run on Linux without modification. This feature greatly simplifies software distribution and makes Java ideal for cross-platform development, especially in web applications and distributed systems.

Data Types

In Java, **data types** specify the kind of values a variable can hold. Java is a **strongly typed language**, which means every variable must be declared with a data type.

Java has two broad categories of data types:

1. Primitive Data Type: These are the basic building blocks that store simple values directly in memory. Primitive data store only single values and have no additional capabilities. There are **8 primitive data types** Examples of primitive data types are

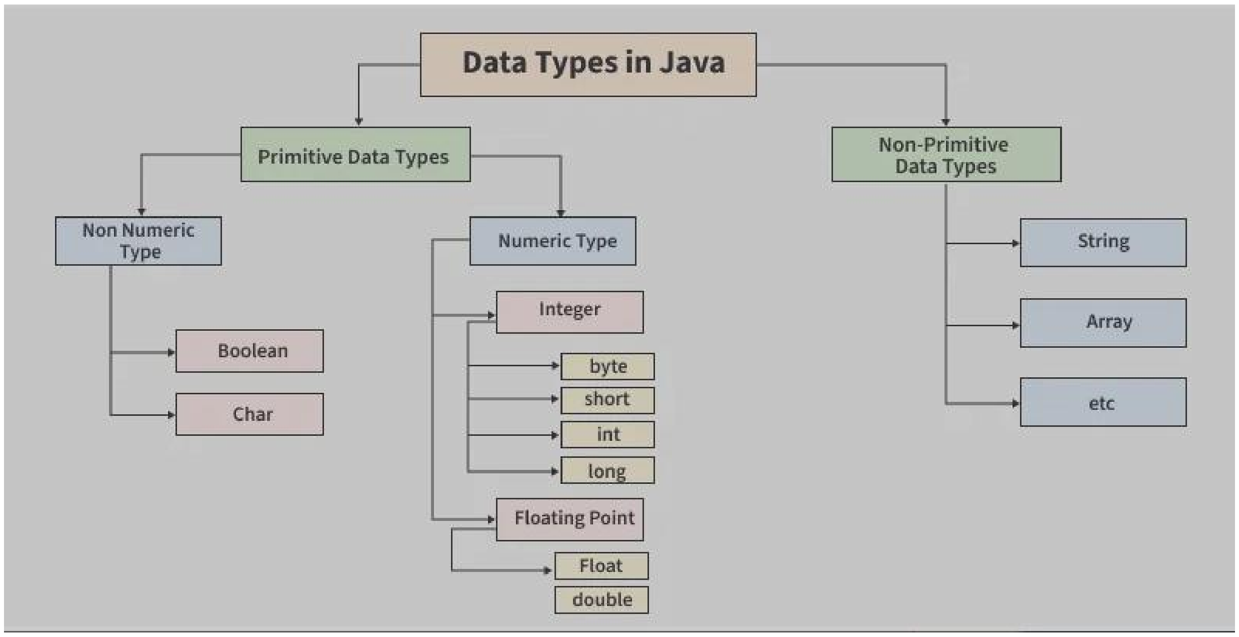
- **Boolean**
- **char**
- **byte**
- **short**
- **int**
- **long**
- **float**
- **double**

Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	JVM-dependent (typically 1 byte)	true, false	true, false
byte	8-bit signed integer	0	1 byte	(none)	-128 to 127
char	Unicode character(16 bit)	\u0000	2 bytes	'a', '\u0041', '\101', '\\', '\', '\n', 'β'	0 to 65,535 (unsigned)
short	16-bit signed integer	0	2 bytes	(none)	-32,768 to 32,767
int	32-bit signed integer	0	4 bytes	-2,0,1	-2,147,483,648 to 2,147,483,647
long	64-bit signed integer	0L	8 bytes	-2L,0L,1L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	32-bit IEEE 754 floating-point	0.0f	4 bytes	3.14f, -1.23e-10f	~6-7 significant decimal digits
double	64-bit IEEE 754 floating-point	0.0d	8 bytes		

2. Non-Primitive Data Types (Object Types): These are reference types that store memory addresses of objects. Examples of Non-primitive data types are

- **String**
- **Array**
- **Class**
- **Interface**
- **Object**

The below diagram demonstrates different types of primitive and non-primitive data types in Java.



Java Variables

In Java, variables are containers that store data in memory. Understanding variables plays a very important role as it defines how data is stored, accessed, and manipulated.

Key Components of Variables in Java:

A variable in Java has three components, which are listed below:

- **Data Type:** Defines the kind of data stored (e.g., int, String, float).

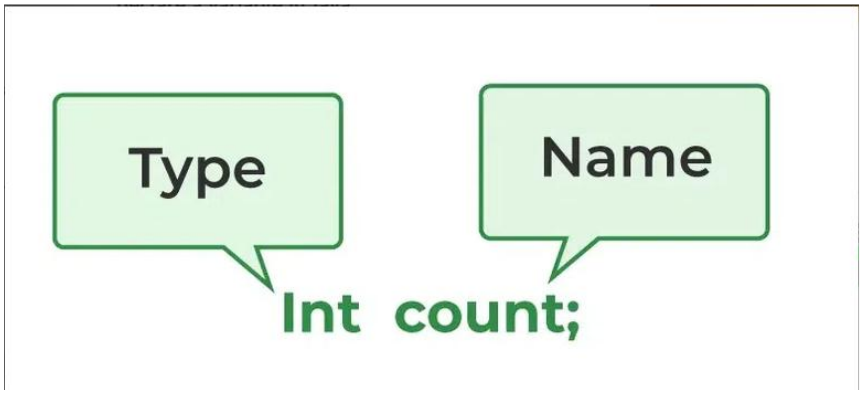
- **Variable Name:** A unique identifier following Java naming rules.
- **Value:** The actual data assigned to the variable.

Note: There are three types of variables in Java: **Local, Instance** and **Static**.

How to Declare Java Variables?

The image below demonstrates how we can declare a variable in Java:

Variable Declaration



From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

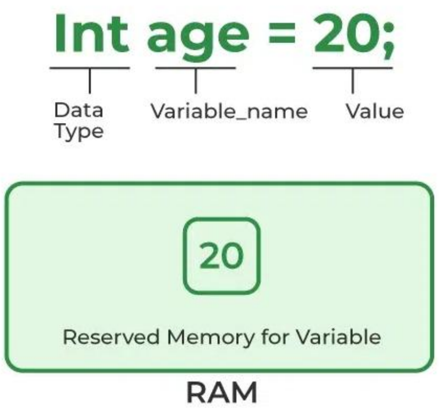
1. **data type**: In Java, a data type define the type of data that a variable can hold.
2. **variable name:** Must follow Java naming conventions (e.g., camelCase).

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input

How to Initialize Java Variables?

It can be perceived with the help of 3 components explained above:



Variable Initialization

Example:-

```
// Demonstrating how to intialize variables
// of different types in Java


class VariableDemo{
    public static void main(String[] args) {
        // Declaring and initializing variables


        // Initializing integer variables
        int t = 10;
        int s = 20;


        // Initializing character variable
        char var = 'h';
        // Displaying the values of the variables


        System.out.println("Speed: " + s);

        System.out.println("Time: " + t);

        System.out.println("Character: " + var);

    }

}
```

OUTPUT:-

Output

```
Speed: 20

Time: 10

Character: h
```

Constant

In Java, a constant is a variable whose value cannot be changed after it has been initialized. This immutability is enforced by the `final` keyword.

Key characteristics of constants in Java:

- Declaration with `final`: The `final` keyword is used in the declaration of a variable to designate it as a constant. Once a `final` variable is assigned a value, that value cannot be reassigned.

  Java
  ```
  final int MAX_VALUE = 100;
  final String GREETING = "Hello";
  ```

Literal

In Java, a literal is a fixed value that is directly represented in the source code. It is a constant value that cannot be changed during program execution. Literals are used to assign values to variables, specify values in expressions, and define constant values within a program.

There are several types of literals in Java:

- **Integer Literals:**

Represent whole numbers. They can be expressed in:

  - **Decimal (base 10):** Standard numbers (e.g., 10, 123).
  - **Octal (base 8):** Prefixed with 0 (e.g., 012 which is 10 in decimal).
  - **Hexadecimal (base 16):** Prefixed with 0x or 0X (e.g., 0xFF, 0x1A).
  - **Binary (base 2):** Prefixed with 0b or 0B (e.g., 0b1010).

Floating-Point Literals:

Represent numbers with decimal points or exponential components. By default, they are of type double. To specify a float literal, append f or F (e.g., 3.14, 2.5f, 1.23E-05d).

Character Literals:

Represent a single character enclosed within single quotes (e.g., 'A', 'c'). They can also include escape sequences for special characters like \n (newline), \t (tab), \\ (backslash), etc.

String Literals:

Represent a sequence of characters enclosed within double quotes (e.g., "Hello, World!", "Java Programming"). These create String objects.

Boolean Literals:

Represent the logical values true or false.

Null Literal:

Represents the absence of a value, used particularly for object references (e.g., String s = null;). It is case-sensitive and must be null in lowercase. It cannot be used with primitive data types

What is an Array in Java?

An **array** is a **container object** that holds a **fixed number of elements** of the **same data type**. Arrays in Java are **indexed**, meaning each element is accessed using its position (index starts from 0).

Key Characteristics of Arrays:

- **Fixed Size**: Size must be defined at the time of creation and cannot be changed later.
- **Homogeneous Elements**: All elements must be of the same type.
- **Indexed Access**: Each element is accessed using an index.
- **Stored in Contiguous Memory Locations**.

Array Declaration and Initialization

Declaration:

```
int[] numbers;          // Recommended
int numbers[];          // Also valid
```

Memory Allocation:

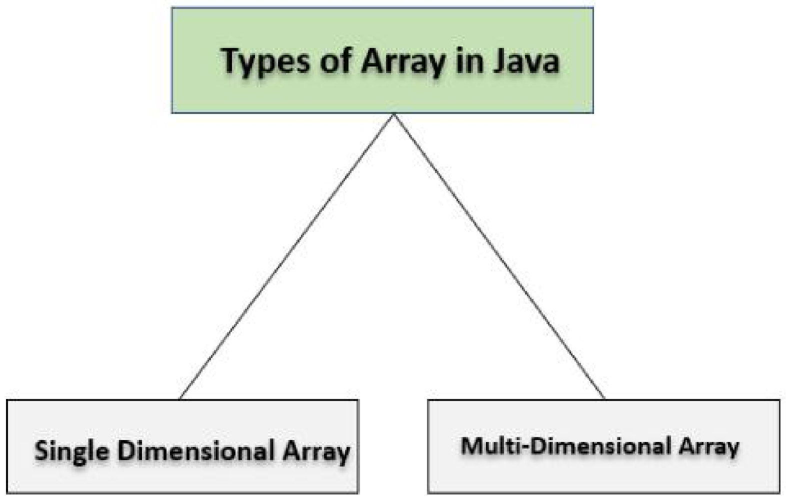
```
numbers = new int[5];   // Allocates memory for 5 integers
```

Declaration + Initialization:

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

In Java, an array is a container object that holds a fixed number of values of a single data type. The elements of an array are stored in contiguous memory locations, and each element can be accessed by its index, which starts from 0.

There are two primary types of arrays in Java:



Single-Dimensional Arrays:

- These are the most basic type, representing a linear collection of elements.
- Elements are arranged in a single row and accessed using a single index.
- Example:

```
int[] numbers = {10, 20, 30, 40, 50}; // An array of integers
String[] names = {"Alice", "Bob", "Charlie"}; // An array of strings
```

Example:-

```
public class Geeks {
    public static void main(String[] args) {

        // initializing array
        int[] arr = { 40,55,63,17,22,68,89,97,89};

        // size of array
        int n = arr.length;

        // traversing array
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
    }
}
```

- Multi-Dimensional Arrays:
    - These are arrays of arrays, meaning they can store data in a grid-like or tabular structure.
    - The most common multi-dimensional array is the two-dimensional array, often visualized as a matrix or table with rows and columns.

    - Elements are accessed using multiple indices (e.g., one for the row and one for the column).
    - Example (Two-Dimensional Array):

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
}; // A 3x3 array
```

- Three-Dimensional Arrays: These are arrays of two-dimensional arrays, conceptually representing a cube or layers of tables.

Java

```
int[][][] cube = {
    {{1, 2}, {3, 4}},
    {{5, 6}, {7, 8}}
}; // A 2x2x2 array
```

Example:-

```
public class MatrixAddition {


    public static void main(String[] args) {

        // Define two matrices

        int[][] matrix1 = {

            {1, 2, 3},

            {4, 5, 6},

            {7, 8, 9}

        };


        int[][] matrix2 = {

            {9, 8, 7},
            {6, 5, 4},
            {3, 2, 1}
        };

        // Get dimensions of the matrices
        int rows = matrix1.length;
        int cols = matrix1[0].length;

        // Create a new matrix to store the sum
        int[][] sumMatrix = new int[rows][cols];

        // Perform matrix addition
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                sumMatrix[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }

        // Print the resulting sum matrix
        System.out.println("Sum of the two matrices:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(sumMatrix[i][j] + " ");
            }
            System.out.println(); // Move to the next line after each row
```

```
        }
    }
}
```

What are Operators in Java?

Operators are special symbols or keywords that perform operations on variables and values. Java provides many types of operators to perform different operations such as arithmetic, comparison, logical, etc.

Types of Operators in Java

Operator Type	Examples	Description
1. Arithmetic	`+`, `-`, `*`, `/`, `%`	Perform mathematical operations
2. Relational	`==`, `!=`, `>`, `<`	Compare two values (return `true` or `false`)
3. Logical	`&&`, `	
4. Assignment	`=`, `+=`, `-=`, etc.	Assign values to variables
5. Unary	`++`, `--`, `+`, `-`	Operate on a single operand
6. Bitwise	`&`, `	`, ^, ~`
7. Ternary (?:)	`condition ? x : y`	Shorthand for `if-else`

1. Arithmetic Operators

Operator	Meaning	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus	`a % b`

2. Relational (Comparison) Operators

Operator	Meaning	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

3. Logical Operators

Operator	Meaning	Example
&&	Logical AND	a > 0 && b > 0
`		`
!	Logical NOT	!(a > 0)

4. Assignment Operators

Operator	Meaning	Example
=	Assign value	a = 10
+=	Add and assign	a += 5 ⟶ a = a + 5
-=	Subtract and assign	a -= 3 ⟶ a = a - 3
*=	Multiply and assign	a *= 2
/=	Divide and assign	a /= 2

5. Unary Operators

Operator	Meaning	Example
++	Increment	a++ or ++a
--	Decrement	a-- or --a
+	Unary plus (positive)	+a
-	Unary minus (negative)	-a

6. Ternary Operator

```
condition ? value_if_true : value_if_false
```

Example:

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

Java Program Demonstrating All Major Operators

```java
public class OperatorExamples {
    public static void main(String[] args) {
        int a = 10, b = 5;

        // Arithmetic Operators
        System.out.println("Arithmetic Operators:");
        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));

        // Relational Operators
        System.out.println("\nRelational Operators:");
        System.out.println("a == b: " + (a == b));
        System.out.println("a != b: " + (a != b));
        System.out.println("a > b: " + (a > b));
        System.out.println("a < b: " + (a < b));

        // Logical Operators
        System.out.println("\nLogical Operators:");
        boolean x = true, y = false;
        System.out.println("x && y: " + (x && y));
        System.out.println("x || y: " + (x || y));
        System.out.println("!x: " + (!x));

        // Assignment Operators
        System.out.println("\nAssignment Operators:");
        int c = 20;
        System.out.println("Initial c: " + c);
        c += 5; // c = c + 5
        System.out.println("c += 5 → " + c);
        c *= 2;
        System.out.println("c *= 2 → " + c);

        // Unary Operators
        System.out.println("\nUnary Operators:");
        int d = 7;
        System.out.println("d = " + d);
        System.out.println("d++ = " + d++); // post-increment
        System.out.println("Now d = " + d);
        System.out.println("++d = " + ++d); // pre-increment

        // Ternary Operator
        System.out.println("\nTernary Operator:");
        int max = (a > b) ? a : b;
        System.out.println("Max of a and b is: " + max);
    }
}
```

Output:

```
Arithmetic Operators:
a + b = 15
a - b = 5
a * b = 50
a / b = 2
a % b = 0

Relational Operators:
a == b: false
a != b: true
a > b: true
a < b: false

Logical Operators:
x && y: false
x || y: true
!x: false

Assignment Operators:
Initial c: 20
c += 5 → 25
c *= 2 → 50

Unary Operators:
d = 7
d++ = 7
Now d = 8
++d = 9

Ternary Operator:
Max of a and b is: 10
```

Conditional Statements

Conditional statements in Java are utilized to make decisions in a program based on some specific conditions. The main constraints are 'if,' 'else,' and 'else if.' These statements allow the program to execute different lines of code depending on the scenario, whether a particular condition is true or false. These are vital for controlling the flow of a program that enables the developers to create the logic that is able to respond dynamically to the data or the input, which further helps in making the program more powerful and flexible.

Why Are Conditional Statements Important in Programming?

Conditional statements are essential in programming because they enable a program to make decisions based on certain conditions. Just like in real life, where we act differently depending on situations (e.g., "if it rains, take an umbrella"), programs also need to choose what to do depending on input or data. In Java, conditional statements

like if, else if, and else control the flow of execution by deciding which block of code should run.
They are used in almost every application—such as validating login credentials, checking user inputs, or deciding actions based on system status. For example:

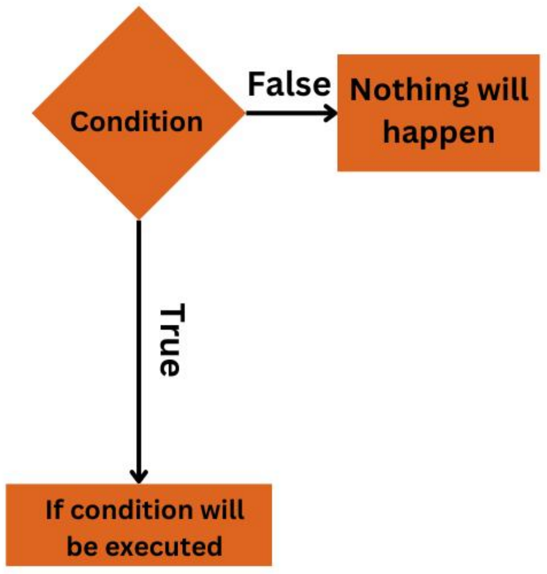
```
if(password.equals("admin123")) {
   System.out.println("Access Granted");
} else {
   System.out.println("Access Denied");
}
```

Types of Java Conditional Statements

There are five types of Java conditional statements. All of these are widely used according to their syntaxes and use cases. We will discuss them further in detail so let us note them down below:

1. Java If Statement
2. Java If-Else Statement
3. Java If-Else-If Ladder Statement
4. Java Nested If Statement
5. Java Switch Statement

Java If Statement



If Statement

Suppose a condition is true **if a statement** is used to run the program. It is also known as a one-way selection statement. If a condition is used, an argument is passed, and if it is satisfied, the corresponding code is executed; otherwise, nothing happens.

Syntax

The syntax of the If statement is:-

```
if (expression) {
   // You can enter the code here
}
```

Working of Java If Statement

If conditional statements in java works on the following given statement:-

1. Statements in the block are executed if the expression evaluates to nonzero.
2. Control is transferred to the statement that follows the expression if it evaluates to zero (false).

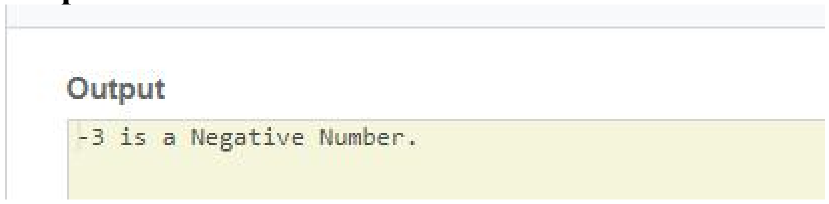
Example

The below given example helps in understanding the if statement in Java.

- Java

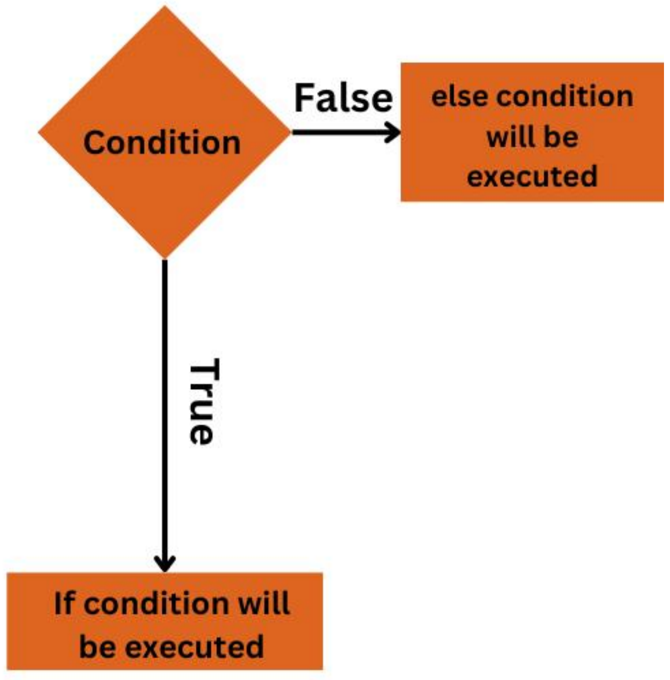
```
import java.util.*;
class sample {
   public static void main(String[] args) {
      int a = -3;
      if (a < 0)   {
         System.out.println(a + " is a Negative Number.");
      }
   }
}
```

Output:



```
-3 is a Negative Number.
```

Java If-Else Statement


Condition
False
else condition will be executed
True
If condition will be executed

If-Else Statement

An if-else statement is a control structure that determines which statements to choose based on a set of conditions.

Syntax

The syntax of the If-Else conditional statements in java is:-

```
if (condition) {
   // Statements that will be carried out if the condition is satisfied
} else {
   // Statements that will be carried out if the condition is not met
}
```

Working of Java If-Else Statement

The statements in the if block is carried out if the condition is met. Otherwise, the statements in the else block are carried out. Always follow an if clause with an else clause. If not, the compiler will produce a "misplaced else" error.

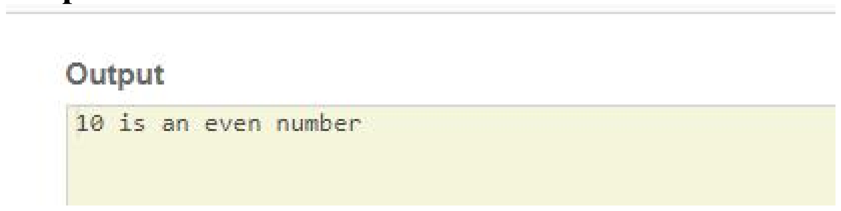
Example

The example below helps us understand the if-else conditional statements in Java.

- Java

```
import java.util.*;
```

```
class sample {
   public static void main(String[] args) {
      int a = 10;
      if (a % 2 == 0)     {
         System.out.println(a + " is an even number");
      }
      else     {
         System.out.println(a + " is an odd number");
      }
   }
}
```
Output:



Output

```
10 is an even number
```

Using Ternary Operator

The Java ternary operator offers a concise syntax for determining if a condition is true or false. It returns a value based on the outcome of the Boolean test.
One can replace their Java if-else statement using a ternary operator to create compact code. Expert coders enjoy the clarity and simplicity the Java ternary operator adds to their code.
The syntax of the Java Ternary operator is as follows:

```
(condition) ? (return if true) : (return if false);
```

Let's understand the concept with the help of a code example. We will see the above code using the ternary operator instead of the Java If-else statement.

- Java

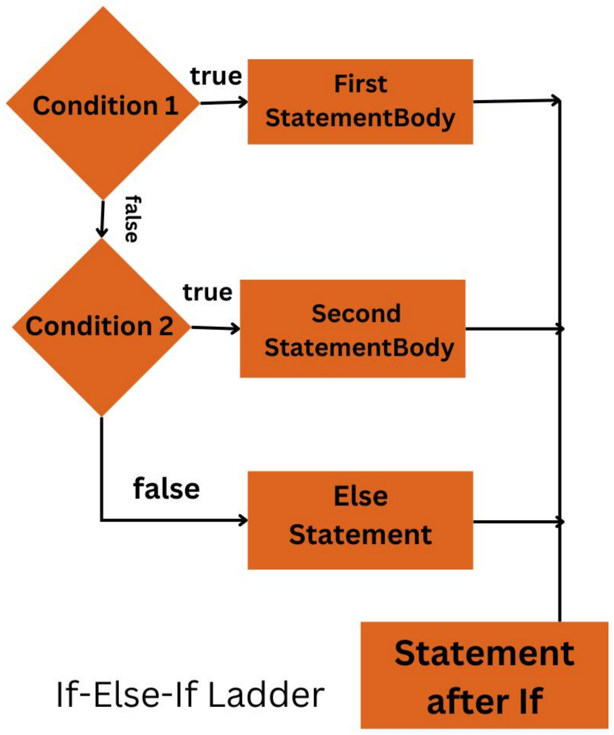
```
class example {
  public static void main( String args[] ) {
    int a = 10;
    String teropt =  (a % 2 == 0) ? a + " is an even number" : a + " is an odd number";
    System.out.println(teropt);
  }
}
```

Output

```
10 is an even number
```

Java If-Else-If Ladder Statement

An if-else-if ladder in Java can execute one code block while multiple other blocks are executed.



Syntax

The syntax of the If-Else-If conditional statements in java is:-

```
if (condition1) {
   // Executable code if condition1 is true,
} else if (condition2) {
   // Executable code if condition2 is true
}



...
else {
   // Executable code if all the conditions are false
}
```

Working of Java If-Else-If Ladder Statement

The conditional expressions are evaluated in the If-Else-If statement from top to bottom. The statement associated with a true condition is executed as soon as it is found, and the rest of the ladder is skipped. The final else statement is executed if all the conditions are false.

Example

The below given example helps in understanding the if-else-if ladder conditional statements in Java.

- Java

```
import java.util.*;
class sample {
    public static void main(String[] args) {
        int i = 3;
        if (i == 1)
            System.out.println("January");
        else if (i == 2)
            System.out.println("February");
        else if (i == 3)
            System.out.println("March");
        else
            System.out.println("April");
    }
}
```

Output:

Output

```
March
```

Java Nested If Statement

If conditions inside another if conditions are called as Nested If conditional statements in java.

Syntax

The syntax of the nested if conditional statements in java are:-

```
if (condition1) {
    // Statement 1 will execute

    if (condition2)   {
      // Statement 2 will execute
    }
}
```

Working of Java Nested If Statement

The way nested if statements operate assumes that they must first become true and sufficient for the other states with the second condition for them to be used, even though other statements may choose to proceed with a false condition if the first condition is met.

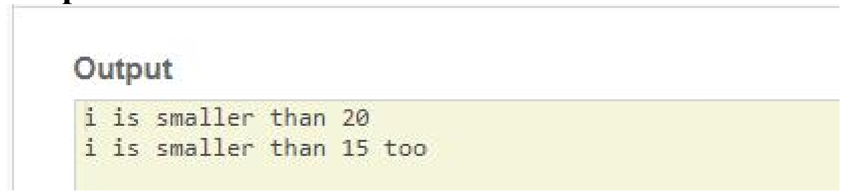
Example

The example below helps us understand the Nested If conditional statements in Java.

- Java

```
        System.out.println("i is smaller than 15 too");
      else
        System.out.println("i is greater than 20");
    }
  }
}
```

Output:



Output

```
i is smaller than 20
i is smaller than 15 too
```

Java Switch Statement

Unlike the if-else statement, the switch statement has more than one way to be executed. Additionally, it compares the expression's value to its cases by focusing its evaluation on a few primitive or class types.

Syntax

The syntax of the Switch statement is:-

```
switch (Expression) {
    case value 1:
        // Statement 1;
    case value 2:
        // Statement 2;
    case value 3:
        // Statement 3;

         ...
    case value n:
        // Statement n;
    Default:
```

```
    // default statement;
}
```

Working of Java Switch Statement

Java's switch case allows multiple conditions to be checked simultaneously, similar to an if-else ladder. A constant or a literal expression that can be evaluated is given to Switch.
Each test case's value is compared to the expression's value until a match is made. The associated code is executed if the specified default keyword does not find a match. Otherwise, the code designated for the test case that matches it is run.

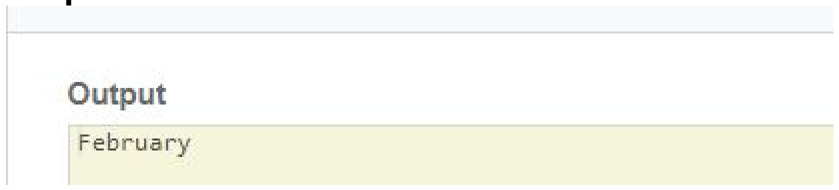
Example

- Java

```
import java.util.*;
class Sample {
  public static void main(String[] args) {
    int mon = 2;
    switch (mon) {
    case 1:
      System.out.println("January");
      break;

    case 2:
      System.out.println("February");
      break;

    case 3:
      System.out.println("March");
      break;

    case 4:
      System.out.println("April");
      break;

    case 5:
      System.out.println("May");
      break;

    case 6:
      System.out.println("June");
      break;

    default:
```

```
    System.out.println("Not present in first half month of year");
   }
  }
}
```

Output:



Output

```
February
```

Java Loops

Loops in programming allow a set of instructions to run multiple times based on a condition. In Java, there are three types of Loops, which are explained below:

1. for loop

The for loop is used when we know the number of iterations (we know how many times we want to repeat a task). The for statement includes the initialization, condition, and increment/decrement in one line.

Example: The below Java program demonstrates a for loop that prints numbers from 0 to 10 in a single line.

```
// Java program to demonstrates the working of for loop

import java.io.*;


class Geeks {

    public static void main(String[] args)

    {

        for (int i = 0; i <= 10; i++) {

            System.out.print(i + " ");

        }

    }
```

}

Output

```
0 1 2 3 4 5 6 7 8 9 10
```

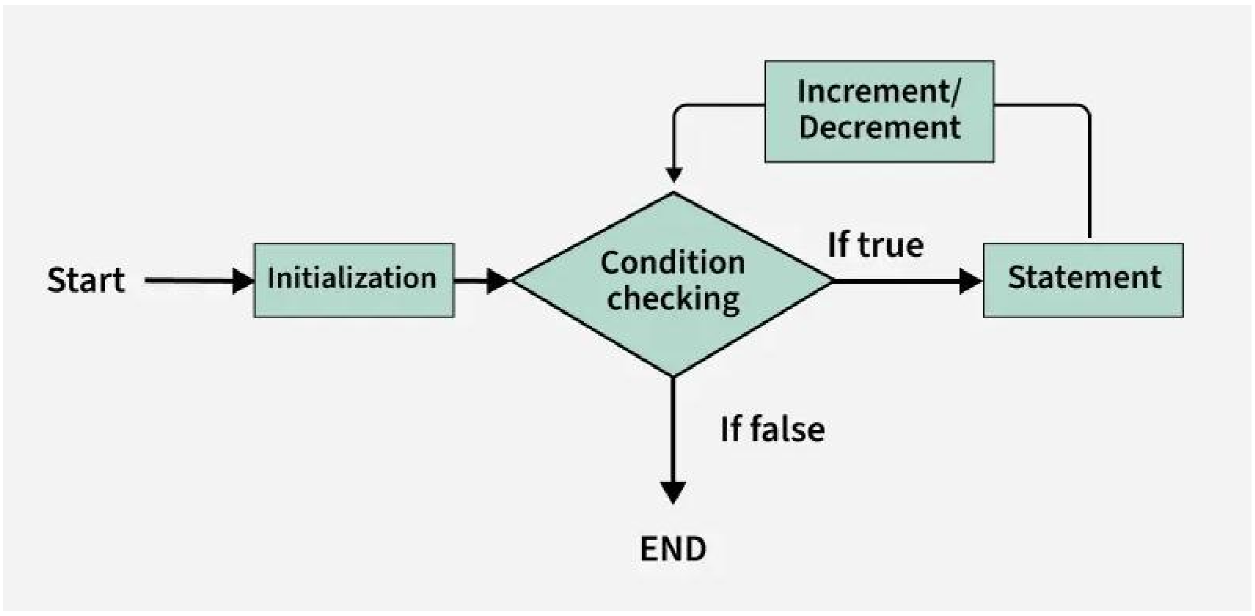
Syntax:

for (initialization; condition; increment/decrement) {

// code to be executed

}

The image below demonstrates the flow chart of a for loop:



Flowchart of for -loop

- **Initialization condition**: Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an Entry Control Loop as the condition is checked prior to the execution of the loop statements.
- **Statement execution**: Once the condition is evaluated to true, the statements in the loop body are executed.
- **Increment/ Decrement**: It is used for updating the variable for next iteration.

- **Loop termination**:When the condition becomes false, the loop terminates marking the end of its life cycle.

Note: There is another form of the for loop known as **Enhanced for loop** or (for each loop).

Enhanced for loop (for each)

This loop is used to iterate over arrays or collections.

Example: The below Java program demonstrates an Enhanced for loop (for each loop) to iterate through an array and print names.

```
// Java program to demonstrate

// the working of for each loop

import java.io.*;


class Geeks {

    public static void main(String[] args)

    {

        String[] names = { "Sweta", "Gudly", "Amiya" };


        for (String name : names) {

            System.out.println("Name: " + name);

        }

    }

}
```

Output

```
Name: Sweta

Name: Gudly

Name: Amiya
```

Syntax:

```
for (dataType variable : arrayOrCollection) {
// code to be executed
}
```

2. while Loop

A while loop is used when we want to check the condition before executing the loop body.

Example: The below Java program demonstrates a while loop that prints numbers from 0 to 10 in a single line.

```
// Java program to demonstrates

// the working of while loop

import java.io.*;


class Geeks {

    public static void main(String[] args)

    {

        int i = 0;

        while (i <= 10) {

            System.out.print(i + " ");

            i++;

        }

    }

}
```

Output

```
0 1 2 3 4 5 6 7 8 9 10
```

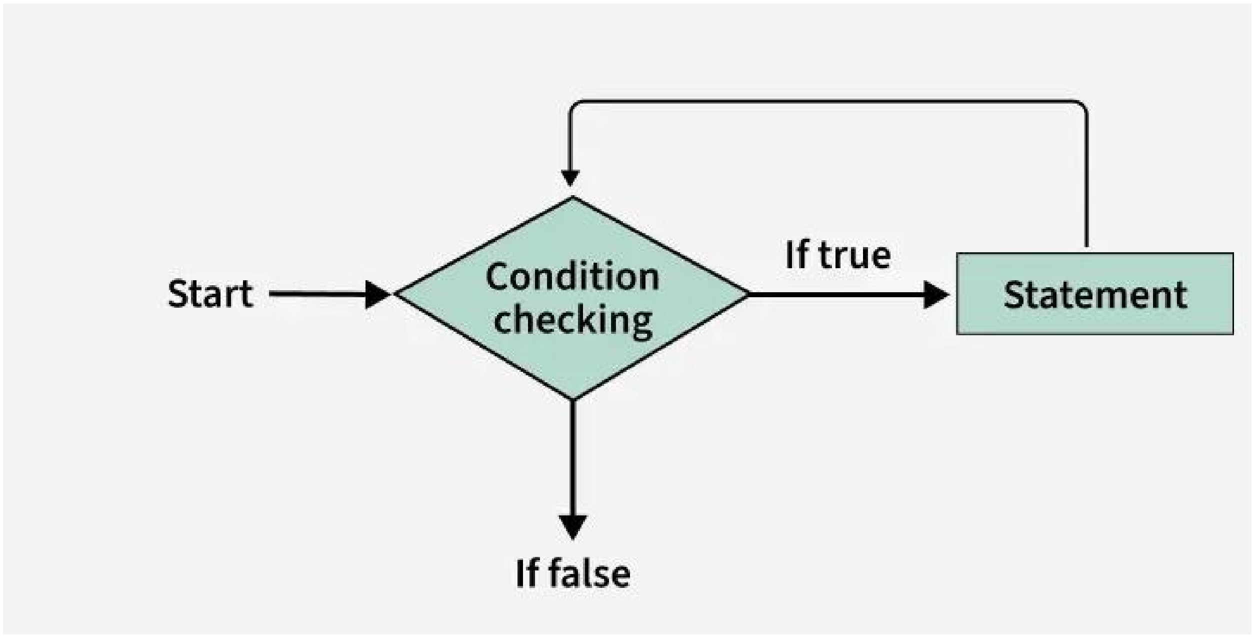
Syntax:

while (condition) {

// code to be executed

}

The below image demonstrates the flow chart of a while loop:



Flowchart of while-loop

- While loop starts with the checking of Boolean condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called Entry control loop
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

3. do-while Loop

The do-while loop ensures that the code block executes **at least once** before checking the condition.

Example: The below Java program demonstrates a do-while loop that prints numbers from 0 to 10 in a single line.

// Java program to demonstrates

```
// the working of do-while loop
import java.io.*;

class Geeks {
    public static void main(String[] args)
    {
        int i = 0;
        do {
            System.out.print(i + " ");
            i++;
        } while (i <= 10);
    }
}
```

Output

```
0 1 2 3 4 5 6 7 8 9 10
```

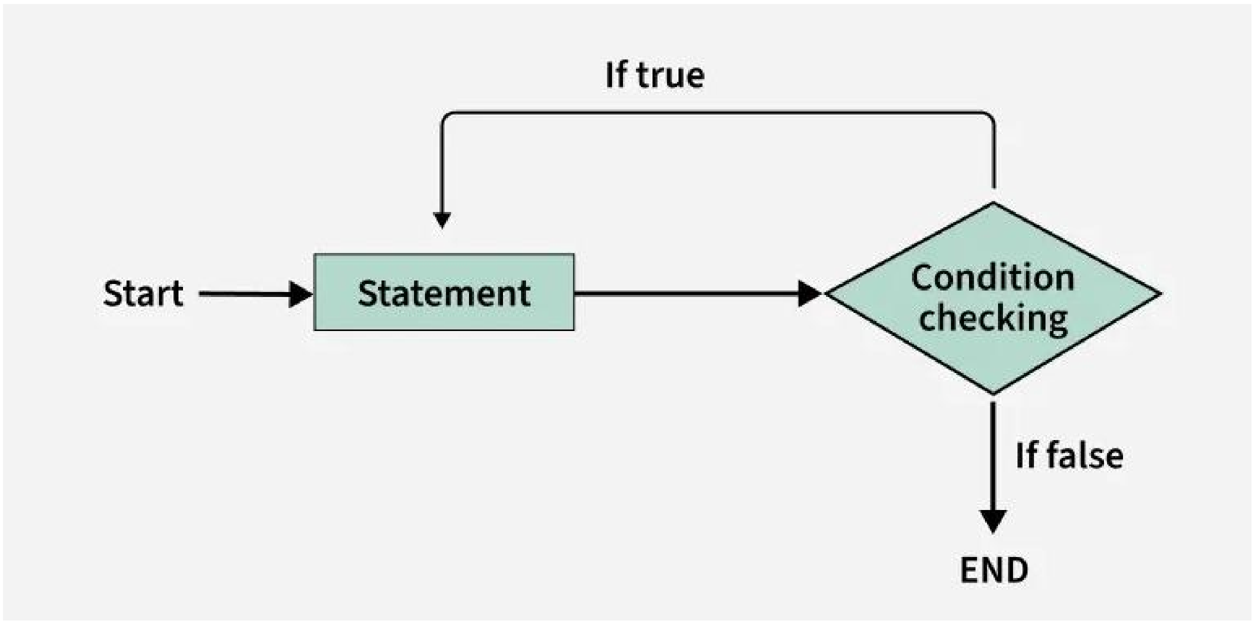
Syntax:

do {

// code to be executed

} while (condition);

The below image demonstrates the flow chart of a do-while loop:



Flowchart of do-while loop

- do while loop starts with the execution of the statement. There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements a tleast once before any condition is checked, and therefore is an example of exit control loop.

Class, Object and Method

A class is a blueprint for the object. Before we create an object, we first need to define the class.

We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Create a class in Java

We can create a class in Java using the class keyword. For example,

```
class ClassName {
  // fields
  // methods
}
```

Here, fields (variables) and methods represent the **state** and **behavior** of the object respectively.

- fields are used to store data
- methods are used to perform some operations

For our bicycle object, we can create the class as

```
class Bicycle {

  // state or field
  private int gear = 5;

  // behavior or method
  public void braking() {
    System.out.println("Working of Braking");
  }
}
```

In the above example, we have created a class named Bicycle. It contains a field named gear and a method named braking().

Here, Bicycle is a prototype. Now, we can create any number of bicycles using the prototype. And, all the bicycles will share the fields and methods of the prototype.

Note: We have used keywords private and public. These are known as access modifiers. To learn more, visit Java access modifiers.

Java Objects

An object is called an instance of a class. For example, suppose Bicycle is a class then MountainBicycle, SportsBicycle, TouringBicycle, etc can be considered as objects of the class.

Creating an Object in Java

Here is how we can create an object of a class.

```
className object = new className();


// for Bicycle class

Bicycle sportsBicycle = new Bicycle();


Bicycle touringBicycle = new Bicycle();
```

We have used the new keyword along with the constructor of the class to create an object. Constructors are similar to methods and have the same name as the class. For example, Bicycle() is the constructor of the Bicycle class. To learn more, visit Java Constructors.

Here, sportsBicycle and touringBicycle are the names of objects. We can use them to access fields and methods of the class.

As you can see, we have created two objects of the class. We can create multiple objects of a single class in Java.

Note: Fields and methods of a class are also called members of the class.

Access Members of a Class

We can use the name of objects along with the Operator to access members of a class. For example,

```
class Bicycle {

  // field of class
  int gear = 5;

  // method of class
  void braking() {
    ...
  }
}

// create object
Bicycle sportsBicycle = new Bicycle();

// access field and method
sportsBicycle.gear;
sportsBicycle.braking();
```

In the above example, we have created a class named Bicycle. It includes a field named gear and a method named braking(). Notice the statement,

```
Bicycle sportsBicycle = new Bicycle();
```

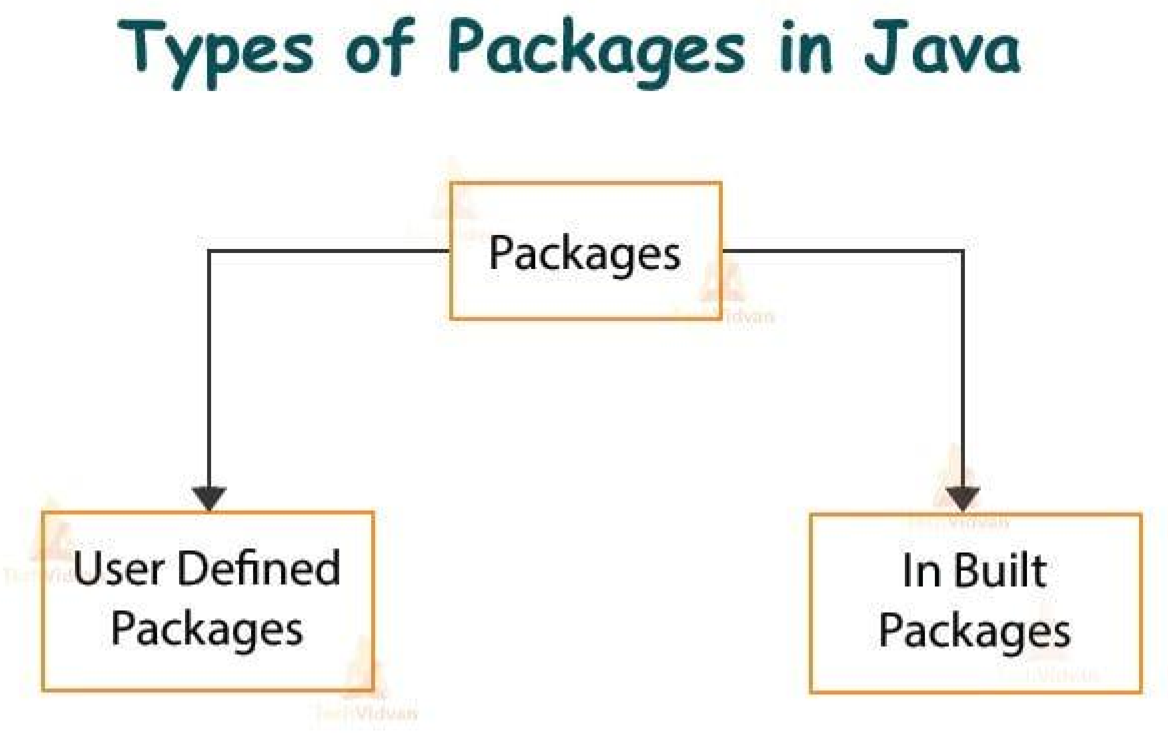
Here, we have created an object of Bicycle named sportsBicycle. We then use the object to access the field and method of the class.

- **sportsBicycle.gear** - access the field gear
- **sportsBicycle.braking()** - access the method braking()

We have mentioned the word **method** quite a few times. You will learn about Java methods in detail in the next chapter.

Packages in Java

Packages in Java are used to group related classes, interfaces, and sub-packages into a namespace. They provide a way to organize files in larger projects, avoid name conflicts, and manage access control. Packages also help in making code modular and manageable, especially when dealing with large-scale applications.



This tutorial will cover:

1. What is a Package in Java?

1. What is a Package in Java?

A package is a namespace that organizes a set of related classes and interfaces. Packages are used to prevent name clashes and to control the access of classes, interfaces, and methods.

Basic Definition:

package com.example.utils;

In the above example, `com.example.utils` is a package that could contain utility classes related to an application. Classes inside the package can be imported and used by other classes.

2. Why Use Packages?

1. Organizing Code: Packages help group similar functionality into modules. For example, all database-related classes can go into a `com.example.database` package.

2. Avoiding Naming Conflicts: In large projects, it’s possible to have classes with the same names. Packages provide namespaces, so two classes with the same name can exist in different packages.

3. Access Control: Java packages provide access control mechanisms. Classes, methods, and fields can have package-level visibility.

4. Reusability: Code can be organized into packages that can be reused across projects.

3. Types of Packages in Java

Java has two types of packages:

1. Built-in Packages: These are provided by the Java API (e.g., `java.util`, `java.io`, `java.lang`).

2. User-defined Packages: These are created by users to organize their classes and interfaces.

4. How to Create a Package?

To create a package in Java, you use the `package` keyword. The package statement must be the first line in the source file (excluding comments).

Creating a Package:

```
// File: com/example/HelloWorld.java
package com.example;

public class HelloWorld {
    public void greet() {
        System.out.println("Hello from the com.example package!");
    }
}
```

In this example:

- The `package com.example;` statement defines that the class `HelloWorld` belongs to the `com.example` package.

- The folder structure should mirror the package name (`com/example/`).

5. Accessing Classes from Packages

To access a class from a package, the fully qualified name of the class (including the package name) should be used, or the class should be imported.

Accessing Without Import:

```
com.example.HelloWorld hello = new com.example.HelloWorld();
hello.greet();
```

Accessing With Import:

You can import the class to avoid writing the full package name every time:

```
import com.example.HelloWorld;

public class Main {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld();
        hello.greet();
    }
}
```

6. Importing Packages

Java provides the `import` statement to import classes or entire packages.

1. Importing a Single Class:

```
import com.example.HelloWorld;
```

2. Importing All Classes in a Package:

```
import com.example.*;
```

In the second example, all classes within the `com.example` package will be imported.

7. Sub-packages in Java

Java supports sub-packages, which are simply packages within other packages. There is no special syntax for sub-packages; they are treated like regular packages, but the hierarchy is created by adding dots in the package name.

Example:

package com.example.utils;

The `utils` package is a sub-package of `com.example`.

You can nest packages as deeply as required:

package com.example.database.connection;

This would represent a `connection` package under `database` under `com.example`.

8. Package Naming Conventions

Java follows certain conventions for naming packages to ensure uniqueness and clarity:

1. Reverse Domain Name: Use your domain name in reverse as the base of your package names. For example, if your domain is `example.com`, start your package with `com.example`.

2. Lowercase Names: Use lowercase letters to avoid conflicts with class names.

3. Meaningful Names: Use descriptive and meaningful names to reflect the content of the package.

Example:

package com.mycompany.product.service;

9. Java Built-in Packages

Java comes with several built-in packages that provide a wide range of functionalities. Some common built-in packages are:

- java.lang: Provides classes fundamental to the Java language (automatically imported).

- java.util: Contains utility classes, including collections framework, date, time facilities, and more.

- java.io: Contains classes for input/output operations (e.g., file handling).

- java.net: Contains classes for networking operations.

- java.sql: Contains classes for database access and processing.

Example of Using Built-in Packages:

```
import java.util.ArrayList;

public class BuiltInExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        System.out.println(list);
    }
}
```

10. Example Program Using Packages

Here's a full example of creating and using user-defined packages.

Step 1: Define a Package

Create a file `com/example/greetings/Greeter.java`:

```
package com.example.greetings;

public class Greeter {
    public void sayHello() {
        System.out.println("Hello from the Greeter class!");
    }
}
```

Step 2: Create Another Class to Use This Package

Create another file in a different package `com/example/app/MainApp.java`:

```
package com.example.app;

// Importing the Greeter class from com.example.greetings
import com.example.greetings.Greeter;

public class MainApp {
    public static void main(String[] args) {
        Greeter greeter = new Greeter();
        greeter.sayHello();
    }
}
```

Step 3: Compiling and Running

To compile and run this example, follow these steps:

1. Compile:

Navigate to the root directory and compile both files:

```
javac com/example/greetings/Greeter.java
javac com/example/app/MainApp.java
```

2. Run:

Run the `MainApp` class:

```
java com.example.app.MainApp
```

Output:

```
Hello from the Greeter class!
```

11. Use of `package-info.java`

Java allows the creation of a `package-info.java` file to provide package-level documentation and annotations.

Example:

```
/**
 * This is the com.example.app package for the application.
 */
package com.example.app;
```

Nested and Inner Classes

Nested Classes

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the nested class, and the class that holds the inner class is called the outer class.

Syntax

Following is the syntax to write a nested class. Here, the class Outer_Demo is the outer class and the class Inner_Demo is the nested class.

```
class Outer_Demo {
   class Inner_Demo {
   }
}
```

Nested classes are divided into two types −

Non-static nested classes − These are the non-static members of a class.

Static nested classes − These are the static members of a class.

Inner Classes

Inner Classes (Non-static Nested Classes)

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier private, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Types of Java Inner Classes

Inner classes are of three types depending on how and where you define them. They are −

Inner Class

Method-local Inner Class

Anonymous Inner Class

Creating an Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

Example: Creating an inner class in Java

Open Compiler

```
class Outer_Demo {
   int num;

   // inner class
   private class Inner_Demo {
      public void print() {
         System.out.println("This is an inner class");
      }
   }

   // Accessing he inner class from the method within
   void display_Inner() {
      Inner_Demo inner = new Inner_Demo();
      inner.print();
   }
}

public class My_class {

   public static void main(String args[]) {
      // Instantiating the outer class
      Outer_Demo outer = new Outer_Demo();
      // Accessing the display_Inner() method.
      outer.display_Inner();
   }
}
```

Here you can observe that Outer_Demo is the outer class, Inner_Demo is the inner class, display_Inner() is the method inside which we are instantiating the inner class, and this method is invoked from the main method.

If you compile and execute the above program, you will get the following result −

Output

```
This is an inner class.
```

Accessing the Private Members

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, getValue(), and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

```
Outer_Demo outer = new Outer_Demo();
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

The following program shows how to access the private members of a class using inner class.

Example: Accessing the Private Members Using Inner Class

```
class Outer_Demo {
  // private variable of the outer class
  private int num = 175;

  // inner class
  public class Inner_Demo {
    public int getNum() {
      System.out.println("This is the getnum method of the inner class");
      return num;
    }
  }
}

public class My_class2 {

  public static void main(String args[]) {
    // Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();
```

```
      // Instantiating the inner class
      Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
      System.out.println(inner.getNum());
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is the getnum method of the inner class: 175
```

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

Example: Method-local Inner Class

Open Compiler

```
public class Outerclass {
   // instance method of the outer class
   void my_Method() {
      int num = 23;
      // method-local inner class
      class MethodInner_Demo {
         public void print() {
            System.out.println("This is method inner class "+num);
         }
      } // end of inner class

      // Accessing the inner class
      MethodInner_Demo inner = new MethodInner_Demo();
      inner.print();
   }

   public static void main(String args[]) {
      Outerclass outer = new Outerclass();
      outer.my_Method();
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is method inner class 23
```

Anonymous Inner Class

An inner class declared without a class name is known as an anonymous inner class. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows −

Syntax: Anonymous Inner Class

```
AnonymousInner an_inner = new AnonymousInner() {
   public void my_method() {
      ........
      ........
   }
};
```

The following program shows how to override the method of a class using anonymous inner class.

Example: Anonymous Inner Class

Open Compiler

```
abstract class AnonymousInner {
   public abstract void mymethod();
}

public class Outer_class {

   public static void main(String args[]) {
      AnonymousInner inner = new AnonymousInner() {
         public void mymethod() {
            System.out.println("This is an example of anonymous inner class");
         }
      };
      inner.mymethod();
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is an example of anonymous inner class
```

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument −

```
obj.my_Method(new My_Class() {
   public void Do() {
      .....
      .....
   }
});
```

The following program shows how to pass an anonymous inner class as a method argument.

Example

Open Compiler

```
// interface
interface Message {
   String greet();
}

public class My_class {
   // method which accepts the object of interface Message
   public void displayMessage(Message m) {
      System.out.println(m.greet() +
         ", This is an example of anonymous inner class as an argument");
   }
```

```
  public static void main(String args[]) {
    // Instantiating the class
    My_class obj = new My_class();

    // Passing an anonymous inner class as an argument
    obj.displayMessage(new Message() {
      public String greet() {
        return "Hello";
      }
    });
  }
}
```

If you compile and execute the above program, it gives you the following result −

Output

```
Hello, This is an example of anonymous inner class as an argument
```

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows −

Syntax

```
class MyOuter {
   static class Nested_Demo {
   }
}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

Example

Open Compiler

```
public class Outer {
   static class Nested_Demo {
      public void my_method() {
         System.out.println("This is my nested class");
      }
   }

   public static void main(String args[]) {
      Outer.Nested_Demo nested = new Outer.Nested_Demo();
      nested.my_method();
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is my nested class
```

Java Constructors

In Java, **constructors** play an important role in object creation. A constructor is a special block of code that is called when an object is created. Its main job is to initialize the object, to set up its internal state, or to assign default values to its attributes. This process happens automatically when we use the "new" keyword to create an object.

Characteristics of Constructors:

- **Same Name as the Class:** A constructor has the same name as the class in which it is defined.
- **No Return Type:** Constructors do not have any return type, not even void. The main purpose of a constructor is to initialize the object, not to return a value.
- **Automatically Called on Object Creation:** When an object of a class is created, the constructor is called automatically to initialize the object's attributes.
- **Used to Set Initial Values for Object Attributes:** Constructors are primarily used to set the initial state or values of an object's attributes when it is created.

Now, let us look at a simple example to understand how a constructor works in Java.

Example: This program demonstrates how **a constructor is automatically called when an object is created in Java.**

```java
// Java Program to demonstrate
// Constructor usage
import java.io.*;

// Driver Class
class Geeks {

    // Constructor
    Geeks()
    {
        super();
        System.out.println("Constructor Called");
    }

    // main function
    public static void main(String[] args)
    {
        Geeks geek = new Geeks();
    }
}
```

Output

```
Constructor Called
```

Note: It is not necessary to write a constructor for a class. It is because the Java compiler creates a default constructor (constructor with no arguments) if your class doesn't have any.

Types of Constructors in Java

Now is the correct time to discuss the types of the constructor, so primarily there are three types of constructors in Java are mentioned below:

- Default Constructor
- Parameterized Constructor
- Copy Constructor

1. Default Constructor in Java

A constructor that has no parameters is known as default constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. Once you define a constructor (with or without parameters), the compiler no longer provides the default constructor. Defining a parameterized constructor does not automatically create a no-argument constructor, we must explicitly define if needed. The default constructor can be implicit or explicit.

- **Implicit Default Constructor:** If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn't take any parameters and initializes the object with default values, such as 0 for numbers, null for objects.
- **Explicit Default Constructor:** If we define a constructor that takes no parameters, it's called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically. Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you.

Example: This program demonstrates **the use of a default constructor, which is automatically called when an object is created.**

```
// Java Program to demonstrate
// Default Constructor
```

```
import java.io.*;

// Driver class
class Geeks{

    // Default Constructor
    Geeks() {
        System.out.println("Default constructor");

    }

    // Driver function
    public static void main(String[] args)
    {
        Geeks hello = new Geeks();
    }
}
```

Output

```
Default constructor
```

Note: Default constructor provides the default values to the object like 0, null, false etc. depending on the type.

2. **Parameterized Constructor in Java**

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Example: This program demonstrates the use of a parameterized constructor to initialize an object's attributes with specific values.

```
// Java Program for Parameterized Constructor
import java.io.*;

class Geeks {

    // data members of the class
    String name;
    int id;

    Geeks(String name, int id) {
        this.name = name;
        this.id = id;
    }
}

class GFG
{
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor
        Geeks geek1 = new Geeks("Sweta", 68);
        System.out.println("GeekName: " + geek1.name
                           + " and GeekId: " + geek1.id);
    }
}
```

Output

```
GeekName: Sweta and GeekId: 68
```

Remember: Does constructor return any value?
There are no "return value" statements in the constructor, but the constructor returns the current class instance. We can write 'return' inside a constructor.

3. Copy Constructor in Java

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.
Note: Java does not provide a built-in copy constructor like C++. We can create our own by writing a constructor that takes an object of the same class as a parameter and copies its fields.
Example: This example, demonstrates how a copy constructor can be used to create a new object by copying the values of another object's attributes.

```
// Java Program for Copy Constructor
import java.io.*;

class Geeks {

    // data members of the class
    String name;
    int id;
    // Parameterized Constructor
    Geeks(String name, int id)
    {
        this.name = name;
        this.id = id;
    }

    // Copy Constructor
    Geeks(Geeks obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}

class GFG {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor
        System.out.println("First Object");
        Geeks geek1 = new Geeks("Sweta", 68);
        System.out.println("GeekName: " + geek1.name
                           + " and GeekId: " + geek1.id);

        System.out.println();

        // This would invoke the copy constructor
        Geeks geek2 = new Geeks(geek1);
        System.out.println(
            "Copy Constructor used Second Object");
        System.out.println("GeekName: " + geek2.name
                           + " and GeekId: " + geek2.id);
    }
}
```

Output

```
First Object

GeekName: Sweta and GeekId: 68
```

```
Copy Constructor used Second Object
GeekName: Sweta and GeekId: 68
```