JAVA PROGRAMMING LANGUAGE

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

SAVE AND SHARE

Curated By- HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

TOPICS COVERED-

PART-1 :-

- Introduction to Java and Features
- Writing First Program and Basic Input/Output in Java
- Identifiers in Java
- Variables in Java
- Data Types in Java
- Operators in Java
- Sample Problems|Operators, Data Type, Input/Output
- Decision Control Statements
  (If,Else,Switch,Break,Continue)

PART -2 :- (UPCOMING)

- Loops in Java
- Methods in Java
- Sample Problems|(Decision, Loops,Arrays & Strings)
- Arrays in Java
- Strings in Java

- **Immutable Strings in Java**
- **BigInteger Class in Java**
- **ArrayList in Java**

PART-1 :-

1.Introduction to Java and Features-

Getting started with any programming language should be purposeful, we must understand the **features and power** of the programming language to develop a deep interest in it.
Java was developed after the C and C++ around 1991. We can consider Java as a **complete package** for software developers to play with it for a variety of applications they want to build.

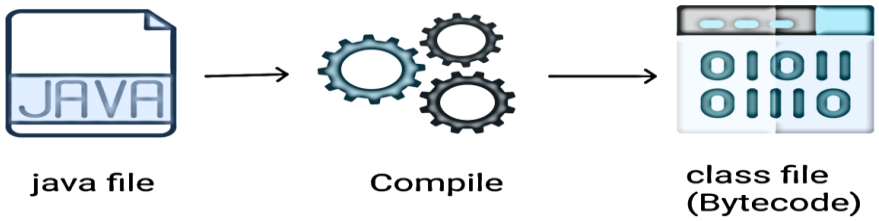
Java is currently the most used programming language worldwide. The biggest tech giants such as **Amazon, Google, TCS, Wipro, etc.,** have a huge percentage of code in Java.

Java is power-packed with ample of features. Here are some of the most important features of Java Programming Language:

- **Simple -** Java was designed for professional developers to learn and use easily . If you are aware of basic programming principles, then Java is not a hard nut to crack.
- **Secure -** Java achieves security by confining Java Programs to Java Execution Environment and not allowing Java programs to access other parts of the computer system.
- **Portable -** Portable code must run on a variety of operating systems, CPU's, and browsers. When a Java program is compiled, a Bytecode is generated, this bytecode can be executed on any platform using JVM (Java Virtual Machine). We will discuss JVM details in the latter part of the tutorial.

- **Object-Oriented -** Java provides a clean, usable, and pragmatic approach to objects. It was designed on the principle that "Everything under the sun is an Object."
- **Robust -**The robustness of the program is the reliable execution on a variety of systems. To better understand this, consider one of the main reasons for program failure is memory management. In C/C++, a programmer must manually allocate and free dynamic memory. There are cases where programmers forgot to free up the dynamic memory which may cause program failure. Java virtually solves this problem as unused objects are garbage collected automatically.
- **Architecture Neutral -** The central issue for Java Designers was that of code longevity and portability. One of the main problems for programmers was that there was no guarantee that the program running perfectly fine today will run tomorrow because of the system upgrades, processor upgrades, or changes in core system resources. Java Designers made several hard decisions and designed the Java Virtual Machine to solve this problem. Their goal was "write once, run anywhere, any time, forever".
- **Distributed -** Java is designed for distributed environment because it handles TCP/IP protocol. It is very much useful in building large scale distributed computing based applications. Java also supports Remote Method Invocation. This feature enables a program to invoke methods across a network.
- **Dynamic -** Java programs have run time meta information to verify and resolve accesses of the objects at runtime. This makes it possible to dynamically link code in a safe and secure environment. It also gives us the feature to update small fragments of bytecode, dynamically updated on a running system.

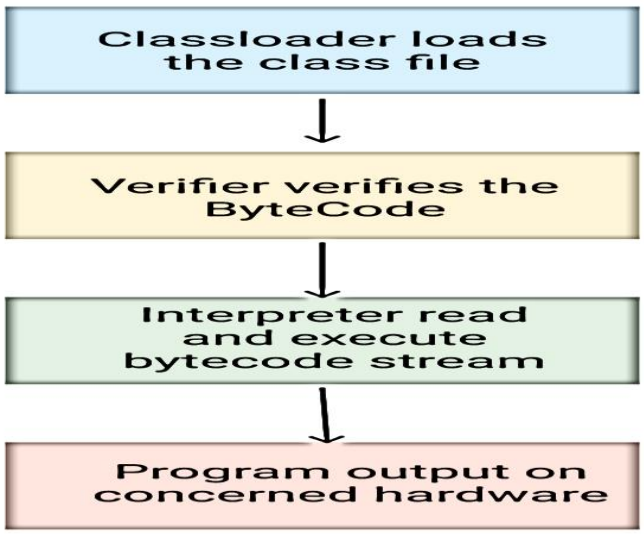
Java Program Compilation - Java program compilation converts a java program file to a highly optimized class file which is called bytecode.



Java Program Execution - Java program execution is completed with the help of the following tools.

- **Classloader -** It loads the class file for JVM execution.
- **Bytecode Verifier -** It verifies the bytecode and restrict the objects for illegal access of other parts of the system.
- **Interpreter -** It read the bytecode instructions and execute them line by line.

Java program Execution Flow:

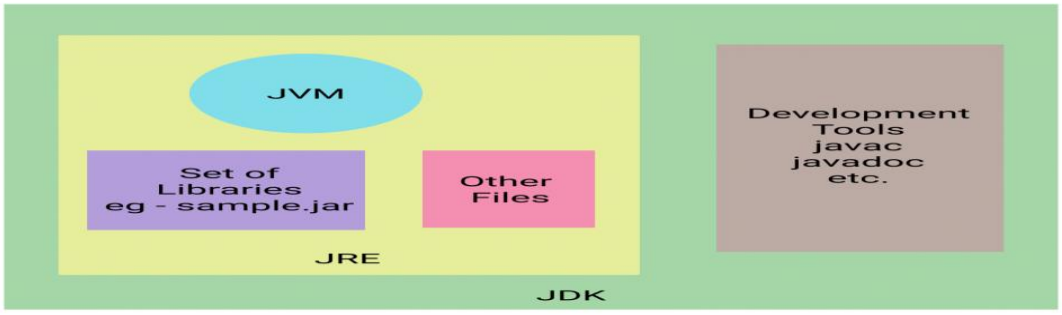


Java Virtual Machine (JVM)
JVM is called an abstract virtual machine because it does not exist physically, but it is a kind of specification that provides a secure runtime environment to execute the bytecode generated through the compiler, JVM actually invokes the main( ) method present in the Java program. JVM is a part of the JRE(Java Runtime Environment).
Java applications are called WORA (Write once, run anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

Java Runtime Environment (JRE)
JRE stands for “Java Runtime Environment” and may also be written as “Java RTE.” The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

- A **specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- An **implementation** is a computer program that meets the requirements of the JVM specification
- **Runtime Instance** Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

Java Development Kit (JDK)
Java Development Kit (in short JDK) is a Kit that provides the environment to develop and execute. The JDK contains a private Java Virtual Machine (JVM), interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



Difference between Java and C++

Comparison Metric	C++	Java
Domain and Usage	C++ is mainly used for system programming.	Java is mainly used for building desktop, web, and Mobile Applications.
Compiler and Interpreter	In C++ program is only compiled and converted into machine code. That is why it is platform dependent.	In Java , program once compiled converted into bytecode ,then interpreter read the bytecode stream to execute line by line through JVM. That is why it is platform independent.
Pointers	C++ supports pointers for memory allocation , You can write pointer programs.	Java supports pointer internally , however you can not write pointer programs.
Structure and Union	C++ have Structures and Unions.	Java don't have these , however you can implement this using classes.
Thread Support	C++ don't have inbuilt thread support , you have to use third party library.	Java has built in thread support.
Documentation Comment	C++ don't have documentation comment.	Java have documentation comment , which is useful for maintaining the code.
Call by value and reference	C++ supports both call by value and call by reference.	Java supports only call by value.
Operator Overloading	C++ supports operator overloading.	Java don't have support for operator overloading.

2.Writing First Program and Basic Input/Output in Java-

First Program - "Hello World"

The below-given program is the simplest program of Java, printing "Hello World" to the screen. Let us try to understand every bit of the code step by step.

```
// This is a simple Java program.
class HelloWorld
{
   // Your program begins with a call to main().
   // Prints "Hello, World" to the terminal window.
   public static void main(String args[])
   {
      System.out.println("Hello, World");
   }
}
```

Output:

```
Hello, World
```

The "Hello World!" program consists of three primary components: the HelloWorld class definition, the `main` method, and the source code comments. The following explanation will provide you with a basic understanding of the code:

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

   ```
   class HelloWorld
   ```

HelloWorld 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 **}**.

2. **main method:** In Java programming language, every application must contain a main method whose signature is:

```
public static void main(String[] args)
public: So that JVM can execute the method from anywh
ere.
static: Main method is to be called without object.
The modifiers public and static can be written in eit
her order.
void: The main method doesn't return anything.
main(): Name configured in the JVM.
String[]: The main method accepts a single argument:
          an array of elements of type String.
```

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

3. The next line of code is shown here. Notice that it occurs inside main( ).

```
System.out.println("Hello, World");
```

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

4. Comments: They can either be multi-line or single line comments.

```
/* This is a simple Java program.
Call this file "HelloWorld.java". */
```

This is a multiline comment. This type of comment must begin with /* and end with */. For single line you may directly use // as in C/C++.

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 which contain the main method) should match the name of the file that holds the program.

Java Basic Console Input/Output

In Java, there are three different ways of reading input from the user in the command line environment(console).

1.Using Buffered Reader Class

This is the classical Java method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader. We can read input from the user in the command line.

Advantages :

- The input is buffered for efficient reading.

Drawback: :

- The wrapping code is hard to remember.

```
// Java program to demonstrate BufferedReader

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test
{
   public static void main(String[] args) throws IOException
   {
      //Enter data using BufferReader
      BufferedReader reader =
               new BufferedReader(new
InputStreamReader(System.in));

      // Reading data using readLine
      String name = reader.readLine();

      // Printing the read line
      System.out.println(name);
   }
}
```

Note: To read other types, we use functions such as Integer.parseInt(), Double.parseDouble(). To read multiple values, we use split().

2. Using Scanner Class

This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions; however, it can also be used to read input from the user in the command line.

Advantages:

- Convenient methods for parsing primitives (nextInt(), nextFloat(), ...) from the tokenized input.

- Regular expressions can be used to find tokens.

Drawback:

- The reading methods are not synchronized.

```
// Java program to demonstrate working of Scanner in Java
import java.util.Scanner;

class GetInputFromUser
{
    public static void main(String args[])
    {
        // Using Scanner for Getting Input from User
        Scanner in = new Scanner(System.in);

        String s = in.nextLine();
        System.out.println("You entered string "+s);

        int a = in.nextInt();
        System.out.println("You entered integer "+a);

        float b = in.nextFloat();
        System.out.println("You entered float "+b); }}
```

3. Using Console Class

It has been becoming a preferred way for reading user's input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).

Advantages:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Drawback:

- Does not work in non-interactive environment (such as in an IDE).

```
// Java program to demonstrate working of System.console()
// Note that this program does not work on IDEs as
// System.console() may require console
public class Sample
{
    public static void main(String[] args)
    {
        // Using Console to input data from user
        String name = System.console().readLine();
        System.out.println(name);
    }
```

3.Identifiers in Java-

Java Identifiers

In programming languages, identifiers are used for identification purposes. In Java, an identifier can be a class name, method name, variable name, or label. For example :

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

```
    }
}
```

In the above java code, we have the following identifiers namely:

- **Test** : class name.
- **main** : method name.
- **a** : variable name.

Rules for defining Java Identifiers

There are certain rules for defining a valid Java identifier. These rules must be followed; otherwise, we get a compile-time error. These rules are also valid for other languages such as C, C++.

- The only allowed characters for identifiers are all alphanumeric characters([**A-Z**],[**a-z**],[**0-9**]), '**$**'(dollar sign), and '**_**' (underscore).For example "geek@" is not a valid java identifier as it contain '@', which is a special character.
- Identifiers should **not** start with digits(**[0-9]**). For example "123geeks" is a not a valid java identifier.
- Java identifiers are **case-sensitive**.
- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 - 15 letters only.
- **Reserved Words** can't be used as an identifier. For example "int while = 20;" is an invalid statement as while is a reserved word. There are **53** reserved words in Java.

Examples of valid identifiers :

```
MyVariable
MYVARIABLE
myvariable
x
i
x1
i1
_myvariable
$myvariable
```

```
sum_of_array
geeks123
```

Examples of invalid identifiers :

```
My Variable  // contains a space
123geeks   // Begins with a digit
a+c // plus sign is not an alphanumeric character
variable-2 // hyphen is not an alphanumeric character
sum_&_difference // ampersand is not an alphanumeric chara
cter
```

Reserved Words

Any programming language reserves some words to represent functionalities defined by that language. These words are called reserved words. They can be briefly categorised into two parts: **keywords**(50) and **literals**(3).

keywords define functionalities, and literals define a value.

```
MyVariable
MYVARIABLE
myvariable
x
i
x1
i1
_myvariable
$myvariable
sum_of_array
```

Identifiers are used by symbol tables in various analyzing phases(like lexical, syntax, semantics) of a compiler architecture.

Note : The keywords const and goto are reserved, even though they are not currently used. In place of const, the final keyword is used. Some keywords like strictfp are included in later versions of Java.

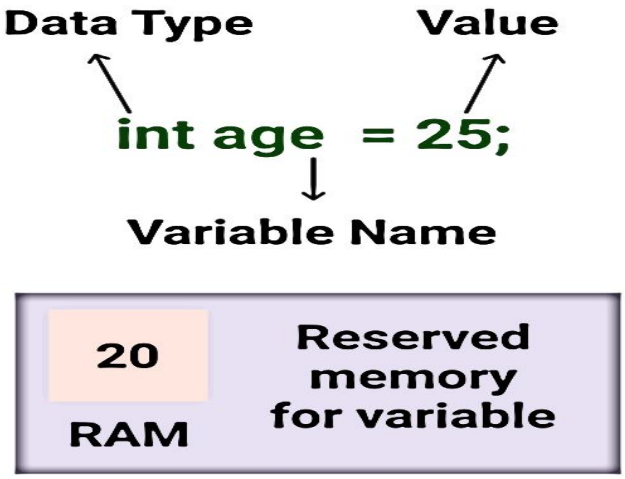
4.Variables in Java-

A **variable** is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.

How to declare variables?

We can declare variables in java as follows:



- **Data Type**: Type of data that can be stored in this variable.
- **Variable Name**: Name given to the variable.
- **Value**: It is the initial value stored in the variable.

Examples:

```
//Declaring float variable
float simpleInterest;
//Declaring and Initializing integer variable
int time = 10, speed = 20;
```

```
// Declaring and Initializing character variable
char var = 'h';
```

Types of variables

There are three types of variables in Java:

- Local Variables
- Instance Variables
- Static Variables

Let us now learn about each one of these variables in detail.

1. **Local Variables**: A variable defined within a block or method or constructor is called local variable.
    - These variables are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
    - The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
    - Initilisation of local variable is mandatory.

```
public class StudentDetails

{

   public void StudentAge()

   {  //local variable age

      int age = 0;

      age = age + 5;

      System.out.println("Student age is : " + age);

   }

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

```
    {
        StudentDetails obj = new StudentDetails();
        obj.StudentAge();
    }
}
```

Output:

```
Student age is : 5
```

In the above program the variable age is a local variable in the function StudentAge(). If we use the variable age outside StudentAge() function, the compiler will produce an error as shown in below program.

Sample Program 2:

```
public class StudentDetails
{
    public void StudentAge()
    {   //local variable age
        int age = 0;
        age = age + 5;
    }

    public static void main(String args[])
    {
        //using local variable age outside it's scope
        System.out.println("Student age is : " + age);
    }
}
```

Output:

```
error: cannot find symbol
 " + age);
```

1. **Instance Variables**: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

    - As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
    - Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
    - Initilisation of Instance Variable is not mandatory. Its default value is 0
    - Instance Variable can be accessed only by creating objects.

Sample Program:

```
import java.io.*;

class Marks

{

   //These variables are instance
variables.

   //These variables are in a class and
are not inside any function

   int engMarks;

   int mathsMarks;

   int phyMarks;

}

class MarksDemo

{

   public static void main(String args[])

   {  //first object

      Marks obj1 = new Marks();

      obj1.engMarks = 50;

      obj1.mathsMarks = 80;

      obj1.phyMarks = 90;

      //second object

      Marks obj2 = new Marks();

      obj2.engMarks = 80;

      obj2.mathsMarks = 60;
```

```
        obj2.phyMarks = 85;

        //displaying marks for first object

        System.out.println("Marks for
first object:");


System.out.println(obj1.engMarks);


System.out.println(obj1.mathsMarks);


System.out.println(obj1.phyMarks);

        //displaying marks for second
object

        System.out.println("Marks for
second object:");


System.out.println(obj2.engMarks);


System.out.println(obj2.mathsMarks);


System.out.println(obj2.phyMarks);

    }

}
```

Output:

```
Marks for first object:
50
80
90
Marks for second object:
80
60
85
```

As you can see in the above program the variables, *engMarks* , *mathsMarks* , *phyMarks* are instance variables. In case we have multiple objects as in the above program, each object will have its own copies of instance variables. It is clear from the above output that each object will have its own copy of instance variable.

1. **Static Variables**: Static variables are also known as Class variables.

    - These variables are declared similarly as instance variables, the difference is that static variables are declared using the

static keyword within a class outside any method, constructor, or block.

- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at the start of the program execution and are destroyed automatically when the execution ends.
- Initilisation of Static Variable is not mandatory. Its default value is 0
- If we access the static variable like Instance variable (through object), compiler will show the warning message and it wont hault the program. Compiler will replace the object name to class name automatically.
- If we access the static variable without classname, the compiler will automatically append the class name.

To access static variables, we need not create any object of that class, we can simply access the variable as:

```
class_name.variable_name;
```

Sample Program:

```
import java.io.*;

class Article {

  // static variable salary

  public static double rating;

  public static String name = "Variables Declaration";

}

public class ArticleDemo

{

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

```
        //accessing static variable without object
        Article.rating = 4.5;
        System.out.println(Article.name + "'s  rating:" + Article.rating);
    }
}
```

output:

```
Variables Declaration's rating:4.5
```

Instance variable Vs Static variable

- Each object will have its **own copy** of instance variable whereas We can only have **one copy** of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of instance variable. In case of static variable, changes **will be reflected** in other objects as static variables are common to all object of a class.
- We can access instance variables **through object references** and Static Variables can be accessed **directly using class name.**
- Syntax for static and instance variables:

```
class Example
    {
        static int a; //static variable
        int b;        //instance variable
    }
```

5.Data Types in Java-

Java Data Types

There are majorly two types of languages. The first one is **Statically typed language** where each variable and expression type is already known at compile time. Once a variable is declared to be of a certain data type, it cannot hold values of other data types. Example: C,C++, Java. The Other one is **Dynamically typed languages:** These languages can receive different data types over time. Example: Ruby, Python.

Java is a **statically typed and also a strongly typed language** because in Java each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.

Java has two categories of data:

- Primitive data (e.g., number, character)
- Object data (programmer created types)

Primitive data

Primitive data are only single values; they have no special capabilities. There are 8 primitive data types

Type	Description	Default	Size	Examples
boolean	true or false	false	1 bit	true, false
byte	two's complement integer	0	8 bits	(none)
char	unicode character	\u0000	16 bits	'a','\n','\u0041'

short	two's complement integer	0	16 bits	(none)
int	two's complement integer	0	32 bits	-1,0,1
long	two's complement integer	0	64 bits	-1L,0L,1L
float	IEEE 754 floating point	0.0	32 bits	-1.23e100f,2.45e100f
double	IEEE 754 floating point	0.0	64 bits	-3.45e300d,4.56e245d

Let us look at each of these types in detail:

- **boolean:** boolean data type represents only one bit of information, **either true or false** . Values of type boolean are not converted implicitly or explicitly (with casts) to any other type; however, the programmer can easily write the conversion code.

```
// A Java program to demonstrate boolean data type

Class DEMO

{

   public static void main(String args[])

   {

      boolean b = true;

      if (b == true)
```

```
            System.out.println("Hi DEMO");
        }
    }
```

- **Output**:

```
Hi DEMO
```

- **byte:** The byte data type is an 8-bit signed two's complement integer. The byte data type is useful for saving memory in large arrays.
    - Size: 8-bit
    - Value: -128 to 127

```
// Java program to demonstrate byte data type in Java
class DEMO
{
   public static void main(String args[])
   {
      byte a = 126;
      // byte is 8 bit value
      System.out.println(a);
      a++;
      System.out.println(a);
      // It overflows here because
      // byte can hold values from -128 to 127
      a++;
      System.out.println(a);
```

```
        // Looping back within the range
        a++;
        System.out.println(a);
    }
}
```

Output:

```
126
127
-128
-127
```

- **short:** The short data type is a 16-bit signed two's complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.
    - **Size:** 16 bit
    - **Value:** -32,768 to 32,767 (inclusive)
- **int**: It is a 32-bit signed two's complement integer.
    - **Size:** 32 bit
    - **Value:** -2^{31} to $2^{31}-1$

  Note: In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has value in the range [0, $2^{32}-1$]. Use the Integer class to use int data type as an unsigned integer.

- **long:** The long data type is a 64-bit two's complement integer.
    - Size: 64 bit
    - Value: -2^{63} to $2^{63}-1$.

Note: In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2^{64}-1. The Long class also contains methods like compareUnsigned, divideUnsigned etc., to support arithmetic operations for unsigned long.

- **Floating point Numbers : float and double**
  float: The float data type is a single-precision 32-bit IEEE 754 floating point. Use a float (instead of double) if you need to save memory in large arrays of floating point numbers.

    - **Size:** 32 bits
    - **Suffix :** F/f Example: 9.8f

  double: The double data type is a double-precision 64-bit IEEE 754 floating point. For decimal values, this data type is generally the default choice.

  Note: Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable. If accuracy is the most prior concern then, it is recommended not to use these data types and use BigDecimal class instead.

- **char**: The char data type is a single 16-bit Unicode character. A char is a single character.

    - Value: '\u0000' (or 0) to '\uffff' 65535

// Java program to demonstrate primitive data types in Java

```
Class DEMO
{
   public static void main(String args[])
   {
      // declaring character
       char a = 'G';
      // Integer data type is generally
      // used for numeric values
      int i=89;
      // use byte and short if memory is a
constraint
      byte b = 4;
      // this will give error as number is
      // larger than byte range
      // byte b1 = 7888888955;
      short s = 56;
      // this will give error as number is
      // larger than short range
      // short s1 = 87878787878;
      // by default fraction value is double in
java
      double d = 4.355453532;

      // for float use 'f' as suffix
      float f = 4.7333434f;
      System.out.println("char: " + a);
      System.out.println("integer: " + i);
      System.out.println("byte: " + b);
      System.out.println("short: " + s);
      System.out.println("float: " + f);
      System.out.println("double: " + d);
   }
}
```

Output:

```
char: G
integer: 89
byte: 4
short: 56
float: 4.7333436
double: 4.355453532
```

6.Operators in Java-

Java provides many types of operators, which can be used according to the need. They are classified based on the functionality they provide. Mostly used operators are discussed below :

1. **Arithmetic Operators:** They are used to perform simple arithmetic operations on primitive data types.
   - ***** Multiplication
   - **/ :** Division
   - **% :** Modulo
   - **+ :** Addition
   - **- :** Subtraction

```
// Java program to illustrate

// arithmetic operators

public class operators

{

   public static void main(String[] args)

   {

      int a = 20, b = 10, c = 0, d = 20, e = 40, f =
30;

      String x = "Thank", y = "You";

      // + and - operator

      System.out.println("a + b = "+(a + b));

      System.out.println("a - b = "+(a - b));

      // + operator if used with strings

      // concatenates the given strings.

      System.out.println("x + y = "+x + y);

      // * and / operator

      System.out.println("a * b = "+(a * b));

      System.out.println("a / b = "+(a / b));

      // modulo operator gives remainder

      // on dividing first operand with second

      System.out.println("a % b = "+(a % b));

      // if denominator is 0 in division

      // then Arithmetic exception is thrown.

      // uncommenting below line would throw

      // an exception

      // System.out.println(a/c);

   }

}
```

Output:

```
a+b = 30
a-b = 10
x+y = ThankYou
a*b = 200
a/b = 2
a%b = 0
```

Unary Operators: Unary operators need only one operand. They are used to increment, decrement, or negate a value.

- **- :Unary minus** is used for negating the values.
- **+ :Unary plus** is used for giving positive values. Only used when deliberately converting a negative value to a positive value.
- **++ :Increment operator** is used for incrementing the value by 1. There are two varieties of increment operator.
  - **Post-Increment :** Value is first used for computing the result, and then it is incremented.
  - **Pre-Increment :** Value is incremented first, and then the result is computed.
- **-- : Decrement operator** is used for decrementing the value by 1. There are two varieties of decrement operator.
  - **Post-decrement :** Value is first used for computing the result, and then it is decremented.
  - **Pre-Decrement :** Value is decremented first, and then the result is computed.
- **! : Logical not operator** is used for inverting a boolean value.

// Java program to illustrate

// unary operators

```
public class operators

{

   public static void main(String[] args)

   {

      int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;

      boolean condition = true;
```

```
        // pre-increment operator

        // a = a+1 and then c = a;

        c = ++a;

        System.out.println("Value of c (++a) = " + c);

        // post increment operator

        // c=b then b=b+1

        c = b++;

        System.out.println("Value of c (b++) = " + c);

        // pre-decrement operator

        // d=d-1 then c=d

        c = --d;

        System.out.println("Value of c (--d) = " + c);

        // post-decrement operator

        // c=e then e=e-1

        c = e--;

        System.out.println("Value of c (e--) = " + c);

        // Logical not operator

        System.out.println("Value of !condition =" + !condition);

    }

}
```

Output:

```
Value of c (++a) = 21
Value of c (b++) = 10
Value of c (--d) = 19
Value of c (e--) = 40
Value of !condition =false
```

1. **Assignment Operator : '='** Assignment operator is used to assign a value to any variable. It has a right to left associativity, i.e., the value given on the right hand side of the operator is assigned to the variable on the left; therefore, right hand side value must be declared before using it or it should be a constant.
   General format of assignment operator is,

   ```
   variable = value;
   ```

   In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a = a+5 , we can write a **+=** 5.

   - **+=**, for adding left operand with right operand and then assigning it to variable on the left.
   - **-=**, for subtracting left operand with right operand and then assigning it to variable on the left.
   - ***=**, for multiplying left operand with right operand and then assigning it to variable on the left.
   - **/=**, for dividing left operand with right operand and then assigning it to variable on the left.
   - **%=**, for assigning modulo of left operand with right operand and then assigning it to variable on the left.

   ```
   int a = 5;
   a += 5; //a = a+5;
   ```

// Java program to illustrate

// assignment operators

```
public class operators

{

    public static void main(String[] args)

    {

        int a = 20, b = 10, c, d, e = 10, f = 4, g = 9;

        // simple assignment operator

        c = b;
```

```
System.out.println("Value of c = " + c);

// This following statement would throw an exception

// as value of right operand must be initialised

// before assignment, and the program would not

// compile.

// c = d;

// instead of below statements,shorthand

// assignment operators can be used to

// provide same functionality.

a = a + 1;

b = b - 1;

e = e * 2;

f = f / 2;

System.out.println("a,b,e,f = " + a + ","   + b + "," + e + "," + f);

a = a - 1;

b = b + 1;

e = e / 2;

f = f * 2;

// shorthand assignment operator

a += 1;

b -= 1;

e *= 2;

f /= 2;
```

```
        System.out.println("a,b,e,f (using shorthand operators)= " +
                           a + "," + b + "," + e + "," + f);
    }
}
```

Output :

```
Value of c =10
a,b,e,f = 21,9,20,2
a,b,e,f (using shorthand operators)= 21,9,20,2
```

1. **Relational Operators :** These operators are used to check for relations such as equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as in conditional if else statements. General format is,

   ```
   variable relation_operator value
   ```

   Some of the relational operators are:

   - **== , Equal to :** returns true of left hand side is equal to right hand side.
   - **!= , Not Equal to :** returns true of left hand side is not equal to right hand side.
   - **< , less than :** returns true of left hand side is less than right hand side.
   - **<= , less than or equal to :** returns true of left hand side is less than or equal to right hand side.
   - **> , Greater than :** returns true of left hand side is greater than right hand side.
   - **>= , Greater than or equal to:** returns true of left hand side is greater than or equal to right hand side.

// Java program to illustrate

// relational operators

```
public class operators                                  {
{                                                         int a = 20, b = 10;
   public static void main(String[] args)                 String x = "Thank", y = "Thank";
```

```
        int ar[] = { 1, 2, 3 };

        int br[] = { 1, 2, 3 };

        boolean condition = true;

        //various conditional 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));

        System.out.println("a != b :" + (a
!= b));

        // Arrays cannot be compared
with

        // relational operators because
objects

        // store references not the value

        System.out.println("x == y : " + (ar
== br));



System.out.println("condition==true :"
+ (condition == true));

    }

}
```

Output :

```
a == b :false
a < b :false
a <= b :false
a > b :true
a >= b :true
a != b :true
x == y : false
condition==true :true
```

Logical Operators : These operators are used to perform "logical AND" and "logical OR" operations, i.e., the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is that the second condition is not evaluated if the first one is false, i.e., it has short-circuiting effect. It is used extensively to test for several conditions for making a decision.
Conditional operators are:

- **&& , Logical AND :** returns true when both conditions are true.
- **|| , Logical OR :** returns true if at least one condition is true.

// Java program to illustrate

// logical operators

```
public class operators

{

   public static void main(String[] args)

   {

      String x = "Sher";

      String y = "Locked";

      Scanner s = new
Scanner(System.in);

      System.out.print("Enter
username:");

      String uuid = s.next();

      System.out.print("Enter
password:");

      String upwd = s.next();

      // Check if user-name and
password match or not.

      if ((uuid.equals(x) &&
upwd.equals(y)) ||

            (uuid.equals(y) &&
upwd.equals(x))) {

         System.out.println("Welcome
user.");

      } else {

         System.out.println("Wrong uid
or password");

      }

   }

}
```

Output :

```
Enter username:Sher
Enter password:Locked
Welcome user.
```

Ternary operator : Ternary operator is a shorthand version of if-else statement. It has three operands, and hence the name ternary. General format is:

```
condition ? if true : if false
```

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

```
// Java program to illustrate

// max of three numbers using
```

```
// ternary operator.

public class operators

{

   public static void main(String[] args)

   {

      int a = 20, b = 10, c = 30, result;

      //result holds max of three

      //numbers

      result = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);

      System.out.println("Max of three numbers = "+result);

   }

}
```

Output :

```
Max of three numbers = 30
```

Bitwise Operators : These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.

- **& , Bitwise AND operator:** returns bit by bit AND of input values.
- **| , Bitwise OR operator:** returns bit by bit OR of input values.
- **^ , Bitwise XOR operator:** returns bit by bit XOR of input values.
- **~ , Bitwise Complement Operator:** This is a unary operator which returns the one's compliment representation of the input value, i.e., with all bits inversed.

```
// Java program to illustrate
// bitwise operators
public class operators
{
   public static void main(String[] args)
   {
//if int a = 010 java considers it as
octal value of 8 as number starts with
0.
      int a = 0x0005;
      int b = 0x0007;
      // bitwise and
      // 0101 & 0111=0101
      System.out.println("a&b = " + (a &
b));
      // bitwise and
      // 0101 | 0111=0111
      System.out.println("a|b = " + (a |
b));
      // bitwise xor
      // 0101 ^ 0111=0010
      System.out.println("a^b = " + (a ^
b));
      // bitwise and
      // ~0101=1010
      System.out.println("~a = " + ~a);

      // can also be combined with
      // assignment operator to provide
shorthand
      // assignment
      // a=a&b
      a &= b;
      System.out.println("a= " + a);
  } }
```

Output :

```
a&b = 5
a|b = 7
a^b = 2
~a = -6
a= 5
```

Shift Operators : These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a

number by two. General format-

```
number shift_op number_of_places_to_shift;
```

- **<< , Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
- **>> , Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
- **>>> , Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

```
// Java program to illustrate
// shift operators
public class operators
{
    public static void main(String[] args)
    {
        int a = 0x0005;
        int b = -10;
        // left shift operator
        // 0000 0101<<2 =0001 0100(20)
        // similar to 5*(2^2)
        System.out.println("a<<2 = " + (a << 2));
        // right shift operator
        // 0000 0101 >> 2 =0000 0001(1)
        // similar to 5/(2^2)
```

```
        System.out.println("a>>2 = " + (a >> 2));

        // unsigned right shift operator

        System.out.println("b>>>2 = "+ (b >>> 2));

    }

}
```

Output :

```
a2 = 1
b>>>2 = 1073741821
```

Precedence and Associativity of Operators

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude with the top representing the highest precedence and the bottom shows the lowest precedence.

Operators	Associativity	Type
++,--	Right to Left	Unary Postfix
++,--,+,-,!	Right to Left	Unary Prefix
/,*,%	Left to Right	Multiplicative
+,-	Left to Right	Additive
<,<=,>,>=	Left to Right	Relational
==,!=	Left to Right	Equality
/,*,%	Left to Right	Multiplicative
&	Left to Right	Boolean Logical AND
^	Left to Right	Boolean Logical XOR
|	Left to Right	Boolean Logical OR

&&	Left to Right	Conditional AND
||	Left to Right	Conditional OR
?:	Right to Left	Conditional
=,+=,-=,*=,/=,%=	Right to Left	Assignment

Interesting Questions on Operators

1. **Precedence and Associativity:** There is often confusion when it comes to hybrid equations i.e, the equations having multiple operators. The problem is, which part to solve first. There is a golden rule to follow in these situations. If the operators have different precedence, solve the higher precedence first. If they have the same precedence, solve according to associativity, that is either from right to left or from left to right. Explanation of the below program is well written in comments within the program itself.

```
public class operators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
      // precedence rules for arithmetic operators.
      // (* = / = %) > (+ = -)
      // prints a+(b/d)
      System.out.println("a+b/d = "+(a + b / d));
      // if same precendence then associative
      // rules are followed.
      // e/f -> b*d -> a+(b*d) -> a+(b*d)-(e/f)
      System.out.println("a+b*d-e/f = "+(a + b * d - e / f));
   }
}
```

Output:

```
a+b/d = 20
a+b*d-e/f = 219
```

Be a Compiler: Compiler in our systems uses lex tool to match the greatest match when generating tokens. This creates a bit of problem if overlooked. For example, consider the statement **a=b+++c;**, to many of the readers this might seem to create compiler error. But this statement is absolutely correct as the tokens created by lex

are a, =, b, ++, +, c. Therefore this statement has similar effect of first assigning b+c to a and then incrementing b. Similarly, a=b+++++c; would generate error as tokens generated are a, =, b, ++, ++, +, c. which is actually an error as there is no operand after second unary operand.

```
public class operators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10, c = 0;
      // a=b+++c is compiled as
      // b++ +c
      // a=b+c then b=b+1
      a = b+++c;
      System.out.println("Value of a(b+c),b(b+1),c = " + a + "," + b + "," + c);
      // a=b+++++c is compiled as
      // b++ ++ +c
      // which gives error.
      // a=b+++++c;
      // System.out.println(b+++++c);
   }
}
```

Output:

```
Value of a(b+c),b(b+1),c = 10,11,0
```

Using + over (): When using + operator inside system.out.println() make sure to do addition using parenthesis. If we write something before doing addition, then string addition takes place, that is associativity of addition is left to right, and hence integers are added to string first producing a string, and string objects concatenates when using +, therefore it can create unwanted results.

```
public class operators
{
   public static void main(String[] args)
   {
      int x = 5, y = 8;
      // concatenates x and y
```

```
        // as first x is added to "concatenation (x+y) = "
        // producing "concatenation (x+y) = 5" and then 8 is
        // further concatenated.
        System.out.println("Concatenation (x+y)= " + x + y);
        // addition of x and y
        System.out.println("Addition (x+y) = " + (x + y));
    }
}
```

Output:

```
Concatenation (x+y)= 58
Addition (x+y) = 13
```

7.Java Sample Problems | Operators, Data Type, Input/Output-

The following are some basic implementation problems covering the topics discussed until now.

- **Problem 1 :** Change the case of the character entered. (**using operators only**).
- **Problem 2 :** Write a program to convert temperature given in Celsius (user input) to Fahrenheit.
- **Problem 3 :** Write a program to find the area of a triangle. Take the length of sides as user input. (Area printed should be rounded off to two decimal places).
- **Problem 4 :** Take user input amount of money and consider an infinite supply of denominations 1, 20, 50 and 100. What is the minimum number of denominations to make the change?
- **CHECK IMPLEMENTATION ON GOOGLE**

8.Decision Control Statements (If,Else,Switch,Break,Continue) in Java-

Decision Making in programming is similar to decision making in real life. In programming too, we face some situations where we want a certain block of code to be executed when some condition is fulfilled.
A programming language uses control statements to control the flow of execution of the program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.
Java's Selection statements:

- **if**
- **if-else**
- **nested-if**
- **if-else-if**
- **switch-case**
- **jump - break, continue, return**

These statements allow you to control the flow of your program's execution based upon conditions known only during the run time.

If : if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e., if a certain condition is true, then a block of statement is executed otherwise not.
Syntax:

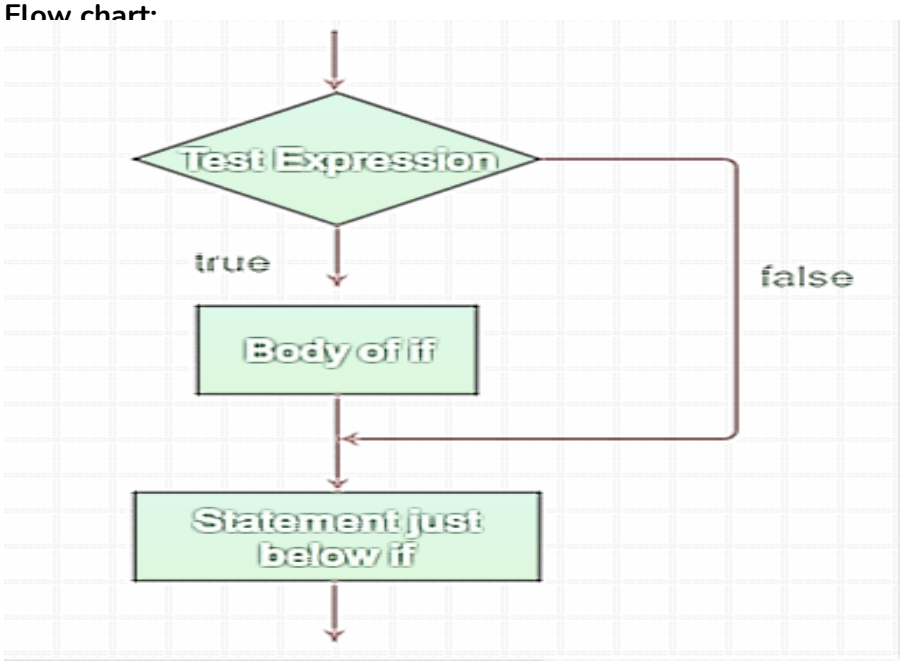
```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

Here, **condition** after evaluation will be either true or false. if statement accepts boolean values - if the value is true, then it will execute the block of statements under it.
If we do not provide the curly braces '{' and '}' after **if(condition)**, then by default if statement will consider the immediate one statement to be inside its block. For example,

```
if(condition)
   statement1;
   statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

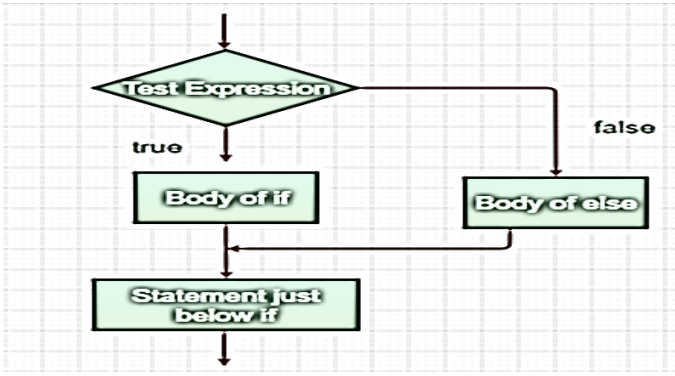
Flow chart:



if-else: The if statement alone tells us that if a condition is true it will execute a block of statements, and if the condition is false it won't. But what if we want to do something else when the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```



if-else-if ladder: Here, a user can decide among multiple options. The if statements are executed from the top to bottom. As soon as one of the conditions controlling

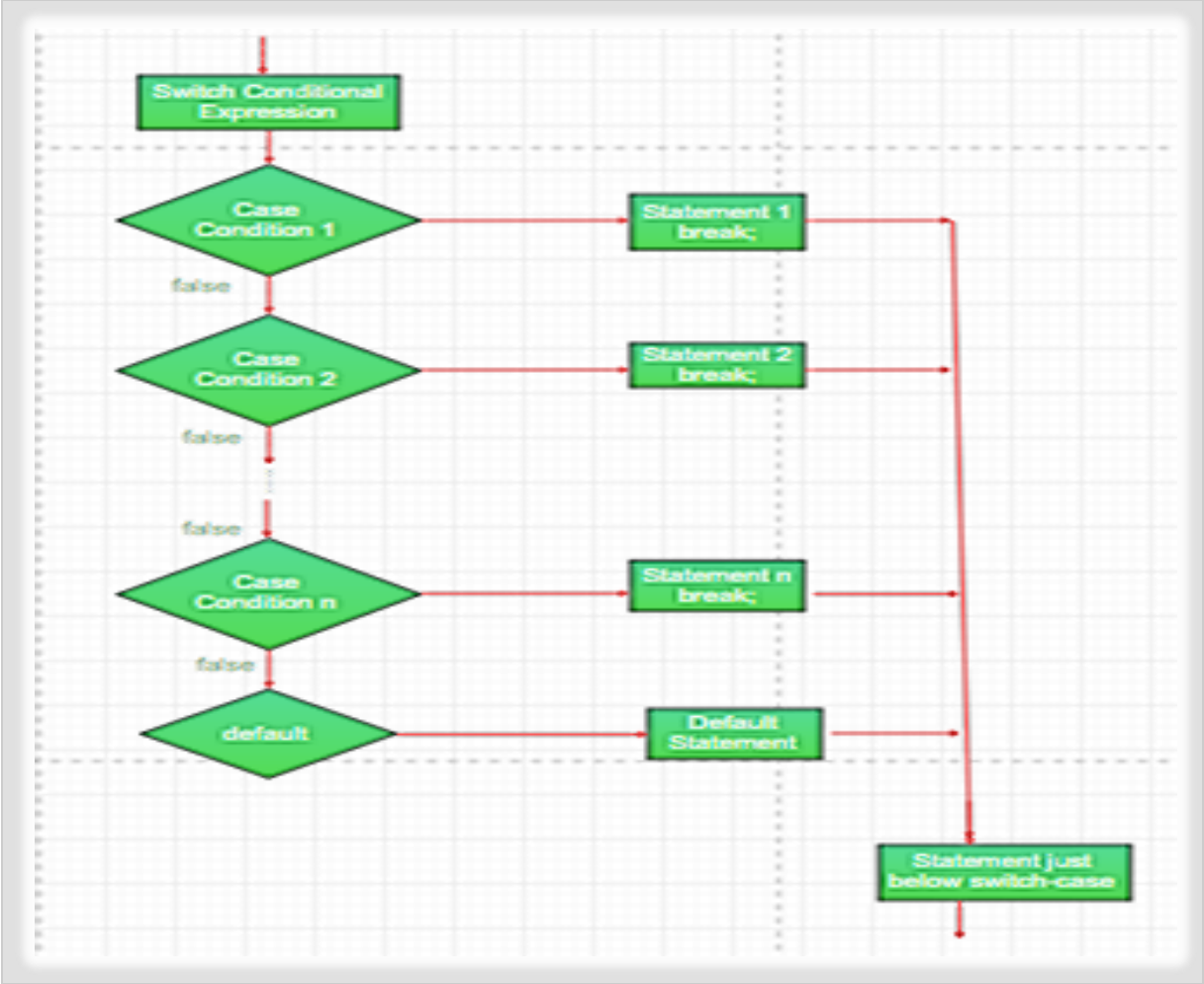
the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```
if (condition)
    statement;
else if (condition)
    statement;
else
    statement;
```

switch-case: The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
Syntax:

```
switch (expression)
{
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;
  .
  .
  case valueN:
    statementN;
    break;
  default:
    statementDefault;
}
```

- Expression can be of type byte, short, int char or an enumeration. Beginning with JDK7, *expression* can also be of type String.
- Dulplicate case values are not allowed.
- The default statement is optional.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If omitted, execution will continue on into the next case.



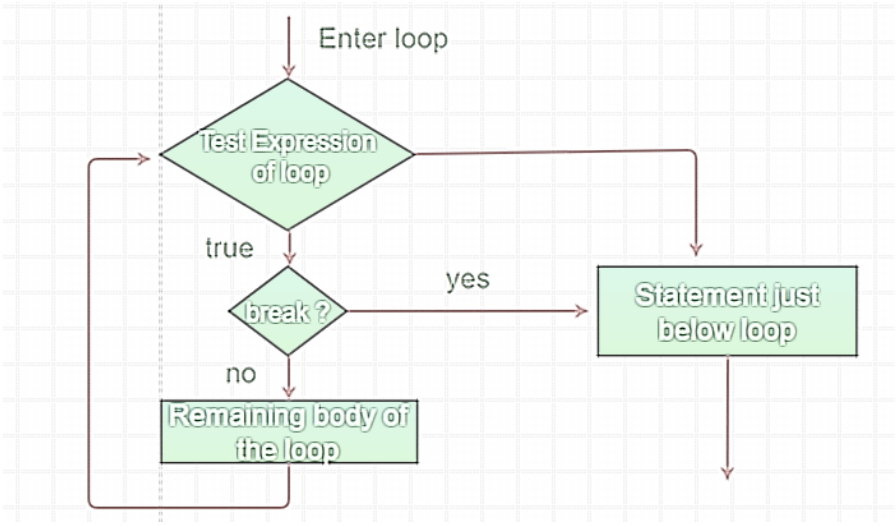
jump: J ava supports three jump statement: **break, continue**, and **return**. These three statements transfer control to other part of the program.

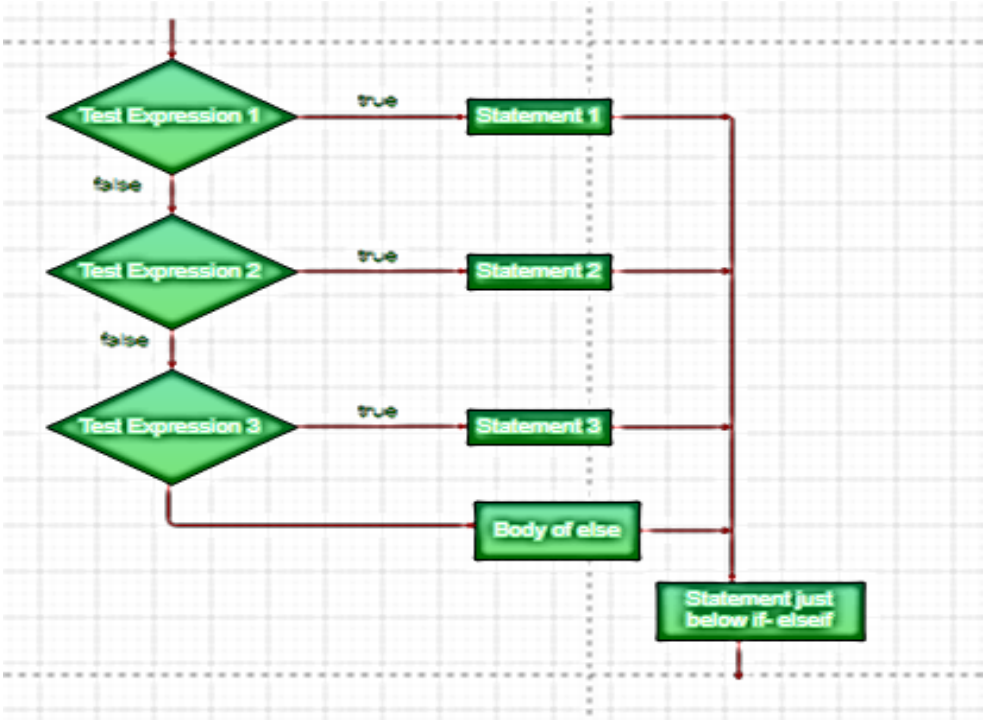
1. **Break:** In Java, break is majorly used for the following:
    - To terminate a sequence in a switch statement (discussed above).
    - To exit a loop.
    - Used as a "civilized" form of goto.

Using break to exit a Loop

Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
Note: Break, when used inside a set of nested loops, will only break out of

the innermost loop.





Using break as a Form of Goto

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. Java uses label. A Label is use to identifies a block of code.

Syntax:

```
label:
{
  statement1;
  statement2;
  statement3;
  .
```

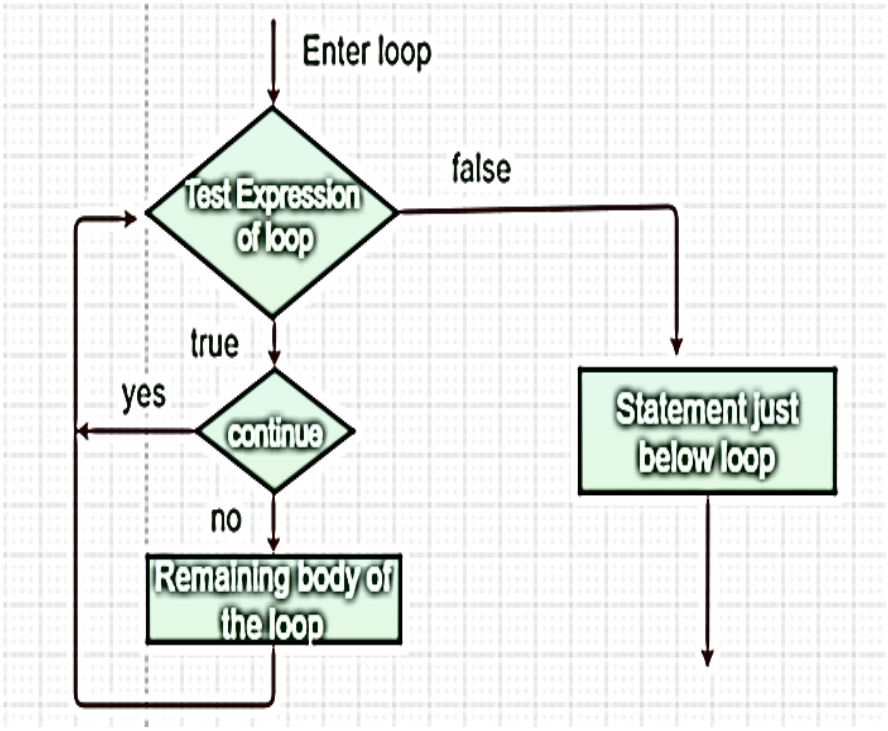
```
    .
}
```

Now, break statement can be use to jump out of target block.
Note: You cannot break to any label which is not defined for an enclosing block.
Syntax:

```
break label;
```

Continue:

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.



Return:

The return statement is used to explicitly return from a method. That is, it causes a program control to transfer back to the caller of the method.

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

CREDITS- INTERNET

DISCLOSURE- ALL THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.