# Java Operators with Examples

**Operators** constitute the basic building block to any programming language. Java too provides many types of operators which can be used according to the need to perform various calculations and functions, be it logical, arithmetic, relational, etc. They are classified based on the functionality they provide. Here are a few types:

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators

This article explains all that one needs to know regarding Arithmetic Operators.

## Arithmetic Operators

These operators involve the mathematical operators that can be used to perform various simple or advanced arithmetic operations on the primitive data types referred to as the operands. These operators consist of various unary and binary operators that can be applied on a single or two operands. Let's look at the various operators that Java has to provide under the arithmetic operators.

| Operators | Result |
|---|---|
| + | Addition of two numbers |
| - | Subtraction of two numbers |
| * | Multiplication of two numbers |
| / | Division of two numbers |
| % | (Modulus Operator)Divides two numbers and returns the remainder |

Now let's look at each one of the arithmetic operators in Java:

**1. Addition(+):** This operator is a binary operator and is used to add two operands.
**Syntax:**

```
num1 + num2
```

**Example:**

```
num1 = 10, num2 = 20
```

```
sum = num1 + num2 = 30
```

- Java

```java
// Java code to illustrate Addition operator


import java.io.*;


class Addition {
    public static void main(String[] args)
    {
        // initializing variables
        int num1 = 10, num2 = 20, sum = 0;


        // Displaying num1 and num2
        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);


        // adding num1 and num2
        sum = num1 + num2;
        System.out.println("The sum = " + sum);
    }
}
```

**Output**

```
num1 = 10
num2 = 20
The sum = 30
```

**2. Subtraction(-):** This operator is a binary operator and is used to subtract two operands.

**Syntax:**

```
num1 - num2
```

**Example:**

```
num1 = 20, num2 = 10
sub = num1 - num2 = 10
```

- Java

```java
// Java code to illustrate Subtraction operator


import java.io.*;


class Subtraction {

    public static void main(String[] args)

    {

        // initializing variables

        int num1 = 20, num2 = 10, sub = 0;


        // Displaying num1 and num2

        System.out.println("num1 = " + num1);

        System.out.println("num2 = " + num2);


        // subtracting num1 and num2

        sub = num1 - num2;

        System.out.println("Subtraction = " + sub);

    }

}
```

**Output**

```
num1 = 20

num2 = 10

Subtraction = 10
```

**3. Multiplication(*):** This operator is a binary operator and is used to multiply two operands.

**Syntax:**

```
num1 * num2
```

**Example:**

```
num1 = 20, num2 = 10

mult = num1 * num2 = 200
```

- Java

```java
// Java code to illustrate Multiplication operator
```

```java
import java.io.*;


class Multiplication {

    public static void main(String[] args)

    {

        // initializing variables

        int num1 = 20, num2 = 10, mult = 0;


        // Displaying num1 and num2

        System.out.println("num1 = " + num1);

        System.out.println("num2 = " + num2);


        // Multiplying num1 and num2

        mult = num1 * num2;

        System.out.println("Multiplication = " + mult);

    }

}
```

**Output**

```
num1 = 20

num2 = 10

Multiplication = 200
```

**4. Division(/):** This is a binary operator that is used to divide the first operand(dividend) by the second operand(divisor) and give the quotient as a result.

**Syntax:**

```
num1 / num2
```

**Example:**

```
num1 = 20, num2 = 10

div = num1 / num2 = 2
```

- Java

```java
// Java code to illustrate Division operator


import java.io.*;
```

```java
class Division {

    public static void main(String[] args)

    {

        // initializing variables

        int num1 = 20, num2 = 10, div = 0;


        // Displaying num1 and num2

        System.out.println("num1 = " + num1);

        System.out.println("num2 = " + num2);


        // Dividing num1 and num2

        div = num1 / num2;

        System.out.println("Division = " + div);

    }

}
```

**Output**

```
num1 = 20

num2 = 10

Division = 2
```

**5. Modulus(%):** This is a binary operator that is used to return the remainder when the first operand(dividend) is divided by the second operand(divisor).

**Syntax:**

```
num1 % num2
```

**Example:**

```
num1 = 5, num2 = 2

mod = num1 % num2 = 1
```

- Java

```java
// Java code to illustrate Modulus operator


import java.io.*;


class Modulus {

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

```
    {
        // initializing variables
        int num1 = 5, num2 = 2, mod = 0;

        // Displaying num1 and num2
        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Remaindering num1 and num2
        mod = num1 % num2;
        System.out.println("Remainder = " + mod);
    }
}
```

**Output**

```
num1 = 5
num2 = 2
Remainder = 1
```

Here is an example program in Java that implements all basic arithmetic operators for user input:

- Java

```
import java.util.Scanner;

public class ArithmeticOperators {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the first number: ");
        double num1 = sc.nextDouble();

        System.out.print("Enter the second number: ");
        double num2 = sc.nextDouble();
```

```
        double sum = num1 + num2;
        double difference = num1 - num2;
        double product = num1 * num2;
        double quotient = num1 / num2;

        System.out.println("The sum of the two numbers is: " + sum);
        System.out.println("The difference of the two numbers is: " + difference);
        System.out.println("The product of the two numbers is: " + product);
        System.out.println("The quotient of the two numbers is: " + quotient);
    }
}
```

*Input*

```
Enter the first number: 20
Enter the second number: 10
```

*Output*

```
The sum of the two numbers is: 30.0
The difference of the two numbers is: 10.0
The product of the two numbers is: 200.0
The quotient of the two numbers is: 2.0
```

## Explanation

- The program implements basic arithmetic operations using user input in Java. The program uses the **Scanner class from the java.util package** to read user input from the console. The following steps describe how the program works in detail:
- **Import the java.util.Scanner class:** The program starts by importing the Scanner class, which is used to read input from the console.
- **Create a Scanner object**: Next, a Scanner object sc is created and associated with the standard input stream System.in.
- Read the first number from the user: The program prompts the user to enter the first number and uses the nextDouble() method of the Scanner class to read the input. The input is stored in the num1 variable of type double.
- **Read the second number from the user**: The program prompts the user to enter the second number and uses the nextDouble() method of the Scanner class to read the input. The input is stored in the num2 variable of type double.

- **Perform arithmetic operations:** The program performs the four basic arithmetic operations (addition, subtraction, multiplication, and division) using the num1 and num2 variables and stores the results in separate variables sum, difference, product, and quotient.
- **Print the results:** The program prints out the results of the arithmetic operations using the println method of the System.out object.
- **This program demonstrates how to implement basic arithmetic operations using user input in Java.** The Scanner class makes it easy to read user input from the console, and the basic arithmetic operations are performed using standard mathematical operators in Java.

Operators constitute the basic building block to any programming language. Java too provides many types of operators which can be used according to the need to perform various calculations and functions be it logical, arithmetic, relational, etc. They are classified based on the functionality they provide. Here are a few types:

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators

## Unary Operators in Java

Java unary operators are the types that need only one operand to perform any operation like increment, decrement, negation, etc. It consists of various arithmetic, logical and other operators that operate on a single operand. Let's look at the various unary operators in detail and see how they operate.

### Operator 1: Unary minus(-)

This operator can be used to convert a positive value to a negative one.

**Syntax:**

```
-(operand)
```

**Illustration:**

```
a = -10
```

**Example:**

- Java

```
// Java Program to Illustrate Unary - Operator


// Importing required classes
```

```java
import java.io.*;


// Main class
class GFG {


    // Main driver method
    public static void main(String[] args)
    {
        // Declaring a custom variable
        int n1 = 20;


        // Printing the above variable
        System.out.println("Number = " + n1);


        // Performing unary operation
        n1 = -n1;


        // Printing the above result number
        // after unary operation
        System.out.println("Result = " + n1);
    }
}
```

**Output**

```
Number = 20
Result = -20
```

## Operator 2: 'NOT' Operator(!)

This is used to convert true to false or vice versa. Basically, it reverses the logical state of an operand.

**Syntax:**

```
!(operand)
```

**Illustration:**

```
cond = !true;

// cond < false
```

**Example:**

- Java

```java
// Java Program to Illustrate Unary NOT Operator


// Importing required classes
import java.io.*;


// Main class
class GFG {


    // Main driver method
    public static void main(String[] args)
    {
        // Initializing variables
        boolean cond = true;
        int a = 10, b = 1;


        // Displaying values stored in above variables
        System.out.println("Cond is: " + cond);
        System.out.println("Var1 = " + a);
        System.out.println("Var2 = " + b);


        // Displaying values stored in above variables
        // after applying unary NOT operator
        System.out.println("Now cond is: " + !cond);
        System.out.println("!(a < b) = " + !(a < b));
        System.out.println("!(a > b) = " + !(a > b));
    }
}
```

**Output:**

```
Cond is: true
Var1 = 10
Var2 = 1
Now cond is: false
```

```
!(a < b) = true
!(a > b) = false
```

### Operator 3: Increment(++)

It is used to increment the value of an integer. It can be used in two separate ways:

**3.1:** Post-increment operator
When placed after the variable name, the value of the operand is incremented but the previous value is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

**Syntax:**

```
num++
```

**Illustration:**

```
num = 5
num++ = 6
```

**3.2:** Pre-increment operator
When placed before the variable name, the operand's value is incremented instantly.

**Syntax:**

```
++num
```

**Illustration:**

```
num = 5
++num = 6
```

### Operator 4: Decrement(–)

It is used to decrement the value of an integer. It can be used in two separate ways:

**4.1:** Post-decrement operator
When placed after the variable name, the value of the operand is decremented but the previous values is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

**Syntax:**

```
num--
```

**Illustration:**

```
num = 5
num-- = 5 // Value will be decremented before execution of next statement.
```

**4.2:** Pre-decrement operator
When placed before the variable name, the operand's value is decremented instantly.

**Syntax:**

```
--num
```

**Illustration:**

```
num = 5

--num = 5 //output is 5, value is decremented before execution of next
statement
```

## Operator 5: Bitwise Complement(~)

This unary operator returns the one's complement representation of the input value or operand, i.e, with all bits inverted, which means it makes every 0 to 1, and every 1 to 0.

**Syntax:**

```
~(operand)
```

**Illustration:**

```
a = 5 [0101 in Binary]

result = ~5


This performs a bitwise complement of 5

~0101 = 1010 = 10 (in decimal)


Then the compiler will give 2's complement

of that number.

2's complement of 10 will be -6.

result = -6
```

**Example:**

- Java

```java
// Java program to Illustrate Unary

// Bitwise Complement Operator


// Importing required classes

import java.io.*;


// Main class

class GFG {


    // Main driver method

    public static void main(String[] args)

    {

        // Declaring a variable
```

```
        int n1 = 6, n2 = -2;

        // Printing numbers on console
        System.out.println("First Number = " + n1);
        System.out.println("Second Number = " + n2);

        // Printing numbers on console after
        // performing bitwise complement
        System.out.println(
            n1 + "'s bitwise complement = " + ~n1);
        System.out.println(
            n2 + "'s bitwise complement = " + ~n2);
    }
}
```

**Output:**

```
First Number = 6
Second Number = -2
6's bitwise complement = -7
-2's bitwise complement = 1
```

Example program in Java that implements all basic unary operators for user input:

- Java

```
import java.util.Scanner;

public class UnaryOperators {
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        // fro user inputs here is the code.
```

```java
        // System.out.print("Enter a number: ");
        // int num = sc.nextInt();
        int num = 10;

        int result = +num;
        System.out.println(
            "The value of result after unary plus is: "
            + result);

        result = -num;
        System.out.println(
            "The value of result after unary minus is: "
            + result);

        result = ++num;
        System.out.println(
            "The value of result after pre-increment is: "
            + result);

        result = num++;
        System.out.println(
            "The value of result after post-increment is: "
            + result);

        result = --num;
        System.out.println(
            "The value of result after pre-decrement is: "
            + result);

        result = num--;
        System.out.println(
            "The value of result after post-decrement is: "
```

```
            + result);
    }
}
```

**Output**

```
The value of result after unary plus is: 10
The value of result after unary minus is: -10
The value of result after pre-increment is: 11
The value of result after post-increment is: 11
The value of result after pre-decrement is: 11
The value of result after post-decrement is: 11
```

## Explanation

The above program implements all basic unary operators in Java using user input. The program uses the Scanner class from the java.util package to read user input from the console. The following steps describe how the program works in detail:

- **Import the java.util.Scanner class:** The program starts by importing the Scanner class, which is used to read input from the console.
- **Create a Scanner object:** Next, a Scanner object sc is created and associated with the standard input stream System.in.
- **Read the number from the user:** The program prompts the user to enter a number and uses the nextInt() method of the Scanner class to read the input. The input is stored in the num variable of type int.
- **Use unary plus operator:** The program uses the unary plus operator + to perform a positive operation on num. The result of the operation is stored in the result variable of type int.
- **Use unary minus operator:** The program uses the unary minus operator – to perform a negative operation on num. The result of the operation is stored in the result variable.
- **Use pre-increment operator:** The program uses the pre-increment operator ++ to increment the value of num before using it in an expression. The result of the operation is stored in the result variable.
- **Use post-increment operator:** The program uses the post-increment operator ++ to increment the value of num after using it in an expression. The result of the operation is stored in the result variable.
- **Use pre-decrement operator:** The program uses the pre-decrement operator — to decrement the value of num before using it in an expression. The result of the operation is stored in the result variable.

- **Use post-decrement operator:** The program uses the post-decrement operator — to decrement the value of num after using it in an expression. The result of the operation is stored in the result variable.
- **Print the results**: The program prints out the final values of result using the println method of the System.out object after each operation.
- This program demonstrates how to use basic unary operators in Java. The Scanner class makes it easy to read user input from the console, and various unary operators are used to modify the value of the num variable in the program.

Adavantages

The main advantage of using unary operators in Java is that they provide a simple and efficient way to modify the value of a variable. Some specific advantages of using unary operators are:

1. **Concise and Easy to Use**: Unary operators are simple to use and require only one operand. They are easy to understand and make code more readable and concise.
2. **Faster than Other Operators**: Unary operators are faster than other operators as they only require one operand. This makes them ideal for operations that need to be performed quickly, such as incrementing a counter.
3. **Pre- and Post-Increment/Decrement**: Unary operators provide both pre- and post-increment and decrement options, which makes them useful for a variety of use cases. For example, the pre-increment operator can be used to increment the value of a variable before using it in an expression, while the post-increment operator can be used to increment the value of a variable after using it in an expression.
4. **Modifying Primitive Types:** Unary operators can be used to modify the value of primitive types such as int, long, float, double, etc.

Overall, unary operators provide a simple and efficient way to perform operations on variables in Java, and they can be used in a variety of scenarios to make code more readable and concise.

**Operators** constitute the basic building block of any programming language. Java too provides many types of operators which can be used according to the need to perform various calculations and functions, be it logical, arithmetic, relational, etc. They are classified based on the functionality they provide.

**Types of Operators:**

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators

6. Ternary Operator
7. Bitwise Operators
8. Shift Operators

This article explains all that one needs to know regarding Assignment Operators.

## Assignment Operators

These operators are used to assign values to a variable. The left side operand of the assignment operator is a variable, and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type of the operand on the left side. Otherwise, the compiler will raise an error. This means that the assignment operators have 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, the right-hand side value must be declared before using it or should be a constant. The general format of the assignment operator is,

```
variable operator value;
```

*Types of Assignment Operators in Java*

The Assignment Operator is generally of two types. They are:

**1. Simple Assignment Operator:** The Simple Assignment Operator is used with the "=" sign where the left side consists of the operand and the right side consists of a value. The value of the right side must be of the same data type that has been defined on the left side.

**2. Compound Assignment Operator:** The Compound Operator is used where +,-,*, and / is used along with the = operator.

Let's look at each of the assignment operators and how they operate:

*1. (=) operator:*

This is the most straightforward assignment operator, which is used to assign the value on the right to the variable on the left. This is the basic definition of an assignment operator and how it functions.

**Syntax:**

```
num1 = num2;
```

**Example:**

```
a = 10;
ch = 'y';
```

- Java

```java
// Java code to illustrate "=" operator


import java.io.*;


class Assignment {

    public static void main(String[] args)

    {
```

```
        // Declaring variables
        int num;
        String name;

        // Assigning values
        num = 10;
        name = "GeeksforGeeks";

        // Displaying the assigned values
        System.out.println("num is assigned: " + num);
        System.out.println("name is assigned: " + name);
    }
}
```

**Output**

```
num is assigned: 10
name is assigned: GeeksforGeeks
```

*2. (+=) operator:*

This operator is a compound of '+' and '=' operators. It operates by adding the current value of the variable on the left to the value on the right and then assigning the result to the operand on the left.

**Syntax:**

```
num1 += num2;
```

**Example:**

```
a += 10

This means,
a = a + 10
```

- Java

```
// Java code to illustrate "+="

import java.io.*;

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

```
        // Declaring variables
        int num1 = 10, num2 = 20;

        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Adding & Assigning values
        num1 += num2;

        // Displaying the assigned values
        System.out.println("num1 = " + num1);
    }
}
```

**Output**

```
num1 = 10
num2 = 20
num1 = 30
```

***Note:*** *The compound assignment operator in Java performs implicit type casting. Let's consider a scenario where* `x` *is an* `int` *variable with a value of 5.*

***int x = 5;***

*If you want to add the double value 4.5 to the integer variable* `x` *and print its value, there are two methods to achieve this:*

***Method 1:*** *x = x + 4.5*

***Method 2:*** *x += 4.5*

*As per the previous example, you might think both of them are equal. But in reality, Method 1 will throw a runtime error stating the "**incompatible types: possible lossy conversion from double to int**", Method 2 will run without any error and prints 9 as output.*

## Reason for the Above Calculation

***Method 1*** *will result in a runtime error stating "incompatible types: possible lossy conversion from double to int." The reason is that the addition of an* `int` *and a* `double` *results in a* `double` *value. Assigning this* `double` *value back to the* `int` *variable* `x` *requires an explicit type casting because it may result in a loss of precision. Without the explicit cast, the compiler throws an error.*

***Method 2*** *will run without any error and print the value 9 as output. The compound assignment operator* `+=` *performs an implicit type conversion, also known as*

*an **automatic narrowing primitive conversion** from **`double`** to **`int`**. It is equivalent to **`x = (int) (x + 4.5)`**, where the result of the addition is explicitly cast to an **`int`**. The fractional part of the **`double`** value is truncated, and the resulting **`int`** value is assigned back to **`x`**.*
*It is advisable to use Method 2 (`x += 4.5`) to avoid runtime errors and to obtain the desired output.*

Same **automatic narrowing primitive conversion** is applicable for other compound assignment operators as well, including **`-=`**, **`*=`**, **`/=`**, and **`%=`**.

*3. (-=) operator:*

This operator is a compound of '-' and '=' operators. It operates by subtracting the variable's value on the right from the current value of the variable on the left and then assigning the result to the operand on the left.

**Syntax:**

```
num1 -= num2;
```

**Example:**

```
a -= 10
```

```
This means,
a = a - 10
```

- Java

```java
// Java code to illustrate "-="

import java.io.*;

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

        // Declaring variables
        int num1 = 10, num2 = 20;

        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Subtracting & Assigning values
        num1 -= num2;
```

```
        // Displaying the assigned values
        System.out.println("num1 = " + num1);
    }
}
```

**Output**

```
num1 = 10
num2 = 20
num1 = -10
```

*4. (*=) operator:*

This operator is a compound of '*' and '=' operators. It operates by multiplying the current value of the variable on the left to the value on the right and then assigning the result to the operand on the left.

**Syntax:**

```
num1 *= num2;
```

**Example:**

```
a *= 10
This means,
a = a * 10
```

- Java

```
// Java code to illustrate "*="

import java.io.*;

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

        // Declaring variables
        int num1 = 10, num2 = 20;

        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Multiplying & Assigning values
```

```
        num1 *= num2;


        // Displaying the assigned values
        System.out.println("num1 = " + num1);
    }
}
```

**Output**

```
num1 = 10
num2 = 20
num1 = 200
```

*5. (/=) operator:*

This operator is a compound of '/' and '=' operators. It operates by dividing the current value of the variable on the left by the value on the right and then assigning the quotient to the operand on the left.

**Syntax:**

```
num1 /= num2;
```

**Example:**

```
a /= 10
This means,
a = a / 10
```

- Java

```
// Java code to illustrate "/="

import java.io.*;

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

        // Declaring variables
        int num1 = 20, num2 = 10;

        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);
```

```
        // Dividing & Assigning values
        num1 /= num2;

        // Displaying the assigned values
        System.out.println("num1 = " + num1);
    }
}
```

**Output**

```
num1 = 20
num2 = 10
num1 = 2
```

*6. (%=) operator:*

This operator is a compound of '%' and '=' operators. It operates by dividing the current value of the variable on the left by the value on the right and then assigning the remainder to the operand on the left.

**Syntax:**

```
num1 %= num2;
```

**Example:**

```
a %= 3

This means,
a = a % 3
```

- Java

```
// Java code to illustrate "%="

import java.io.*;

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

        // Declaring variables
        int num1 = 5, num2 = 3;
```

```
        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Moduling & Assigning values
        num1 %= num2;

        // Displaying the assigned values
        System.out.println("num1 = " + num1);
    }
}
```

**Output**

```
num1 = 5
num2 = 3
num1 = 2
```

**Operators** constitute the basic building block to any programming language. Java too provides many types of operators which can be used according to the need to perform various calculations and functions, be it logical, arithmetic, relational, etc. They are classified based on the functionality they provide.

**Types of Operators:**

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators

**Java Relational Operators** are a bunch of binary operators used to check for relations between two operands, including equality, greater than, less than, etc. They return a boolean result after the comparison and are extensively used in looping statements as well as conditional if-else statements and so on. The general format of representing relational operator is:

**Syntax:**

```
variable1 relation_operator variable2
```

Let us look at each one of the relational operators in Java:

**Operator 1: 'Equal to' operator (==)**

This operator is used to check whether the two given operands are equal or not. The operator returns true if the operand at the left-hand side is equal to the right-hand side, else false.

**Syntax:**

```
var1 == var2
```

**Illustration:**

```
var1 = "GeeksforGeeks"
var2 = 20
var1 == var2 results in false
```

**Example:**

- Java

```java
// Java Program to Illustrate equal to Operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Initializing variables
        int var1 = 5, var2 = 10, var3 = 5;

        // Displaying var1, var2, var3
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 == var2: "
                           + (var1 == var2));
```

```
        // Comparing var1 and var3 and
        // printing corresponding boolean value
        System.out.println("var1 == var3: "
                           + (var1 == var3));
    }
}
```

**Output**

```
Var1 = 5
Var2 = 10
Var3 = 5
var1 == var2: false
var1 == var3: true
```

**Operator 2: 'Not equal to' Operator(!=)**

This operator is used to check whether the two given operands are equal or not. It functions opposite to that of the equal-to-operator. It returns true if the operand at the left-hand side is not equal to the right-hand side, else false.

**Syntax:**

```
var1 != var2
```

**Illustration:**

```
var1 = "GeeksforGeeks"
var2 = 20

var1 != var2 results in true
```

**Example:**

- Java

```
// Java Program to Illustrate No- equal-to Operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {
```

```
    // Main driver method
    public static void main(String[] args)
    {
        // Initializing variables
        int var1 = 5, var2 = 10, var3 = 5;

        // Displaying var1, var2, var3
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 != var2: "
                           + (var1 != var2));

        // Comparing var1 and var3 and
        // printing corresponding boolean value
        System.out.println("var1 != var3: "
                           + (var1 != var3));
    }
}
```

**Output**

```
Var1 = 5
Var2 = 10
Var3 = 5
var1 != var2: true
var1 != var3: false
```

**Operator 3: 'Greater than' operator(>)**

This checks whether the first operand is greater than the second operand or not. The operator returns true when the operand at the left-hand side is greater than the right-hand side.

**Syntax:**

var1 > var2

**Illustration:**

var1 = 30

var2 = 20

var1 > var2 results in true

**Example:**

- Java

```java
// Java code to Illustrate Greater than operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Initializing variables
        int var1 = 30, var2 = 20, var3 = 5;

        // Displaying var1, var2, var3
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 > var2: " + (var1 > var2));

        // Comparing var1 and var3 and
        // printing corresponding boolean value
```

```
        System.out.println("var3 > var1: "
                           + (var3 >= var1));
    }
}
```

**Output**

```
Var1 = 30
Var2 = 20
Var3 = 5
var1 > var2: true
var3 > var1: false
```

**Operator 4: 'Less than' Operator(<)**

This checks whether the first operand is less than the second operand or not. The operator returns true when the operand at the left-hand side is less than the right-hand side. It functions opposite to that of the greater-than operator.

**Syntax:**

```
var1 < var2
```

**Illustration:**

```
var1 = 10
var2 = 20

var1 < var2 results in true
```

**Example:**

- Java

```
// Java code to Illustrate Less than Operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
```

```
    {
        // Initializing variables
        int var1 = 10, var2 = 20, var3 = 5;

        // Displaying var1, var2, var3
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 < var2: " + (var1 < var2));

        // Comparing var2 and var3 and
        // printing corresponding boolean value
        System.out.println("var2 < var3: " + (var2 < var3));
    }
}
```

**Output**

```
Var1 = 10
Var2 = 20
Var3 = 5
var1 < var2: true
var2 < var3: false
```

**Operator 5: Greater than or equal to (>=)**

This checks whether the first operand is greater than or equal to the second operand or not. The operator returns true when the operand at the left-hand side is greater than or equal to the right-hand side.

**Syntax:**

```
var1 >= var2
```

**Illustration:**

```
var1 = 20
var2 = 20
var3 = 10
```

var1 >= var2 results in true

var2 >= var3 results in true

**Example:**

- Java

```java
// Java Program to Illustrate Greater than or equal to
// Operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Initializing variables
        int var1 = 20, var2 = 20, var3 = 10;

        // Displaying var1, var2, var3
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 >= var2: "
                           + (var1 >= var2));

        // Comparing var2 and var3 and
        // printing corresponding boolean value
        System.out.println("var2 >= var3: "
```

```
                            + (var2 >= var3));
    }
}
```

**Output**

```
Var1 = 20
Var2 = 20
Var3 = 10
var1 >= var2: true
var2 >= var3: true
```

**Operator 6: Less than or equal to (<=)**

This checks whether the first operand is less than or equal to the second operand or not. The operator returns true when the operand at the left-hand side is less than or equal to the right-hand side.

**Syntax:**

```
var1 <= var2
```

**Illustration:**

```
var1 = 10
var2 = 10
var3 = 9

var1 <= var2 results in true
var2 <= var3 results in false
```

**Example:**

- Java

```
// Java Program to Illustrate Less
// than or equal to operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {
```

```
    // Main driver method
    public static void main(String[] args)
    {
        // Initializing variables
        int var1 = 10, var2 = 10, var3 = 9;

        // Displaying var1, var2, var3
        System.out.println("Var1 = " + var1);
        System.out.println("Var2 = " + var2);
        System.out.println("Var3 = " + var3);

        // Comparing var1 and var2 and
        // printing corresponding boolean value
        System.out.println("var1 <= var2: "
                           + (var1 <= var2));

        // Comparing var2 and var3 and
        // printing corresponding boolean value
        System.out.println("var2 <= var3: "
                           + (var2 <= var3));
    }
}
```

**Output**

```
Var1 = 10
Var2 = 10
Var3 = 9
var1 <= var2: true
var2 <= var3: false
```

program that implements all relational operators in Java for user input:

- Java

```
import java.util.Scanner;
```

```java
public class RelationalOperators {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);

    //System.out.println("Enter first number: ");
   // int num1 = scan.nextInt();

   // System.out.println("Enter second number: ");
   // int num2 = scan.nextInt();

    int num1 =1;
    int num2 = 2;



    System.out.println("num1 > num2 is " + (num1 > num2));
    System.out.println("num1 < num2 is " + (num1 < num2));
    System.out.println("num1 >= num2 is " + (num1 >= num2));
    System.out.println("num1 <= num2 is " + (num1 <= num2));
    System.out.println("num1 == num2 is " + (num1 == num2));
    System.out.println("num1 != num2 is " + (num1 != num2));
  }
}
```

**Output**

```
num1 > num2 is false
num1 < num2 is true
num1 >= num2 is false
num1 <= num2 is true
num1 == num2 is false
num1 != num2 is true
```

## Explanation

*The code above implements all relational operators in Java for user input. The following is an explanation of the code in detail:*

- **Importing the Scanner class**: The code starts by importing the Scanner class, which is used to read input from the user. The Scanner class is part of the java.util package, so it needs to be imported in order to be used.
- **Defining the main method:** The program then defines the main method, which is the starting point of the program. This is where the program logic is executed.
- **Reading user input:** The code uses the Scanner object to read two integers from the user. The first number is stored in the num1 variable, and the second number is stored in the num2 variable.
- **Using relational operators:** The code then uses the relational operators >, <, >=, <=, ==, and != to compare the values of num1 and num2. The results of these comparisons are stored in boolean variables, which are then displayed to the user.
- **Displaying results**: The results of the comparisons are displayed to the user using the System.out.println method. This method is used to print a message to the console.
- **Closing the Scanner object:** Finally, the code closes the Scanner object to prevent memory leaks and to ensure that the program terminates cleanly.

The relational operators in Java return a boolean value of true or false, depending on the result of the comparison. For example, num1 > num2 returns true if num1 is greater than num2, and false otherwise. Similarly, num1 == num2 returns true if num1 is equal to num2, and false otherwise.

## Advantages

There are several advantages of using relational operators in Java, including:

1. **Easy to understand**: Relational operators are simple and easy to understand, making it easy to write code that performs comparisons.
2. **Versatile:** Relational operators can be used to compare values of different data types, such as integers, floating-point numbers, and strings.
3. **Essential for making decisions:** Relational operators are essential for making decisions in a program, as they allow you to control the flow of a program based on the results of comparisons.
4. **Efficient:** Relational operators are efficient, as they perform comparisons quickly and accurately.
5. **Reusable code**: The code that uses relational operators can be reused in different parts of a program, making it easier to maintain and update the code.

6. **Improved code readability**: By using relational operators, you can make your code more readable and understandable, as the comparisons are clearly stated in the code.
7. **Debugging made easier**: Relational operators make debugging easier, as you can use them to check the values of variables and to find out where a problem is occurring in your code.

Logical operators are used to perform logical “AND”, “OR” and “NOT” operations, i.e. the function similar to AND gate and OR gate in digital electronics. They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition under particular consideration. One thing to keep in mind is, while using AND operator, the second condition is not evaluated if the first one is false. Whereas while using OR operator, the second condition is not evaluated if the first one is true, i.e. the AND and OR operators have a short-circuiting effect. Used extensively to test for several conditions for making a decision.

1. **AND Operator** ( **&&** ) – if( a && b ) [if true execute else don’t]
2. **OR Operator** ( **||** ) – if( a || b) [if one of them is true to execute else don’t]
3. **NOT Operator** ( **!** ) – !(a<b) [returns false if a is smaller than b]

## Example For Logical Operator in Java

Here is an example depicting all the operators where the values of variables a, b, and c are kept the same for all the situations.

```
a = 10, b = 20, c = 30

For AND operator:

Condition 1: c > a
Condition 2: c > b

Output: True [Both Conditions are true]

For OR Operator:

Condition 1: c > a
Condition 2: c > b

Output: True [One of the Condition if true]

For NOT Operator:
Condition 1: c > a
Condition 2: c > b

Output: False [Because the result was true and NOT operator did it's opposite]
```

## 1. Logical ‘AND’ Operator (&&)

This operator returns true when both the conditions under consideration are satisfied or are true. If even one of the two yields false, the operator results false. In Simple terms, ***cond1 && cond2 returns true when both cond1 and cond2 are true (i.e. non-zero***).

**Syntax:**

```
condition1 && condition2
```

**Illustration:**

```
a = 10, b = 20, c = 20

condition1: a < b
condition2: b == c

if(condition1 && condition2)
d = a + b + c

// Since both the conditions are true
d = 50.
```

**Example**

- Java

```java
// Java code to illustrate
// logical AND operator

import java.io.*;

class Logical {
    public static void main(String[] args)
    {
        // initializing variables
        int a = 10, b = 20, c = 20, d = 0;

        // Displaying a, b, c
        System.out.println("Var1 = " + a);
        System.out.println("Var2 = " + b);
        System.out.println("Var3 = " + c);

        // using logical AND to verify
        // two constraints
        if ((a < b) && (b == c)) {
```

```
            d = a + b + c;
            System.out.println("The sum is: " + d);
        }
        else
            System.out.println("False conditions");
    }
}
```

**Output**

```
Var1 = 10
Var2 = 20
Var3 = 20
The sum is: 50
```

Now in the below example, we can see the short-circuiting effect. Here when the execution reaches to if statement, the first condition inside the if statement is false and so the second condition is never checked. Thus the ++b(pre-increment of b) never happens and b remains unchanged.

**Example:**

- Java

```
import java.io.*;

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

        // initializing variables
        int a = 10, b = 20, c = 15;

        // displaying b
        System.out.println("Value of b : " + b);

        // Using logical AND
        // Short-Circuiting effect as the first condition is
```

```
        // false so the second condition is never reached
        // and so ++b(pre increment) doesn't take place and
        // value of b remains unchanged
        if ((a > c) && (++b > c)) {
            System.out.println("Inside if block");
        }

        // displaying b
        System.out.println("Value of b : " + b);
    }
}
```

**Output:**

```
Value of b : 20
Value of b : 20
```

*The output of AND Operator*

## 2. Logical 'OR' Operator (||)

This operator returns true when one of the two conditions under consideration is satisfied or is true. If even one of the two yields true, the operator results true. To make the result false, both the constraints need to return false.

**Syntax:**

```
condition1 || condition2
```

**Example:**

```
a = 10, b = 20, c = 20

condition1: a < b
condition2: b > c

if(condition1 || condition2)
d = a + b + c

// Since one of the condition is true
d = 50.
```

- Java

```java
// Java code to illustrate
// logical OR operator

import java.io.*;

class Logical {
    public static void main(String[] args)
    {
        // initializing variables
        int a = 10, b = 1, c = 10, d = 30;

        // Displaying a, b, c
        System.out.println("Var1 = " + a);
        System.out.println("Var2 = " + b);
        System.out.println("Var3 = " + c);
        System.out.println("Var4 = " + d);

        // using logical OR to verify
        // two constraints
        if (a > b || c == d)
            System.out.println(
                "One or both + the conditions are true");
        else
            System.out.println(
                "Both the + conditions are false");
    }
}
```

**Output**

```
Var1 = 10
Var2 = 1
Var3 = 10
Var4 = 30
One or both + the conditions are true
```

Now in the below example, we can see the short-circuiting effect for OR operator. Here when the execution reaches to if statement, the first condition inside the if statement is true and so the second condition is never checked. Thus the ++b (pre-increment of b) never happens and b remains unchanged.

**Example**

- Java

```
import java.io.*;


class ShortCircuitingInOR {

    public static void main (String[] args) {


      // initializing variables

        int a = 10, b = 20, c = 15;


        // displaying b

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


        // Using logical OR

        // Short-circuiting effect as the first condition is true

        // so the second condition is never reached

        // and so ++b (pre-increment) doesn't take place and

        // value of b remains unchanged

        if((a < c) || (++b < c))

            System.out.println("Inside if");


        // displaying b

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


    }
}
```

**Output**

```
Value of b: 20
Inside if
Value of b: 20
```

## 3. Logical ‘NOT’ Operator (!)

Unlike the previous two, this is a unary operator and returns true when the condition under consideration is not satisfied or is a false condition. Basically, if the condition is false, the operation returns true and when the condition is true, the operation returns false.

**Syntax:**

```
!(condition)
```

**Example:**

```
a = 10, b = 20


!(a<b) // returns false

!(a>b) // returns true
```

- Java

```java
// Java code to illustrate

// logical NOT operator

import java.io.*;


class Logical {

    public static void main(String[] args)

    {

        // initializing variables

        int a = 10, b = 1;


        // Displaying a, b, c

        System.out.println("Var1 = " + a);

        System.out.println("Var2 = " + b);


        // Using logical NOT operator

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

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

    }

}
```

**Output**

```
Var1 = 10
```

```
Var2 = 1
!(a < b) = true
!(a > b) = false
```

4. Implementing all logical operators on Boolean values (By default values – TRUE or FALSE)

*Syntax –*

```
boolean a = true;
boolean b = false;
```

*program –*

- Java

```java
public class LogicalOperators {

    public static void main(String[] args) {

        boolean a = true;

        boolean b = false;


        System.out.println("a: " + a);

        System.out.println("b: " + b);

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

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

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

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

    }
}
```

**Output**

```
a: true
b: false
a && b: false
a || b: true
!a: false
!b: true
```

*Explanation:*

- The code above is a Java program that implements all logical operators with default values. The program defines a class LogicalOperators with a main method.
- In the main method, two boolean variables a and b are defined and assigned default values true and false, respectively.
- The program then prints the values of a and b to the console using the System.out.println method. This allows us to verify the values assigned to a and b.
- Next, the program calculates the result of the logical operators && (and), || (or), and ! (not) applied to a and b. The results of these operations are also printed to the console using the System.out.println method.
- The && operator returns true if both operands are true; otherwise, it returns false. In this case, the result of a && b is false.
- The || operator returns true if either operand is true; otherwise, it returns false. In this case, the result of a || b is true.
- The ! operator negates the value of its operand. If the operand is true, the result is false, and if the operand is false, the result is true. In this case, the results of !a and !b are false and true, respectively.
- The output of the program shows the truth table for all logical operators. This table provides a visual representation of how these operators behave for all possible combinations of true and false inputs.

## 5. Advantages of logical operators:

*The advantages of using logical operators in a program include:*

1. **Readability**: Logical operators make code more readable by providing a clear and concise way to express complex conditions. They are easily recognizable and make it easier for others to understand the code.
2. **Flexibility**: Logical operators can be combined in various ways to form complex conditions, making the code more flexible. This allows developers to write code that can handle different scenarios and respond dynamically to changes in the program's inputs.
3. **Reusability**: By using logical operators, developers can write code that can be reused in different parts of the program. This reduces the amount of code that needs to be written and maintained, making the development process more efficient.
4. **Debugging**: Logical operators can help to simplify the debugging process. If a condition does not behave as expected, it is easier to trace the problem by examining the results of each logical operator rather than having to navigate through a complex code structure.

**6. Disadvantages of logical operators :**

1. **Short-circuit evaluation:** One of the main advantages of logical operators is that they support short-circuit evaluation. This means that if the value of an expression can be determined based on the left operand alone, the right operand is not evaluated at all. While this can be useful for optimizing code and preventing unnecessary computations, it can also lead to subtle bugs if the right operand has side effects that are expected to be executed.
2. **Limited expressiveness:** Logical operators have a limited expressive power compared to more complex logical constructs like if-else statements and switch-case statements. This can make it difficult to write complex conditionals that require more advanced logic, such as evaluating multiple conditions in a specific order.
3. **Potential for confusion:** In some cases, the use of logical operators can lead to confusion or ambiguity in the code. For example, consider the expression a or b and c. Depending on the intended order of operations, this expression can have different interpretations. To avoid this kind of confusion, it is often recommended to use parentheses to explicitly specify the order of evaluation.
4. **Boolean coercion:** Logical operators can sometimes lead to unexpected behavior when used with non-Boolean values. For example, when using the or operator, the expression a or b will evaluate to a if a is truthy, and b otherwise. This can lead to unexpected results if a or b are not actually Boolean values, but instead have a truthy or falsy interpretation that does not align with the programmer's intentions.

*Overall, logical operators are an important tool for developers and play a crucial role in the implementation of complex conditions in a program. They help to improve the readability, flexibility, reusability, and debuggability of the code.*

**Operators** constitute the basic building block to any programming language. Java too provides many types of operators which can be used according to the need to perform various calculations and functions, be it logical, arithmetic, relational, etc. They are classified based on the functionality they provide. Here are a few types:

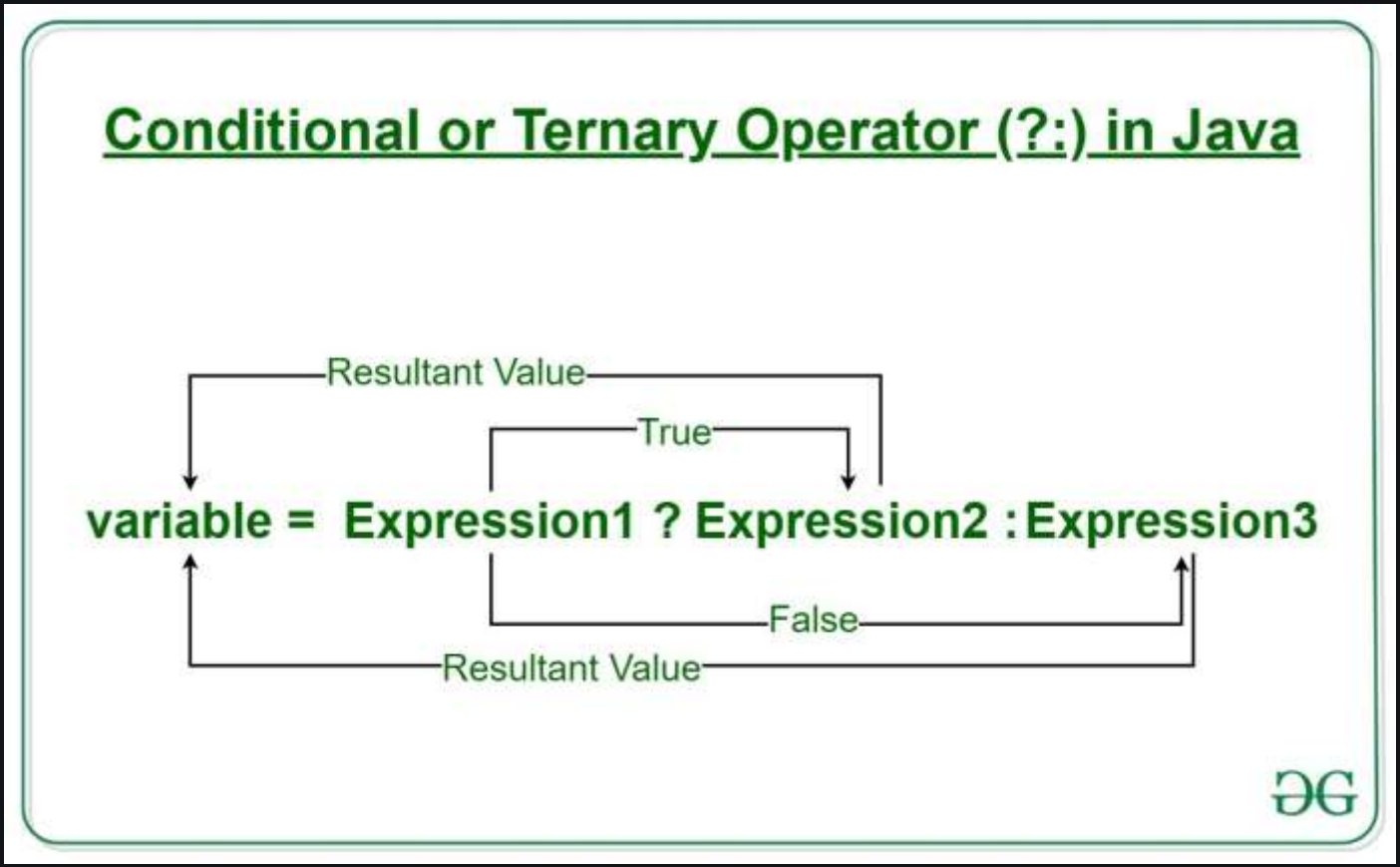
1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators

This article explains all that one needs to know regarding Arithmetic Operators.

## Ternary Operator

Java ternary operator is the only conditional operator that takes three operands. It's a one-liner replacement for the if-then-else statement and is used a lot in Java

programming. We can use the ternary operator in place of if-else conditions or even switch conditions using nested ternary operators. Although it follows the same algorithm as of if-else statement, the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.




**Syntax:**

```
variable = Expression1 ? Expression2: Expression3
```

If operates similarly to that of the if-else statement as in *Exression2* is executed if *Expression1* is true else *Expression3* is executed.

```
if(Expression1)
{
    variable = Expression2;
}
else
{
    variable = Expression3;
}
```

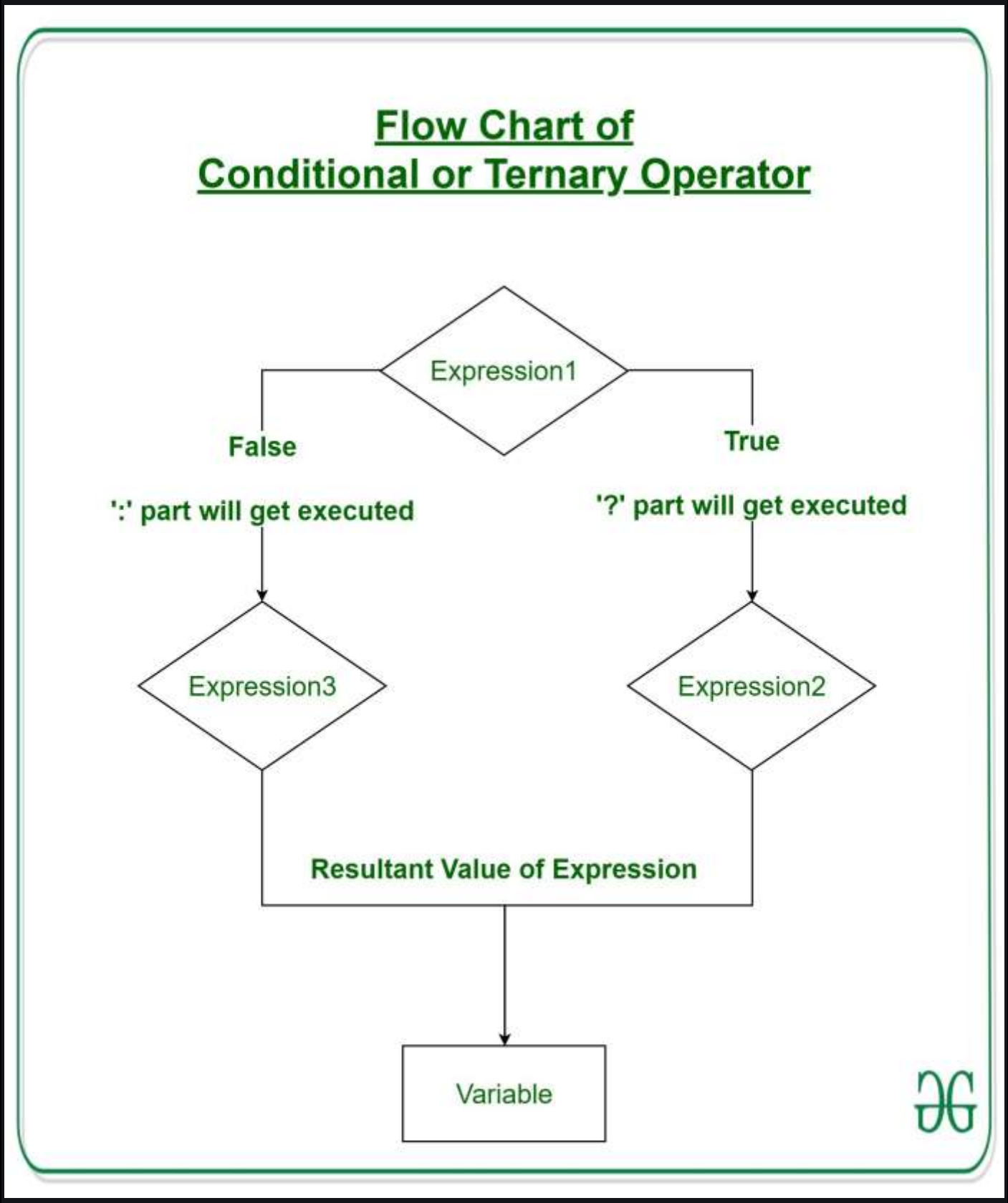
**Example:**

```
num1 = 10;
num2 = 20;

res=(num1>num2) ? (num1+num2):(num1-num2)
```

```
Since num1<num2,
the second operation is performed
res = num1-num2 = -10
```

*Flowchart of Ternary Operation*




**Example 1:**

- Java

```java
// Java program to find largest among two
// numbers using ternary operator

import java.io.*;

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

        // variable declaration
        int n1 = 5, n2 = 10, max;

        System.out.println("First num: " + n1);
        System.out.println("Second num: " + n2);

        // Largest among n1 and n2
        max = (n1 > n2) ? n1 : n2;

        // Print the largest number
        System.out.println("Maximum is = " + max);
    }
}
```

**Output**

```
First num: 5
Second num: 10
Maximum is = 10
```

**Time Complexity:** O(1)
**Auxiliary Space:** O(1)
**Example 2:**

- Java

```java
// Java code to illustrate ternary operator
```

```java
import java.io.*;

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

        // variable declaration
        int n1 = 5, n2 = 10, res;

        System.out.println("First num: " + n1);
        System.out.println("Second num: " + n2);

        // Performing ternary operation
        res = (n1 > n2) ? (n1 + n2) : (n1 - n2);

        // Print the largest number
        System.out.println("Result = " + res);
    }
}
```

**Output**

```
First num: 5
Second num: 10
Result = -5
```

**Time Complexity:** O(1)
**Auxiliary Space:** O(1)

*Example no 3:*
*Implementing ternary operator on Boolean values-*

- Java

```java
public class TernaryOperatorExample {
    public static void main(String[] args)
    {
        boolean condition = true;
```

```
        String result = (condition) ? "True" : "False";

        System.out.println(result);

    }
}
```

**Output**

```
True
```

*Explanation –*

In this program, a Boolean variable condition is declared and assigned the value true. Then, the ternary operator is used to determine the value of the result string. If the condition is true, the value of result will be “True”, otherwise it will be “False”. Finally, the value of result is printed to the console.

*Advantages of ternary operator –*

- **Compactness**: The ternary operator allows you to write simple if-else statements in a much more concise way, making the code easier to read and maintain.
- **Improved readability**: When used correctly, the ternary operator can make the code more readable by making it easier to understand the intent behind the code.
- **Increased performance:** Since the ternary operator evaluates a single expression instead of executing an entire block of code, it can be faster than an equivalent if-else statement.
- **Simplification of nested if-else statements:** The ternary operator can simplify complex logic by providing a clean and concise way to perform conditional assignments.
- **Easy to debug**: If a problem occurs with the code, the ternary operator can make it easier to identify the cause of the problem because it reduces the amount of code that needs to be examined.

It’s worth noting that the ternary operator is not a replacement for all if-else statements. For complex conditions or logic, it’s usually better to use an if-else statement to avoid making the code more difficult to understand.

## Bitwise Operators

Bitwise operators are used to performing the manipulation of individual bits of a number. They can be used with any integral type (char, short, int, etc.). They are used when performing update and query operations of the Binary indexed trees.

Now let’s look at each one of the bitwise operators in Java:

**1. Bitwise OR (|)**

This operator is a binary operator, denoted by '|'. It returns bit by bit OR of input values, i.e., if either of the bits is 1, it gives 1, else it shows 0.

**Example:**

```
a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7
  0101
| 0111
 ________
  0111  = 7 (In decimal)
```

**2. Bitwise AND (&)**

This operator is a binary operator, denoted by '&.' It returns bit by bit AND of input values, i.e., if both bits are 1, it gives 1, else it shows 0.

**Example:**

```
a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise AND Operation of 5 and 7
  0101
& 0111
 ________
  0101  = 5 (In decimal)
```

**3. Bitwise XOR (^)**

This operator is a binary operator, denoted by '^.' It returns bit by bit XOR of input values, i.e., if corresponding bits are different, it gives 1, else it shows 0.

**Example:**

```
a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7
  0101
^ 0111
 ________
  0010  = 2 (In decimal)
```

**4. Bitwise Complement (~)**

This operator is a unary operator, denoted by '~.' It returns the one's complement representation of the input value, i.e., with all bits inverted, which means it makes every 0 to 1, and every 1 to 0.

**Example:**

```
a = 5 = 0101 (In Binary)


Bitwise Complement Operation of 5


~ 0101

 ________
  1010  = 10 (In decimal)
```

***Note:*** *Compiler will give 2's complement of that number, i.e., 2's complement of 10 will be -6.*

- Java

```java
// Java program to illustrate
// bitwise operators


public class operators {
    public static void main(String[] args)
    {
        // Initial values
        int a = 5;
        int b = 7;


        // bitwise and
        // 0101 & 0111=0101 = 5
        System.out.println("a&b = " + (a & b));


        // bitwise or
        // 0101 | 0111=0111 = 7
        System.out.println("a|b = " + (a | b));


        // bitwise xor
        // 0101 ^ 0111=0010 = 2
```

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


        // bitwise not
        // ~00000000 00000000 00000000 00000101=11111111 11111111 11111111 11111010
        // will give 2's complement (32 bit) of 5 = -6
        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
```

Auxiliary space:O(1)

Time complexity:O(1)

- Java

```
// Demonstrating the bitwise logical operators



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


        String binary[]={
```

```
          "0000","0001","0010","0011","0100","0101",
          "0110","0111","1000","1001","1010",
          "1011","1100","1101","1110","1111"
        };

      // initializing the values of a and b
      int a=3; // 0+2+1 or 0011 in binary
      int b=6; // 4+2+0 or 0110 in binary

      // bitwise or
      int c= a | b;

      // bitwise and
      int d= a & b;

      // bitwise xor
      int e= a ^ b;

      // bitwise not
      int f= (~a & b)|(a &~b);
      int g= ~a & 0x0f;


      System.out.println(" a= "+binary[a]);
      System.out.println(" b= "+binary[b]);
      System.out.println(" a|b= "+binary);
      System.out.println(" a&b= "+binary[d]);
      System.out.println(" a^b= "+binary[e]);
      System.out.println("~a & b|a&~b= "+binary[f]);
      System.out.println("~a= "+binary[g]);
    }
}
```

**Output**

```
 a= 0011
 b= 0110
 a|b= 0111
 a&b= 0010
 a^b= 0101
~a & b|a&~b= 0101
~a= 1100
```

**Bit-Shift Operators (Shift Operators)**

Shift 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.

**Syntax:**

```
number shift_op number_of_places_to_shift;
```

**Types of Shift Operators:**

Shift Operators are further divided into 4 types. These are:

1. Signed Right shift operator (>>)
2. Unsigned Right shift operator (>>>)
3. Left shift operator(<<)
4. Unsigned Left shift operator (<<<)

***Note:*** *For more detail about the Shift Operators in Java, refer Shift Operator in Java.*

*program to implement all Bitwise operators in java for user input*

- Java

```java
import java.util.Scanner;


public class BitwiseOperators {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);


        System.out.print("Enter first number: ");

        int num1 = input.nextInt();


        System.out.print("Enter second number: ");

        int num2 = input.nextInt();


        System.out.println("Bitwise AND: " + (num1 & num2));
```

```
        System.out.println("Bitwise OR: " + (num1 | num2));
        System.out.println("Bitwise XOR: " + (num1 ^ num2));
        System.out.println("Bitwise NOT: " + (~num1));
        System.out.println("Bitwise Left Shift: " + (num1 << 2));
        System.out.println("Bitwise Right Shift: " + (num1 >> 2));
        System.out.println("Bitwise Unsigned Right Shift: " + (num1 >>> 2));

        input.close();
    }
}
```

**Input**

```
Enter first number: 4
Enter second number: 8
```

**Output**

```
Bitwise AND: 0
Bitwise OR: 12
Bitwise XOR: 12
Bitwise NOT: -5
Bitwise Left Shift: 16
Bitwise Right Shift: 1
Bitwise Unsigned Right Shift: 1
```

*Explanation*

This program prompts the user to enter two numbers, num1 and num2. It then performs the following bitwise operations using the &, |, ^, ~, <<, >>, and >>> operators:

Bitwise AND
Bitwise OR
Bitwise XOR
Bitwise NOT
Bitwise Left Shift
Bitwise Right Shift
Bitwise Zero Fill Right Shift

*Advantages*

The advantages of using Bitwise Operators in Java are:

1. **Speed:** Bitwise operations are much faster than arithmetic operations as they operate directly on binary representations of numbers.

2. **Space Optimization:** Bitwise operations can be used to store multiple values in a single variable, which can be useful when working with limited memory.
3. **Bit Manipulation:** Bitwise operators allow for precise control over individual bits of a number, which can be useful in various applications such as cryptography, error detection, and compression.
4. **Code Simplification**: Bitwise operations can simplify the code by reducing the number of conditional statements and loops required to perform certain tasks.
5. **Improved Readability**: Bitwise operations can make the code more readable by encapsulating complex logic into a single operation, making the code easier to understand and maintain.

In summary, Bitwise Operators are an important tool for optimizing performance, improving code readability, and reducing code complexity in Java applications.